Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Une approche méthodologique au développement de transactions e-business application au domaine de la construction

Absil, Fabrice

Award date:
2004

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 18. Apr. 2024

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Une approche méthodologique au développement de transactions e-business : Application au domaine de la construction.

Fabrice ABSIL

Année Académique 2003 – 2004

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique

TABLE DES MATIÈRES	**1 - 4**
GLOSSAIRE	**5 - 6**
INTRODUCTION	**7 - 8**
PARTIE 1 : Fondements théoriques	**9 - 36**
Chapitre 1 : UML	**9 - 20**
1.1- Introduction 1.1.1 UML : un standard 1.1.2 Principes UML 1.2- Diagramme de cas d'utilisation 1.2.1 Les acteurs 1.2.2 Les cas d'utilisation 1.2.3 Les relations entre cas d'utilisation 1.2.4 Description des cas d'utilisation 1.3 Diagramme de classes 1.3.1 Les classes 1.3.2 Les attributs 1.3.3 Les associations 1.3.4 Les classes-associations 1.3.5 Les agrégations 1.3.6 Les compositions 1.3.7 La généralisation 1.3.8 Les contraintes et les propriétés de généralisation 1.4- Diagramme d'activités 1.4.1 Les états-action 1.4.2 Les transitions 1.4.3 Les barres de synchronisation 1.4.4 Les travées (swimlanes) 1.4.5 Les flots d'objets entre actions 1.5- Les stéréotypes	
Chapitre 2 : XML	**21 - 36**
2.1- Introduction 2.1.1 XML : un standard 2.1.2 Principes XML 2.2-Les documents XML 2.2.1 Un document bien formé 2.2.2 Un document valide 2.3 La syntaxe 2.3.1 Les éléments 2.3.2 Les attributs 2.3.3 Les commentaires 2.3.4 Les instructions de traitement 2.3.5 La section de données de caractères 2.3.6 La déclaration d'un type de document 2.3.7 Les namespaces 2.4 La DTD : comment structurer les documents XML 2.4.1 Qu'est ce qu'une DTD ? 2.4.2 Déclarer un élément	

2.4.3 Déclarer un attribut
2.4.4 Limites des DTD

2.5. Schémas XML
2.5.1 Objectifs
2.5.2 Avantages des Schémas XML
2.5.3 Déclarer les Schémas XML
2.5.4 Déclarer un élément
2.5.5 Déclarer un attribut
2.5.6 Le typage des données
2.5.7 Le modèle de contenu et les connecteurs
2.5.8 L'occurrence d'un élément
2.5.9 Elément local ou global
2.5.10 Création de types complexes à partir de types simples
2.5.11 Les types complexes vides
2.5.12 Limites des Schémas XML

PARTIE 2 : Conception et développement de transactions E-business 37 - 60

Chapitre 3 : Transaction e-Business 37 - 40

3.1 Contexte actuel
3.2 Les principaux modèles d'échange électronique
3.3 Les implications technologiques d'une solution B2B
3.4 L'approche EDI
3.4.1 Historique
3.4.2 EDIFACT
3.4.3 Quel avenir pour EDIFACT ?

Chapitre 4 : ebXML 41 - 48

4.1 XML pour les échanges électroniques
4.1.1 Avantages de XML
4.1.2 Limites de XML
4.1.3 Quel avenir pour XML dans les échanges électroniques ?

4.2 Le projet ebXML
4.2.1 Historique
4.2.2 Mission
4.2.3 Apport d'ebXML

4.3 L'architecture technique d'ebXML
4.3.1 Principaux concepts
4.3.2 Composants de l'architecture ebXML
4.3.3 Scénario d'utilisation ebXML

4.4 Démarche de développement d'un service ebXML
4.4.1 L'infrastructure technique
4.4.2 L'infrastructure sémantique

Chapitre 5 : Méthodologie de modélisation 49 - 60

5.1 Introduction

5.2 Le niveau Business : Représentation du domaine
5.2.1 Représentation des business process par un diagramme de cas d'utilisation
5.2.2 Représentation du domaine par un diagramme de classes

5.3 Le niveau Logique : Modélisation de la transaction
5.3.1 Représentation d'une transaction par diagramme d'activités
5.3.2 Représentation d'un message par un diagramme de classes « restreint »

5.4 Le niveau Physique : Transformation de UML vers XML 5.4.1 Génération de la transaction 5.4.2 Règles de construction des messages 5.4.3 Règles de dérivation d'un Schéma XML	
PARTIE 3 : Application au domaine de la Construction	**61 - 106**
Chapitre 6 : Le domaine de la Construction	**61 - 82**
6.1 Contexte général 6.2 Les acteurs 6.2.1 Les familles d'acteurs 6.2.2 Relations entre acteurs 6.2.3 Indépendance des organisations 6.3 Les documents 6.3.1 Les types de documents 6.3.2 Les flux d'échanges entre les différents acteurs 6.3.3 Mode d'échange de documents 6.4 Moyens de coordination 6.4.1 La réunion 6.4.2 La planification 6.5 Le processus global et processus spécifiques de la construction 6.5.1 Le processus global 6.5.2 Les processus spécifiques 6.5.3 Interactions entre les partenaires 6.6 Informatisation des professionnels 6.6.1 Développement de l'informatisation 6.6.2. Limites à l'informatisation 6.6.3 Influences de l'informatisation 6.7 Le modèle IFC 6.7.1 Le modèle IFC, un standard 6.7.2 Principe du modèle IFC 6.7.3 Architecture du modèle IFC 6.7.4 La plate-forme IFC-2X 6.7.4 Evolution 6.7.5 Limites 6.7.6 Obstacles 6.8 Problèmes et difficultés 6.8.1 Nature des problèmes 6.8.2. Identification des problèmes 6.9 Avantages attendus d'un meilleur système d'échange	
Chapitre 7 : Le niveau business	**83 - 97**
7.1 Représentation du domaine par un diagramme de classes 7.1.1 La plate-forme IFC-2X 7.1.2 Sélection des entités représentatives du domaine 7.1.3 Règles de mapping EXPRESS - diagramme de classes UML 7.1.4 Diagramme de classes 7.1.5 Description des classes 7.2 Représentation des business process par un diagramme de cas d'utilisation 7.2.1 Le système retenu 7.2.2 Profil des acteurs 7.2.3 Fonctionnalités du système 7.2.5 Diagramme des cas d'utilisation 7.2.6 Description des cas d'utilisation	

Chapitre 8 : Le niveau logique 8.1 Description de la transaction « Demande de validation stabilité » 8.1.1 Diagramme d'activités 8.1.2 Description des activités 8.2 Description du contenu des messages 8.2.1 Message « Demande de validation » 8.2.2 Message « Demande de consultation » 8.2.3 Message « Objet proposition [transfert] »	**98 - 105**
Chapitre 9 : Le niveau physique 9.1 Traduction de la dynamique de la transaction en XPDL 9.2 Traduction des messages en Schémas XML	**106**
CONCLUSION	**107-109**
BIBLIOGRAPHIE	**110-112**
ANNEXES (sur CD-rom)	
Annexe 1 : ➢ Traduction de la dynamique de la transaction en XPDL	**CD**
Annexe 2 : ➢ Traduction des messages en Schémas XML	**CD**
Annexe 3 : ➢ Spécifications UML : "OMG Unified Modeling Language Specification, March 2003, version 1.5, formal/03-03-01"	**CD**
Annexe 4 : ➢ Spécifications XML : "Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation 04 February 2004"	**CD**
Annexe 5 : ➢ Spécifications IFC : "Industry Foundation Classes 2X édition 2, International Alliance for Interoperability, 2003"	**CD**
Annexe 6 : ➢ Spécifications Schéma XML (partie 0) : "XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001" ➢ Spécifications Schéma XML (partie 1) : "XML Schema Part 1: Structures, W3C Recommendation, 2 May 2001" ➢ Spécifications Schéma XML (partie 2) : "XML Schema Part 2: Datatypes, W3C Recommendation, 2 May 2001"	**CD**
Annexe 7 : ➢ Spécifications ebXML : "ebXML Technical Architecture Specification v1.0.4, UN/CEFAC - OASIS, 16 February 2001"	**CD**

GLOSSAIRE

AEC	Architecture Engineering Construction	Maître d'oeuvre du secteur de la construction.
ANSI	American National Standards Institute	Principal organisme de normalisation des états unis d'Amérique.
B2B	Business to Business	Echanges commerciaux entre entreprises.
B2C	Business to Consumer	Echanges directs, via le Web, entre entreprises et clients.
CEFACT	Centre for Trade Facilitation and Electronic Business.	Consortium
C.R.	Compte Rendu	Compte rendu de réunion de chantier.
DOE	Dossier des Ouvrages Exécutés	Dossier reprenant toutes les informations utiles à l'utilisation et la maintenance du bâtiment.
DTD	Document Type Description	Standard XML définissant la grammaire d'un document XML
DXF	Drawing eXchange File Format	Format de fichier de dessin propriétaire (AUTODESK) qui est devenu un standard de fait.
ebXML	Electronic Business XML	Ensemble des spécifications XML pour le commerce électronique.
EDI	Electronique Data Interchange	Système d'échange de données entre organisations autonomes utilisant des normes publiques et cherchant à assurer l'interopérabilité. Il permet de multiples échanges simultanés pour atteindre un but professionnel explicite et partagé.
EDIFACT	Electronic Data Interchange for Administration, Commerce and Transport	Norme mondiale de communication
EFFICIENT	E-business Framework For an efflcient Capture and Implementation of ENd-to-end Transactions	Atelier de prototypage de transactions d'e-commerce.
ERP	Entreprise Resource Planning	Progiciel de gestion d'entreprise.
EXPRESS	/	Langage formel normalisé, pour décrire la structure de bases de données orientées objets.
HTML	HyperText Markup Language	Langage de description de page Web.
HVAC	Heating, Ventilation, Air Conditioning	Etudes concernant le chauffage, la ventilation et l'air conditionné.

IAI	International Alliance for Interopérability	Association internationale chargée de mettre au point le standard d'échange IFC.
IFC	Industry Foundation Classes	Spécifications de classes d'objets fondamentaux dans le domaine de l'AEC, nécessaires à la réalisation d'une intégration.
ISO	International Organization for Standardisation	Organisation Internationale de Normalisation.
TIC	Technologies de l'Information et de la Communication.	/
OASIS	Organisation for the Advancement of Structured Information and Standards	Consortium prenant en charge l'élaboration de standards fondés sur XML, notamment ebXML.
OMG	Object Management Group	Consortium qui a pour but de spécifier une architecture à objets répartis.
PAS	Publicly Available Specification	Catégorie applicable à une norme ISO, indiquant que les spécifications de la norme ont été élaborées à l'extérieur de l'ISO.
PME	Petites et Moyennes Entreprises	/
SGML	Standard Generalized Markup Language	Ancêtre de XML, utilisé dans le secteur de la gestion documentaire.
STEP	Standard for Exchange of Product Data Model	Norme ISO de spécification et d'échange de modèles de produits. Les IFC s'appuient sur cette norme.
TDID	Trade Data Interchange Directory	Répertoires d'information de EDIFACT. Ils contiennent des définitions des données échangées et les codifications associées.
UML	Unified Modeling Language	Notation graphique et méthodologique de développement d'applications orientées objets.
UMM	Unified Modeling Methodology	Méthode de modélisation basée sur UML, visant à garantir la cohérence sémantique des concepts utilisés.
W3C	Word Wide Web Consortium	Organisme officiel chargé de "Coordonner" l'évolution des standards du Web.
XMI	XML Metadata Interchange Format	Dialecte XML de représentation des modèles objet.
XML	eXtensible Mark-up Language.	Standard de balisage de documents structurés.
XSL	eXtensible Style Sheet Language	Technologie de feuilles de style utilisées pour transformer des structures XML en d'autres structures.
XSLT	XSL Transformation	Application des feuilles de style XML à la transformation de données.

INTRODUCTION

L'utilisation de l'Internet et des réseaux conquiert chaque jour une place plus importante chez les professionnels qui profitent de son faible coût, de sa facilité d'usage et de son universalité. Cette évolution a deux conséquences : d'une part, elle multiplie les échanges de données et d'autre part, elle pose la question de la coopération.

L'essor de tels réseaux a eu un impact sur les frontières de l'entreprise. Alors qu'avant elle vivait dans un environnement fermé où les informations ne sortaient pas de ses murs, maintenant, ces mêmes informations voyagent dans un monde ouvert où elles peuvent être échangées avec des filiales délocalisées, voire d'autres entreprises, d'autres acteurs. On voit apparaître un nouveau type de commerce, appelé « e-business » qui confère une importance économique aux échanges électroniques.

Cet essor a eu pour conséquences une remise en question du fonctionnement des systèmes d'information existants et une adaptation de ceux-ci, principalement au niveau de l'échange de données sur les réseaux.

Cette interopérabilité des systèmes d'information nécessite l'adoption de normes (standards) quant au format d'échange des informations. En effet, il est impératif que les applications qui sont appelées à communiquer entre elles, puissent se comprendre mutuellement, récupérer les données qui leur sont envoyées mais également comprendre la structure et la sémantique de celles-ci.

Pour combler ce besoin réel, le langage XML est proposé dans un souci de faciliter les échanges de données. XML est souvent présenté comme la solution idéale pour résoudre tous les problèmes d'interopérabilité. Cependant XML n'offre qu'une partie de la solution en se basant sur une syntaxe standardisée pour la communication et aussi la disponibilité de nombreux outils permettant le traitement de cette syntaxe (Dom, Xslt, Xpath, Xlink, etc.)

Au delà des aspects syntaxiques, la résolution des problèmes d'interopérabilité nécessite également l'adoption d'une approche plus « sémantique » impliquant une compréhension plus approfondie du 'business' et son domaine associé. Des approches telles que ebXML et RosettaNet encouragent l'utilisation d'UML pour la représentation de ces aspects plus sémantiques. Une approche méthodologique adéquate permet de dériver systématiquement la solution XML de sa spécification UML.

Le but de ce mémoire est de formaliser une approche méthodologique pour la modélisation de transactions e-business et ensuite d'appliquer cette méthode à un cas concret lié au domaine de la construction.

Le présent mémoire se structure en 3 parties dont nous allons maintenant donner une brève description :

Partie 1 : Fondements théoriques : Cette première partie a pour but de fournir les outils nécessaires à la compréhension et à la modélisation de transactions e-business. L'approche de conception de transaction e-business qui a été adoptée dans ce travail repose sur deux standards : UML et XML.

Dans un premier temps, nous aborderons le langage UML. Ce langage définit plusieurs types de diagrammes qui couvrent un large spectre de la modélisation du logiciel. Dans le cadre de ce travail, on se limitera à décrire les diagrammes UML utiles à la conception de transactions e-business, c'est-à-dire le diagramme de cas d'utilisation, le diagramme de classes et le diagramme d'activités. Nous décrirons le rôle et la syntaxe de chacun de ces diagrammes.

Ensuite, nous aborderons le langage XML. Ce langage permet de modéliser des structures de données, ceci grâce à la possibilité d'associer aux données une sémantique appropriée

qui aide à les décrire et les identifier. Dans le cadre de ce travail, nous utiliserons XML d'une part pour définir la séquence des messages échangés et d'autre part pour décrire la structure de chacun des messages. Afin de pouvoir comprendre et utiliser le langage XML, nous décrirons les principes de ce langage ainsi que sa syntaxe.

Partie 2 : Conception et développement de transactions e-business : cette partie a pour but de définir la notion de transaction e-business, d'établir un état de l'art concernant la mise en œuvre des solutions d'échange de données et de définir une méthode de conception de transactions e-business.

Tout d'abord, nous ferons le point sur la situation actuelle et sur les perspectives futures, nous décrirons les différents modèles d'échange électronique, nous établirons les implications technologiques que la mise en place d'une solution de commerce électronique impose et nous analyserons l'approche EDI afin d'en dégager les avantages et les limites.

Ensuite, nous aborderons l'initiative ebXML. D'une part, nous montrerons que même si le langage XML est une technologie majeure de commerce électronique, il n'est pas suffisant pour définir des transactions e-business. En plus du langage XML, il est nécessaire de pouvoir s'entendre sur le format des messages que les acteurs vont s'échanger ainsi que sur la séquence des échanges. C'est en réponse à cette problématique que nous aborderons l'initiative ebXML ; nous décrirons les missions, les avantages et l'architecture de ebXML.

Enfin, nous définirons les différentes étapes de la méthodologie de modélisation de transactions e-business que nous avons adoptée. Cette méthodologie vise à favoriser la compréhension de la sémantique de la transaction et la compréhension de la sémantique des messages avant de passer à leur implémentation. L'approche envisagée est composée de trois étapes : le niveau business, le niveau logique et le niveau physique. Nous décrirons chacune de ces étapes.

Partie 3 : Application au domaine de la construction : cette partie a pour but d'appliquer la méthodologie de modélisation de transaction au domaine de la construction.

Tout d'abord, nous allons consacrer un chapitre à la description du contexte spécifique de notre étude, le secteur de la construction. Les caractéristiques principales du secteur y seront développées.

Ensuite nous appliquerons la méthodologie de conception de transaction e-business développée dans la partie précédente, au secteur de la construction. Nous nous concentrerons sur la problématique des échanges de données et de la coopération entre les différents acteurs intervenant dans l'acte de conception et de réalisation d'un bâtiment.

Dans le cadre d'un projet de construction, les différents systèmes informatiques en présence, hétérogènes par nature, doivent pouvoir être capables de s'échanger leurs données et leurs résultats. Pour satisfaire ce besoin d'échange, la première étape consiste à définir un langage commun. Il est ensuite indispensable d'aller plus loin car tout projet de construction nécessite des échanges de données multiples et complexes entre partenaires. La conception des bâtiments, par exemple, suit un processus d'échanges interactifs et souvent itératifs entre architectes et ingénieurs. Ce mode de travail, qualifié d'ingénierie collaborative, nécessite la mise en commun des données du projet et illustre le besoin de partage de données autour d'un modèle unique. Le concept de maquette numérique répond à cette problématique. C'est sur ce concept de maquette numérique, que nous allons envisager les échanges dans le cadre de la conception et la réalisation d'un ouvrage.

Ce mémoire a donc pour motivation, d'une part de définir une méthode de modélisation de transaction e-business et d'autre part d'appliquer cette méthode au domaine de la construction et de montrer les apports de cette approche structurante sur le secteur.

PARTIE 1

Fondements théoriques

CHAPITRE 1 : UML

1.1 Introduction

Ce chapitre a pour objectif de fournir au lecteur des informations suffisantes pour qu'il soit en mesure de comprendre les principes et la syntaxe des diagrammes du langage UML qui seront utiles pour la conception et le développement de transactions e-business [Muller 2000] [Carlson 2001].

1.1.1 UML : un standard

La modélisation UML a été définie et spécifiée par un consortium de partenaires formé en 1996. C'est en 1997 que l'OMG (Object Management Group) a retenu UML (Unified Modeling Language) comme standard de modélisation, choix largement entériné depuis par l'ensemble des industriels du génie logiciel.

1.1.2 Principes UML

Comme son nom l'indique, UML n'est pas une méthode mais un langage de modélisation qui unifie le meilleur des différentes approches méthodologiques, orientées objet et traditionnelles. UML définit un langage abstrait pour la description de la structure et du comportement des systèmes informatiques.

UML introduit neuf diagrammes pour représenter les différents points de vue de la modélisation. Ils permettent de visualiser et de manipuler les éléments de la modélisation.

Le présent travail fait appel à trois types de diagrammes pour décrire l'analyse et la conception de transactions e-business. Dans la suite de ce chapitre, nous allons décrire chacun de ces trois diagrammes. Dans la partie applicative de ce travail, ces diagrammes seront appliqués à la modélisation de transactions e-Business.

Les diagrammes utilisés définis par UML sont les suivants :

- Les diagrammes de cas d'utilisation : représentation des services du système du point de vue de l'utilisateur; (vue statique)
- Les diagrammes de classes : représentation de la structure statique des informations en terme de classes et de relations ; (vue statique)
- Les diagrammes d'activité : représentation du comportement d'une transaction en termes d'action ; (vue dynamique)

La notation UML présente l'avantage d'être un langage standardisé et donc compris de tous. De plus, son emploi s'avère très utile lorsqu'il est appliqué aux activités commerciales et industrielles, impliquant les échanges qui en découlent. En effet, comme nous le verrons dans la suite de ce travail, UML permet de modéliser le domaine d'affaires, les transactions et le contenu des messages.

1.2 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation saisit un modèle des interdépendances entre plusieurs cas d'utilisation et des interactions avec un ou plusieurs acteurs [Schneider 1998].
Ce diagramme a pour but de :

- Représenter les services que doit remplir le système, exprimés du point de vue de l'utilisateur.
- Contrôler l'élaboration des autres modèles sur cette base commune.
- Favoriser le caractère évolutif du système.

Dans le diagramme, deux types d'éléments interviennent : les acteurs et les cas d'utilisation.

1.2.1 Les acteurs

Un acteur représente un rôle générique d'utilisateur qui ne renvoie pas nécessairement à une personne ou à une fonction identifiable. L'acteur est souvent un être humain, mais il peut également s'agir d'un autre système ou d'un agent logiciel automatisé situé dans l'environnement du système étudié.

Représentation d'un acteur :

Figure 1.1 : acteur

Les acteurs se déterminent en observant les utilisateurs directs du système, ceux qui sont responsables de son exploitation ou de sa maintenance, ainsi que les autres systèmes qui interagissent avec le système en question. La même personne physique peut jouer le rôle de plusieurs acteurs. D'autre part, plusieurs personnes peuvent jouer le même rôle, et donc agir comme un seul et même acteur. Le nom de l'acteur décrit son rôle lorsqu'il interagit avec le système.

Les acteurs sont les personnes et les choses extérieures à un système qui interagissent avec lui en échangeant de l'information (en entrée et en sortie). La détermination des acteurs permet de préciser les limites du système de manière progressive : floues au départ, elles se précisent au fur et à mesure de l'élaboration des différents cas d'utilisation. Cette activité de délimitation est extrêmement importante car elle définit ce qui fait partie du système à développer et ce qui n'en fait pas partie.

Une fois identifiés, les acteurs doivent être décrits d'une manière claire et concise, avec notamment le détail de leurs responsabilités respectives, en quelques lignes.

Un acteur peut participer à des relations de généralisation avec d'autres acteurs. Les acteurs « fils » seront alors capables de communiquer avec les même cas d'utilisation que le ou les acteurs « parents ».

Les relations de généralisation se présentent sous la forme suivante :

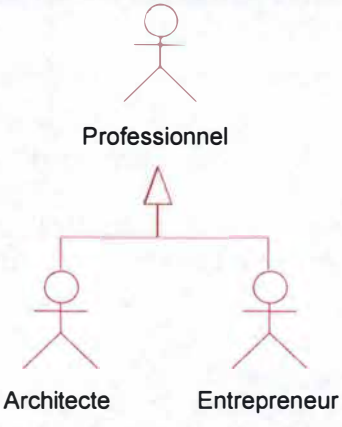


Figure 1.2 : relation de généralisation entre acteurs

1.2.2 Les cas d'utilisation

Un cas d'utilisation est un scénario d'exécution d'un service du système. L'instanciation d'un cas d'utilisation se traduit par l'échange de messages entre le système et ses acteurs.

Les cas d'utilisation se représentent sous la forme suivante :

Figure 1.3 : cas d'utilisation

Les cas d'utilisation peuvent être contenus dans un rectangle qui représente les limites du système. Les acteurs sont alors forcément à l'extérieur du rectangle puisqu'ils ne font pas partie du système. La relation entre un acteur et un cas d'utilisation est une association représentée par une flèche.

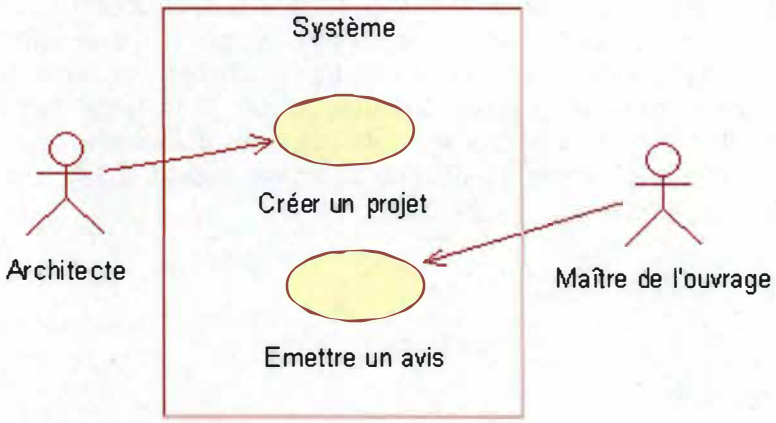


Figure 1.4 : diagramme de cas d'utilisation

Le cas d'utilisation se détermine en observant et en précisant, acteur par acteur, les séquences d'interaction –les scénarios- du point de vue de l'utilisateur. Ils se décrivent en termes d'informations échangées et d'étapes dans la manière d'utiliser le système. Un cas d'utilisation regroupe une famille de scénarios d'interaction possibles entre l'environnement et le système. Les cas d'utilisation sont des abstractions du dialogue entre les acteurs et le système : ils décrivent des interactions potentielles, sans entrer dans les détails de chaque scénario. Lors de la définition des cas d'utilisation, il est important de veiller à ce que ces différents cas d'utilisation aient la même granularité.

Chaque fois qu'un acteur interagit avec le système, le cas d'utilisation instancie un scénario ; ce scénario correspond au flot de messages échangés par le système et les acteurs durant l'interaction particulière qui correspond au scénario. L'analyse des besoins au moyen des cas d'utilisation s'accommode très bien d'une approche itérative et incrémentale.
La portée des cas d'utilisation dépasse largement la définition des seuls besoins du système.

Il est possible de préciser le comportement d'un cas d'utilisation par du texte, des diagrammes d'activités, des machines à états, etc.

1.2.3 Les relations entre cas d'utilisation

Les dépendances entre cas d'utilisation peuvent être de deux types :

La relation d'inclusion : « include »

Dans une relation d'inclusion entre cas d'utilisation, une instance du cas d'utilisation source comprend également le comportement décrit par le cas d'utilisation destination. L'inclusion a un caractère obligatoire, la source spécifiant à quel endroit le cas d'utilisation cible doit être

inclus. Cette relation permet aussi de décomposer des comportements et de définir des comportements partageables entre plusieurs cas d'utilisation. Elle est représentée par une flèche en traits pointillés pointant vers le cas d'utilisation destination et par le stéréotype « include » placé à proximité de cette flèche. La notion de stéréotype sera décrite dans la rubrique 1.5.

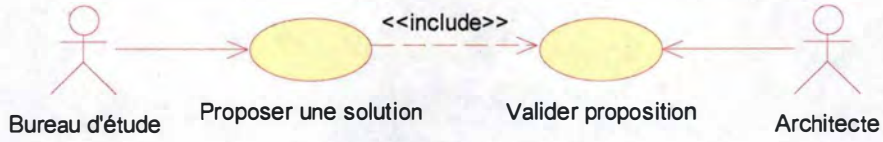


Figure 1.6 : relation d'inclusion entre cas d'utilisation

La relation d'extension : « extend »

Dans une relation d'extension entre cas d'utilisation, le cas d'utilisation source ajoute son comportement au cas d'utilisation destination. L'extension peut être soumise à une condition. Le comportement ajouté est inséré au niveau d'un point d'extension défini dans le cas d'utilisation destination. Cette relation permet de modéliser des variantes de comportement d'un cas d'utilisation (selon les interactions des acteurs et l'environnement du système). Elle est représentée à l'aide d'une flèche en traits pointillés qui pointe vers la classe destination et du mot clé « extend ». Si une condition d'extension existe, il est possible de la préciser à proximité du mot clé « extend ».

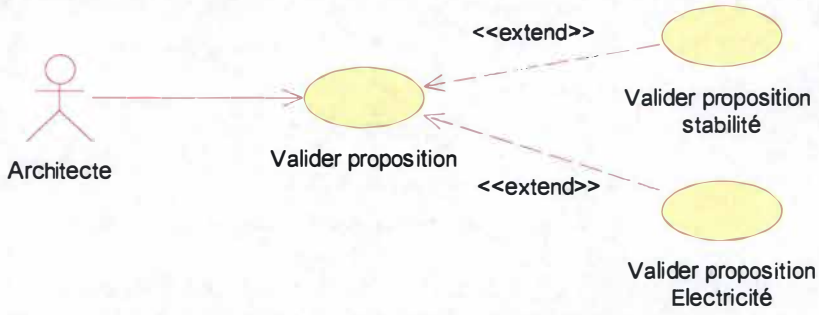


Figure 1.7 : relation d'extension entre cas d'utilisation

1.2.4 Description des cas d'utilisation

La description des cas d'utilisation devra être plus ou moins détaillée en fonction des besoins mais toujours abstraite des détails de l'implémentation. Chaque cas d'utilisation doit faire l'objet d'une description structurée regroupant les informations suivantes :

- Le nom du cas d'utilisation
- Une brève description.
- Les pré-conditions (conditions devant être réalisées en préalable).
- Les post-conditions (conditions remplies à la fin si flux d'événement normal)

Exemple de description de cas d'utilisation

UC1 : Demander information.	
Objectif :	Un utilisateur introduit une demande d'information ou une demande de précision concernant un quelconque point du projet. Cette demande n'est associée à aucun objet en particulier.
Description :	L'initiateur introduit une demande d'information et l'adresse à un destinataire, le destinataire réceptionne la demande d'information et répond à l'initiateur. L'initiateur reçoit et consulte la réponse.
Pré-condition :	Le système est alimenté et fonctionne correctement. Les utilisateurs font partie de l'équipe projet et ont accès au système.
Post-condition :	L'initiateur a reçu la réponse correspondant à la demande d'information qu'il a envoyée.

1.3 Diagrammes de classes

Les diagrammes de classes expriment de manière générale la structure statique du domaine, en termes de classes et de relations entre ces classes.

De même qu'une classe décrit un ensemble d'objets, une association décrit un ensemble de relations ; les objets sont des instances des classes et les relations sont des instances des associations. Un diagramme de classes n'exprime rien de particulier sur les relations d'un objet donné, mais décrit de manière abstraite les relations potentielles d'un objet vers d'autres objets.

1.3.1 Les classes

Une classe est une description abstraite (condensée) d'un ensemble d'objets du domaine de l'application ; elle définit leur structure et leurs relations. Les classes sont représentées par des rectangles compartimentés. Le premier compartiment contient le nom de la classe. En général, le nom d'une classe utilise le vocabulaire du domaine. Le nom de la classe doit exprimer ce que la classe est, et non ce qu'elle fait. Un autre compartiment est généralement ajouté ; il contient les attributs (propriétés) de la classe. Les compartiments d'une classe peuvent être cachés (en totalité ou en partie) lorsque leur contenu n'est pas pertinent dans le contexte d'un diagramme (voir figure 1.9).

Une classe peut servir pour modéliser deux notions différentes. En effet, dans un diagramme de classe, il faut distinguer :

- La modélisation de classe « objets » qui est associée au monde réel et dont les instances sont des objets du domaine.
- La modélisation de classe « type », qui décrit statiquement les éléments non fonctionnels associés aux attributs d'une classe.

Le standard UML définit un certain nombre de types primitifs. Les types suivants sont de types primitifs du métamodèle UML :

- Entier : élément compris dans l'ensemble infini des entiers {..., -2, -1, 0, 1, 2, ...}.
- Booléen : type énuméré qui comprend les deux valeurs vrai et faux.
- Chaîne : suite de caractères.
- Nom : chaîne de caractères qui permet de désigner un élément.
- Temps : valeur qui représente un temps absolu ou relatif.
- ...

Il est également possible de définir ses propres types primitifs et des types énumérés particuliers. Les stéréotypes standard « type primitif » et « énumération » ainsi que des stéréotypes personnels sont appliqués aux classes pour spécifier des types particuliers. Les littéraux d'énumération sont spécifiés dans le compartiment des attributs. La notion de stéréotype sera développée dans la rubrique 1.5.

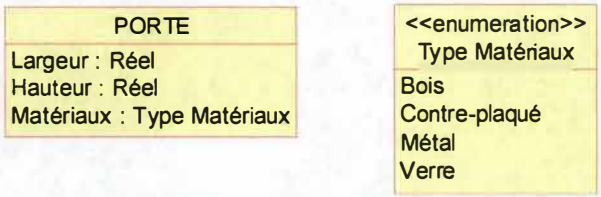


Figure 1.9 : Exemple de type énuméré défini par l'utilisateur.

1.3.2 Les attributs

Les attributs peuvent figurer de manière exhaustive ou non, dans les compartiments des classes. En général, il vaut mieux limiter la visualisation des attributs en se focalisant sur les éléments pertinents dans un diagramme donné. Le type des attributs peut être un type primitif (entier, réel, chaîne de caractères, booléen, date, ...) ou une classe (type construit). Exemple : Voir figure 1.9

1.3.3 Les associations

Les associations représentent des relations structurelles entre classes d'objets. Une association compte au moins deux extrémités d'association, reliées à des classificateurs.

La plupart des associations sont binaires, c'est-à-dire qu'elles connectent deux classes. Les associations se représentent en traçant une ligne entre les classes associées.
Les instances d'une association sont des tuples des instances des classificateurs reliés par cette association. Chaque valeur de tuple doit être unique ; cela signifie que deux instances ne sont jamais reliées par plusieurs liens, instances de la même association. UML impose ainsi une sémantique d'ensemble aux relations.

Arité des associations
La plupart des associations sont dites binaires car elles relient deux classes. Des arités supérieures peuvent cependant exister et se représenter alors au moyen d'un losange sur lequel arrivent les différentes composantes de l'association.

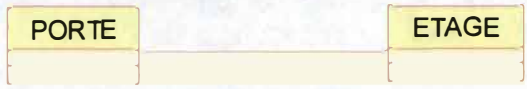


Figure 1.11 : association binaire

Nommage des associations
Les associations peuvent être nommées ; le nom de l'association figure alors au milieu de la ligne qui symbolise l'association.

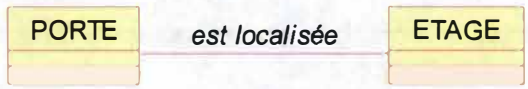


Figure 1.12 : nommage d'une association

Rôles des extrémités d'association
L'extrémité d'une association possède un rôle, qui décrit comment une classe source voit une classe destination au travers de l'association. Chaque association binaire possède deux rôles, un à chaque extrémité.

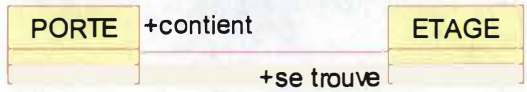


Figure 1.13 : rôles d'une association

Multiplicité des associations
Chaque extrémité d'une association peut porter une indication de multiplicité qui montre combien d'objets de la classe considérée peuvent être liés à un objet de l'autre classe. La multiplicité est une information portée par l'extrémité d'association, sous la forme d'une expression entière.

Valeurs de multiplicité conventionnelles :

1	Un et un seul
0..1	Zéro ou un
N	N (entier naturel)
M..N	De M à N (entiers naturels)
*	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	D'un à plusieurs

Une valeur de multiplicité supérieure à 1 implique une collection d'objets. Cette collection n'est pas bornée dans le cas du * ; en d'autres termes, plusieurs objets participent à la relation, le nombre maximum d'objets n'étant pas fixé. Le terme collection, plus général que liste ou ensemble, est employé afin d'éviter toute supposition sur la structure de données qui contient les objets.

Les valeurs de multiplicité expriment des contraintes liées au domaine de l'application, valables durant toute la vie des objets. Il est important de déterminer les valeurs de multiplicité optimales pour trouver le bon équilibre, d'une part, entre souplesse et possibilité d'extension et, d'autre part, entre complexité et efficacité.

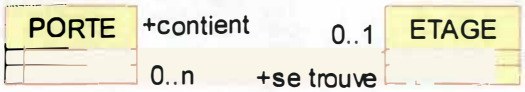


Figure 1.14 : multiplicité d'une association

Contraintes sur les associations :
Certaines contraintes peuvent être définies sur une relation ou sur un groupe de relations. Par exemple, la contrainte ou-exclusif {XOR} permet d'éviter l'introduction de sous-classes artificielles comme c'est le cas dans les constructions à base de généralisation.

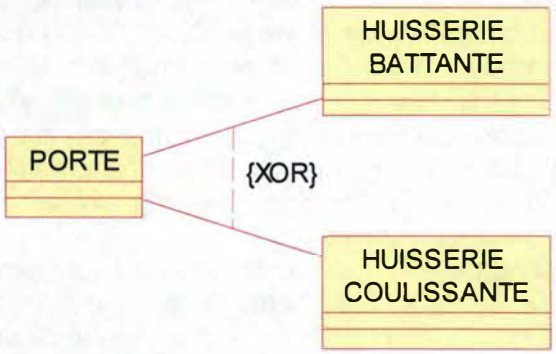


Figure 1.15 : contrainte ou-exclusif

1.3.4 Les classes-associations

Il est possible de représenter une association par une classe pour ajouter, par exemple, des attributs et des opérations dans l'association. Une classe-association possède à la fois les caractéristiques d'une classe et d'une association et peut, à ce titre, participer à d'autres relations dans le modèle. La notation utilise une ligne pointillée pour attacher une classe à une association.

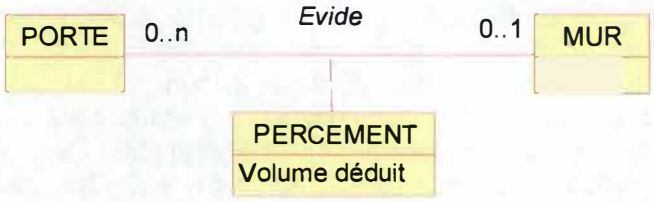


Figure 1.16 : classe association

1.3.5 Les agrégations

Une agrégation représente une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. Quelle que soit l'arité, l'agrégation ne peut concerner qu'un seul rôle d'une association.

L'agrégation se représente en ajoutant un petit losange du côté de l'agrégat.
L'agrégation permet de modéliser une contrainte d'intégrité et de désigner l'agrégat comme garant de cette contrainte.

A travers une telle contrainte d'intégrité, il est possible de représenter par exemple :

- La propagation des valeurs d'attributs d'une classe vers une autre classe.
- Une action sur une classe qui implique une action sur une autre classe.
- Une subordination des objets d'une classe à ceux d'une autre.

Les agrégations peuvent être multiples, comme les associations.
Cela signifie, en particulier, que la multiplicité du côté de l'agrégat peut être supérieure à 1.
La notion d'agrégation ne suppose aucune forme de réalisation particulière.

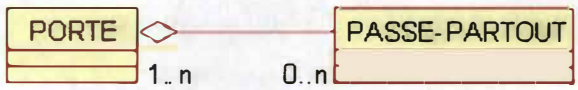


Figure 1.17 : agrégation faible

1.3.6 Les compositions

La composition est un cas particulier d'agrégation avec un couplage plus important. La classe ayant le rôle prédominant dans une composition est appelée classe composite ou classe conteneur.
La composition implique, en plus des propriétés d'agrégation, une coïncidence des durées de vie des composants et du composite : la destruction du composite implique automatiquement la destruction de tous ses composants. La création, la modification et la destruction des divers composants sont de la responsabilité du composite. La composition implique également une contrainte sur la valeur de la multiplicité du côté de l'agrégat : elle peut prendre les valeurs 0 ou 1 uniquement (un composant ne peut être partageable). La limite inférieure de multiplicité 0 du côté du composant correspond à un attribut non renseigné. La composition se représente dans les diagrammes par un losange de couleur noire, placé du côté de la classe composite.
La composition et les attributs sont sémantiquement équivalents ; leurs représentations graphiques sont interchangeables. La notation par composition s'emploie dans un diagramme de classe lorsqu'un attribut participe à d'autres relations dans le modèle.
La différence entre une composition et un attribut nécessite la spécification implicite de son type, et non un sous-type, alors que la relation de composition accepte le polymorphisme.

Figure 1.18 : agrégation forte

1.3.7 La généralisation

UML emploie le terme « généralisation » pour désigner la relation de classification entre un élément plus général et un élément plus spécifique. En fait, le terme « généralisation » désigne un point de vue porté sur un arbre de classification.
L'élément plus spécifique peut contenir des informations qui lui sont propres, à condition que ces informations et la description de l'élément plus général soient totalement cohérentes.
La relation de généralisation se représente au moyen d'une flèche orientée de la classe plus spécialisée vers la classe plus générale. La tête de la flèche possède un petit triangle vide.
Dans le cas de sous-classes multiples, les flèches peuvent être agrégées.
Les sous-classes héritent des attributs, des services, des relations et des contraintes définies dans les superclasses.

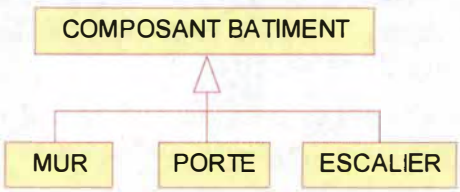


Figure 1.19 : relation de généralisation

1.3.8 Les contraintes et les propriétés de généralisation

Différentes contraintes peuvent être appliquées aux relations de généralisation, pour distinguer, par exemple, les formes de généralisation exclusives des formes inclusives.
Les contraintes sur les relations de généralisation se représentent au moyen d'expressions entre accolades, directement attachées aux généralisations agrégées ou associées à une ligne en pointillés qui relie les relations de généralisation concernées.
Par défaut, la généralisation symbolise une décomposition exclusive ; en d'autres termes, un objet est, au plus, instance d'une des sous-classes.

La contrainte {disjoint} indique qu'une classe descendante d'une classe A peut être descendante d'une seule sous-classe de A.

La contrainte {non-disjoint} indique qu'une classe descendante d'une classe A appartient au produit cartésien des sous-classes de la classe A. Un objet concret se construit alors à partir d'une classe obtenue par mélange de plusieurs superclasses.

La contrainte {complet} indique que la généralisation est terminée et qu'il est impossible d'ajouter des sous-classes. Inversement, la contrainte {incomplet} désigne une généralisation extensible.

{feuille} et {racine} sont des propriétés booléennes associées aux éléments généralisables. Il est possible d'utiliser ces propriétés dans un modèle pour préciser respectivement qu'une classe ne possède pas de sous classes (feuilles d'un arbre de généralisation) et qu'une classe ne possède pas de parent (racine d'un arbre de généralisation).

1.4 Diagrammes d'activités

Un diagramme d'activités modélise le comportement interne d'un cas d'utilisation ou plus généralement d'un processus impliquant l'utilisation d'un ou plusieurs classificateurs.

Un diagramme d'activités représente l'état de l'exécution d'un mécanisme, sous la forme d'un déroulement d'étapes regroupées séquentiellement dans des branches parallèles de flot de contrôle. Le début et la fin d'un mécanisme sont définis respectivement par un état initial et un état final.

1.4.1 Les états-action

Un état-action modélise une étape dans l'exécution d'un algorithme ou d'un workflow. Il consiste en un état simplifié dans lequel figure une action d'entrée avec au moins une transition automatique vers un autre état, déclenchée par la fin de l'action d'entrée et en une action non-interruptible.
Représentation d'un état-action :

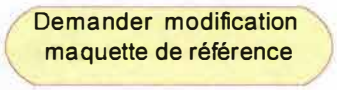


Figure 1.20 : état-action

NB : le nom de l'action n'est pas nécessairement unique dans un diagramme d'activités.
Si des transitions sont liées à des événements explicites ou qu'il faut définir des transitions internes, l'utilisation de simples états s'impose.

Tout diagramme d'activités comporte un et un seul état initial. En revanche, il est possible d'avoir, pour un niveau hiérarchique donné, plusieurs états finaux qui correspondent chacun à une condition de fin différente.

L'état initial est un pseudo-état qui désigne le premier état actif d'un niveau hiérarchique via une unique transition.

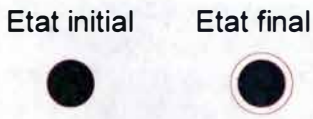


Figure 1.21 : état initial et final

1.4.2 Les transitions

Chaque état-action représente une étape particulière du traitement englobant.
En général, les états-action sont reliés par des transitions automatiques, aussi dénommées transition sans déclencheur ou transition de terminaison.
Elles sont représentées par des flèches. De telles transitions ne possèdent pas de signature d'événement. Il est inutile de faire figurer un nom d'événement sur la transition.
Lorsque l'action de l'état se termine, la transition est automatiquement déclenchée et l'action du prochain état démarre.

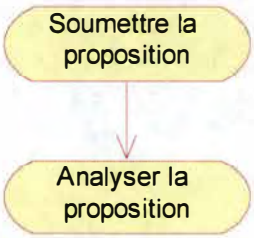


Figure 1.22 : transition

Les transitions peuvent également être déclenchées par l'occurrence d'un signal ou la réalisation d'une certaine condition (un objet se trouvant dans un certain état, par exemple).
Les transitions entre états-action peuvent également dépendre de conditions booléennes, mutuellement exclusives, appelées décisions. Une décision peut être représentée par un losange (= ou exclusif) où arrive une unique transition et d'où partent plusieurs transitions. Le ou exclusif peut également être utilisé pour modéliser une alternative.

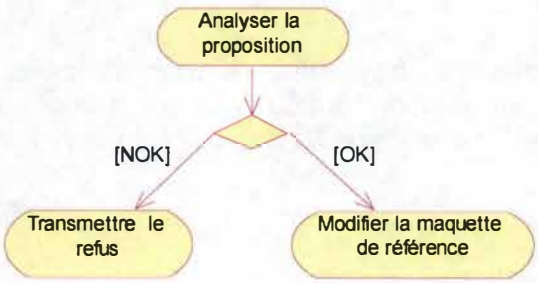
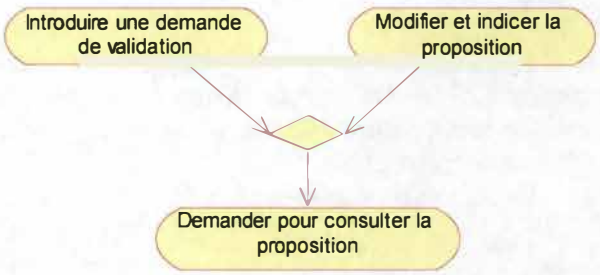


Figure 1.23 : représentation d'une décision



représentation d'une alternative

1.4.3 Les barres de synchronisation

Les diagrammes d'activités représentent les synchronisations entre flots de contrôle, au moyen de transitions concurrentes. Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution d'un cas d'utilisation. Les transitions de débranchement au départ d'une barre de synchronisation sont déclenchées simultanément.
Inversement, une barre de synchronisation ne peut être franchie que lorsque toutes les transitions en entrée sur la barre sont déclenchées.

Figure 1.24 : synchronisation de flots de contrôle parallèles à partir de barres de synchronisation

1.4.4 Les travées (swimlanes)

Les diagrammes d'activités peuvent être découpés en travées pour montrer les différentes responsabilités au sein d'un mécanisme ou d'une organisation. Chaque responsabilité est assurée par un ou plusieurs acteurs et chaque action est allouée à une travée donnée.
Une travée partitionne ainsi les états d'un diagramme d'activités. La position relative des travées n'est pas significative ; les transitions peuvent traverser librement les travées. Chaque travée est délimitée par des lignes verticales.

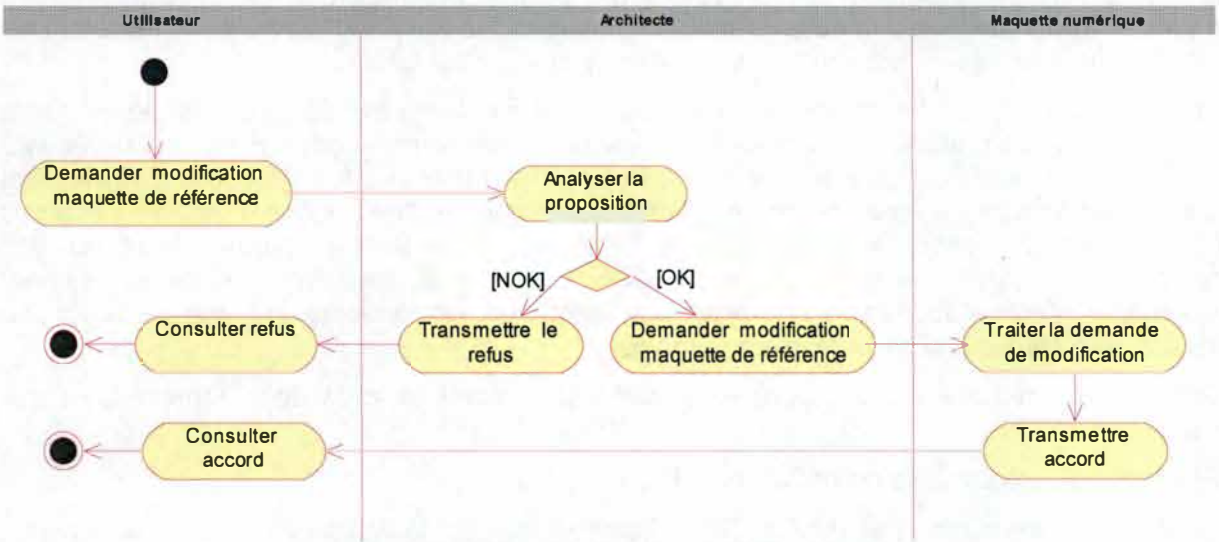


Figure 1.25 : partition d'un diagramme d'activités en travées

1.4.5 Les flots d'objets entre actions

Il est possible de faire apparaître clairement les objets dans un diagramme d'activités, éventuellement au sein de travées. Les objets représentés sont ceux qui initient des actions, qui sont utilisés par des actions ou qui sont modifiés par les actions. Un objet utilisé par une action est visualisé en connectant l'objet à l'état-action associé à cette action, via une flèche en traits pointillés (cette flèche pointe vers l'état-action).

Un objet modifié par une action est représenté à l'aide d'une flèche en traits pointillés orientée de l'état-action concerné vers l'objet. Un même objet peut être modifié par une action puis utilisé par une ou plusieurs actions. Souvent, différentes actions manipulent un même objet qui change alors d'état au cours de l'exécution du mécanisme. Pour augmenter la lisibilité, cet objet peut figurer à plusieurs reprises dans les diagrammes ; son état est alors précisé à chaque occurrence de l'objet dans une expression entre crochets.

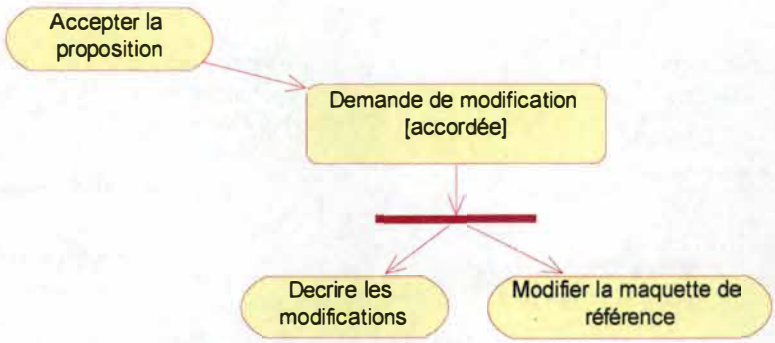


Figure 1.26: représentation des objets manipulés

1.5 Les Stéréotypes.

Les stéréotypes constituent le principal mécanisme d'extension fourni par UML. Les stéréotypes sont utilisés pour préciser la sémantique d'un élément, en général un type d'objet ou de relation. Un certain nombre de stéréotypes sont définis par UML.

Un stéréotype introduit une nouvelle classe dans le méta-modèle par dérivation d'une classe existante (le nom d'un stéréotype nouvellement défini ne doit pas être déjà attribué à un autre élément du métamodèle). Un stéréotype permet la métaclassification d'un élément d'UML. Les utilisateurs peuvent ainsi ajouter de nouvelles classes d'éléments de modélisation au métamodèle, en plus du noyau prédéfini par UML.

Un stéréotype sert à définir une utilisation particulière d'éléments de modélisation existants ou à modifier la signification d'un élément. L'élément stéréotypé et son parent non stéréotypé ont une structure identique mais une sémantique différente. Ainsi, les instances d'un élément E stéréotypé ont les mêmes propriétés (attributs, opérations, associations) que les instances de l'élément E ; elles peuvent toutefois avoir des propriétés supplémentaires via des contraintes ou des valeurs marquées, associées à leur stéréotype. Chaque élément stéréotypé peut s'appuyer sur d'autres stéréotypes, essentiellement par le biais du mécanisme de généralisation entre stéréotypes.

Le nom du stéréotype est placé entre guillemets avant le nom de l'élément auquel il s'applique.

Exemples de stéréotypes prédéfinis par UML :

- « énumération » : ensemble d'identificateurs formant le domaine de valeur d'un type.
- « document » : composant qui représente un document.

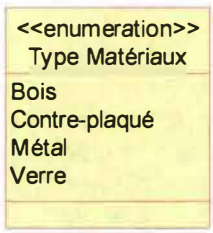


Figure 1.27: stéréotype « enumeration »

Exemple de stéréotypes définis spécifiquement dans le cadre du projet Efficient[1] :

- « BusinessDocument » : composant qui représente un message Business.
- « HumanChoice » : action nécessitant qu'un acteur humain effectue un choix. Cette action est toujours suivie d'un point de décision.

[1] Atelier de prototypage de transactions d'e-commerce, Projet de recherche développé par le centre de recherche Publique Henri Tudor, Luxembourg [Gregoire 2004]

CHAPITRE 2 : XML

2.1 Introduction

Ce chapitre a pour objectif de fournir au lecteur des informations suffisantes pour qu'il soit en mesure de comprendre le choix de cette technologie et de déchiffrer les messages XML produits par l'atelier logiciel Efficient. Pour ce faire, nous allons décrire les principes et la syntaxe du métalangage XML [Boukhors 2002] [Hancke 2002].

2.1.1 XML : un standard

La technologie XML a été définie et spécifiée par le XML Working Group formé en 1996 par le W3C. C'est en 1998 que le projet XML est devenu le standard que l'on connaît aujourd'hui.

2.1.2 Principes XML

L'objectif principal du XML Working Group au moment de sa formation etait de créer une technologie universelle pour structurer l'information ; cette technologie se devait d'être particulièrement adaptée à la diffusion et à l'échange d'informations, notamment à travers Internet. Il ne faut pas percevoir XML comme une application, mais bien comme une forme d'expression standardisée caractérisée par des règles de grammaire.

Le standard XML est un standard ouvert, gratuit, libre de droits, adapté au Web et indépendant des plates-formes, des langages et des systèmes d'exploitation. Toutefois, comme nous le verrons par la suite, XML peut être utilisé pour répondre à des besoins d'échanges de données.

XML est issu du langage SGML et utilise tout comme lui un système de balisage. Du point de vue de l'utilisation, XML est beaucoup moins complexe, tout en offrant un grand nombre de fonctionnalités. Le langage XML est un métalangage, c'est-à-dire que c'est un langage extensible à souhait, permettant de créer de nouveaux langages.

Le langage XML, avec sa volonté de fournir une information structurée et s'autodécrivant, favorise l'organisation des éléments et leur sémantique. Néanmoins, il conserve le même esprit que HTML : utilisation des balises, simplicité et adaptabilité au Web. Il faut noter que dans un document HTML, les balises ont pour unique but de mettre en forme les données, alors qu'avec XML, les balises nous renseignent sur la hiérarchie et la sémantique des données, sans aucune notation de mise en forme. XML ne s'occupe pas de la mise en forme mais de la description des données contenues dans les fichiers, il délègue la mise en page aux feuilles de styles XSL.

La capacité du langage XML à décrire les données et leurs relations entre elles lui promet de devenir un vecteur d'information essentiel sur le Web, d'autant plus que les documents XML sont des documents enregistrés au format texte, format exploitable pour tous les systèmes d'exploitation. Ceci rend possible la communication entre des applications fonctionnant sur des plates-formes potentiellement différentes et écrites dans des langages hétérogènes. L'utilisation du langage XML dans les projets informatiques fournit une garantie d'interopérabilité des applications.

Un des atouts de XML est que, dans la mesure où le concepteur de la grammaire d'un document XML définit des balises possédant des noms explicites, le document sera lisible par n'importe quel être humain et exploitable par la machine. Toutes les informations contenues dans un document XML sont encadrées par les balises XML.

Les informations manipulées sont décrites sous la forme d'une structure arborescente. Cette structure est autodescriptive car la sémantique y est incluse. Comme dans la conception d'un quelconque arbre, on doit d'abord définir une racine, dont vont dépendre toutes les informations, qui sont, soit composites, c'est-à-dire des noeuds, soit une information brute, c'est-à-dire des feuilles.

Un document XML est composé de deux éléments : il contient d'une part, la structure des données et d'autre part, les données elles-mêmes. En résumé, on peut aisément indiquer que la balise qui fait partie de la structure de données, représente le contenant, et que c'est à l'intérieur de ces balises que figurent les données, c'est-à-dire le contenu. Les données représentent l'information brute.

2.2 Les documents XML

La création de documents XML requiert une certaine rigueur. Pour pouvoir être diffusé et compris de tous, un document XML se doit d'être bien formé et valide. Ceci implique d'être très attentif au respect de la syntaxe du langage [Boukhors 2002].

2.2.1 Un document bien formé

Un document XML est bien formé lorsqu'il est syntaxiquement correct. Contrairement aux interpréteurs HTML, les interpréteurs XML se doivent d'être très stricts sur la syntaxe du document fourni, conformément à la norme XML.

Tout d'abord, il faut prendre en considération le fait que le langage XML effectue la distinction entre les lettres majuscules et les lettres minuscules. Cette distinction permet de créer un vocabulaire encore plus large, mais implique que le créateur d'un document XML s'oblige à respecter quelques règles afin de ne pas perdre un temps précieux à rechercher les raisons d'un document mal formé ; des fautes de frappe ou des fautes d'orthographe peuvent être difficiles à déceler avec un éditeur de texte classique.

Pour que le document soit bien formé, il est obligatoire de fermer toutes les balises ; la moindre balise mal fermée provoquera la non interprétation du document par l'interpréteur XML. Si un document XML contient une instruction incorrecte, un message d'erreur sera affiché. C'est là encore une différence avec les interpréteurs HTML, qui en général tolèrent l'oubli de fermeture d'une balise.

Enfin, il n'est pas autorisé d'utiliser des balises croisées, c'est-à-dire que le fils d'une balise doit impérativement se refermer avant la fermeture de son père.

2.2.2 Un document valide

La première étape pour vérifier qu'un document XML peut être validé par un interpréteur XML, consiste à s'assurer qu'il est bien formé. Toutefois, cette première vérification, qui est indispensable, n'est pas suffisante pour faire d'un document XML, un document valide.

Pour être valide, un document XML doit aussi répondre aux règles définies dans la grammaire associée au document XML, car tout document valide est un document dont la structure de données répond à la structure définie par ce document. Si le document répond aux règles définies, on dit que le contenu du document est valide. Le processus qui permet de s'assurer du respect de la structure définie, est nommé la validation. Ces grammaires seront définies sous la forme d'une DTD, ou sous la forme d'un schéma. La description de ces deux types de grammaire est abordée dans ce chapitre respectivement aux points 2.4 et 2.5.

2.3 La syntaxe

Dans un document XML, on rencontre un certain nombre de types de balises : nous allons en définir la syntaxe et l'utilisation [Boukhors 2002] [Ibrahim 2000].

2.3.1 Les éléments

Les balises de début et de fin d'un élément constituent les blocs élémentaires d'un document XML. Ces balises doivent porter un nom explicite.

La particularité des balises SGML et, par héritage, celles des balises HTML et XML est d'encadrer le nom des balises par les caractères « < » et « > ». La balise de fin allant de paire avec la balise de début devra s'écrire de la façon suivant « </nom_balise> ».

Figure 2.1 : élément

```
<matériau>bloc de terre cuite</matériau>
```

Il est possible de créer des balises ne comprenant pas de balise fermante. Ces balises ne contiennent pas de données et ont une syntaxe particulière : elles n'ont pas de balise de fin car la balise de fin est en quelque sorte comprise dans la balise de début. De telles balises peuvent être utilisées lorsqu'un élément est obligatoire dans un document XML, mais ne contient pas de données, c'est un élément vide.

Figure 2.2 : élément vide

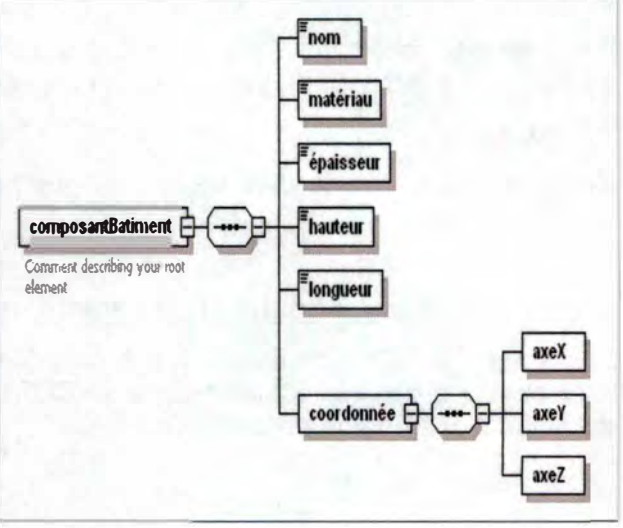
```
</matériau>
```

Pour être valides, les balises ne doivent pas contenir d'espace, ni entre les différents mots qui composent le nom de la balise, ni entre les caractères « < », « > » et le nom de la balise. D'autre part, les balises ne doivent contenir que des lettres et des chiffres, à condition que les chiffres ne se trouvent pas en première position. Les caractères spéciaux ne peuvent pas être utilisés pour nommer une balise. Ces règles de nommage sont valables aussi pour les attributs, que nous verrons par la suite.

Nous avons vu comment nommer nos balises ; maintenant intéressons-nous à la sémantique de nos données. Pour définir une structure XML, il faut raisonner en terme de hiérarchie. En effet, nous devons définir une racine et une arborescence, comprenant des nœuds et des feuilles, les feuilles contenant les données brutes.

Figure 2.3 : représentation arborescente d'un document XML

```
<composantBatiment>
    <nom>Mur</nom>
    <matériau>Bloc de terre cuite</matériau>
    <épaisseur>9</épaisseur>
    <hauteur>230</hauteur>
    <longueur>900</longueur>
    <coordonnée>
        <axeX>125</axeX>
        <axeY>2350</axeY>
        <axeZ>0</axeZ>
    </coordonnée>
</composantBatiment>
```



2.3.2 Les attributs

Les éléments peuvent contenir des attributs. Ce sont, en général des données qui sont associées à de la méta-information concernant l'élément et qui viennent compléter la description d'un élément. La valeur associée à un attribut doit impérativement être contenue entre guillemets.

Figure 2.4 : élément contenant un attribut

```
<épaisseur unité="centimètre"> 9 </épaisseur>
```

Il est possible d'associer plusieurs attributs à un élément, la syntaxe est la suivante :

Figure 2.5 : élément contenant plusieurs attributs

```
<longueur    unité="centimètre"
             codeMesurage="NBN-2456">
             2548
</longueur>
```

2.3.3 Les commentaires

Il est possible d'insérer des lignes de commentaires dans un document XML afin de décrire les fonctionnalités du document XML ou encore pour mettre des lignes de codes temporairement en commentaire. Toute ligne mise en commentaire sera ignorée par le parser et ne sera donc pas interprétée.

Pour insérer une ligne de commentaire dans un document XML, on utilise la syntaxe suivante :

Figure 2.6 : commentaires

```
< !--Voici une ligne de commentaires-->
< !--Voici plusieurs lignes de commentaires
qui vont se terminer
par cette dernière ligne-->
```

2.3.4 Les instructions de traitement

Les instructions de traitement permettent des déclarations spécifiques concernant une application.

Par exemple, pour définir la version du XML et le type d'encodage utilisés dans le document, on insère, au début du document XML, la ligne suivante :

< ?xml version="1.0" encoding="UTF-16" ?>

Cette instruction est aussi appelée prologue ; elle doit figurer dans tous les documents XML créés.

2.3.5 La section de données de caractères : CDATA

La section CDATA est une section importante ; elle permet d'indiquer au parser de ne pas interpréter ce qui est contenu dans la CDATA. En fait, une section CDATA concerne la définition d'une chaîne de caractères.

Il est intéressant de noter que l'utilisation de caractères spéciaux tels que <, par exemple, ne pose pas de problème car le parser n'intervient pas dans la section CDATA.

Une section CDATA représente une feuille de l'arbre défini par un document XML.

CDATA est une instruction particulière qui permet de considérer une chaîne de caractères comme une simple chaîne à afficher ou à transmettre et non pas à interpréter.

2.3.6 La déclaration d'un type de document

Il ne faut pas confondre la déclaration d'un type de document avec une DTD (Data type description) qui permet de définir un ensemble de règles.

Ce type de balise permet de déclarer l'utilisation d'une DTD par un document. Une déclaration de ce type indique au parser quelle DTD il doit utiliser au moment où il va vérifier la validité du document.

Figure 2.7 : déclaration d'un type de document

```
< !DOCTYPE batiment SYSTEM "batiment.dtd">
```

2.3.7 Les namespaces

La notion de namespace est apparue en janvier 1999 et est recommandée par le W3C.

Les namespaces ou les espaces de noms de XML permettent de qualifier de façon unique des éléments et des attributs. Grâce aux espaces de noms, nous pouvons rapidement identifier à quel domaine se rapporte un élément ou un attribut et comment il doit être interprété [Ibrahim 2000].

Différencier les espaces de noms autorise la cohabitation, dans un même document, d'objets portant le même nom, mais ayant une connotation différente, par exemple, parce que ces objets sont liés à un modèle de contenu différent. Dans le monde de la messagerie, il est fréquent que des métiers différents utilisent des concepts portant le même nom mais ayant des significations différentes.

Donner aux éléments et aux attributs des noms uniques, sur Internet, permet d'avoir une connaissance mutuelle des objets, entre plusieurs DTD, sans pour autant qu'il y ait contradiction entre l'interprétation des éléments et, surtout, leurs contenus.

Figure 2.8 : exemple d'utilisation de namespace

```
<élément
xmlns : archi="http://www.archi.org"
xmlns : peintre="http://www.peintre.org">
   <archi:couleur ref="200456">vert bouteille</archi:couleur>
   <peintre:couleur>
      <peintre:référence>200456</peintre:référence>
      <peintre:composition>
         <peintre:bleu>30</peintre:bleu>
         <peintre:rouge>20</peintre:rouge>
         <peintre:vert>50</peintre:vert>
      </peintre:composition>
   </peintre:couleur>
</élément>
```

2.4 La DTD : comment structurer les documents XML

Nous avons vu comment structurer un document en définissant les règles de la structure et en fournissant les données en même temps. Voyons maintenant comment séparer le document XML de sa DTD [Boukhors 2002] [Robbe 2002].

2.4.1 Qu'est-ce qu'une DTD ?

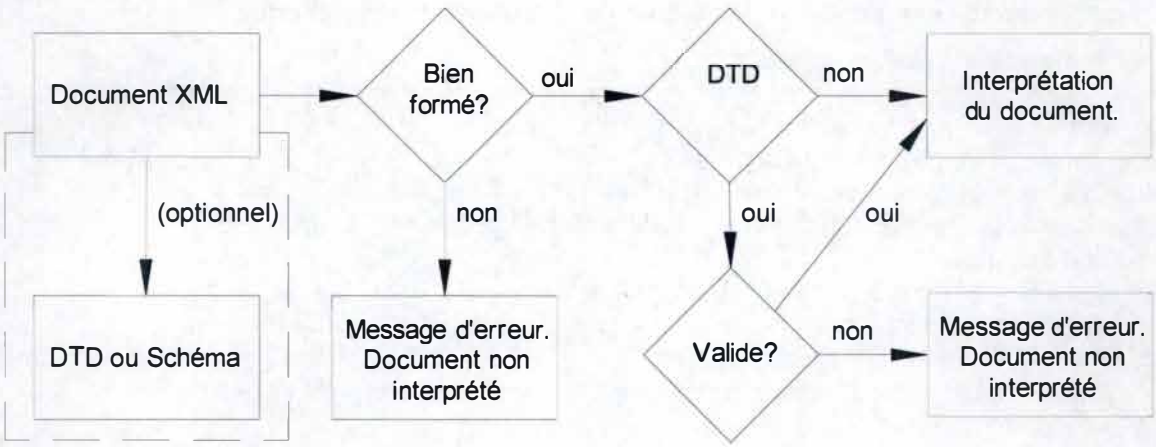
Le besoin de structurer des informations d'une manière précise se fait souvent sentir.
Pour être certain que l'information va être correctement structurée, il sera opportun de créer un ensemble de règles pour les documents XML au moyen d'une DTD. L'ensemble des définitions des données, ainsi que leur type sont définis dans la DTD.
La norme DTD a été émise en octobre 2000 par le W3C dans le but de pouvoir définir la grammaire des documents XML [Bray 2000].

Une DTD peut être comparée à une série d'expressions qui définissent la structure logique du ou des documents XML censés y obéir. Comme nous l'avons vu plus haut, il est possible d'utiliser une grammaire sans pour autant l'externaliser dans un fichier. Mais, en pratique, dans les applications, pour des raisons de facilité, de lisibilité, et de réutilisabilité, il est utile de définir une ou plusieurs DTD, notamment si on prévoit des échanges de données avec d'autres services. L'utilisation d'une DTD permet d'améliorer le partage des données entre plusieurs applications n'ayant pas pour autant les mêmes besoins.

La modélisation d'une DTD doit pouvoir être évolutive et doit pouvoir être facilement utilisable par les autres applications.

Lorsqu'une DTD est associée à un document XML, le parser est en charge de vérifier que le document obéit aux règles fixées par cette même DTD. Cette vérification se déroule en plusieurs étapes. Tout d'abord, le parser vérifie que le document XML est bien formé et c'est seulement après cette première vérification qu'il va pouvoir vérifier que le document répond aux règles spécifiées par la DTD. Une fois ces vérifications terminées, le parser lit les informations dans le document XML et les rend accessibles dans une application XML.

Figure 2.9 : Interprétation d'un document XML par un parser



2.4.2 Déclarer un élément

L'élément est le composant le plus important d'un document XML. En effet, les documents XML sont en grande partie régis par les éléments définis dans la DTD.

Une déclaration d'un type d'éléments est une instruction qui permet de créer et de définir la structure d'une classe d'un élément.

En général, cette déclaration fournit au parser les 3 informations suivantes : son nom, le modèle du contenu et les attributs associés.

Pour déclarer un élément, nous devons utiliser la balise de définition nommée ELEMENT, précédée du point d'exclamation. Ensuite, il faut spécifier le nom de l'élément. Enfin, vient le type de contenu entouré de parenthèses. Le modèle de contenu peut être :

- simple : définit un type de contenu qui contient une chaîne de caractères ;
- complexe : contient un agrégat de plusieurs éléments qui définissent un sous-modèle.

Pour la définition de chaque élément, la syntaxe suivante doit être respectée :
< !ELEMENT nom_élément (modèle_contenu)>

Dans le cas de l'élément matériau, il s'agit d'un type simple, puisque l'élément contiendra une chaîne de caractères.

Figure 2.10 : déclaration d'un élément de type simple

```
< !ELEMENT matériau (#PCDATA)>
```

Maintenant, nous voulons définir un composant de bâtiment, qui est une donnée complexe puisque nous souhaitons pouvoir la décomposer en nom, couche, hauteur, longueur et coordonnée. Il s'agit d'un agrégat de plusieurs éléments, les définitions de nom, couche, hauteur, longueur et coordonnée doivent être retrouvées dans la DTD.

Figure 2.11 : déclaration d'un élément de type complexe

```
< !ELEMENT composantBatiment (nom, couche, hauteur, longueur, coordonnée)>
```

Dans une DTD, l'ordre dans lequel sont déclarés les différents composants d'un élément composite est très important : dans le document XML ces éléments devront être organisés en rapport avec cette déclaration, sinon le document sera rejeté car non valide.

Si nous définissons un sous-modèle dans un élément, sans contraintes d'occurrence, par défaut toutes les balises composant cet élément doivent être présentes dans le document utilisant l'élément en question. Par contre, elles peuvent ne pas être renseignées si on a recours aux opérateurs définis dans le tableau suivant :

Contraintes d'occurrence	Signification
?	0 ou 1
*	0 ou plus
+	1 ou plus
Par défaut :	1 et 1 seul

Figure 2.12 : synthèse des contraintes d'occurrences.

Pour les types composites, il est possible de combiner des contraintes.

Pour illustrer l'utilisation des contraintes d'occurrence, nous allons, à présent, détailler la décomposition de l'élément mur,

Figure 2.13 : utilisation des contraintes d'occurrence dans la définition d'une DTD

```
< !ELEMENT mur (nom, couche+, hauteur, longueur, coordonnée, ouverture*)>
< !ELEMENT couche (épaisseur, matériau?, coefThermique?, coefAcoustique?)>
< !ELEMENT coordonnée (axeX, axeY, axeZ)>
< !ELEMENT ouverture (hauteur, longueur)>
< !ELEMENT nom (#PCDATA)>
< !ELEMENT hauteur (#PCDATA)>
< !ELEMENT longueur (#PCDATA)>
< !ELEMENT épaisseur (#PCDATA)>
< !ELEMENT matériau (#PCDATA)>
< !ELEMENT coefThermique (#PCDATA)>
< !ELEMENT coefAcoustique (#PCDATA)>
< !ELEMENT axeX (#PCDATA)>
< !ELEMENT axeY (#PCDATA)>
< !ELEMENT axeZ (#PCDATA)>
```

Le document XML suivant répond correctement à la grammaire définie précédemment.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mur SYSTEM "C:\1-NOS DOCUMENTS\dtdattr.dtd">
<mur>
    <nom>Mur 2.1</nom>
    <couche>
        <matériau>Bloc portant en béton</matériau>
        <épaisseur>14</épaisseur>
        <coefThermique>1.330</coefThermique>
        <coefAcoustique>0.29</coefAcoustique>
    </couche>
    <couche>
        <matériau>Laine de roche</matériau>
        <épaisseur>6</épaisseur>
        <coefThermique>0.034</coefThermique>
        <coefAcoustique>0.54</coefAcoustique>
    </couche>
    <couche>
        <matériau>Lame d'aire</matériau>
        <épaisseur>7</épaisseur>
        <coefThermique>2.500</coefThermique>
        <coefAcoustique>0.00</coefAcoustique>
    </couche>
    <couche>
        <matériau>Brique de parement</matériau>
        <épaisseur>9</épaisseur>
        <coefThermique>1.230</coefThermique>
        <coefAcoustique>0.33</coefAcoustique>
    </couche>
    <hauteur>240</hauteur>
    <longueur>500</longueur>
    <coordonnée>
        <axeX>2340</axeX>
        <axeY>235</axeY>
        <axeZ>0</axeZ>
    </coordonnée>
</mur>
```

Parfois il peut être intéressant de laisser le choix au développeur du document, quant à la nature de l'élément, c'est-à-dire simple ou contenant un sous-modèle, le mot-clé permettant de définir ce type d'élément est ANY.

Figure 2.14 : élément dont le type n'est pas défini

```
< !ELEMENT mur ANY>
```

2.4.3 Déclarer un attribut

Nous avons déjà abordé le thème des attributs (2.3.2) ; maintenant nous allons voir comment ceux-ci doivent être définis dans une DTD.

Pour que l'attribut d'un élément soit accepté dans le document par le parser, il est nécessaire qu'il ait été déclaré dans la DTD. Les attributs sont déclarés comme suit :
< !ATTLIST non-élément nom-attribut type-attribut valeur-par-défaut>. La déclaration d'un attribut comporte donc les éléments suivants :

- l'élément auquel l'attribut est associé ;
- le nom de l'attribut ;
- le type de l'attribut ;
- la valeur par défaut ou une contrainte d'occurrence.

Les types d'attributs disponibles sont les suivants :

- une chaîne ;
- une énumération, (valeur1 I valeur2 I valeur3) ;
- une entité (ENTITY et ENTITIES) ;
- un identifiant (ID, IDREF, IDREFS) ;
- un symbole (NMTOKEN, NMTOKENS) ;
- une notation : notation prédéfinie (pas besoin de spécifier une valeur par défaut).

Quatre qualificatifs sont disponibles :

- obligatoire (#REQUIRED) : ne peut être laissé en blanc dans le document XML ;
- optionnelle (#IMPLIED) : le parser indique à l'application XML que l'attribut peut être laissé en blanc.
- fixée (#FIXED) : le domaine de valeurs possibles est inclus dans la déclaration. Prend la 1ère valeur de la liste pour valeur par défaut ;
- valeur ("valeur") : indique la valeur prise par défaut parmi les éléments de la liste, si aucune valeur n'est saisie dans le document.

Figure 2.15 : Déclaration d'attributs dans une DTD

```
< !ELEMENT porte (hauteur, longueur)>
< !ELEMENT hauteur (#PCDATA)>
< !ELEMENT longueur (#PCDATA)>
< !ATTLIST hauteur unité CDATA #REQUIRED "cm">
< !ATTLIST largeur unité CDATA #REQUIRED "cm">
< !ATTLIST porte type ("battante"|"coulissante"|"pliante"|"pivotante"|"autre") "battante" >
< !ATTLIST porte couleur CDATA #IMPLIED>
```

Le document XML suivant répond correctement à la grammaire définie précédemment.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE porte SYSTEM "C:\1-NOS DOCUMENTS\porte.dtd">
<porte couleur="" typeP="coulissante">
    <hauteur unité="cm">210</hauteur>
    <longueur unité="cm">90</longueur>
 </porte>
```

2.4.4 Limites des DTD

Bien que la syntaxe des DTD soit relativement simple à appréhender et qu'elle permette l'emploi d'analyseurs de validation efficaces et de taille modeste, elle est trop limitée pour décrire les schémas terminologiques orientés *données* mis en œuvre dans les applications B2B. Nous allons maintenant décrire les principales limites de cette syntaxe.

La structuration des DTD est écrite avec une syntaxe particulière différente de la syntaxe XML. Elle n'est pas toujours supportée par les mêmes outils que les documents XML. Ainsi, il est nécessaire de se procurer un outil spécifique pour pouvoir développer des DTD.

La syntaxe des DTD est relativement limitée ; par exemple un seul type primitif est défini, il s'agit de PCDATA qui représente le type texte. Il est alors impossible de typer les données ou de poser des contraintes sur les valeurs (virgules flottantes, format de date, longueur d'une chaîne, etc.)

Outre le manque de typage pour les données, il n'est pas non plus possible de définir un nombre précis d'occurrences d'un élément dans une grammaire XML. Les contraintes d'occurrence ont le mérite d'exister mais sont bien loin d'être suffisantes.

Une DTD est rigide : elle décrit des structures de documents dans lesquelles chaque élément acceptable doit être connu et codé à l'avance.

Un des problèmes importants est aussi que la validation par une DTD d'un fichier XML peut échouer dans le cas où il contiendrait plusieurs balises portant le même nom et placées à des niveaux d'arborescence différents.

Aucune prise en charge de la documentation n'existe, à l'exception d'un commentaire général.

Les DTD furent créées, dans le cadre de XML "document-oriented" qui prévalait avec le SGML. Il semble que maintenant, les Schémas XML, initiative du W3C soient plus adaptés aux besoins actuels, plus centrés sur l'interopérabilité et "data-oriented".

Le manque de précision des DTD a accéléré la mise au point de langages de schémas plus robustes et plaidé en faveur de l'adoption d'un standard pour une nouvelle génération de schémas, le « Schéma XML » du W3C.

2.5 SCHEMA XML

Il existe plusieurs normes XML Schémas. Celle que nous utilisons dans ce chapitre fait référence au W3C XML Schéma. Cette norme a été adoptée en mai 2001 par le W3C. [Sperberg 2002] [Boukhors 2002].

2.5.1 Objectifs

Un Schéma XML a le même objectif qu'une DTD : le développeur souhaite que son document soit soumis à une validation.

Les Schémas XML permettent de décrire les classes de documents XML. Ils définissent un ensemble de composants :

- la signification, l'utilisation et les relations entre différents éléments et leur contenu
- les attributs et leurs valeurs
- les types de données
- les entités
- les notations

2.5.2 Avantages des Schéma XML

Les Schémas XML ont l'avantage d'être plus précis et plus souples. Ils sont écrits au format XML contrairement aux DTD. Ceci leur donne comme avantage de pouvoir être édités par un éditeur XML classique et non plus par un programme dédié, comme c'est le cas avec les DTD.

Avec les Schémas XML, il est possible de typer les données, des types simples ou complexes sont disponibles.

Les Schémas permettent de décrire l'imbrication et l'ordre d'apparition des éléments et de leurs attributs. En d'autres termes, un Schéma XML permet de définir la grammaire d'un langage particulier.

Les Schémas XML offre donc une meilleure structuration et permettent une plus grande réutilisation.

2.5.3 Déclarer les Schémas XML

Par convention, le préfixe xsd est utilisé pour associer un document XML à l'espace de nommage des Schémas :

Figure 2.16 : déclaration d'un Schéma XML

```
<xsd : schema xmlns : xsd="http://www.w3.org/2001/XMLSchema">
```

2.5.4 Déclarer un élément

Chaque élément déclaré dans un Schéma a un type associé qui définit le genre de contenu que l'élément peut avoir. Pour la définition de chaque élément, la syntaxe suivante doit être respectée : <xsd:element name="NomElement" type="modele_contenu">

Un élément ne contenant pas de sous-élément ni d'attribut est considéré comme étant de type simple (élément feuille).

Figure 2.17 : déclaration d'un élément de type simple

```
<xsd:element name="materiau" type="xsd:string">
```

Un élément contenant soit des sous-éléments, soit des attributs, ou les deux à la fois est considéré de type complexe (élément nœud).

Figure 2.18 : déclaration d'un élément de type complexe

```
<xsd:element name="composantBatiment" type="TypeComposant">
```

Les types utilisés dans le modèle de contenu peuvent être eux-mêmes des types simples ou complexes.

Figure 2.19 : définition du type complexe déclaré ci-dessus

```
<xsd : complexType name="TypeComposant">
        <xsd:element name="nom" type="xsd:string">
        <xsd:element name="couche" type="TypeCouche">
        <xsd:element name="hauteur" type="xsd:decimal">
        <xsd:element name="longueur" type="xsd:decimal">
        <xsd:element name="coordonnée" type="TypeCoord">
</ xsd : complexType>
```

2.5.5 Déclarer un attribut

Un attribut est un qualificatif qui vient compléter un élément. Contrairement aux éléments, les attributs sont toujours des types simples. Les attributs sont déclarés de la même manière que les éléments simple mais en utilisant la balise <attribute>. Tous les éléments contenant des attributs sont de type complexe.

Figure 2.20 : déclaration d'un attribut

```
<xsd:attribute name="Unité" type="xsd:sting">
```

Un attribut peut lui-même contenir les attributs use et/ou value. Ces attributs permettent de donner une valeur par défaut à l'attribut.

Figure 2.21 : contraintes d'occurrences sur un attribut avec les Schémas XML

Valeur de l'attribut use	Valeur de l'attribut value	Conséquence sur l'occurrence de l'attribut
Required	Aucune	L'attribut doit apparaître. Il peut prendre n'importe quelle valeur.
Required	« architecte »	L'attribut doit apparaître, en prenant « architecte » pour valeur.
Optional	Aucune	L'attribut est optionnel et il peut avoir n'importe quelle valeur.
Fixed	« architecte »	L'attribut peut apparaître ; si tel est le cas, sa valeur est « architecte ».
Default	« architecte »	L'attribut peut apparaître ; s'il n'apparaît pas, alors sa valeur est « architecte ». S'il apparaît, sa valeur est celle que l'on voudra bien lui donner.
Prohibited	Aucune	L'attribut ne doit pas apparaître.

Figure 2.22 : déclaration d'attributs dans un Schéma XML

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="porte">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="hauteur">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="xs:string">
                                <xs:attribute name="unité" type="xs:string" use="optional" default="cm"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
                <xs:element name="longueur">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="xs:string">
                                <xs:attribute name="unité" type="xs:string" use="optional" default="cm"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="typeP" use="optional" default="battante">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="battante"/>
                        <xs:enumeration value="coulissante"/>
                        <xs:enumeration value="pliante"/>
                        <xs:enumeration value="pivotante"/>
                        <xs:enumeration value="autre"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="couleur" type="xs:string" use="optional"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Le document XML repris dans la figure 2.15 répond correctement à la grammaire définie ci-dessus.

2.5.6 Le typage des données

L'avantage majeur des Schémas par rapport aux DTD est le typage des données, il est quasiment inexistant avec les DTD, alors qu'avec les Schémas, est fournie une bibliothèque complète de types de données prédéfinis. Les types simples les plus courants y sont représentés : chaîne de caractères, date, réel, entier, etc.

Grâce aux types de données, il sera possible, dès la validation du document XML par le parser, de détecter et de rejeter un document dans lequel une donnée ne correspond pas au type qui lui a été associé.

Figure 2.23 : déclaration d'attributs dans un Schéma XML

Déclaration d'un élément	Bien-formé	Valide
<xsd:element name="materiau" type="xsd:string">		
Document XML respectant la déclaration de l'élément	**Bien-formé**	**Valide**
<materiau>Bloc de terre cuite</materiau>		
Document XML ne respectant pas le Schéma	**Bien-formé**	**Non-Valide**
<materiau>22,34</materiau>		

Les types prédéfinis peuvent être utilisés pour définir les attributs et les éléments simples.

2.5.7 Le modèle de contenu et les connecteurs

Un type de données complexes, comme une adresse, va être caractérisé par un modèle de contenu. Il s'agit en fait d'une séquence de sous-éléments. Chaque élément sera défini ou déclaré par la balise <element>.

Il existe trois connecteurs différents pour les Schémas XML :

1. Le connecteur de séquence (<sequence>) permet de définir une suite d'éléments ; il impose que ceux-ci apparaissent dans le document XML, dans l'ordre dans lequel ils ont été déclarés dans le Schéma. L'inversion de deux éléments dans un document associé à un Schéma provoquera l'invalidation du document.
2. Le connecteur de choix (<choice>) permet de choisir entre plusieurs possibilités.
3. Le connecteur all (<all>) permet de faire apparaître les sous-éléments une fois ou pas du tout, et dans n'importe quel ordre.

2.5.8 L'occurrence d'un élément

Par défaut, lorsque l'on déclare un élément sans préciser le nombre d'occurrences, il doit figurer une et une seule fois dans le document XML. Mais il est possible de définir une occurrence différente de 0, 1 ou n (seules possibilités qu'offrait une DTD). En effet, les attributs minOccurs et maxOccurs permettent de varier à souhait le nombre d'occurrences.

- maxOccurs indique l'occurrence maximale ; les valeurs possibles pour cet attribut sont "unbounded" ou n'importe quel entier souhaité. Par défaut, maxOccurs est égal à 1.
- minOccurs indique l'occurrence minimale ; les valeurs possibles pour cet attribut sont de 0 à n. Si minOccurs est égal à 0, cela indique que l'élément est considéré comme optionnel. Par défaut, minOccurs est égal à 1.

Il faut conserver à l'esprit qu'une instance unique d'un élément est attendue dans le document XML, si rien n'est précisé. Ainsi pour indiquer que le document XML peut contenir zéro ou plusieurs instances d'un élément, il faut renseigner les attributs maxOccurs et minOccurs correctement.

Figure 2.24 : utilisation des contraintes d'occurrence dans la définition d'un Schéma XML

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:element name="mur">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="nom" type="xs:string"/>
                <xs:element name="couche" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="épaisseur" type="xs:decimal"/>
                            <xs:element name="materiau" type="xs:string" minOccurs="0"/>
                            <xs:element name="coefThermique" type="xs:decimal" minOccurs="0"/>
                            <xs:element name="coefAcoustique" type="xs:decimal" minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="hauteur" type="xs:decimal"/>
                <xs:element name="longueur" type="xs:decimal"/>
                <xs:element name="coordonnée">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="axeX" type="xs:decimal"/>
                            <xs:element name="axeY" type="xs:decimal"/>
                            <xs:element name="axeZ" type="xs:decimal"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="ouverture" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="hauteur" type="xs:decimal"/>
                            <xs:element name="largueur" type="xs:decimal"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Le document XML repris dans la figure 2.13 répond correctement à la grammaire définie ci-dessus.

2.5.9 Elément local ou global

Il est possible de déclarer des éléments localement ou globalement.

Pour déclarer des éléments globalement, les éléments doivent être déclarés comme fils de l'élément Schéma. L'attribut ***ref*** permet d'indiquer sous quel nom est défini le sous-élément global.

Les attributs peuvent aussi être déclarés de manière globale. De la même manière, les attributs devront être déclarés comme fils du schéma et seront référencés dans les éléments complexes grâce à l'attribut ***ref***.

2.5.10 Création de types complexes à partir de types simples

Il est possible d'associer des attributs à des éléments simples mais la syntaxe est un peu verbeuse.

D'abord, il va falloir définir un nouveau type complexe.

Dans ce type complexe, on définit la balise <simpleContent> qui indique que nous définissons un élément qui ne contient pas de sous-éléments.

Dans l'élément simpleContent, on définit la balise <extension> qui permet de créer un élément simple dérivé d'un type de base.

Cet héritage permet de créer un type simple pourvu d'attributs. C'est entre les balises <extension> et </extension> que les attributs vont être déclarés.

2.5.11 Les types complexes vides

Il peut arriver que certains éléments soient vides, c'est-à-dire sans contenu, mais qui comprennent cependant des attributs. Pour ce faire, il faut déclarer un élément complexe particulier : un type complexe vide.

2.5.12 Limites des Schémas XML

Comme nous l'avons constaté, les Schémas XML complètent les DTD en de nombreux points. Les Schémas XML sont mieux adaptés aux besoins actuels et sont plus centrés sur l'interopérabilité. Cela est visible grâce au typage très complet, la possibilité de dériver des types, la précision de dénombrement des éléments, etc.

Mais là où quelques lignes suffisaient à définir la structure d'un document avec une DTD, on est amené à rédiger près d'une page avec les Schémas XML. On a donc gagné en précision pour perdre en concision.

PARTIE 2

Conception et développement de transactions E-business

CHAPITRE 3 : Transaction e-Business

3.1 Contexte actuel

Les années 1990 ont vu l'avènement d'une intégration poussée des systèmes d'information favorisant la communication de données entre les différentes entités de l'entreprise. Depuis quelques années, l'émergence d'Internet et des réseaux ont dématérialisé les frontières qui existaient et ont ainsi permis des échanges d'informations qui ne se limitent plus aux filiales délocalisées mais qui permettent des échanges avec d'autres entreprises et des acteurs divers répartis un peu partout sur la planète. Cette ouverture a permis le développement d'une nouvelle approche d'échange d'informations, appelé « e-business » qui confère une importance économique aux échanges électroniques.

Ce passage d'une économie "matérielle" à une économie "immatérielle" a des conséquences très importantes sur la vie des entreprises et leurs évolutions. L'enjeu stratégique se situe au niveau du partage de l'information [EDIFRANCE 2002] [b-process 2002].

3.2 Les principaux modèles d'échange électronique

En matière de commerce électronique, on peut distinguer deux modèles principaux d'échange électronique :

- **B2C** : vente directe au client

  Cette partie qui est la plus visible du grand public est appelée à connaître un grand essor dans les années qui viennent. Le concept B2C introduit le concept de la désintermédiation. Le fournisseur peut, en effet, proposer directement ses produits à l'utilisateur final et supprimer le réseau de distribution (par exemple, Amazon.com.).

- **B2B** : transactions inter-entreprises

  Ce mode de commerce électronique constitue la partie immergée de l'iceberg. Les transactions électroniques permettent de couvrir la totalité du flux commercial. L'avantage évident de ce modèle est la suppression des échanges de documents papier à traiter qui ralentissent et ajoutent un surcoût aux échanges commerciaux. Un autre avantage est l'intégration poussée qui peut être réalisée avec les systèmes de gestion de l'entreprise.

Dans le cadre de ce travail et plus particulièrement dans la troisième partie, nous nous intéresserons exclusivement aux échanges « Business to Business » ce qui signifie l'échange de données et de services entre professionnels.

3.3 Les implications technologiques d'une solution B2B

La mise en place d'une solution de commerce électronique dans une organisation demande de formaliser trois éléments importants.

1/ Définir les processus métier[2]

Un projet de commerce électronique est avant tout un projet d'organisation. Les échanges de données électroniques ne sont possibles que lorsque les procédures métier sont établies.

[2] Processus métier : Dans le contexte de l'ingénierie des processus, il s'agit d'un flux d'activités concourant à la réalisation d'un objectif identifié dans le domaine d'activité de l'entreprise ou de l'organisation. [Fannader 2002]

Ces procédures définissent en effet précisément la manière de commercer avec ses partenaires. Dans le contexte du « commerce mondial », les procédures doivent pouvoir être définies de manière non ambiguë et au moyen d'un langage universel : UML s'offre comme une réponse à ce problème.
UML est aujourd'hui une technologie stable et mature couvrant les nombreux besoins de modélisation des processus métiers. De plus, de nombreux travaux s'attachent à l'échange de ces modèles à travers la définition de XMI (XML Metadata Interchange).

2/ Se comprendre à travers un langage d'échange d'informations

Le langage servant de base à la conversation commerciale doit être formalisé et compris de chacun des participants à la transaction. Dans ce domaine comme dans beaucoup d'autres, les technologies ouvertes et standardisées prennent le pas sur les initiatives propriétaires. Un effort significatif avait été fait dans ce domaine avec la définition de l'EDI, mais aujourd'hui la technologie XML prend le pas rapidement.

3/ Posséder une infrastructure de communication sécurisée

L'infrastructure de communication est disponible aujourd'hui grâce au développement d'Internet. Par ailleurs, pour résoudre le problème de la sécurité qui est crucial pour l'établissement de transactions électroniques sur le « réseau des réseaux », il apparaît aujourd'hui que les technologies de sécurisation sont, elles aussi, matures et disponibles : les mécanismes de réseau privé virtuel sont intégrés dans les protocoles d'Internet. Une infrastructure basée sur les certificats peut également être mise en place pour répondre à des besoins élevés de sécurité.

3.4 L'approche EDI

3.4.1 Historique

L'échange informatisé de données existe depuis une trentaine d'années. De nombreuses entreprises des secteurs de l'automobile, de la grande distribution ou de l'industrie électronique ont réussi, bien avant le développement du Web et de l'Internet, à mettre en place des solutions propriétaires d'échanges de données, celles-ci s'accompagnant de réduction des coûts et d'amélioration des processus métiers. Ces développements ont souligné le besoin d'une normalisation des échanges afin d'éviter une prolifération des formats d'échange.

Un certain nombre d'acteurs se sont alors rassemblés pour essayer de fédérer les développements et travailler sur la normalisation des formats, dans l'objectif de définir un standard d'échange interopérable. Ces travaux ont débouché, d'un part sur la norme ANSI X12 principalement utilisée aux Etats-Unis et, d'autre part, sur la norme EDIFACT (Electronic Data Interchange for Administration, Commerce and Transport) utilisée partout ailleurs.

Dans le cadre de l'EDI, pour pouvoir s'échanger des données, les partenaires ont recours à :

- une sémantique définie comme un dictionnaire de données ;
- une syntaxe : des normes bien particulières (ANSI X12, EDIFACT...) définie par des non informaticiens ;
- des moyens d'échanges réalisés au moyen de réseaux à valeur ajoutée.

3.4.2 EDIFACT

EDIFACT a pour vocation de pouvoir effectuer des transactions répétitives de manière électronique en éliminant ainsi les flux papier, et ceci dans un cadre normalisé. L'origine de EDIFACT, contrairement à XML, n'est pas informatique mais fonctionnelle et le système a donc été conçu d'abord par des utilisateurs. C'est la raison pour laquelle, il a été décidé de

mettre en place un ensemble très strict de règles mêlant syntaxe, dictionnaires (nomenclature internationale), messages, codes, règles de fonctionnement et de créer simultanément l'organisme d'enregistrement et ses ramifications, les groupes d'utilisateurs.

L'objectif d'EDIFACT est de permettre à des utilisateurs, quelle que soit leur localisation géographique, d'utiliser un message en précisant tous les aspects de l'échange afin de réduire l'ambiguïté au minimum. Pour cela, un message EDIFACT ne correspond à la norme que si sa définition est enregistrée dans un dictionnaire mondial public.

Avantages d'EDIFACT

Les répertoires d'informations sont le point fort d'EDIFACT. Ces répertoires appelés TDID (Trade Data Interchange Directory) contiennent des définitions des données échangées et les codifications associées. Ils se sont construits au fil des années et sont le fruit d'une multitude de groupes dans tous les secteurs d'activités. Ces répertoires, mis à jour de manière semestrielle, représentent la valeur ajoutée sémantique d'EDIFACT.

Limites d'EDIFACT

- La norme EDIFACT précise les règles syntaxiques de structuration de l'information pour réaliser des échanges. Cependant, à l'opposé de XML qui s'appuie sur un métalangage formel au sens mathématique, EDIFACT repose sur des règles empiriques qui conduisent souvent à des ambiguïtés de compréhension, qui rendent difficile le développement des applications pouvant traiter ces flux de données.
- La syntaxe développée demande un effort important d'implémentation pour chaque nouveau message.
- Au niveau sémantique, la constitution de dictionnaires comprenant la définition des différents « mots » est réalisée en langage naturel. Ce recours au langage naturel introduit, dans certains cas, des problèmes d'ambiguïté et d'interprétation multiple.
- La description des règles de validation est souvent insuffisante et se limite au message sans prendre en compte la transaction dans sa globalité. En effet, une plus grande attention est portée sur le contenu du message que sur le concept de transaction.
- EDIFACT est rarement appliqué à la lettre par les utilisateurs en raison de son manque de souplesse. On peut voir là un échec de la normalisation puisque la rigidité du format a conduit les entreprises utilisatrices à s'approprier le format en le modifiant pour le faire correspondre à leurs besoins spécifiques.
- On peut aussi mentionner un problème de coût de développement : la réutilisation cross-sectorielle est difficile, ce qui revient à maintenir des relations 1 to 1 entre les acteurs.

Evaluation d'EDIFACT

Aujourd'hui, une majorité des grandes entreprises qui font des échanges de données électroniques le font par EDI de type EDIFACT. Ce système, par son coût, sa lourdeur et sa complexité à mettre en œuvre a eu une très faible pénétration dans le secteur des PME. Bien que l'EDI apporte une réponse fonctionnelle et technique intéressante à la transmission des informations inter-entreprises, il a été subi par un grand nombre d'entreprises (surtout des PME, sous la pression d'un grand donneur d'ordre) et est resté cantonné à quelques secteurs d'activité.

EDIFACT est un langage structuré qui convient parfaitement à des échanges réguliers et nombreux entre partenaires stables. EDIFACT n'est pas adapté à des échanges épisodiques entre partenaires irréguliers.

3.4.3 Quel avenir pour EDIFACT ?

Cela ne signifie pas pour autant que l'EDI est à remplacer. Même si la solution EDI est coûteuse à maintenir, on choisit de la maintenir, lorsqu'elle est déjà en place dans l'entreprise : il serait en effet encore plus coûteux de changer cette technologie. Cela aurait un impact sur les processus profonds du système d'information, sans compter l'impact sur les partenaires. C'est pourquoi des ponts XML-EDI ont été mis en place.

Les acteurs traditionnels et historiques de l'EDI s'impliquent toutefois dans les comités de réflexion XML pour que les avantages de l'EDI soient conservés. Ils peuvent en effet faire bénéficier cette nouvelle technologie des années de travail qui ont abouti à la normalisation des dictionnaires EDI. Leur participation vise aussi à permettre une transition entre les deux mondes.

Aujourd'hui, le monde EDI se tourne avec intérêt vers les perspectives ouvertes par XML.

CHAPITRE 4 : ebXML

4.1 XML pour les échanges électroniques

Comme nous l'avons mentionné précédemment, le métalangage XML présente un certain intérêt dans le cadre des échanges électroniques entre entreprises et entre systèmes d'information. Nous allons, à présent, exposer ses principaux atouts et ensuite ses limites [Dubois 2002] [b-process 2002].

4.1.1 Avantages de XML

Technologie ouverte basée sur des standards internationaux

La technologie XML est non-propriétaire : sa spécification et ses évolutions sont pilotées par une instance neutre qui force les différents acteurs à coopérer dans des groupes de travail et qui empêche un des acteurs de s'approprier la technologie. Sa rapide standardisation (due à son héritage SGML) a placé dans les mains des éditeurs un outil à la fois simple et puissant pour structurer les données inter-applicatives.

Portabilité, interopérabilité, réutilisation

XML a pour force son extrême portabilité car il est défini avec une indépendance totale par rapport au système qui le réceptionne ou au protocole qui le transporte. Cette caractéristique en fait un media de choix pour les transactions du commerce global.

L'interopérabilité est atteinte par la mécanique de transformation d'arbre mise en place dans XML au travers de la technologie XSLT.

Enfin, la réutilisation est naturelle grâce aux structures de documents (DTD ou schémas XML) qui peuvent être repris pour servir de base à un développement, puis étendu. C'est le rôle des registres de Schema XML tel que ebXML de favoriser la réutilisation en publiant ces schémas.

Fondements théoriques, flexibilité, évolutivité

XML repose, contrairement à EDI, sur des fondements théoriques forts (théorie des arbres et des graphes). Ceci lui confère une richesse importante en terme d'algorithmes de manipulation et d'optimisation des structures.

La flexibilité de XML permet une manipulation des données au sein d'outils standard avec une grande facilité. Ces composants applicatifs sont modulaires et l'on peut donc remplacer un élément par un autre qui apporte de meilleures performances.

Les recherches menées actuellement dans le cadre de XML montrent le potentiel qui lui reste (XML Query, Xforms, XML Digital Signature, ...)

Adoption massive, banalisation, réduction des coûts

XML a rencontré un succès immédiat auprès des différents éditeurs de logiciels dans les deux dernières années. Des rivaux industriels tels que IBM, Microsoft, Sun et Oracle supportent tous le standard XML 1.0, développent des produits majeurs sur cette technologie et collaborent pour développer des standards qui lui sont liés.

Cette adoption massive, et pas seulement par les acteurs du commerce électronique, induit une banalisation et une mise à disposition des outils de base (ceux-ci n'ont en effet plus de valeur ajoutée). Les compétences autour de cette technologie seront également beaucoup plus répandues que pour l'EDI, ce qui facilitera sa mise en œuvre par les équipes techniques.

4.1.2 Limites de XML

Plusieurs styles de description des messages

XML laisse la possibilité de représenter un élément par un « élément XML » ou par un « attribut XML ». Le choix dépend principalement de la connaissance que l'on a des applications qui traiteront ce type de document XML.

Si une application XML est imprévisible dans son choix entre éléments et attributs, le processeur sera bien plus sollicité lors de l'interprétation des documents. Par conséquent, il est nécessaire de faire preuve de cohérence dans la manipulation des éléments et des attributs XML pour une terminologie donnée.

Sémantique business des noms

La définition du nom des balises selon un vocabulaire propre au business peut entraîner des ambiguïtés et des difficultés de lecture pour une personne étrangère au domaine d'application.

Règles de validation limitées

La limite connue des DTD et des schémas XML ne permet pas d'exercer des contrôles de validation très poussés sur le document XML. En effet, la validation se limite, d'une part, pour une DTD, à la structure du message et, d'autre part, pour un schéma, à la structure du message et au typage des champs.

4.1.3 Quel avenir pour XML dans les échanges électroniques ?

L'EDI a su s'imposer dans un certain nombre de secteurs d'activités où il va continuer à être utilisé pendant quelques années. Dans un premier temps XML va permettre de généraliser les échanges de données informatisées à tous les secteurs d'activités, à toutes les typologies d'organisations ainsi qu'à tous les types de données. Dans un deuxième temps, il est probable qu'il supplantera complètement l'EDI.

Bien que XML soit une des technologies majeures du commerce électronique, il faut noter que XML ne suffit pas, en effet, XML ne définit pas :

- de langage commun pour le commerce électronique ;
- de structure standard d'un document de commerce électronique ;
- de standard pour le transport et la sécurité de l'information ;
- une architecture standard commune.

Ces rôles sont dévolus à plusieurs organisations. Le but de ces organisations est de fournir un forum indépendant, afin de permettre l'établissement de standards techniques ouverts. Elles vont tenter de définir un langage commun ainsi qu'une structure standard de document, afin de rendre l'interopérabilité proposée par XML, une réalité certaine.
Citons ebXML, une initiative conjointe des Nations Unies et du CEFACT, qui crée une architecture d'échange fondée sur des scénarios d'affaires types et non sur des formats de données précis. La spécification a été adoptée en mai 2001. EbXML est une des initiatives les plus prometteuses à l'heure actuelle.

L'importance de ces organisations est indéniable : l'interopérabilité fournie par XML est virtuelle, encore faut-il pouvoir s'entendre sur le format de l'information que l'on va faire transiter. Le dépôt dans une organisation indépendante des schémas est garant du bon fonctionnement du système, XSLT permettant ensuite d'adapter ce schéma standard aux besoins de l'entreprise.

La rubrique qui suit tentera de faire comprendre les principes généraux de l'architecture ebXML et plus particulièrement le concept de transaction.

4.2 Le projet ebXML

4.2.1 Historique

EbXML a débuté en novembre 1999. La première phase du projet a été achevée en mai 2001 et s'est concrétisée par la définition d'une architecture et la publication d'un ensemble de spécifications clés pour la mise en œuvre de commerce électronique. Les entreprises disposent maintenant d'une méthode standard basée sur XML pour échanger des messages d'affaires, opérer leurs relations commerciales, communiquer des données partageant la même sémantique, définir et enregistrer des processus d'affaires.

Les travaux d'ebXML se poursuivent sous le contrôle de deux instances qui ont supervisé la phase initiale du projet, UN/CEFACT et OASIS. Les spécifications ont été améliorées et les projets en cours sont en voie de faire la preuve de la viabilité de l'utilisation d'ebXML dans le monde réel [Girard 2001] [Chauvet 2001].

4.2.2 Mission

L'initiative ebXML a été mise en place en raison du besoin largement reconnu de permettre aux entreprises, quelles que soient leur taille et leur localisation géographique, de mettre en œuvre leurs opérations de commerce électronique de manière simple, économique et fiable.

L'objectif principal est de fournir une infrastructure ouverte basée sur XML permettant à tous les partenaires commerciaux potentiels d'utiliser l'information du commerce électronique de manière universelle, interopérable, sécurisée et cohérente.

Pour répondre à cet objectif, le projet ebXML couvre les domaines suivants :

- Description des processus
- Enregistrement des processus et modèle d'échanges
- Interface pour supporter les processus
- Définition des configurations techniques
- Définition des messages entre partenaires
- Outils de configuration des services sur les messages et les processus

ebXML recommande de plus une méthode (UMM), basée sur UML, pour garantir la cohérence sémantique des concepts utilisés [CEFACT 2001].

UMM est basé sur UML ; il utilise un certain nombre de diagrammes UML qui doivent être développés à différents stades d'un projet B2B. UMM peut-être considéré comme un sous- modèle de UML. Les différents diagrammes utilisés dans le cadre de la méthode UMM ont été définis dans le premier chapitre de ce travail.

4.2.3 Apport d'ebXML

ebXML est le nouveau standard pour les échanges électroniques professionnels. Il propose une infrastructure ouverte, entièrement basée sur les technologies d'Internet pour les protocoles d'échanges et de messageries, et utilisant XML pour la syntaxe des messages et le vocabulaire.

L'idée générale de ebXML est de prendre le meilleur des deux mondes, celui de l'EDI traditionnel d'une part, celui de la facilité et du faible coût de mise en œuvre d'Internet d'autre part.

Par rapport à EDIFACT, le projet va beaucoup plus loin, car il modélise les processus d'affaires, décrit l'interaction entre les processus d'affaires et les données traitées, décrit les

moyens techniques utilisables par les partenaires et permet de déterminer entre deux partenaires les dénominateurs communs techniques utilisables.

De plus, le projet ebXML se caractérise par une centralisation des données et de processus communs dans des répertoires ou référentiels qui peuvent être mis à jour par les utilisateurs et services de base pour le téléchargement des logiciels applicatifs ebXML qui auraient été développés selon ces spécifications.

Donc, au-delà des données et messages EDIFACT, ebXML propose une gestion des processus d'affaires, une composition dynamique du contenu des messages en fonction des exigences d'une profession ou des partenaires, et une infrastructure technique de mise en œuvre.

L'intérêt de ce projet normatif est qu'il s'appuie de manière implicite sur les technologies objet, sur les architectures applicatives à base de composants et sur l'infrastructure technique et de communication de l'Internet.

En ce qui concerne les spécifications de contenu, il existe également des approches verticales, visant à regrouper le plus grand nombre d'acteurs d'un secteur industriel donné. Par exemple, RosettaNet fournit une spécification pour les fabricants de composants électroniques ou semi-conducteurs et les entreprises des technologies de l'information. RosettaNet a une portée moins générale que les spécifications ebXML mais certaines notions de son architecture et certaines notions métier dégagées au cours de la formalisation des processus métier trouvent un champ d'application plus vaste dans le domaine du commerce électronique en général. C'est sur ces points que des rapprochements entre RosettaNet et ebXML ont eu lieu pour harmoniser les définitions et les vocabulaires XML.

4.3 L'architecture technique d'ebXML

4.3.1 Principaux concepts

Les principaux concepts utilisés sont les suivants [ebXML 2003] :

- Un mécanisme standard pour décrire les processus d'affaires (Business Process) et leur association avec le modèle d'informations.
- Un mécanisme pour enregistrer et mettre à disposition les processus d'affaires et les méta-modèles d'information (Information Meta Models). Ils peuvent ainsi être partagés et réutilisés.
- La découverte de l'information relative à chaque participant, intégrant :
  - Les processus d'affaires qu'ils sont capables de supporter.
  - Les interfaces de services d'affaires (Business Service Interfaces) qu'ils offrent pour supporter leurs processus d'affaires.
  - Les messages d'affaires qui sont échangés entre leurs interfaces de services d'affaires respectives.
  - Les configurations techniques des protocoles de transport, de sécurité et de codage
- Des mécanismes pour enregistrer les informations mentionnées ci-dessus de manière à ce qu'elles puissent être facilement trouvées et restituées.
- Un mécanisme pour décrire la mise en œuvre d'un agrément mutuel entre deux partenaires (Collaboration Protocol Agreement – CPA)

- Un cadre standardisé de service de messagerie d'affaires (Messaging Service) qui garantit l'échange fiable, sûr et interopérable des messages entre les partenaires.
- Un mécanisme pour la configuration des services de messagerie respectifs chargés d'exécuter les transactions des processus d'affaires convenus conformément aux règles prédéfinies dans l'accord d'échange.

4.3.2 Composants de l'architecture ebXML

A partir de cette description sommaire des concepts et de l'architecture, nous allons décrire sommairement les composants technologiques nécessaires pour compléter l'architecture technique ebXML. Ces composants peuvent être classés suivant deux grandes catégories :

- L'infrastructure sémantique
- L'infrastructure technique

Chaque composant trouve sa place dans l'une des deux catégories d'infrastructure.

La manière dont ces composants sont assemblés les uns avec les autres dans l'infrastructure ebXML est montrée dans le schéma suivant :

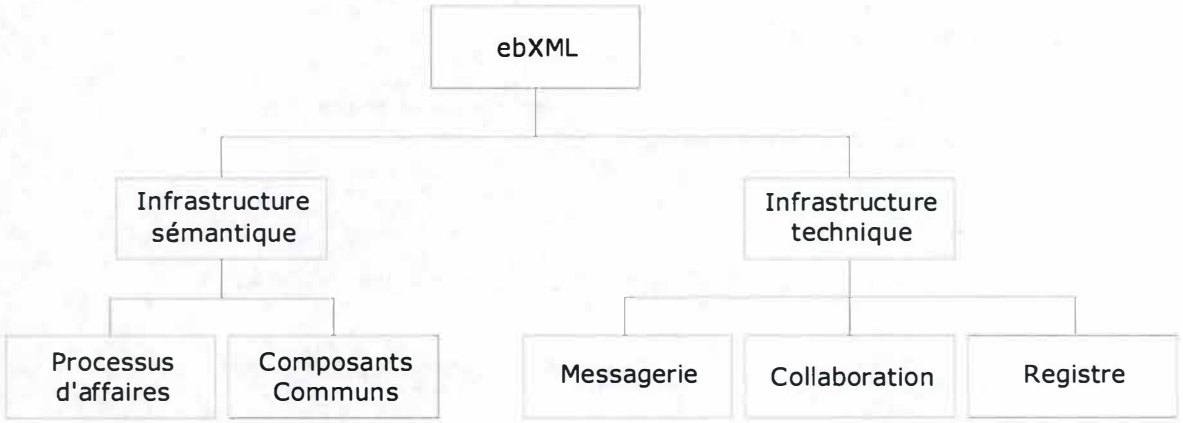


Figure 4.1 : Composants de l'architecture ebXML

4.3.3 Scénario d'utilisation ebXML

Après avoir relevé les principaux concepts, nous allons illustrer les principes de l'architecture technique d'ebXML au travers d'un exemple [ebXML 2003]. Cet exemple commence par la configuration des interfaces ebXML et aboutit à l'exécution d'une transaction d'affaire simple dans un interchange.

Le diagramme qui suit montre un exemple de ce processus et les étapes nécessaires pour configurer puis déployer des applications ebXML et les composants logiciels qui leur sont associés.

Les spécifications de ebXML ne sont pas limitées à cet exemple ; ce dernier a uniquement pour but de fournir un rapide aperçu du fonctionnement d'ebXML.

Nous allons représenter cet exemple à l'aide d'un diagramme de séquence (UML). Ce type de diagramme va nous permettre de décrire l'interaction entre les « acteurs » en mettant en relief la chronologie des envois de messages.

En ce qui concerne la représentation graphique :

- Un acteur est représenté par une ligne verticale.
- Un message est représenté par une flèche horizontale, orientée de l'émetteur du message vers le destinataire.
- La dimension verticale représente l'écoulement du temps de haut en bas.

Diagramme de séquences

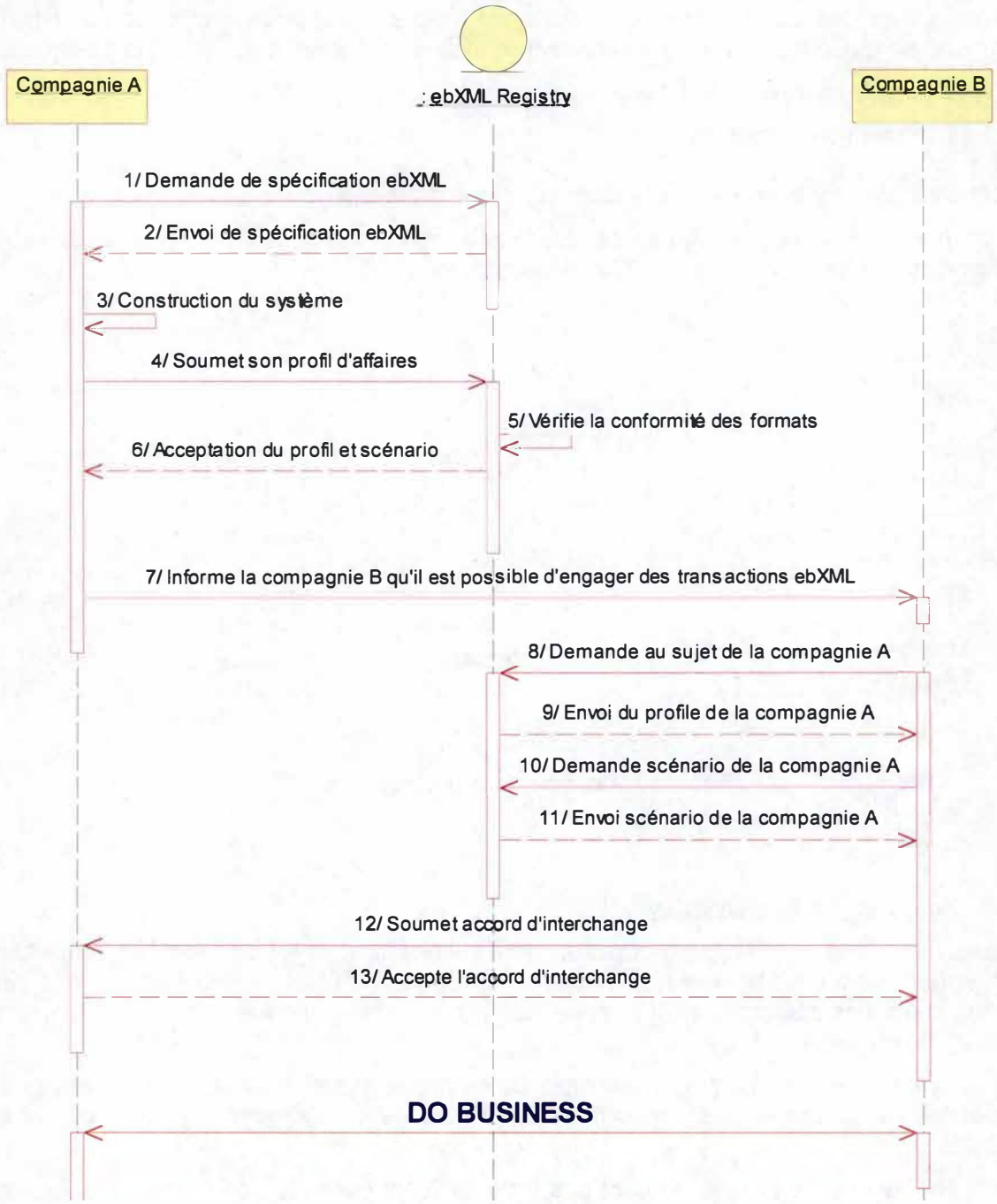


Figure 4.2: Utilisation de ebXML, diagramme d'activité.

Description des différentes étapes

1- Demande de spécification ebXML : la société A consulte les répertoires qui contiennent des ensembles de spécifications d'utilisation d'ebXML pour savoir si elle peut en devenir utilisatrice. Cette demande est guidée à partir de mots clé qui permettent de définir le secteur d'activité, le type de transaction, l'emplacement géographique... de A.

2- Envoi de spécification ebXML : le résultat de cette demande de spécifications est transmis à la société A.

3- Construction du système : en se basant sur les informations disponibles sur le référentiel ebXML, La compagnie A peut construire ou acheter l'implémentation ebXML adaptée à ses transactions ebXML. Cette démarche vise à mettre à niveau son système d'information pour qu'il puisse répondre aux spécifications ebXML dans sa sphère d'utilisation.

4- Soumet son profil d'affaires : La compagnie A soumet alors l'information concernant son profil d'affaires (y compris les détails d'implémentation et les liens de références) au registre ebXML. Le profil d'affaires soumis au registre ebXML décrit les capacités et les contraintes du système ebXML de la compagnie ainsi que les scénarios d'affaires supportés. Les scénarios d'affaires supportés par la compagnie A sont déclarés sous forme de versions XML des processus d'affaires et des documents et blocs d'information qui leurs sont associés (par exemple, règles de calcul d'une taxe sur la vente).

5- Vérifie la conformité des formats : vérification de la conformité des formats et des descriptions des règles d'utilisations du scénario d'affaires.

6- Acceptation du profil et scénario : un accusé est envoyé par le registre à la compagnie A.

7- Informe la compagnie B qu'il est possible d'engager des transactions ebXML : La compagnie B est informée par la compagnie A qu'il est possible d'engager des transactions d'affaires utilisant ebXML.

8- Demande au sujet de la compagnie A : La compagnie B possède une application capable d'assurer une interface ebXML avec ses applications existantes. L'application ebXML de la société B va interroger le répertoire ebXML afin d'obtenir le profil de la compagnie A.

9- Envoi du profil de la compagnie A : le registre ebXML envoie à la compagnie B les spécifications qui sont propres à A.

10- Demande scénario de la compagnie A : L'application ebXML de la société B va interroger le répertoire ebXML afin d'obtenir les scénarios d'affaires supportés par la compagnie A.

11- Envoi scénario de la compagnie A : le registre ebXML envoie à la compagnie B les scénarios d'affaires supportés par la compagnie A.

12- Soumet accord d'interchange : avant de s'engager dans l'exécution du scénario, la compagnie B soumet directement à l'interface ebXML de la compagnie A une proposition d'accord d'interchange. L'accord d'interchange proposé indique que les deux compagnies s'engagent mutuellement sur les scénarios d'affaires et autres spécificités sur lesquels porteront leurs échanges. L'accord d'interchange contient aussi des informations relatives aux spécifications des services de messagerie qui prendront en charge les transmissions, sur les mesures d'exception et procédures dégradées et sur les mécanismes de sécurité associés.

13- Accepte l'accord d'interchange : La compagnie A accepte l'accord d'interchange.

14- Les compagnies A et B sont alors prêtes à engager l'exécution d'échanges électroniques utilisant ebXML.

4.4 Démarche de développement d'un service ebXML

Développer un service ebXML peut être réalisé en suivant deux démarches principales et complémentaires [ebXML 2003] :

4.4.1 L'infrastructure technique

Cette démarche permet de spécifier une infrastructure de collaboration basée sur ebXML. Elle se compose de quatre parties distinctes :

Spécification service de messagerie

Elle fournit un protocole capable d'assurer un transfert fiable et sécurisé de messages entre des partenaires.

Spécification Implémentation, Interopérabilité et Conformité

Elle fournit des moyens pour créer une infrastructure et des applications conformes aux spécifications ebXML et capables d'interopérer.

Spécification Registre

Elle fournit des informations pour le développement de registres interopérables, dotés d'une interface capable d'administrer la soumission, la recherche et l'extraction des contenus enregistrés dans le registre.

Spécification des profils et accords de protocole de collaboration

Un profil d'affaires définit les capacités techniques d'un partenaire à engager des collaborations de commerce électronique avec d'autres partenaires en échangeant des messages électroniques. Un accord d'interchange documente l'accord technique entre deux ou plusieurs partenaires préalablement à l'engagement d'une collaboration de commerce électronique entre ces partenaires.

4.4.2 L'infrastructure sémantique

Cette démarche permet de spécifier les contenus des messages d'affaires. Elle se compose de deux spécifications distinctes :

Schéma de spécification des processus d'affaires

Il permet aux partenaires de définir leurs processus d'affaires d'une manière unique et cohérente. Le but est de fournir une méthode permettant de développer des composants logiciels pour le commerce électronique à partir de la spécification et de la modélisation des processus d'affaires.

Spécification technique des composants communs

Elle permet aux partenaires de développer un ensemble partagé de blocs sémantiques d'information capable de représenter les principaux types de données d'affaires qui sont utilisés actuellement par les applications des entreprises et de mettre ces blocs sémantiques à disposition des utilisateurs pour la création de nouveaux vocabulaires des affaires et la restructuration des vocabulaires existants.

La suite de notre étude se concentre sur la modélisation des aspects sémantiques impliquant une compréhension plus approfondie du « business » et son domaine associé. Nous nous intéresserons donc exclusivement à la démarche concernant l'infrastructure sémantique.

CHAPITRE 5 : Méthodologie de modélisation

Ce chapitre décrit la méthodologie qui sera utilisée dans la troisième partie de ce travail. Cette méthodologie s'appuie sur le projet de recherche « Efficient » réalisé par le Centre de Recherche Public Henri Tudor [Eshuis 2003] [Gregoire 2004]. Cette méthodologique a pour but d'aider des experts métiers à formaliser les transactions qu'ils souhaitent mettre en place et permet, à partir de spécifications UML, de dériver systématiquement la solution XML [Carlson 2001] [Dubois 2002].

5.1 Introduction

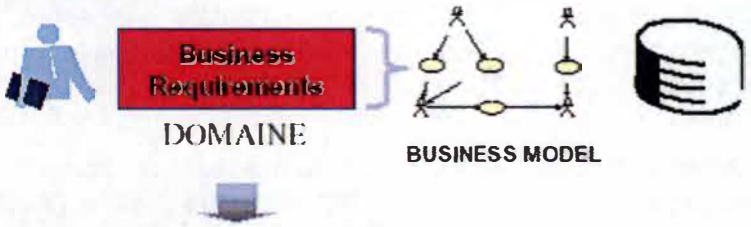
Les transactions électroniques occupent une place croissante dans les activités de nombreuses entreprises. Ces transactions impliquent par nature des acteurs dispersés géographiquement qui échangent des messages complexes. La complexité de ces flux entrave la bonne compréhension de ces transactions, les capacités des experts du métier à les expliquer et des experts informatiques à les comprendre.

Ces difficultés de compréhension, et donc aussi d'amélioration, ont été mises en évidence par des initiatives telles que ebXML, qui propose un ensemble de spécifications pour faciliter la mise en place de telles transactions.

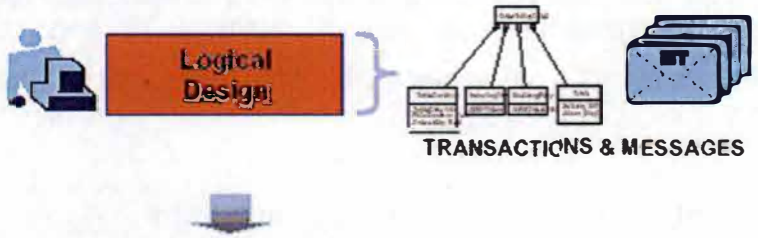
Cette méthodologie cherche à favoriser la compréhension de la sémantique de la transaction et la compréhension de la sémantique des messages avant de passer à leur implémentation XML.

La compréhension d'une transaction e-business selon la proposition d'efficient est progressive, comme préconisée par UMM [CEFACT 2001]. L'approche est composée de 3 étapes :

1. **Le niveau Business**



2. **Le niveau logique**



3. **Le niveau physique**

Figure 5.1 : Etapes d'élaboration progressive d'une transaction. [Dubois 2002]

5.2 Le niveau Business : Représentation du domaine

Ce niveau se concentre sur la compréhension du business ; cette étape est indépendante de la solution qui sera envisagée pour répondre aux besoins. Ce niveau consiste, d'une part, à représenter le domaine par un diagramme de classe et, d'autre part, à représenter les « Business Process » par un diagramme de cas d'utilisation.

5.2.1 Représentation des « Business Process » par un diagramme de cas d'utilisation

Le diagramme de cas d'utilisation saisit un modèle des interdépendances entre plusieurs cas d'utilisation et des interactions avec un ou plusieurs acteurs. Ce diagramme a pour but principal de représenter les services que doit remplir le système, exprimés du point de vue de l'utilisateur.

Le diagramme montre la séquence d'actions qui peut être exécutée en interagissant avec les acteurs (parties prenantes.) Ce diagramme de cas d'utilisation doit être accompagné d'une description comprenant pour chaque cas d'utilisation :

- Le nom : nom apparaissant sur le graphique.
- L'objectif : brève description des objectifs remplis par le process.
- Les préconditions : conditions devant être réalisées en préalable.
- Les postconditions : conditions remplies à la fin.

A ce niveau, il n'y a pas de description du corps du cas d'utilisation (= implémentation) et il n'y a pas encore de concept de message.

5.2.2 Représentation du domaine par un diagramme de classes

Le but de cette représentation est de développer un modèle conceptuel représentant le monde réel afin de dégager une compréhension commune du domaine d'affaires. Ce modèle ne représente ni la base de données, ni les transactions du système informatique. Le vocabulaire utilisé est celui du domaine.

L'ensemble de l'information utilisée au cours de la transaction est structurée dans un diagramme de classe UML [OMG 2001]. Ce diagramme de classe général formalise le domaine d'affaires et permet de visualiser les informations véhiculées par l'ensemble des transactions. Les classes correspondent à des entités connues des experts métier (Business Object), des objets réels, auxquels il est possible d'associer des propriétés (ou attributs). Les relations qui lient ces objets dans le monde réel sont traduites en relations statiques UML.

La vision commune doit aborder un certain nombre de thèmes : objectifs, portée, délimitation du domaine d'affaires, contraintes, besoins respectifs des parties prenantes...

5.3 Le niveau Logique : Modélisation de la transaction

Ce niveau a pour but de modéliser de manière plus détaillée la structure de la transaction. Cette vue consiste, d'une part, à représenter chaque transaction par un diagramme d'activité et, d'autre part, à représenter chaque message par un diagramme de classe, ce diagramme de classe étant une sélection du domaine business défini précédemment.

5.3.1 Représentation d'une transaction par un diagramme d'activité

L'identification des acteurs d'une transaction permet de leur assigner les différentes activités à réaliser. Cette information dynamique de séquence d'activités est décrite dans un

diagramme d'activité [OMG 2001]. Ce diagramme permet d'une part, de spécifier les étapes de la transaction requises pour chaque participant et, d'autre part, de représenter les documents échangés au cours de la transaction. En effet, les participants exécutent tous des activités qui utilisent et/ou produisent des documents.

Le diagramme d'activité spécifie la séquence générale des activités et les messages échangés au cours de la transaction. Pour ce faire, il est utile de respecter les règles de modélisation suivantes :

- Chaque message est représenté par un « Object Flow ».
- Un message doit toujours être « entouré » de 2 activités (modélisation de l'envoi du message entre 2 acteurs).
- Le message se trouve dans la swimlane de l'acteur qui émet le message.

Dans le cadre du projet Efficient, les activités et objets du diagramme d'activité ont été stéréotypés pour plus de clarté. Les stéréotypes suivants ont été introduits :

- « *BusinessDocument* » : décrit les messages business.
- « *CommunicationActivity* » : désigne les activités qui produisent des messages business en sortie
- « *HumanChoice* » : permet à un participant d'évaluer un message qu'il reçoit. Il s'accompagne toujours d'un point de décision en sortie.
- « *Business Activity* » désigne toute autre activité.

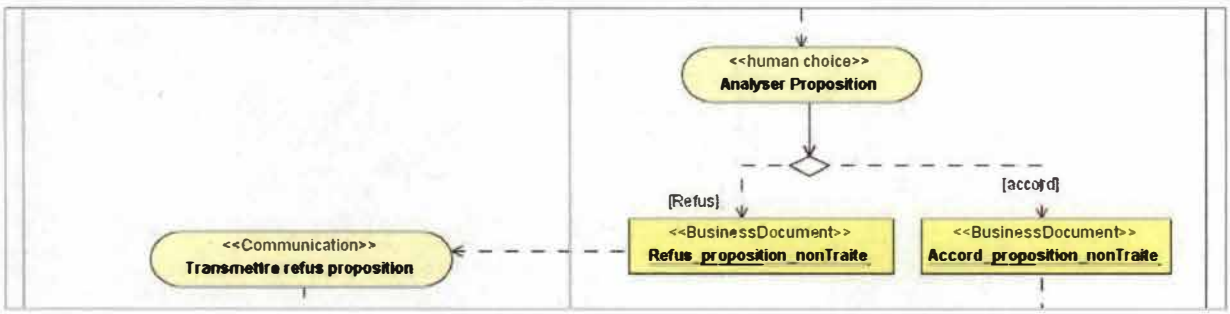


Figure 5.2 : Extrait d'un diagramme d'activité

La méthode employée par Efficient pour représenter les transactions diffère de la spécification ebXML [Eshuis 2003]. En effet, la spécification ebXML, définit la transaction à deux niveaux. Tout d'abord le niveau atomique, décrivant les transactions élémentaires exécutées entre deux partenaires ; ces transactions doivent respecter un protocole standard décrivant l'échange des messages. Ensuite, elle définit la transaction à un niveau général, en orchestrant les transactions atomiques dans un diagramme d'activités. Il faut préciser que les messages n'apparaissent pas dans le diagramme d'activités.

L'outil Efficient quant à lui, ne comporte qu'un niveau représentant la transaction par un diagramme d'activité qui spécifie la séquence générale des activités et les messages échangés. L'approche Efficient se concentre sur le cœur de la transaction, les échanges de messages.

A partir du diagramme d'activités (spécification dynamique UML), l'outil Efficient est capable de générer automatiquement une définition de la transaction.

5.3.2 Représentation d'un message par un diagramme de classes « restreint »

La structure de chacun des messages échangés (objet UML stéréotypé) « *BusinessDocument* » est décrite explicitement dans un diagramme de classes UML.

Le diagramme de classe représentant le message est modélisé de manière à correspondre à une vue du diagramme de classe du domaine. De cette manière, on peut garantir une certaine traçabilité entre le modèle du domaine d'affaire et le modèle du message.

Chaque classe de ce diagramme représente un élément de ce message. Une classe peut être en relation avec n'importe quelle autre classe.

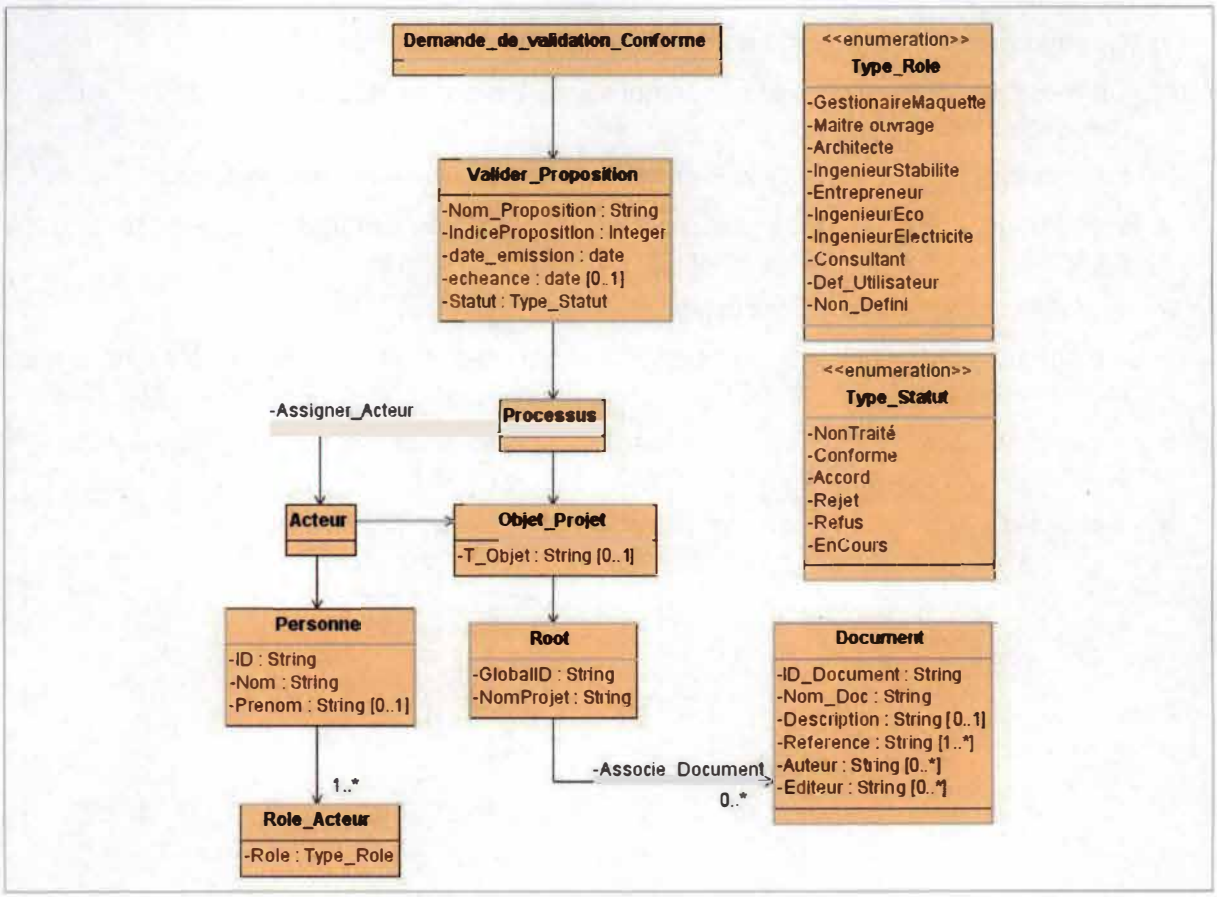


Figure 5.2 : Exemple d'un diagramme de classe représentant un message.

La construction de ce diagramme de classes « restreint » doit respecter un ensemble de règles afin que la traduction automatique du diagramme UML en XMLSchema soit possible.

Ces règles sont les suivantes :

- La racine de chaque diagramme de classe est une classe qui représente le message, elle ne possède ni d'attributs, ni opérations.
- Le diagramme de classe doit uniquement utiliser des relations directionnelles (associations, agrégations, compositions)
- Le diagramme de classe doit avoir une structure arborescente et ne doit pas comprendre de boucles !
- Les classes peuvent posséder des attributs mais pas des opérations.
- L'héritage peut être utilisé dans certaines limites (voir ci-après : pas de substitution)
- Certaines règles (XOR) et certains stéréotypes (énumération) peuvent être utilisés.

A partir du diagramme de classe restreint (spécification statique UML), l'outil Efficient est capable de générer automatiquement le XMLSchema du message.

5.4 Le niveau Physique : Transformation de UML vers XML

Ce niveau a pour but de produire la dynamique des transactions et les messages dans une syntaxe appropriée, c'est-à-dire le code qui y est associé [Ramel 2003].

5.4.1 Génération de la transaction

En ce qui concerne la dynamique de la transaction, le projet Efficient sur lequel s'appuie notre étude, permet aux participants distants d'animer une même procédure afin de la tester et de la valider.

L'outil d'animation développé dans le cadre de ce projet est composé d'une partie centrale « l'animateur » gérant la procédure de l'envoi des messages et de parties « clients » installées sur chacun des postes des participants, permettant de recevoir et d'écrire des messages. Un moteur de Workflow est au cœur de cet outil d'animation, qui exécute une procédure composée d'envois et de réceptions de messages entre des acteurs. La procédure de ce workflow est générée automatiquement au format XPDL depuis le diagramme d'activité qui décrit le flux des messages (voir 5.3.1). Le fichier XPDL généré est complété par des activités supplémentaires qui vérifient la structure et le contenu de chaque message réceptionné. Dans la suite de ce travail, nous modéliserons une transaction et nous en générerons la procédure au format XPDL (voir 8.1 et 9.1)

Nous allons, maintenant décrire brièvement le fonctionnement de cet animateur. Chaque envoi d'un message par un participant passe par l'animateur, qui vérifie que ce message est bien permis à un moment donné de la procédure et qu'il est correct syntaxiquement. Si tout est correct, l'animateur transmet ce message au participant à qui il était adressé. Sinon, il renvoie à l'émetteur accompagné d'un message d'erreur. Ensuite, l'animateur cherche les messages suivants qu'il est possible d'envoyer et prévient les différents clients.

Toutes les actions de l'animateur (validation, complétion, gestion des flux) ont pour but d'alléger la quantité de travail des différents participants qui peuvent de la sorte se concentrer sur le contenu de la transaction en cours.

Dans la partie qui suit, nous allons nous concentrer sur les messages intervenant dans la transaction.

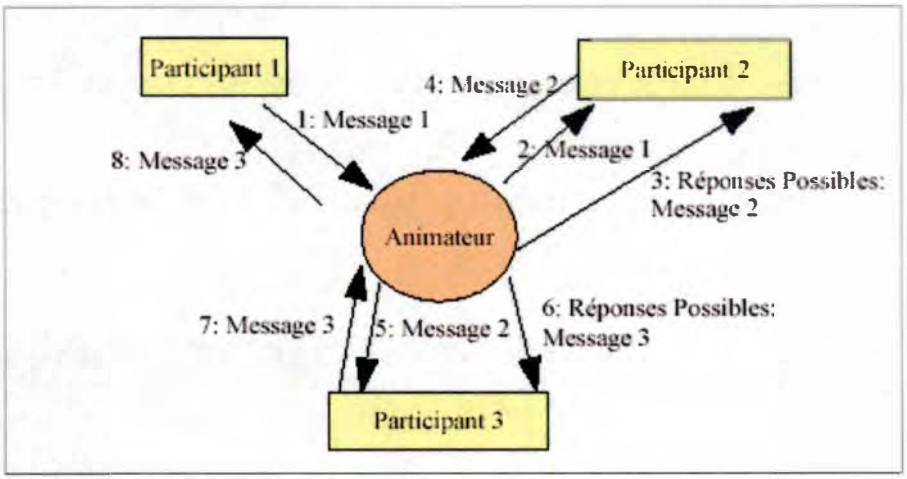


Figure 5.3 : Exemple d'échange de message lors d'une animation [Gregoire 2004]

5.4.2 Règles de construction des messages

La transformation du message UML en langage XML nécessite, lors de l'élaboration du diagramme de classe « message », de respecter les règles de construction suivantes :

- Sélectionner les classes et attributs nécessaires
- Organiser les classes sous la forme d'un arbre respectant la structure initiale en termes d'agrégation, d'héritage et de cardinalités. Ajouter les rôles et autres informations.

5.4.3 Règles de dérivation d'un Schéma XML

Cette partie n'a pas pour but de décrire l'algorithme de transformation complet mais de décrire les principales règles de transformation.

Principes généraux

- Les concepts business sont représentés par des éléments XML.
- Les méta-données (version, références, information de typage, etc.) sont représentées par des attributs XML.
- Les schémas UML sont traduits en SchémaXML. Le moteur de traduction que nous allons utiliser dans le cadre de cet exercice traduit en SchémaXML.
- Les classes sont représentées par des « complexType » et leur utilisation (classe racine, association et attribut) sont représentées par des éléments.
- Tous les éléments XML sont dérivés de la représentation UML du message. Ces éléments peuvent seulement être dérivés à partir des classes, des rôles ou des attributs UML.
- Pour chaque élément XML, il doit être possible de retrouver l'élément qui lui correspond dans le schéma UML.

Définition des tags

- Elément XML dérivé d'une classe : le nom de la classe.
- Elément XML dérivé d'un rôle : le nom du rôle.
- Elément XLM dérivé d'un attribut : le nom de l'attribut.

Définition des Types

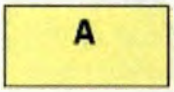
Les types des éléments XML correspondent aux types définis dans le modèle UML.

Règles de conversion

Ces règles de conversion on pour but de pouvoir transformer n'importe quel modèle UML en un SchémaXML et instance.

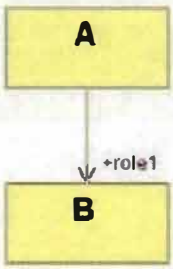
Classe

TRANSFORMATION	
UML	XML
Classe racine du message.	La classe est transformée en élément du nom de la classe contenant un complexType anonyme.
Autre classe	La classe est transformée en complexType du nom de la classe.

EXEMPLE	
UML	XMLSchema
	<xsd:element name="A" type="A"/> <xsd:complexType name="A"> ... </xsd:complexType>
	INSTANCE
	<A> </A>

Association directionnelle

TRANSFORMATION	
UML	XML
Association avec un rôle.	➢ Le rôle est transformé en complexType. ➢ L'association est transformée en élément du nom de rôle et utilisant ce complexType.
Association sans rôle	➢ Le rôle est transformé en complexType. ➢ L'association est transformée en élément du nom de la classe et utilisant ce complexType.

EXEMPLE	
UML	XMLSchema
	<xsd:element name="A" type="A"/> <xsd:complexType name="A"> <xsd:sequence> <xsd:element name="role1" type="B"/> </xsd:sequence> </xsd:complexType > <xsd:complexType name="B"> ...
	INSTANCE
	<A> <role1> data </role1> </A>

Composition simple : Idem association directionnelle.

Agrégation : Idem association directionnelle.

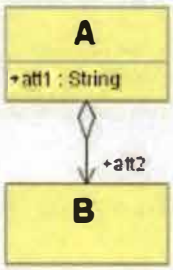
Attributs de classe

Une classe peut aussi contenir des attributs. L'attribut d'une classe est décrit par un nom et un type.

TRANSFORMATION	
UML	**XML**
Classe contenant des attributs.	Les attributs sont transformés en éléments. Ces éléments sont imbriqués dans l'élément qui représente la classe.

EXEMPLE

UML



XMLSchema

```
<xsd:complexType name="A">
    <xsd:sequence>
                <xsd:element name="att1" type="xsd:string"/>
                <xsd:element name="role2"          type="B"/>
    </xsd:sequence>
</xsd:complexType>
```

INSTANCE

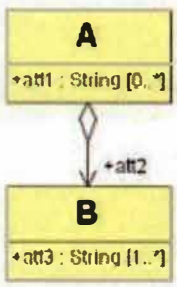
```
<A>
    <attribut1>data</attribut1>
    <role1>data</role1>
</A>
```

Attributs répétitifs de classe

La cardinalité représente le nombre d'occurrences d'attributs ou de rôles. Lorsque la cardinalité n'est pas indiquée de manière explicite, elle vaut implicitement 1. La cardinalité multiple, peut être décrite dans le XMLShema grâce à minOccurs et maxOccurs.

EXEMPLE

UML



XMLSchema

```
<xsd:element name="A" type="A"/>

<xsd:complexType name="A">
        <xsd:sequence>
                <xsd:element name="att1" type="xsd:string"
                minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="att2" type ="B"/>
        </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="B">
        <xsd:sequence>
          <xsd:element name="att3" type="xsd:string"
          minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>
```

INSTANCE

```
<A>
     <att1>data</att1>
    <att1>data</att1>

    <att2>
         <att3>data<att3>
         <att3>data<att3>
    <att2>
</A>
```

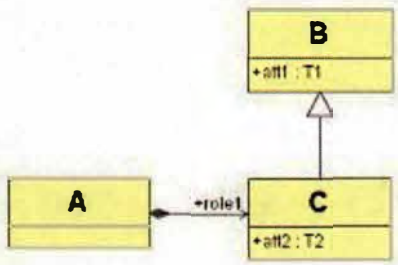
Cardinalité	Description	XMLSchema représentation
1	Exactement un	minOccurs="1" maxOccurs="1"
0..1	Optionnel	minOccurs="0" maxOccurs="1"
0..*	Non déterminé	minOccurs="0" maxOccurs="unbounded"
1..*	Au moins un	minOccurs="1" maxOccurs="unbounded"
1..N	De 1à N	minOccurs="1" maxOccurs="N"

Héritage simple

TRANSFORMATION	
UML	**XML**
L'héritage.	Les standards XML actuels ne proposent pas de mécanismes intégrés de représentation de l'héritage. Traduction de l'héritage par copie descendante pour les attributs, les références d'associations et les compositions.

EXEMPLE : Héritage simple

UML



XMLSchema

```
<xsd:complexType name="A">
      <xsd:sequence>
            <xsd:element name="role1" type="C"/>
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="C">
      <xsd:complexContent>
            <xsd:extension base="B" >
                  <xsd:sequence>
                        <xsd:element name="att2" type="T2"/>
                  </xsd:sequence>
            </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="B">
            <xsd:sequence>
                  <xsd:element name="att1" type="T1"/>
            </xsd:sequence>
</xsd:complexType>
```

INSTANCE

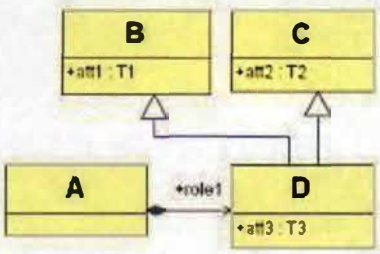
```
<A>
      <role1>
            <att1>data</att1>
            <att2>data</att2>
      </role1>
</A>
```

Héritage multiple

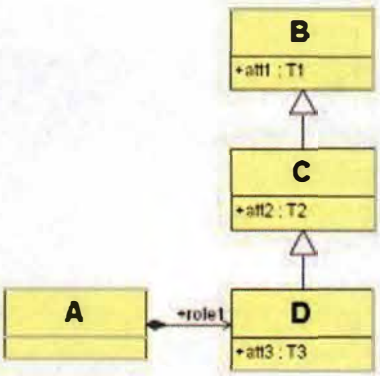
L'héritage multiple n'est pas possible en XMLSchema. Pour représenter un héritage multiple, il est nécessaire de le convertir en héritages simples.

EXEMPLE : Héritage multiple

UML



Transformation



XMLSchema

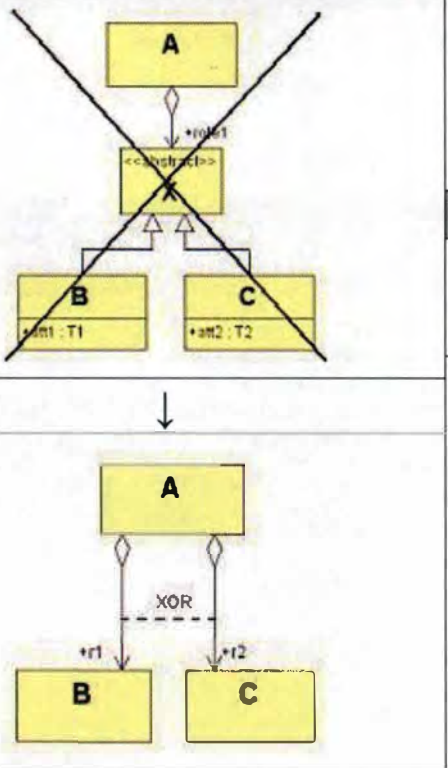
```
<xsd:complexType name="A">
      <xsd:sequence>
            <xsd:element name="role1" type="D"/>
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="D">
      <xsd:complexContent>
        <xsd:extension base="C" >
          <xsd:sequence>
            <xsd:element name="att3" type="T3"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="C">
      <xsd:complexContent>
            <xsd:extension base="B" >
                  <xsd:sequence>
                        <xsd:element name="att2" type="T2"/>
                  </xsd:sequence>
            </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="B">
      <xsd:sequence>
            <xsd:element name="att1" type="T1"/>
      </xsd:sequence>
</xsd:complexType>
```

INSTANCE

```
<A>
    <role1>
        <att1>data</att1>
        <att2>data</att2>
        <att3>data</att3>
    </role1>
</A>
```

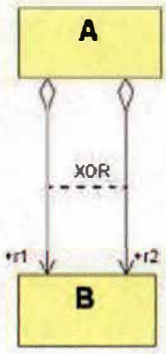
Union disjointe de classe

La substitution ne peut être traduite avec les complexType. Il est nécessaire de la transformer en contrainte d'exclusion.

EXEMPLE : Union disjointe de classe	
UML	**XMLSchema**
	<xsd:complexType name="A"> <xsd:choice> <xsd:element name="r1" type="B"/> <xsd:element name="r2" type="C"/> </xsd:choice> </xsd:complexType>
	INSTANCE
	<A> <r1>data</r1> </A>
	<A> <r2>data</r2> </A>

Exclusion de rôle (XOR)

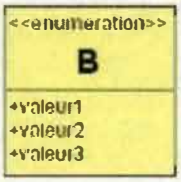
TRANSFORMATION	
UML	**XML**
L'exclusion de rôle.	Traduction de l'exclusion de rôle grâce au connecteur de choix « choice ».

EXEMPLE	
UML	**XMLSchema**
	<xsd:complexType name="A"> <xsd:choice> <xsd:element name="r1" type="B"/> <xsd:element name="r2" type="B"/> </xsd:choice> </xsd:complexType>
	INSTANCE
	<A> <r1> data</r1> </A>
	<A> <r2>data</r2> </A>

Enumération

Une valeur énumérée est une valeur que l'on est contraint de choisir dans une liste de valeurs prédéfinies.

TRANSFORMATION	
UML	XML
L'énumération est représentée dans le modèle comme un type avec la liste des valeurs possibles comme attribut.	L'élément XML contient la valeur choisie.

EXEMPLE	
UML	XMLSchema
	<xsd:simpleType name="B"> <xsd:restriction base="xsd:string"> <xsd:enumeration value="valeur1"/> <xsd:enumeration value="valeur2"/> <xsd:enumeration value="valeur3"/> </xsd:restriction> </xsd:simpleType>

PARTIE 3

Application au domaine de la Construction

CHAPITRE 6 : Le domaine de la construction.

6.1 Contexte général

Le secteur de la construction se distingue des autres secteurs industriels par un certain nombre de particularités liées à la nature du produit fini. En effet, chaque bâtiment est un ouvrage unique dont la réalisation nécessite les efforts concertés de nombreux acteurs exerçant autant de métiers. Il en résulte un certain nombre de caractéristiques propres à ce secteur :

- Des processus complexes avec notamment une succession de phases de nature très différente, impliquant de nombreux acteurs souvent géographiquement dispersés. La constitution de l'équipe évolue dynamiquement tout au long du cycle de vie des projets.
- Le nombre élevé des intervenants, la diversité de leur culture, l'hétérogénéité de leur structure. Une opération de construction associe aussi bien des grandes entreprises, que des PME, des artisans, des professions libérales, des services publics,...
- La recomposition perpétuelle des équipes d'un projet à un autre, qui gêne l'instauration de protocoles valables dans la durée et pour toutes les équipes.
- Au sein d'une même phase, les partenaires doivent échanger et produire de nombreux documents de manière interdépendante. Avec le développement des technologies de l'information et de la communication, les échanges se multiplient de plus en plus.
- La multiplicité des types d'informations (ex. plans, spécifications, rapports de situation, notes de services, etc.), de leur media (ex. papier, parole, maquette, échantillons, électronique, etc.), de leur support (ex. écrit, tabulaire, graphique), et de leur présentation (ex. différence dans la codification graphique des plans).
- L'utilisation de logiciels et de formats de données hétérogènes, propriétaires ou commerciaux.

La complexité du processus de construction tient surtout à la bonne communication des informations entre les personnes. Il convient donc d'être particulièrement attentif aux interactions entre les différents partenaires, car c'est à ce niveau que réside la source du plus grand nombre de non-conformités dans le secteur de la construction et ce, principalement en raison d'une absence de structure dans la communication.

6.2 Les acteurs

6.2.1 Les familles d'acteurs

Le collectif d'acteurs, dans sa taille et dans sa configuration, varie considérablement d'une opération à une autre. Cette variation n'est pas seulement liée à la taille du projet et à son prix, bien qu'elle dépende largement de ces deux facteurs, mais également à sa complexité fonctionnelle et technique ; ainsi, la conception d'une salle d'opérations dans un hôpital fait intervenir des dizaines de spécialistes et des personnes compétentes. À l'opposé, la conception d'une maison individuelle ne requiert dans la majorité des cas que peu d'acteurs : un client, un architecte et une entreprise.

Nous présentons un schéma général des acteurs et de leurs rôles.

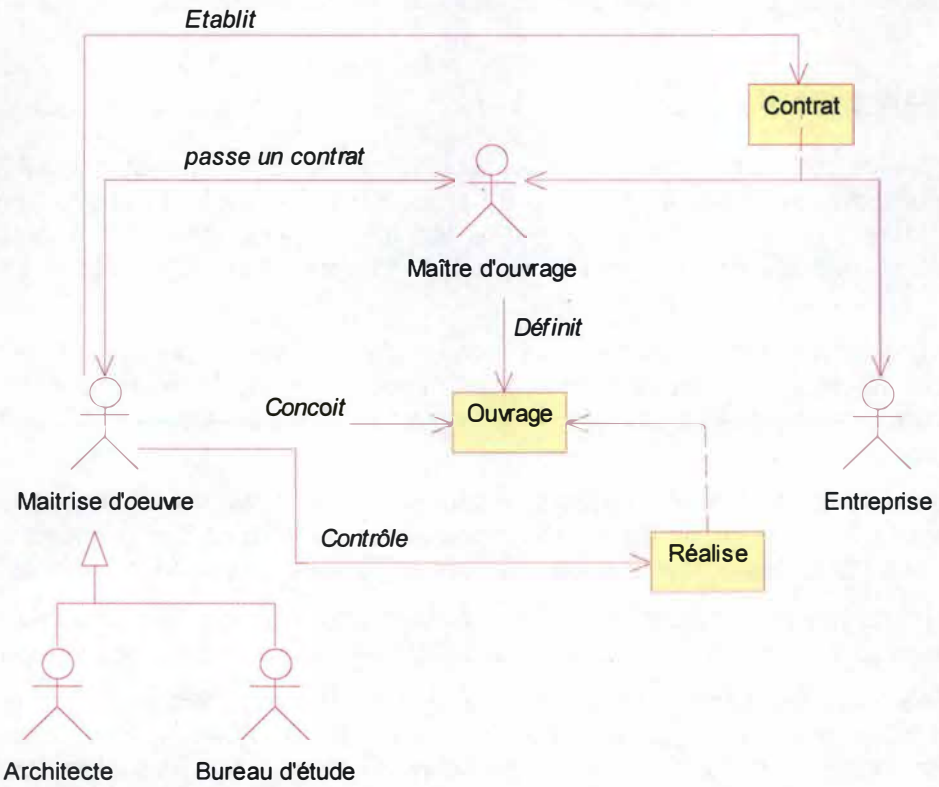


Figure 6.1 : schéma général des acteurs et de leurs rôles

Pour tout projet, on repère trois familles d'intervenant. Nous allons décrire ces différents intervenants et leurs missions respectives dans le processus de construction.

Le maître d'ouvrage

Le maître de l'ouvrage est représenté par la personne physique ou morale pour le compte de qui est construit l'ouvrage ; il est généralement le futur propriétaire. Celui-ci est à l'origine de la décision et dialogue avec l'architecte pour en asseoir sa faisabilité.

Sa mission est, entre autres, de trouver le terrain ou l'espace nécessaire à la construction, de fournir les moyens financiers nécessaires à la réalisation de l'ouvrage et à la rémunération des intervenants, ou encore de préciser et d'affiner l'intention initiale en élaborant un programme détaillé et précis de l'opération.

La mission de maîtrise d'ouvrage peut enfin faire l'objet d'une délégation. Lorsque le maître d'ouvrage n'a pas l'intention, les moyens humains ou la compétence nécessaire pour assurer les tâches opérationnelles qu'implique la fonction, il peut confier ces tâches à un maître d'ouvrage délégué. Celui-ci deviendra alors l'interlocuteur privilégié, voire unique, du maître d'oeuvre, des techniciens et des entrepreneurs.

Le maître de l'ouvrage définit les règles spécifiques de passation et d'exécution des travaux.

La maîtrise d'œuvre

La maîtrise d'œuvre est la personne physique ou morale qui conçoit l'ouvrage pour le compte du maître d'ouvrage et qui assure la responsabilité globale de la qualité architecturale et technique, du délai et des coûts. Les principaux maîtres d'œuvre sont les architectes et les bureaux d'études techniques. Dans la suite de ce travail, nous ferons la distinction entre ces deux types d'acteurs car leurs activités respectives nécessitent des échanges fréquents qui sont souvent source d'erreurs et de conflits.

L'architecte

Le bureau d'architecture a pour mission d'apporter une réponse architecturale, technique et économique au programme défini par le maître de l'ouvrage.

Le bureau d'architecture a un triple rôle à jouer :

- Conception du projet.
- Conseils auprès du maître d'ouvrage.
- Contrôle de la réalisation.

De manière plus précise, la mission de l'architecte comprend les fonctions suivantes : d'une part, s'assurer de la viabilité de l'opération, puis concevoir, représenter, décrire et évaluer l'ouvrage. En outre, il lui faut rechercher et proposer les moyens de produire l'ouvrage, diriger, contrôler la conformité, la qualité et le coût des travaux.

L'architecte traite à la fois avec la maîtrise d'ouvrage et avec tous les acteurs de la construction. Il est le trait d'union entre les deux autres parties. Il peut être assisté d'architectes associés, de dessinateurs, etc.

Le bureau d'architecture intervient dans toutes les phases du cycle de vie de l'ouvrage. Ces étapes seront décrites dans la suite de ce chapitre.

Le bureau d'étude

Il est chargé d'une mission dite d'ingénierie, vérifie la faisabilité du projet sur divers points techniques. En effet, la construction contemporaine fait appel à des techniques de plus en plus complexes dont le niveau de technicité ne peut être maîtrisé que par des spécialistes. Il intervient sur demande du maître d'ouvrage, ou du maître d'oeuvre qu'il assistera sur des points particuliers en établissant avec lui soit une relation de co-traitance, soit une relation de sous-traitance. Le bureau d'étude technique peut également intervenir sur demande d'une des entreprises prenant part à l'opération, quand celle-ci ne possède pas en son sein les spécialistes capables de réaliser certaines études techniques.

L'entreprise

L'équipe de réalisation est celle que le maître de l'ouvrage a chargée, par contrat, de la responsabilité de l'ensemble de la construction d'un ouvrage.

Sa fonction principale est de fournir une prestation composée d'éléments très imbriqués et indissociables les uns des autres, comme une main-d'oeuvre qualifiée et encadrée, le matériel spécialisé nécessaire à l'édification de l'ouvrage, les matériaux manufacturés ou industrialisés, et un savoir-faire spécifique. Ce savoir-faire entraîne pour l'entrepreneur une obligation de conseil tant à l'égard du maître d'ouvrage que du maître d'oeuvre. C'est également en fonction de ce savoir-faire que l'entrepreneur proposera, en concertation avec le maître d'oeuvre, un procédé et un enchaînement des opérations propres à chaque type d'ouvrage.

Autres

Ce sont là des acteurs décisionnaires, directement impliqués dans la réalisation et l'aspect de l'ouvrage. À ceux-là s'ajoutent des acteurs consultatifs, assureurs, fournisseurs, bureaux de contrôle et de contrôle technique chargés, par exemple, de l'information sur des problèmes de sécurité et de règlement à chacune des étapes du projet.

Dans le cadre de ce travail, nous ne tiendrons pas compte de ces acteurs car ils n'ont pas un rôle majeur dans le processus de construction.

6.2.2 Relations entre acteurs

Plus globalement, la relation entre acteurs est marquée par un partenariat de nature inconstant ; l'appel d'offres par projet est le principal mode d'attribution des marchés. Cette mise en concurrence entraîne une forte improbabilité de retrouver la même équipe sur des projets différents. Enfin, pour un projet donné, la multiplicité des acteurs et des centres de décision est un autre facteur de complexité des rapports de coopération en effet :

- la séparation formelle entre clients, concepteurs et réalisateurs rend le centre de gravité décisionnel instable ;
- le décideur n'est pas toujours le client final, c'est le cas, par exemple, lorsqu'un maître d'ouvrage délégué assiste un maître d'ouvrage non-professionnel, ce qui entraîne une démultiplication des pouvoirs ;
- il existe un nombre important de prescripteurs.

6.2.3 Indépendance des organisations

Nous voulons insister sur une caractéristique fondamentale du secteur du bâtiment. Il s'agit de l'indépendance des organisations. Pendant le projet, cette indépendance se manifeste par le fait qu'aucune organisation, aucun acteur n'a les moyens d'imposer son mode d'organisation du travail à ses partenaires. Cette indépendance ne signifie pas autonomie et l'ensemble des organisations doit s'entendre sur les interfaces, documents mais aussi modes de communication, rôles et responsabilités.

6.3 Les documents

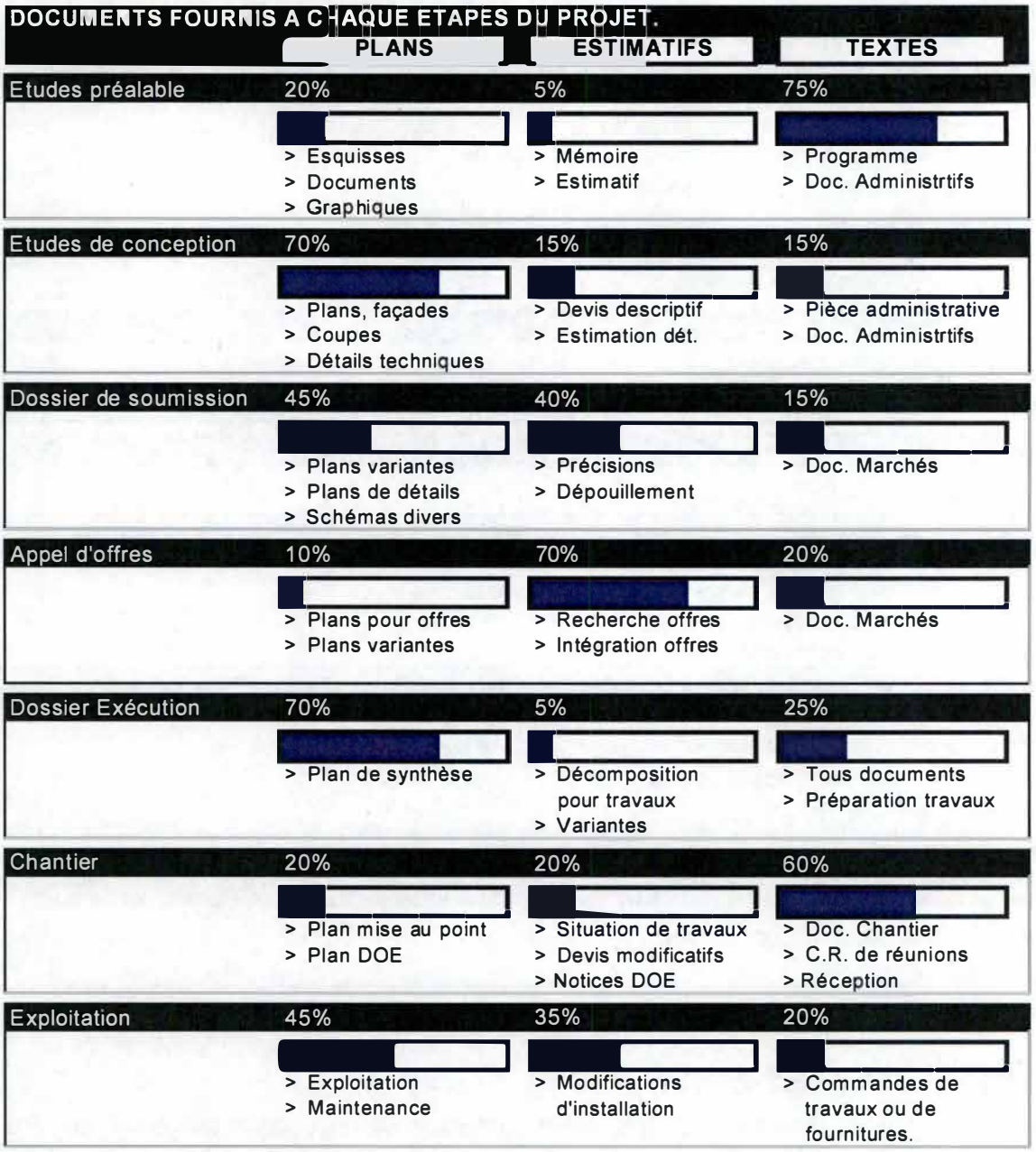
Le projet d'architecture est le lieu d'un échange d'informations diverses. Les documents représentent un type d'objet informationnel particulier, mais l'information produite, utilisée, transformée durant un projet ne se limite pas aux documents. Une conversation téléphonique, le caractère urgent d'un envoi, ou l'expérience passée sur d'autres projets, sont des éléments d'information qui ont leur importance et que doivent prendre en compte les acteurs [Malcurat 2001].

Les documents constituent une ponctuation dans le temps ; ils résultent d'une matérialisation momentanée des interactions. Mais plus que cela, ils transforment l'action collective : par exemple, le document véhicule des demandes, des approbations ou des refus, infléchit des choix, repose la relation entre son émetteur et son destinataire, etc.

6.3.1 Les types de documents

Le nombre et la nature des documents échangés au cours d'un projet sont élevés. Leur répartition relative dans les différentes phases est variable. De plus, la formalisation des documents est loin d'être partagée et il arrive souvent qu'un document soit ressaisi, informatiquement ou non, à l'issue d'un échange pour correspondre au formalisme d'une organisation donnée. On retrouve ici une conséquence de l'indépendance méthodologique des organisations.

Les documents produits sont variables en type et en proportions. Le document, tel que nous l'entendons ici, n'est pas limité à une pièce graphique ou un fichier de dessin, il concerne également les pièces de marchés, les contrats, le cahier des charges,...Le tableau suivant décrit pour chaque étape du projet, les documents produits répartis en trois types.



Il est possible d'ordonner les documents en fonction de la diffusion dont ils font l'objet :

- les documents collectifs : tous les documents qui véhiculent le savoir et l'expérience collective du domaine (livres, manuels, traités, revues, catalogues, bases de données, bases d'images, normes, textes juridiques, règlements divers, etc.)
- les documents d'échange : ce sont les documents véhiculés entre acteurs lors d'une opération de conception / construction. Ils servent à échanger de l'information et à permettre le travail coopératif. Ce sont, par exemple, les pièces du marché et les documents graphiques.
- les documents d'étude : ce sont les documents propres à un acteur et nécessaires à l'accomplissement d'une tâche ou d'une mission. Ces documents de travail n'ont pas à priori, vocation à être échangés. Ce peut être les croquis, les notes sur un cahier...

L'étude qui suit porte sur les échanges dans le cadre de la conception et la réalisation d'un ouvrage ; nous allons donc nous intéresser principalement aux documents d'échange.

6.3.2 Les flux des échanges entre les différents acteurs

Flux 1 : Maître d'ouvrage / Architecte
Programme Esquisses, avant-projet sommaire, avant -projet détaillé Dossier de consultation des entreprises Décompte des travaux Dossier d'exécution des ouvrages Signatures des marchés, paiements

Flux 2 : Maître d'ouvrage / Bureau d'études
Assistance au maître d'ouvrage Etudes de faisabilité

Flux 3 : Maître d'ouvrage / Entreprise
Signatures des marchés, paiements

Flux 4 : Architecte / Bureau d'études
Esquisses, avant-projet sommaire, avant -projet détaillé Dossier de consultation des entreprises Plan détaillé d'exécution des ouvrages

Flux 5 : Architecte / Entreprise
Dossier de consultation des entreprises Plan détaillé d'exécution des ouvrages Etablissement des marchés Suivi de chantier

Flux 6 : Bureau d'études / Entreprise
Plan détaillé d'exécution des ouvrages Suivi de chantier

6.3.3 Mode d'échange de documents

Le mode d'échange actuel de documents est la conséquence du support utilisé de manière majoritaire, le papier. Dans la pratique actuelle, les documents papier sont échangés et ne sont que rarement partagés.

Outre la matérialité du support papier qui explique le mode de l'échange, celui-ci se justifie également pour une autre raison : chaque acteur a la responsabilité des documents qu'il a en charge de produire. Un plan d'architecte, par exemple, reste la propriété de l'architecte. Si un autre acteur veut utiliser ce plan pour continuer son propre travail, il ne peut que travailler à partir d'une copie de ce plan ; en aucun cas, il ne dessine à nouveau sur le plan qu'a en main son propriétaire. Les responsabilités de chacun sont ainsi clairement distinctes et garanties.

L'échange consiste à transmettre un document à l'occasion d'une rencontre entre acteurs ou par courrier postal. Le partage est nettement moins répandu parce qu'il est plus difficile à mettre en œuvre.

L'échange permet le contrôle de la diffusion du document ; l'expéditeur sait à qui et à quel moment il a communiqué un document. Ce mode comporte deux avantages :

- Le destinataire ne peut ignorer le document directement transmis ; en cas de litige, l'expéditeur n'aura pas à se reprocher l'absence de prise en compte du document par ses partenaires.
- L'échange permet de sélectionner les destinataires pour chaque étape de l'évolution du document et une gestion différenciée des relations d'échanges en fonction de la plus ou moins grande proximité conceptuelle des membres de l'équipe. Il est évident, par exemple, que le flux de documents est plus intense entre un bureau d'architecture et un bureau d'études, qu'entre ce même bureau d'architecture et le maître d'ouvrage qui n'est intéressé que par des documents finalisés des grandes phases du projet.

6.4 Moyens de coordination

La coordination est l'activité consistant à gérer les interdépendances entre activités [Malcurat 2001].

Le secteur de la construction utilise deux moyens privilégiés pour coordonner ses acteurs :

- les réunions de chantier, ou plus généralement les rencontres directes entre acteurs
- la planification par la mise en œuvre de techniques graphiques.

6.4.1 La réunion

Une réunion rassemble sur une durée relativement courte les acteurs essentiels aux activités en cours pour valider des choix et des décisions, puis décider de la suite de l'exécution de l'opération.

L'intérêt des réunions pour les acteurs vient justement du contact direct pour prendre rapidement des décisions. Leur importance sociale est telle qu'il serait difficile d'imaginer leur suppression.

On repère au moins quatre fonctions assumées par les réunions :

- une évaluation du travail effectué jusqu'alors, et depuis la dernière réunion ;
- une prise en compte d'éléments nouveaux susceptibles d'orienter l'exécution ;
- une actualisation des objectifs des moyens et des méthodes ;
- l'écriture et la distribution d'un rapport de réunion aux acteurs concernés.

6.4.2 La planification

Dans un sens de contrôle de la coordination, la planification des tâches utilise deux techniques graphiques courantes que sont le diagramme de Gantt et le réseau Pert. Essentielles pendant les phases de chantier, elles sont néanmoins couramment employées sur la totalité de la durée d'une opération. La finalité de ce deux techniques est la même : coordonner les tâches afin de maîtriser les délais et les coûts.

Le réseau PERT montre, davantage que le diagramme de GANTT, la relation de dépendances entre les activités ; c'est en particulier cette caractéristique qui lui donne la capacité de déterminer le chemin critique qui conditionne la durée maximale du projet. Le diagramme de GANTT permet, quant à lui, une bonne appréhension des durées relatives des activités et des recouvrements entre activités.

Dans les deux cas, ces méthodes permettent de :

- définir les activités constituant le projet ;
- organiser les activités dans le temps (ordonnancement) ;
- évaluer les dépendances entre activités ;
- évaluer l'effort nécessaire pour chaque activité ;
- affecter les ressources (personnel, matériel) aux activités.

6.5 Le processus global et les processus spécifiques de la construction

6.5.1 Processus global

Afin de satisfaire le client, il est nécessaire de maîtriser le processus global de construction, depuis la programmation jusqu'à la réception de l'ouvrage, voire jusqu'à la gestion et la démolition de ce dernier. [Gosselin 2000]

Le processus global de toute opération de construction peut-être décomposé en cinq grandes phases :

- La programmation,
- La conception,
- La réalisation (chantier),
- L'exploitation et la maintenance
- La démolition.

Il convient de noter que la phase de conception et la phase de réalisation nécessitent une attention particulière concernant l'aspect collaboratif. En effet, le collectif d'acteurs convoqués lors de la phase de conception et de la phase de réalisation n'est pas le même. L'apparition des entreprises au moment du chantier augmente considérablement le nombre des intervenants. Ceci a une conséquence importante sur l'interdépendance des acteurs et les exigences en matière de coordination. Les diagrammes de Gantt ou de Pert sont alors massivement exploités ; les réunions de synthèse, les réunions de chantier jouent pleinement leur rôle pour synchroniser les interventions. On voit donc se succéder deux moments très différents par la nature des opérations et des procédures engagées et par la structure d'acteurs déployée.

Nous allons maintenant décrire plus en détail les différentes phases d'une opération.

6.5.2 Processus spécifiques

Programmation

La programmation englobe l'ensemble des réflexions, des analyses, des idées, des enquêtes et de la documentation, depuis la naissance de l'intention de construire jusqu'à la définition du programme architectural et la passation du ou des contrats de maîtrise d'œuvre. Elle relève de la responsabilité directe du maître de l'ouvrage.

Suivant la nature et l'importance de l'édifice à construire, le programme sera réduit ou ample, aisé ou complexe à établir. Cette programmation est capitale pour la qualité globale de la construction.

Elle vise essentiellement la satisfaction du client et des utilisateurs par rapport au bien construit. Les exigences retenues lors d'un projet déterminé devraient résulter d'une confrontation entre, d'une part, les exigences d'utilisation de l'ouvrage à construire découlant d'une analyse des besoins et des contraintes utilisateurs-activités-milieu et, d'autre part, les exigences et contraintes administratives, économiques et financières du maître d'ouvrage.

La programmation se place en amont de tout le processus de l'acte de construire.

Conception architecturale et technique

La maîtrise de la conception architecturale et/ou technique nécessite une grande rigueur et un suivi tout au long du processus de conception. Elle est de la responsabilité de l'auteur de projet ou du bureau d'études.

La vérification par l'architecte ou l'ingénieur se fait notamment lors des réunions avec le maître d'ouvrage. Une trace des décisions prises est indispensable.

Des points critiques existent et nécessitent de la part du concepteur une attention particulière, par exemple, en interne, lors du transfert du dossier vers les collaborateurs pour l'élaboration du cahier des charges, des plans de détails, ou encore en externe, lors de la diffusion des plans,...

Cette étape peut-être décomposée en missions :

- Esquisses : Proposition pour une ou plusieurs solutions d'ensemble traduisant les éléments majeurs du programme ; indication des délais de réalisation ; vérification de la faisabilité de l'opération.
- Avant projet : Elaboration d'un document graphique présentant clairement la conception de la future construction (distribution des locaux, façades, volumes) en fonction des contraintes administratives, du programme, du budget disponible et des matériaux.
- Le dossier de demande de permis de bâtir : Réalisation des plans comprenant une indication précise des volumes, du choix des matériaux et de l'implantation. Préparation de tous les documents requis pour l'administration communale.
- Le projet de construction : Conception des plans d'exécution pour l'entrepreneur, ces plans reprennent en détail tous les éléments techniques de la construction. Préparation du dossier de demande des devis aux entrepreneurs, ce dossier comprend les plans d'exécution, les cahiers des charges et les métrés. Les cahiers des charges reprennent les clauses administratives, le cahier général des charges et le cahier spécial de charges.
- Le choix de l'entrepreneur : Demande de prix aux entreprises et conseils au maître de l'ouvrage dans l'étude des différentes offres de prix des entrepreneurs et au niveau du contrat d'entreprise.

Exécution

En général, en dehors des petites interventions, on élabore un planning pour chaque nouveau chantier, afin de préciser au mieux les différentes activités et leurs relations. Ce planning permet d'identifier tous les sous-processus d'exécution et leurs interactions afin de coordonner les actions des différents intervenants dans le temps et l'espace.

L'entreprise doit être attentive aux interactions entre les sous-processus (implantation, terrassement, égouttage, fondations, sous-structure, maçonneries de cave, dalle de rez-de-chaussée,...). Pour chaque sous-processus, l'entreprise a l'obligation :

- d'identifier les exigences du client, précisées généralement sur les plans et/ou dans le cahier des charges
- de mettre les moyens humains compétents et du matériel en état de fonctionnement à la disposition du chantier ou de faire appel à des sous-traitants spécialisés
- d'exécuter les travaux en mettant en évidence les tâches de chacun et les contrôles à faire (éventuellement à l'aide d'une liste de contrôles internes imposés)
- de vérifier si la phase est bien terminée et est susceptible de satisfaire aux exigences du client (plans, cahier des charges...)
- si nécessaire, d'apporter des améliorations.

La vérification de la conformité des travaux aux exigences du client doit au moins être consignée dans le rapport journalier et les rapports de visites (hebdomadaires). Les réunions de chantier fournissent un jugement sur l'évolution des travaux et sur la satisfaction des exigences du client au fur et à mesure de l'avancement des opérations.

Le processus de réception provisoire et définitive permet d'entériner la satisfaction du client par rapport à l'exécution proprement dite. En cas de malfaçons et/ou défauts apparents, l'entrepreneur devra rectifier.

Exploitation et maintenance

La phase d'exploitation / maintenance est la plus longue et aussi la plus coûteuse. Elle commence dès l'achèvement des travaux et fait appel à de nombreuses disciplines : aménagement intérieur, nettoyage, gestion des consommation d'énergie, sécurité, gestion technique et financière, etc.

Les plans «as built», le carnet d'entretien ainsi que le dossier sécurité d'utilisation contribuent à une saine gestion de l'ouvrage. Ils font partie intégrante du processus global de construction.

Dans le cadre de cette phase, il serait intéressant de définir une méthode globale d'évaluation de la satisfaction du client et d'en faire bénéficier tous les intervenants. En effet, il serait judicieux qu'un retour d'informations puisse se faire en complément des plaintes éventuelles émises par les clients dans le cadre de la garantie décennale.

Démolition

La démolition doit être prise en compte dès la phase de conception pour éviter les déboires ultérieurs (tri sélectif difficile, voire impossible). Le concept de construction durable oblige les maîtres d'ouvrage et les concepteurs à mieux prévoir l'adaptation du bien construit à l'évolution des besoins des utilisateurs, tout en respectant l'environnement.

6.5.3 Interactions entre les partenaires

La complexité du processus global de la construction tient non seulement à l'unicité de l'ouvrage à construire, mais aussi et surtout à la bonne communication des informations entre les personnes. Lors de la réalisation d'une construction, plusieurs partenaires interviennent directement au même moment ou à des moments différents. Il convient, d'une part, de bien définir le rôle de chacun dans le projet et, d'autre part, de vérifier régulièrement si les exigences du client sont rencontrées.

Une attention particulière doit systématiquement être accordée aux interactions entre les différents partenaires, car c'est à ce niveau que réside la source du plus grand nombre de non-conformités dans le secteur de la construction et ce, principalement en raison d'une absence de structure dans la communication.

Les fonctions fondamentales à assurer sont donc, la communication entre partenaires, la coopération, la coordination des tâches et l'accès à l'information.

6.6 Informatisation des professionnels

6.6.1 Développement de l'informatisation

La transposition des métiers

Les années 1980 se caractérisent par une diffusion importante de « l'informatique métier ». Dans cette première période, l'informatique s'est contentée de transposer les pratiques de chacun des intervenants. Cette spécialisation des logiciels a renforcé un cloisonnement des métiers ; on peut parler d'une informatisation autonome et indépendante. Elle a augmenté fortement les productivités sectorielles sans favoriser les pratiques d'échanges de l'information et les rapports de collaboration entre acteurs. Il s'agit d'une amélioration locale et non globale.

L'apparition des réseaux

Les années 1990 ont vu l'émergence de l'informatique en réseaux. Ceux-ci introduisent un tout autre changement. En augmentant l'accessibilité en temps réel à une multitude d'informations, en réduisant encore les temps de la conception, en rendant possible une synchronisation forte des décisions, les réseaux induisent le passage d'une ingénierie relativement isolée à une ingénierie concourante.

Les échanges inter métiers

Très rapidement, les différents acteurs se sont rendus compte que pour pouvoir échanger des informations, il était nécessaire d'établir un certain nombre de normes concernant la structuration des données relatives au bâtiment.

Les premières réflexions ont porté sur la structuration des données en « couches ». Ces conventions organisent le dessin en un certain nombre de calques distincts, chaque calque étant exclusivement porteur d'une information particulière ou d'un type de composant du bâtiment. Ce niveau de structuration est sémantiquement très pauvre, et ignore l'organisation interne des entités sur un calque.

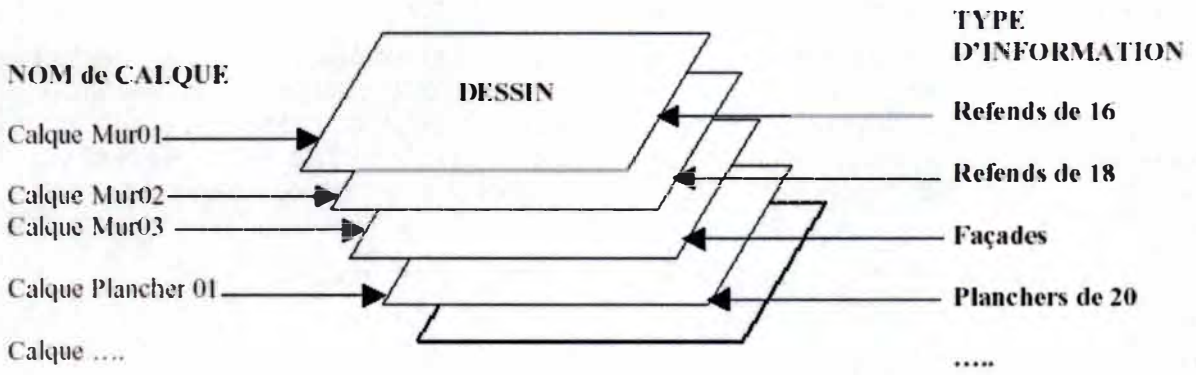


Figure 6.2 : Représentation de la structuration des calques en couches [Billon 1999]

En ce qui concerne les formats d'échange, la définition d'un standard d'échange (DXF) a révélé les limites de compatibilités. En effet, la solution adoptée a conduit au contraire de l'objectif souhaité c'est-à-dire à imposer le même logiciel de dessin aux différents intervenants, ce qui bien sûr résolvait le problème technique, mais constituait un recul aussi bien au plan des avancées technologiques qu'au plan déontologique.

Les armoires à plans

Des systèmes de gestion de projet ont exploré un fonctionnement impliquant des échanges inter-métiers, limités aux « calques » de dessins. C'est le cas des « armoires à plans électroniques » utilisées pour les opérations de synthèse du dossier d'exécution. En fait, il s'agissait surtout d'exploiter au mieux la superposition des plans graphiques d'origine multiple. Le principe est basé sur une convention de structuration de l'information graphique en couches, sur un standard d'échange de dessin, et des règles d'accès aux documents dont la mise à jour et la validation sont contrôlées dans le temps.

On assiste aujourd'hui à l'éclosion d'une offre de gestion de projet en ligne, qui constitue en quelque sorte la version Internet des armoires à plans. Dans ce cas, le poste client est un simple terminal pourvu d'une connexion Internet. Depuis son navigateur, l'abonné au service de gestion peut accéder à diverses fonctions telles que :

- Publier des documents à l'attention des autres utilisateurs du service
- Consulter les documents qui lui ont été transmis
- ...

Ces armoires à plans et ces systèmes de gestion ne peuvent donc répondre que très partiellement aux réels besoins, car la conduite des études de projets est une activité collective qui concerne la totalité des informations techniques et non seulement des plans.

L'ingénierie concourante

Le partage des données techniques et graphiques nécessite la définition d'un standard (ou d'une norme) d'échange qui permette, en tout indépendance, de partager toute l'information du projet. C'est-à-dire non seulement le dessin, mais aussi l'information technique associée, de telle sorte que chaque logiciel métier puisse extraire les données du projet dont il a besoin, automatiquement, et éventuellement en retour enrichir l'information initiale.

C'est ce qu'on appelle « l'ingénierie concourante », qui prend comme sujet d'intervention une représentation numérique, modélisée, centralisée, du bâtiment en projet ou construit.

Si chaque partenaire d'une opération de construction est informatisé, atteindre une nouvelle étape d'économie et de qualité suppose que les différents systèmes informatiques en présence, hétérogènes par nature, puissent échanger leurs données et leurs résultats. Et d'une façon encore plus ambitieuse, puissent être capable de se partager simultanément l'accès à une représentation unique du bâtiment.

C'est le nouveau défi à relever par le secteur de la construction.

En vue de ce nouveau défi, différents éditeurs se sont associés (IAI) afin de spécifier un langage commun propre au secteur du bâtiment, indispensable à l'amélioration de l'interopérabilité des logiciels et à l'enrichissement sémantique du modèle. Cette initiative a donné naissance à un modèle orienté objet qui a pour nom « modèle IFC ». Aujourd'hui, ce modèle fait l'objet d'un consensus au niveau international et a été adopté comme norme au sein de l'ISO.

Dans la suite de ce travail, une rubrique sera consacrée au modèle IFC.

6.6.2 Limites à l'informatisation

On relève deux points, qui ne sont pas à proprement parler des problèmes, mais qui ont limité jusqu'aujourd'hui l'usage de l'informatique et des documents électroniques :

- La prééminence du support papier et plus généralement l'attachement à la matérialité des supports. Il a l'avantage de la souplesse d'utilisation (annotation des plans, visualisation aisée) et une valeur contractuelle car ce sont des documents papier qui sont exigés et non leurs versions électroniques.
- La modélisation graphique est presque toujours effectuée en mode bidimensionnel (2D), le mode tridimensionnel n'est que rarement exigé. Les échanges portent donc sur des représentations traditionnelles (plans, coupes, façades) plutôt que sur des maquettes « virtuelles » en 3D.

6.6.3 Influences de l'informatisation

L'informatisation affecte deux dimensions de la pratique des professionnels :

- Le « métier » : impact sur les modes de conception, de réalisation, de maintenance.
- La coopération : impact sur les modes de coopération entre les acteurs.

6.7 Le modèle IFC

6.7.1 Le modèle IFC, un standard

En 1995, un groupe d'éditeurs de logiciels s'est réuni pour former l'IAI (International Alliance for Interoperability). C'est cet organisme de standardisation qui est à l'origine du modèle IFC [Ferries 2003].

Aujourd'hui l'IAI, compte près de 600 membres provenant d'une vingtaine de pays et regroupant des représentants de tous les métiers de la construction : utilisateurs, éditeurs de logiciels, organismes institutionnels, établissements d'enseignement, fabricants de composants bâtiment,...

L'IAI a adopté plusieurs résultats de la norme STEP[3] (norme ISO 10303) dont le langage EXPRESS avec lequel sont décrites les différentes versions des IFC, et le format neutre pour l'échange de fichiers (partie 21).

Le modèle IFC couvre un spectre très large et est développé de façon progressive et incrémentale. La version la plus récente (2X2) qui contient plus de 600 classes, a été publiée en 2003.

Le modèle fait l'objet d'un consensus au niveau international. La plateforme IFC 2X a été homologuée par l'ISO en novembre 2002 sous la référence ISO/PAS 16739. Cette plate-forme est composée des entités les plus stables du modèle IFC 2X ; c'est donc un sous-ensemble du modèle IFC 2X.

6.7.2 Principe du modèle IFC

Le principe du modèle IFC est de fournir un standard d'échange neutre entre applications afin de permettre et de favoriser l'interopérabilité. Il s'agit d'un standard de description des éléments de construction destiné à répondre aux besoins d'échange et de partage d'informations entre tous les intervenants pour toutes les phases du cycle de vie d'une construction.

Les IFC sont des classes d'objets spécifiques à la construction. La base du format d'échange IFC est une définition commune des objets du bâtiment, répartie dans un grand nombre de classes, d'attributs et de propriétés. Un échange au format IFC est un échange d'objet du projet. Le nom et le type d'information de chacun des paramètres qui définissent l'objet sont décrits par le format.

Ce modèle de description et d'échange de données relatives au bâtiment présente, par rapport aux modèles qui l'ont précédé, une granularité plus fine et une plus grande richesse. Le modèle IFC substitue à une représentation en couche d'éléments graphiques, un véritable modèle orienté objet de tous les composants du projet (pas seulement du bâtiment) et de leurs caractéristiques (y compris mais pas uniquement géométriques). Le fichier IFC est donc sémantiquement beaucoup plus riche qu'un plan DXF par exemple. Les IFC représentent le bâtiment non par des dessins, mais comme une organisation de composants et d'ouvrages. L'échange ne porte pas sur les représentations, mais sur les caractéristiques des objets.

La modélisation du projet devient un système d'information dont chaque partenaire ne retient que la vue métier dont il a l'usage.

[3] STEP signifie STandard for Exchange of Product data. L'objectif de cette norme internationale (ISO 10303) est de décrire un produit tout au long de son cycle de vie, et indépendamment de tout système informatique.

6.7.3 Architecture du modèle

Le modèle IFC est découpé, structuré en quatre niveaux :

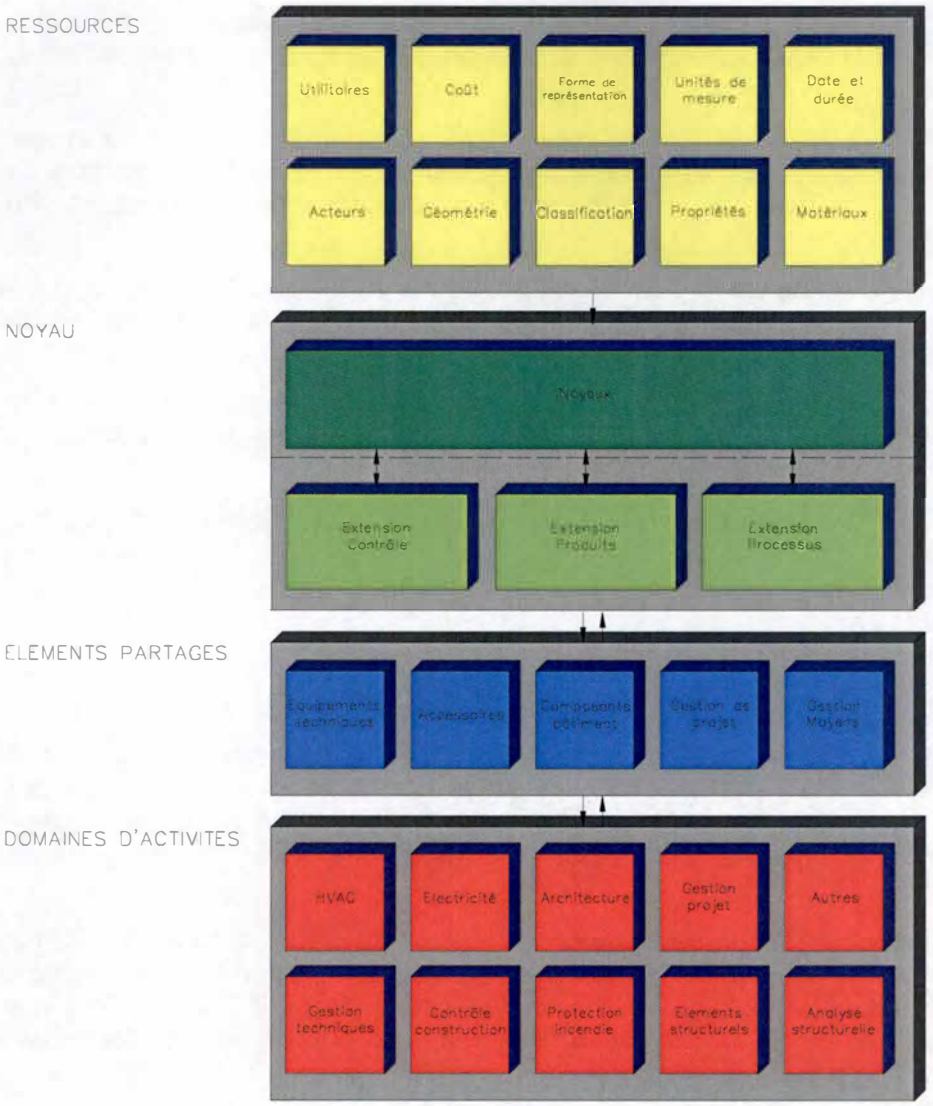


Figure 6.3 : Représentation graphique de l'architecture des IFC.

Ce schéma tente de rendre compte de l'architecture du modèle. Idéalement, il faudrait explorer les schémas conceptuels EXPRESS-G fournis par l'IAI [Pierra 2000]. Mais ces tableaux sont volumineux : plus d'une centaine de pages[4]. Nous n'aborderons pas ici ce niveau de description du modèle IFC.

Il faut bien comprendre que ce modèle ambitionne de représenter toutes les données ayant trait à une opération de conception-construction. Même s'il est à la base un modèle de composants du bâtiment, il permet de représenter également des acteurs et des organisations, des documents, des processus, des plannings, des commandes, etc.

Nous allons maintenant décrire les composants principaux des différents niveaux de l'architecture du modèle IFC

[4] Les spécifications du modèle IFC sont jointes à ce travail et se trouvent sur le cd-rom comprenant les annexes.

Niveau 1 : Les ressources

Le niveau le plus générique est le niveau des ressources (ressource layer), il définit des concepts utilisés par tous les intervenants. Ce niveau de description a déjà été normalisé par l'ISO au sein des groupes de travail de STEP, et sert aussi à d'autres secteurs d'activités que le bâtiment. Les IFC puisent donc directement dans cette bibliothèque publique pour parfaire le niveau le plus détaillé de la description de ses objets.

Par exemple les dessins de plusieurs classes d'objets IFC sont souvent constitués de lignes et autres entités graphiques répétitives, les dimensions s'expriment toujours par les mêmes unités ...

Niveau 2.1 : Le noyau

Le noyau (niveau 2.1) et ses extensions (niveau 2.2) rassemblent les concepts indépendants d'un point de vue particulier. Le noyau définit la partie la plus abstraite et la plus riche de l'architecture IFC. Le noyau est composé de trois types de concepts différents :

- ***Objet Projet***

  Un objet est une classe abstraite qui définit tout objet physique, virtuel ou concept abstrait relatif au projet. Cette classe va servir de base pour la représentation de l'ensemble des éléments contribuant à la réalisation d'une construction. Ces éléments peuvent être de différentes natures : composants matériels (mur, porte,...), éléments spatiaux (étage, local, ...), acteurs (architecte, ingénieur,...), ressources, ...

  Dans le models IFC, le concept d'objet, se spécialise en sept types d'entités fondamentales, ces entités forment le second niveau de spécialisation dans la hiérarchie des classes concernant la branche « Objet ». Les sept types d'entités sont :

  « *Produit* » : Cette classe abstraite représente tout produit manufacturé, découpé ou fabriqué in situ, ou encore résultat de frontières matérielles comme les espaces. Les produits peuvent être définis par une représentation formelle et peuvent avoir une localisation dans l'espace.

  « *Processus* » : Ce sont des actions effectuées dans le cadre d'un projet et qui ont pour but, par exemple, d'acquérir, de construire ou d'entretenir des objets. Les processus sont formalisés sont forme de séquence.

  « *Contrôle* » : Toute méthode, procédure, contrainte s'appliquant à une classe d'objet.

  « *Ressource* » : Il s'agit des informations nécessaires aux procédures concernant par exemple, les études de prix, de planning et d'ordonnancement des travaux.

  « *Acteur* » : Toute personne physique participant aux études, à la construction, à l'entretien ou à l'occupation du bâtiment tout au long de son cycle de vie.

  « *Projet* » : Cette classe va contenir des informations générales concernant le projet. Par exemple son nom, les phases du projet, la liste des unités de mesures qui sont utilisées dans le projet, la localisation du projet, etc...

  « *Groupe* » : Concept abstrait servant à réunir tout ensemble d'objets IFC qui assurent une même fonction, sans considération de leur position ou relations dans le projet. Par exemple, un système, une zone,...

- ***Relation***

  Une relation permet de créer des liens entre les différents objets.

  Dans le models IFC, le concept de relation se spécialise en cinq types de relations fondamentales, ces relations forment le second niveau de spécialisation dans la hiérarchie des classes concernant la branche « Relation ». Les cinq types de relations sont :

« Assigne » : Cette relation représente une association au travers de laquelle un objet fait appel aux services d'autres objets ou au travers de laquelle un objet peut naviguer vers d'autres objets.

« Associer » : cette relation fait référence à des sources d'informations externes (par exemple, un livre, un dictionnaire, des règles,...) et les associent à des objets.

« Décomposer » : Cette relation définit le concept général des éléments composés ou décomposés. Elle permet de naviguer du tout vers les éléments composés et vice-versa.

« Connecter » : capacité à connecter des objets entre eux. Par exemple, un objet fenêtre dans un objet mur.

« Définir » : C'est une relation permettant de lier un type ou des propriétés à un ensemble d'objets. Par exemple, si on veut que les châssis d'un projet possèdent tous les mêmes propriétés.

- ***Propriété***

Les propriétés définissent les caractéristiques des objets.
Dans le modèle IFC, le concept de propriété se spécialise en deux types de propriétés fondamentales, ces propriétés forment le second niveau de spécialisation dans la hiérarchie des classes concernant la branche « Propriété ». Les deux types de propriétés sont :

« Type » : définit l'information spécifique concernant le type de l'objet.

« Set de propriétés » : Définition d'ensemble de propriétés partageables ou extensibles qui peuvent être attachées à une occurrence d'objet.

Niveau 2.2 : Les extensions du noyau

Dans le modèle IFC actuel, le noyau comprend 3 modules d'extensions :

« Extension produit » : c'est le module d'extension le plus important et le plus développé. Comme son nom l'indique, l'« extension produit » est une spécialisation de l'entité produit. Les concepts de base liés à l' « extension produit » sont :

- La structure spatiale : Concept abstrait pour définir un élément spatial composé soit d'un espace mesurable, soit des limites de l'espace. Elle définit les notions de site, de bâtiment, d'étage et d'espace.
- Les éléments : Tout composant matériel du bâtiment, plein ou vide comme par exemple une ouverture, à l'exclusion des espaces. Chaque élément est assigné à la structure spatiale.

« Extension contrôle » : c'est le module qui va permettre d'associer les contraintes et les approbations concernant n'importe quel objet du modèle. A l'heure actuelle, seul le concept est présent, il n'y a encore aucune entité dans le module.

« Extension processus » : c'est le module qui va permettre de développer la notion de processus. L'objectif de ce module est de récolter les informations nécessaires pour le planning et l'ordonnancement de tâches, ainsi que pour définir les ressources utiles à la réalisation de travail. A l'heure actuelle, seules les entités principales sont présentes dans le module.

Niveau 3 : Les éléments partagés

Ce niveau abstrait permet de faire partager des informations et objets communs. Les IFC nomment le niveau des éléments partagés, le niveau de « l'interopérabilité ». Les éléments partagés se répartissent en 5 modules :

« Composant bâtiment » : ils représentent la majorité des éléments qui constituent le bâtiment, par exemple, un mur, une poutre, un châssis,.... Les échanges de données relatifs

au projet porteront principalement sur ces éléments. Pour chacun de ces composants, on prévoit une représentation géométrique qui va définir les informations utiles à la définition de cet élément.

« Accessoires » : ce module permet de représenter de petits accessoires.

« Equipement » : Ce module définit les concepts nécessaire afin d'assurer l'interopérabilité entre différents équipements tels que l'HVAC, la protection incendie, l'électricité,...

« Gestion de projet » : il définit des concepts de base qui sont régulièrement utilisés dans le cadre de la gestion d'un projet de construction. A l'heure actuelle, ce module est très pauvre et doit encore évoluer.

« Patrimoine » : il définit des concepts de base pour la gestion du patrimoine.

Niveau 4 : Les domaine d'activités

Ce niveau correspond aux objets spécifiques des domaines d'activités. Chaque partenaire d'une opération, qui partage les données d'un projet de bâtiment, manipule des éléments qui sont propres à son activité et pour lesquels, il n'est pas nécessaire de partager les concepts et les définitions.

La structure générale des IFC adopte ce principe en définissant des grands domaines d'activités. Par exemple : l'architecture, HVAC, ...

6.7.4 La plate-forme IFC-2X

Un sous-ensemble du modèle IFC-2X appelé Plateforme a été homologué par l'ISO en novembre 2002 sous la référence ISO/PAS[5] 16739. La plate-forme a pour but de stabiliser une partie du système, de manière à inciter les éditeurs de logiciels à investir durablement dans le développement d'applications communiquant via les IFC. Cette plate-forme est conçue de manière à pouvoir intégrer progressivement de nouvelles extensions au modèle.
Le schéma ci-après nous permet de visualiser les modules de l'architecture du modèle IFC qui sont intégrés à la plate-forme IFC-2X.

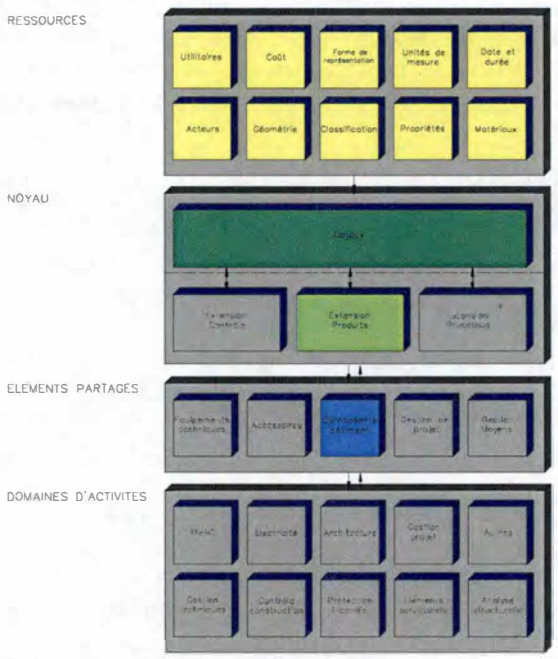


Figure 6.4 : Représentation graphique de la plateforme IFC-2X.

[5] PAS signifie "Publicly Available Specification", catégorie de documents réservée à des spécifications élaborées à l'extérieur de l'ISO et ayant fait l'objet d'un vote positif de la majorité des pays membres du comité. La suite logique sera d'évoluer dans un délai de 3 ans vers un statut de standard international (IS).

6.7.5 Evolution

Le modèle IFC tente de représenter toutes les informations susceptibles d'être échangées ou partagées par les acteurs. Ce modèle est parfois très abstrait et complexe vu le nombre d'entités développées et les potentialités de relations entre elles. Il continue d'évoluer dans le sens d'une augmentation de sa « couverture », notamment du côté des domaines spécifiques d'applications.

6.7.6 Limites

En tant que modèle de données, aussi large soit son spectre, il n'a pas vocation à spécifier ce qui doit être fait des données modélisées. La notion de processus par exemple est développée mais il est à charge d'une application logicielle de la réutiliser pour la proposer à d'éventuels utilisateurs. Il n'y a donc pas, derrière le modèle, un modèle implicite de processus ou de déroulement d'un projet.

6.7.7 Obstacles aux IFC

Les arguments qui s'opposent à l'avènement de tels standards dans un avenir proche ou immédiat, outre l'extrême difficulté à les concevoir, tiennent au complet changement des habitudes de travail qu'ils induisent pour leurs destinataires. À l'heure actuelle, les acteurs travaillent à partir de représentations dont les caractéristiques sont :

- La fragmentation ; c'est par le recoupement de plusieurs documents que les acteurs « reconstituent » l'objet à concevoir.
- Les documents graphiques contractuels servant à construire sont tous créés, sauf rares exceptions, en deux dimensions. L'effort qui demande de modéliser en trois dimensions est encore énorme et surtout souvent inutile pour de petites opérations.
- La progression de la conception architecturale profite de l'imprécision initiale et des lacunes de coordination entre les métiers ; c'est pourquoi il est inutile quelquefois de découvrir trop tôt des problèmes qui plus tard trouveraient « naturellement » réponse.

D'autre part, ces nouveaux standards ont du mal à s'imposer dans le secteur, l'habitude étant prise d'utiliser des formats propriétaires et de réaliser des conversions directement entre ces formats, ce que proposent, à de rares exceptions près, tous les logiciels.

6.8 Problèmes et difficultés

6.8.1 Nature des problèmes

On peut observer deux types de problèmes :

Les problèmes se manifestant sur le bâtiment sont :

- Réclamations des clients (problèmes esthétiques, finitions, malfaçons diverses) ;
- Sinistres couverts par les garanties de bon fonctionnement et garanties décennales.
- Défauts de conformité à la réglementation technique.

Les problèmes d'organisation.

- Problèmes d'organisation interne dans les entreprises, dans les bureaux d'architecture, dans les bureaux d'études.
- Gaspillage de temps pendant les études.
- Reprises d'ouvrages mal exécutés, accidents, litiges, temps perdu sur le chantier.

6.8.2 Identification des problèmes

On peut identifier quatre catégories de problèmes liés au secteur du bâtiment.

Les problèmes structurels

Ce sont de loin les problèmes les plus difficiles à résoudre car ils mettent en cause la nature même des relations entre les acteurs et le système de valeurs des acteurs.

- La coupure entre l'équipe de conception et l'équipe de réalisation. Dans bien des cas (et peut-être dans tous), nous pouvons mettre ces problèmes sur le compte d'une déficience de « l'information ». Par exemple, le manque d'information, le retard de l'information, la fausseté de l'information, l'imprécision de l'information, le manque de spécificité de l'information, etc.
- La coupure entre la maîtrise d'œuvre et la maîtrise d'ouvrage. Car, dans la conception d'un projet architectural, la définition du problème et la résolution du problème sont deux activités inséparables et interagissant. Le programme donné par le maître d'ouvrage et le projet rendu par la maîtrise d'œuvre sont les deux faces d'un même problème. Or, l'équilibre décisionnel n'est pas toujours respecté.
- Les documents contractuels doivent définir clairement les responsabilités de chacun. Or, on remarque que bien souvent il existe une indéfinition relative de certaines tâches imparties à chacun des acteurs. Cette situation met en cause la cohérence de leur production et se trouve à l'origine de conflits.

La défaillance des acteurs

L'absence de lecture des documents, les retards, les interventions non-conformes au stade d'avancement du projet, les lenteurs, sont quelques-unes des défaillances identifiées. Elles engendrent, dans le quotidien des acteurs, une série de problèmes qui appellent une résolution urgente. Elles ont une influence négative sur l'un des facteurs suivants : le délai, le coût ou la qualité.

Les difficultés de l'informatisation

La plupart des intervenants éprouvent des difficultés face à l'informatisation. Nous allons en décrire les principales raisons :

- Ces technologies sont récentes et les professionnels ne sont pas tous informés de leurs possibilités.
- La plupart des petites entreprises éprouvent des difficultés à maîtriser l'informatique la plus élémentaire. Leur réticence provient autant d'un défaut de compétences de leurs équipes que de l'attachement à des modes de communication traditionnels.
- L'utilisation des technologies de l'information induit un profond changement au niveau des méthodes de travail. Par exemple, l'équilibre actuel entre acteurs est menacé par la communication graphique informatisée qui pose le problème du transfert des charges de travail, et par conséquent, le transfert des responsabilités : en effet, il y a un transfert du travail du destinataire, qui devait auparavant ressaisir, vers l'expéditeur, qui assure désormais la saisie au profit du destinataire.
- Le remplacement d'une technologie par une autre est toujours une opération industrielle longue et implique des investissements matériels et humains importants.
- Les utilisateurs vont devoir assumer une période transitoire entre l'exploitation des logiciels conventionnels, structurés en couches et celles des logiciels structurés en objets. Cette période transitoire présente un risque non négligeable pour les entreprises de taille moyenne.
- Le risque d'exclusion de certains professionnels qui n'ont pas fait le choix de l'informatisation.

Les faiblesses des solutions logicielles

- La multiplication des formats d'échanges propriétaires, notamment graphiques, et la mauvaise qualité de conversion des uns aux autres se traduisant notamment par des pertes d'information, voire des distorsions ou des erreurs. Bien souvent, il n'y a pas d'autre solution que la ressaisie de l'information. Cette ressaisie réalisée manuellement n'évite pas les erreurs traditionnelles d'incohérence entre les représentations multiples du bâtiment.
- La faiblesse sémantique des formats, leur incapacité à échanger des données non-graphiques, à gérer les différents sens attachés aux données, à travailler sur des objets incomplètement décrits, etc.
- La lenteur actuelle des communications, surtout lorsque l'on considère le volume des données qui s'échangent, les problèmes de sécurité et de fiabilité.

6.9 Avantages attendus d'un meilleur système d'échange

Il existe aujourd'hui une demande des professionnels pour améliorer les échanges de données. Cette demande s'affirme principalement dans le besoin d'améliorer les transferts d'informations informatisées et d'enrichir sémantiquement les données.

De telles améliorations auraient un impact nous négligeable sur les facteurs suivants :

La compétitivité :

Augmentation de la compétitivité : amélioration de la productivité et de la qualité des opérations de construction.

La productivité :

Augmentation de la productivité : amélioration des opérations quotidiennes, du travail d'équipe et de la gestion du projet en interne et entre partenaires.

Incidence directe :

- Implication plus grande des PME.
- Réduction de la perte de temps à ressaisir l'information communiquée sous forme papier ou sous forme électronique.
- Réduction du nombre d'erreurs et de non-conformités.
- Détection plus rapide des erreurs.
- Réduction de la durée des études.
- Baisse des coûts d'études et de réalisation

La qualité :

Augmentation de la qualité : améliorer la qualité des dossiers d'opérations (par exemple, le dossier d'exécution) à travers une plus grande cohérence des documents qui les constituent : descriptifs, plans, pièces écrites, etc.

Incidence directe :

- Augmentation de la vitesse de diffusion des informations.
- Plus grande fiabilité grâce à une meilleure cohérence de l'information.
- Meilleure maîtrise des coûts.
- Meilleure maîtrise des délais.
- Augmentation de la qualité du produit final.
- Satisfaction du client.

La gestion de l'information :

Une interdépendance simplifiée.

Incidence directe :

- Accès à l'information plus simple.
- Archivage automatique, meilleure traçabilité.
- Augmentation de la sécurité des données.
- Gestion de la diffusion des documents plus simple.
- Meilleure communication entre les partenaires.

La traçabilité :

Une traçabilité complète des informations transmises.

Incidence directe :

- Rigueur dans la production.
- Optimisation du travail de chacun : certitude de disposer du bon document en temps réel.
- Meilleur respect des délais.
- Diminution d'une partie du contentieux (moins de contestations sur les documents disponibles).

Le travail simultané :

Une plus grande capacité à travailler simultanément.

Incidence directe :

- Gestion du travail coopératif
- Atténuation de la contrainte spatiale.
- Fonction d'aide à la cohérence des dossiers produits.

La production de documents de synthèse :

Une facilité plus grande pour la production de documents de synthèse des différents corps d'état. Développer un système qui permet de générer des cahiers des charges et des métrés de manière plus ou moins automatique.

La constitution du dossier d'exploitation :

La constitution progressive du dossier des ouvrages exécutés (plans as-build). Ce dossier est indispensable pour l'exploitation et la maintenance ultérieure de l'ouvrage. La complétude et la qualité du dossier des ouvrages exécutés est bien souvent un point faible de l'achèvement des projets.

6.10 Conclusion

Dans ce chapitre, nous avons tenté de présenter le secteur du bâtiment, en insistant sur ses spécificités. Bien sûr, cette connaissance est nécessaire à la compréhension de notre problématique et les points soulevés sont ceux que pose l'élaboration d'un modèle d'échange pour les acteurs du domaine de la construction.

Dans les chapitres qui suivent, nous allons appliquer la méthodologie permettant de formaliser les transactions au secteur de la construction.

CHAPITRE 7 : Le niveau business

7.1 Représentation du domaine : diagramme de classes

Dans cette partie, nous allons développer un modèle pour représenter les concepts intervenant dans le domaine de la construction. Le but de ce modèle est de permettre aux acteurs d'avoir une compréhension commune du domaine d'affaire. Ce modèle va décrire les composants tangibles comme les portes, les murs, etc... ainsi que les concepts abstraits comme les espaces, les acteurs, etc... Le vocabulaire utilisé est celui du domaine.

Dans le chapitre précédent, nous avons pu constater que pour rendre possible le partage de données techniques et graphiques d'un projet, il s'est avéré utile de définir un standard d'échange, le modèle IFC. Pour structurer l'information descriptive du bâtiment en vue des échanges, nous allons donc nous appuyer sur cette structure normalisée qui permet d'intégrer tous les aspects descriptifs d'un projet aussi bien les aspects techniques que graphiques.

Cependant, le modèle IFC est vaste (plus de 600 entités) et couvre un spectre beaucoup plus large que celui envisagé dans le cadre de notre étude. De plus, seule une partie du modèle est normalisée de manière officielle : la plate-forme 2X. Pour construire notre modèle, nous allons donc n'utiliser qu'un sous ensemble du modèle IFC.

Afin de sélectionner les parties du modèle qui nous sont utiles, nous allons d'une part nous limiter aux classes du modèle qui nous paraissent les plus représentatives dans le contexte de la problématique des échanges entre les différents acteurs de l'acte de bâtir et d'autre part nous concentrer sur la partie du modèle qui a fait l'objet d'une normalisation par l'ISO, c'est-à-dire la plate-forme IFC2X.

Nous éviterons également de surcharger le modèle avec des classes qui ont un intérêt technique mais qui ne sont pas indispensables à la compréhension du domaine.

7.1.1 La plate-forme IFC-2X

Dans le cadre de ce travail, nous avons fait le choix de ne prendre en compte que les modules qui ont atteint une certaine maturité et une certaine stabilité. Nous avons donc considéré uniquement les modules qui ont fait l'objet d'une normalisation officielle, c'est-à-dire appartenant à la plate-forme IFC-2X.

Toutefois, nous avons dû faire une exception par rapport à ce choix. Il nous est apparu en cours d'étude qu'il était nécessaire d'ajouter le module processus à notre sélection. En effet, comme nous le verrons dans la suite de ce travail, nous avons défini différentes fonctionnalités afin de structurer les échanges entre les acteurs. Ces fonctionnalités correspondent à des tâches qui peuvent être initiées par un acteur et affectées à un ou plusieurs acteurs. Or, le modèle IFC permet d'intégrer de tels concepts au travers du module processus. C'est pour cette raison que nous avons ajouté le module processus à notre sélection.

7.1.2 Sélection des entités représentatives du domaine.

A présent, nous allons explorer l'architecture du modèle IFC[6] niveau par niveau afin de déterminer les parties du modèle IFC à partir desquelles nous allons pouvoir développer notre modèle conceptuel. C'est-à-dire les parties de modèle IFC qui interviennent dans le cadre de la problématique des « transactions métier » effectuées lors des différentes étapes de l'acte de construire.

[6] L'architecture du modèle IFC a été décrite dans le chapitre 6.

Niveau 1 : Les ressources

La plupart des données constituant le niveau ressources sont relativement techniques et sont utiles lors de l'implémentation du modèle mais ne sont pas indispensables pour la modélisation et la compréhension des échanges entre acteurs.

Afin de ne pas surcharger inutilement notre modèle, nous avons donc choisi de nous limiter à la représentation des entités qui étaient nécessaires pour modéliser le domaine. Il s'agit exclusivement des entités concernant les acteurs et leur rôle.

Module retenu : « Acteur »

Niveau 2.1 : Le noyau

Le noyau est composé de trois types de concepts différents :

- *Objet du projet* :

  Cette entité sert de base à la représentation de l'ensemble des éléments contribuant à la réalisation d'une construction et est donc indispensable pour la représentation du domaine.

- *Relation* :

  La notion de relation est essentielle. C'est grâce aux différentes relations que l'on va pouvoir agencer les éléments constructifs les uns par rapport aux autres, que l'on va pouvoir situer un objet dans une entité spatiale, que l'on va pouvoir assigner un objet à un acteur,...
  Ce concept est donc indispensable pour le développement de notre modèle.

- *Propriétés* :

  Les propriétés sont constituées des types et des sets de propriétés qui définissent les caractéristiques des objets. Les propriétés sont représentées par des entités distinctes des objets auxquels elles se rapportent. Elles sont organisées selon une arborescence similaire à celle développée pour représenter les objets.
  Nous n'avons pas trouvé utile de modéliser cette partie afin que notre modèle reste lisible. Dans le cadre de cet exercice, nous considérons que les types et les propriétés sont intégrés dans les objets auxquels ils se rapportent.

Niveau 2.2 : Les extensions du noyau

Le noyau comprend 3 modules d'extensions :

- Le module Extensions produits nous est nécessaire pour créer la jonction avec les éléments partagés et plus particulièrement avec les composants bâtiment.
- Le module Extension processus nous est nécessaire car il nous permet représenter les fonctionnalités pour structurer les échanges entre les acteurs.
- Le module Extension contrôle n'a pas été retenu car il n'appartient pas à la plate-forme IFC-2X.

Niveau 3 : Les éléments partagés

Dans le cadre des échanges entre différents acteurs de l'acte de bâtir, la majorité des documents échangés sont des documents décrivant les composants du bâtiment, par exemple, les murs, les dalles, ... Il nous parait donc important de pouvoir représenter ces concepts dans notre modèle conceptuel. Nous avons donc décidé de conserver la partie du modèle décrivant les composants du bâtiment, il s'agit du module « Composants bâtiment ».
Afin de ne pas surcharger le modèle, nous n'aborderons pas les aspectes liés aux techniques spéciales et à la gestion de projet.

De plus, seule la partie Composants bâtiment est intégrée à la plate-forme IFC-2X.

Niveau 4 : Les domaines d'activités

Nous n'avons retenu aucun des domaines d'activités car les objets qu'ils définissent sont des objets spécifiques à une activité et n'ont donc que très peu d'intérêt dans le cadre d'échange d'information entre les différents intervenants de l'acte de construire.

De plus, les domaines d'activité ne sont pas intégrés à la plate-forme 2X.

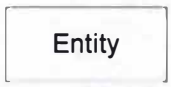
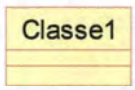
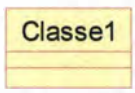
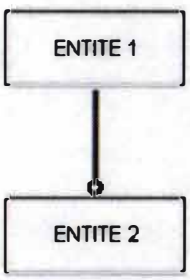
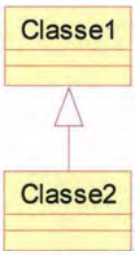
Les attributs

En ce qui concerne les attributs, nous avons décidé pour des raisons de lisibilité, de ne pas les intégrer dans notre diagramme de classes.
La spécification des classes, des attributs et des relations est disponible dans l'annexe 1.

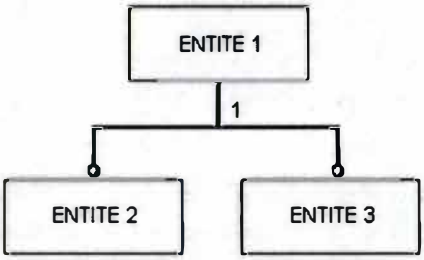
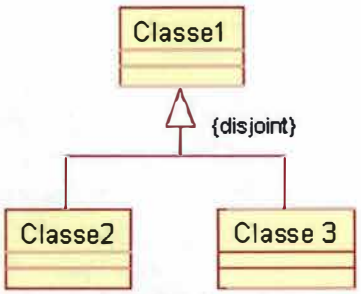
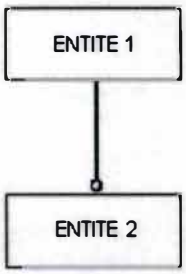
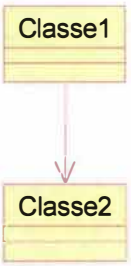
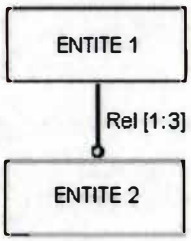
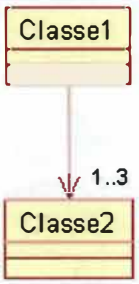
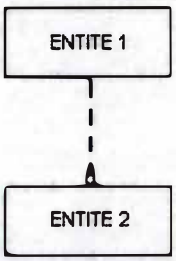
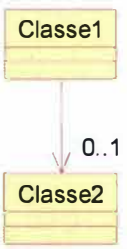
7.1.3 Règles de Mapping EXPRESS – diagramme de classes UML

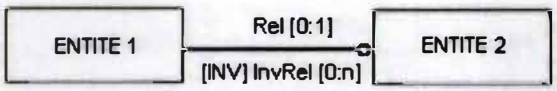
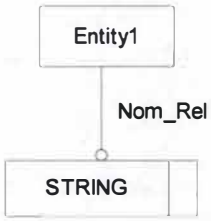
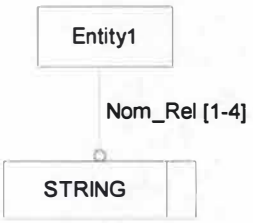
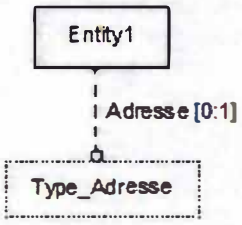
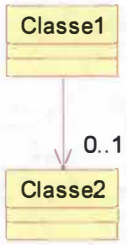
A présent que nous avons déterminé les parties du modèle IFC qui nous étaient utiles pour développer notre modèle conceptuel, il nous reste à traduire ce sous modèle IFC du langage EXPRESS en un diagramme de classes UML. Pour rappel, ces deux types de représentations permettent uniquement de représenter les concepts statiques et ne représentent pas les aspects dynamiques.

Avant de vous présenter le diagramme de classe obtenu, nous allons exposer les principales règles de transformation[7] qui nous ont permis de transformer le modèle EXPRESS en diagramme de classe UML [Arnold 1998].

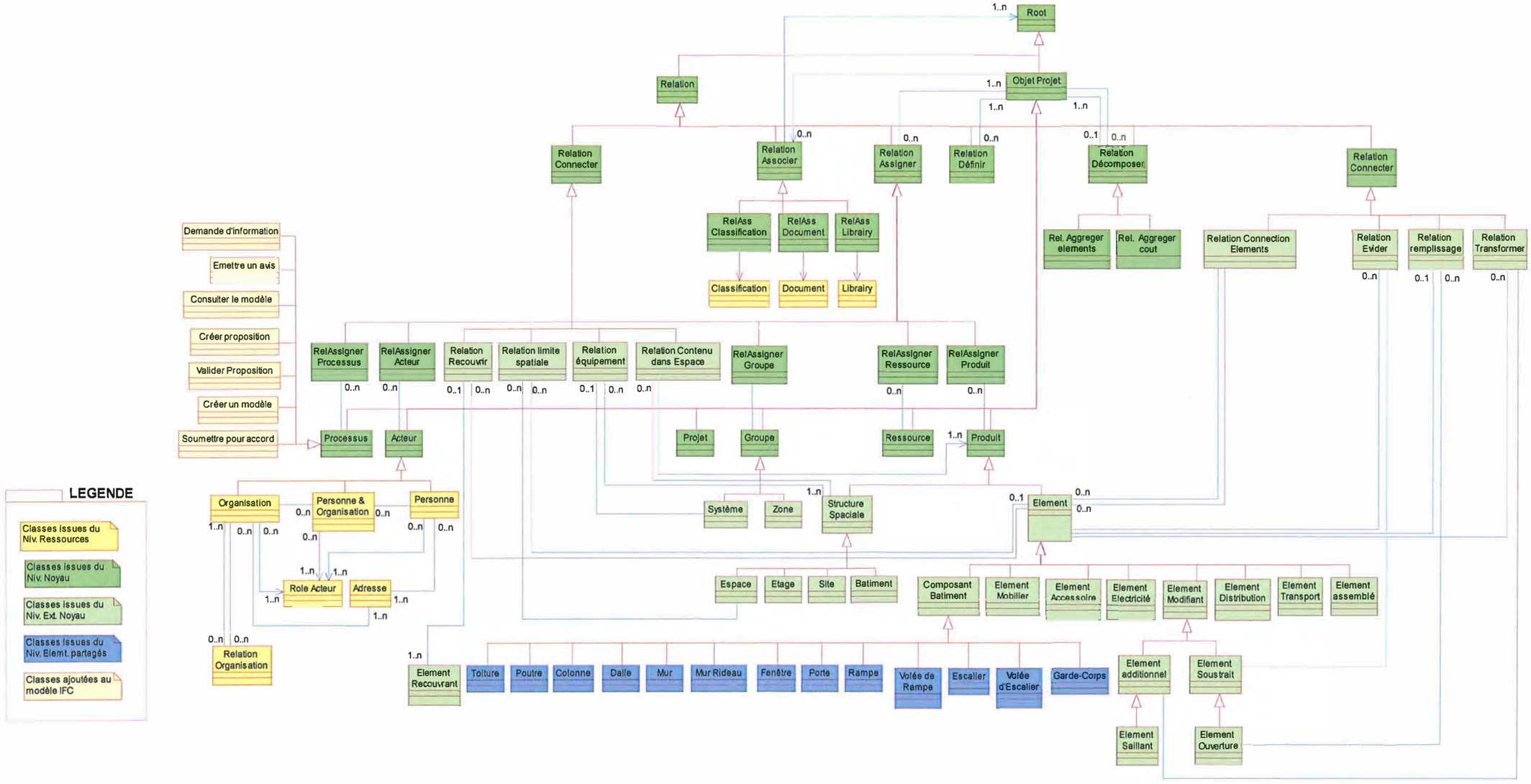
TRANSFORMATION	
EXPRESS	**UML**
L'entité EXPRESS est représentée par un rectangle dans lequel est indiqué le nom de l'entité.	L'entité est transformée en classe UML.
	
Le type primitif EXPRESS est représenté un rectangle à deux compartiments. Le type défini par l'utilisateur est représenté par un rectangle en pointillés.	Un type EXPRESS est transformé en classe UML
	
La relation super-type/sous-type EXPRESS entre deux entités est représentée par une ligne épaisse avec un cercle sur l'extrémité de la ligne située du coté du sous-type.	Cette relation est transformée en relation de généralisation UML.
	

[7] Les règles de mapping sont tirées de [Arnold 1998]

La relation EXPRESS « OneOff » peut être ajoutée à une relation de super-type, sous-type. Elle dénote une disjonction.	La relation EXPRESS « OneOff » est représentée par la contrainte {disjonction} appliquée à la relation de généralisation.
	
La relation EXPRESS entre entités est obligatoire si elle est représentée par une ligne continue avec un cercle à une extrémité. La relation est orientée vers le cercle.	Une relation EXPRESS entre entités est transformée en une relation unidirectionnelle UML.
	
La relation EXPRESS entre entités peut-être accompagnée d'une cardinalité.	Dans le cas d'une relation avec cardinalité, la cardinalité est ajoutée du côté de la flèche terminant la relation unidirectionnelle UML.
	
La relation EXPRESS entre entités est optionnelle lorsqu'elle est représentée par une ligne pointillée avec un cercle à une extrémité. La relation est orientée vers le cercle.	Dans le cas d'un attribut optionnel, une cardinalité de [0..1] est ajoutée du coté de la flèche terminant la relation unidirectionnelle UML.
	

La relation EXPRESS « Inverse » dénote une relation bidirectionnelle entre deux entités. Elle permet de représenter l'inverse d'une relation et de définir pour cette relation une cardinalité.	La relation et son inverse sont transformés en une association bidirectionnelle UML avec des cardinalités aux deux extrémités.
	
La relation EXPRESS entre une entité et un type est obligatoire si elle est représentée par une ligne continue avec un cercle à une extrémité. La relation est orientée vers le cercle. La relation est toujours nommée.	Une relation EXPRESS entre une entité et un type est transformée en un attribut portant le nom de la relation.
	
La relation EXPRESS entre une entité et un type peut-être accompagnée d'une cardinalité.	Dans le cas d'une relation avec cardinalité, la cardinalité est ajoutée à l'attribut.
	
La relation EXPRESS entre une entité et un type est optionnelle lorsqu'elle est représentée par une ligne pointillée avec un cercle à une extrémité. La relation est orientée vers le cercle. La relation est toujours nommée.	Dans le cas d'un attribut optionnel, une cardinalité de [0..1] est ajouté à l'attribut.
	

Certaines contraintes sont définies à l'extérieur de la représentation graphique.
Ces transformations sont possibles car les concepts partagent la même sémantique.


LEGENDE
Classes issues du Niv. Ressources
Classes issues du Niv. Noyau
Classes issues du Niv. Ext. Noyau
Classes issues du Niv. Elemt. partagés
Classes ajoutées au modèle IFC
Root
Relation
Objet Projet
Relation Connecter
Relation Associer
Relation Assigner
Relation Définir
Relation Décomposer
Relation Connecter
RelAss Classification
RelAss Document
RelAss Librairy
Classification
Document
Librairy
Rel. Aggreger elements
Rel. Aggreger cout
Relation Connection Elements
Relation Evider
Relation remplissage
Relation Transformer
Demande d'information
Emettre un avis
Consulter le modèle
Créer proposition
Valider Proposition
Créer un modèle
Soumettre pour accord
RelAssigner Processus
RelAssigner Acteur
Relation Recouvrir
Relation limite spatiale
Relation équipement
Relation Contenu dans Espace
RelAssigner Groupe
RelAssigner Ressource
RelAssigner Produit
Processus
Acteur
Projet
Groupe
Ressource
Produit
Organisation
Personne & Organisation
Personne
Role Acteur
Adresse
Relation Organisation
Système
Zone
Structure Spaciale
Element
Espace
Etage
Site
Batiment
Composant Batiment
Element Mobilier
Element Accessoire
Element Electricité
Element Modifiant
Element Distribution
Element Transport
Element assemblé
Element Recouvrant
Toiture
Poutre
Colonne
Dalle
Mur
Mur Rideau
Fenêtre
Porte
Rampe
Volée de Rampe
Escalier
Volée d'Escalier
Garde-Corps
Element additionnel
Element Soustrait
Element Saillant
Element Ouverture

7.1.4 Diagramme de classe UML

7.1.5 Description des classes

Dans l'annexe 3 jointe à ce mémoire, vous trouverez l'entièreté des spécifications concernant le modèle IFC et notamment la description de toutes les entités du modèle. Toutes les entités composant notre diagramme de classe sont issues de ce modèle ; pour la description des entités, nous vous invitons donc à consulter ces spécifications.

7.2 Représentation des « Business Process » par un diagramme de cas d'utilisation

7.2.1 Le système retenu

Dans le cadre d'un projet de construction, les différents systèmes informatiques en présence, hétérogènes par nature, doivent pouvoir être capables de s'échanger leurs données et leurs résultats. Pour satisfaire ce besoin d'échange, la première étape consiste à définir un langage commun. Comme nous l'avons vu précédemment, ce langage existe et est opérationnel, il s'agit du modèle IFC.

Il est ensuite indispensable d'aller plus loin, en effet, tout projet de construction nécessite des échanges de données multiples et complexes entre partenaires et pas simplement binaires et unilatéraux. La conception des bâtiments, par exemple, suit un processus d'échanges interactifs et souvent itératifs entre architectes et ingénieurs. Ce mode de travail, qualifié d'ingénierie collaborative, nécessite la mise en commun des données du projet et illustre le besoin de partage de données autour d'un modèle unique. Le concept de maquette numérique répond à cette problématique.

C'est sur ce concept de maquette numérique, que nous allons envisager les échanges dans le cadre de la conception et la réalisation d'un ouvrage. La possibilité pour les acteurs de partager une maquette virtuelle de l'ouvrage en cours de conception ou de réalisation, et de fonder leurs échanges sur ce référentiel commun. Par exemple, au lieu d'un commentaire sur un fichier (ou une version de document), les acteurs formulent des commentaires sur un ouvrage ou une partie d'ouvrage comme une cloison, un escalier, une fenêtre, etc.

La solution envisagée a pour objectif de transformer un environnement hétérogène en un vaste système d'information exploitant un modèle numérique normalisé du bâtiment.

L'objectif vise à ce que tout acteur ayant une adresse électronique et un accès Internet ait la possibilité de se coordonner avec d'autres acteurs au sein d'un projet.

La facilitation des échanges d'une part, avec pour conséquence attendue une augmentation du flux d'échanges entre les partenaires du projet, et la plus grande formalisation des échanges d'autre part, et son impact sur la transparence du jeu des acteurs, amènent un questionnement des pratiques de coopération.

Un modèle partagé peut par les avantages qu'il comporte, conduire à améliorer l'efficacité de la conception, du chiffrage, de la construction et de la gestion du bâtiment.

7.1.4 Diagramme de classe UML

A intégrer

7.1.5 Description des classes

Dans l'annexe 3 jointe à ce mémoire, vous trouverez l'entièreté des spécifications concernant le modèle IFC et notamment la description de toutes les entités du modèle. Toutes les entités composant notre diagramme de classe sont issues de ce modèle ; pour la description des entités, nous vous invitons donc à consulter ces spécifications.

7.2 Représentation des « Business Process » par un diagramme de cas d'utilisation

7.2.1 Le système retenu

Dans le cadre d'un projet de construction, les différents systèmes informatiques en présence, hétérogènes par nature, doivent pouvoir être capables de s'échanger leurs données et leurs résultats. Pour satisfaire ce besoin d'échange, la première étape consiste à définir un langage commun. Comme nous l'avons vu précédemment, ce langage existe et est opérationnel, il s'agit du modèle IFC.

Il est ensuite indispensable d'aller plus loin, en effet, tout projet de construction nécessite des échanges de données multiples et complexes entre partenaires et pas simplement binaires et unilatéraux. La conception des bâtiments, par exemple, suit un processus d'échanges interactifs et souvent itératifs entre architectes et ingénieurs. Ce mode de travail, qualifié d'ingénierie collaborative, nécessite la mise en commun des données du projet et illustre le besoin de partage de données autour d'un modèle unique. Le concept de maquette numérique répond à cette problématique.

C'est sur ce concept de maquette numérique, que nous allons envisager les échanges dans le cadre de la conception et la réalisation d'un ouvrage. La possibilité pour les acteurs de partager une maquette virtuelle de l'ouvrage en cours de conception ou de réalisation, et de fonder leurs échanges sur ce référentiel commun. Par exemple, au lieu d'un commentaire sur un fichier (ou une version de document), les acteurs formulent des commentaires sur un ouvrage ou une partie d'ouvrage comme une cloison, un escalier, une fenêtre, etc.

La solution envisagée a pour objectif de transformer un environnement hétérogène en un vaste système d'information exploitant un modèle numérique normalisé du bâtiment.

L'objectif vise à ce que tout acteur ayant une adresse électronique et un accès Internet ait la possibilité de se coordonner avec d'autres acteurs au sein d'un projet.

La facilitation des échanges d'une part, avec pour conséquence attendue une augmentation du flux d'échanges entre les partenaires du projet, et la plus grande formalisation des échanges d'autre part, et son impact sur la transparence du jeu des acteurs, amènent un questionnement des pratiques de coopération.

Un modèle partagé peut par les avantages qu'il comporte, conduire à améliorer l'efficacité de la conception, du chiffrage, de la construction et de la gestion du bâtiment.

7.2.2 Profil des acteurs.

Dans le chapitre précédent, nous avons identifié les intervenants classiques d'un projet de construction et nous avons défini leurs missions respectives dans le cadre du processus de construction.

La solution que nous avons envisagée a introduit deux nouveaux acteurs : Serveur de projet et Gestionnaire maquette. Nous les décrirons ci-après.

La figure qui suit, nous montre les relations qu'il existe entre les acteurs :

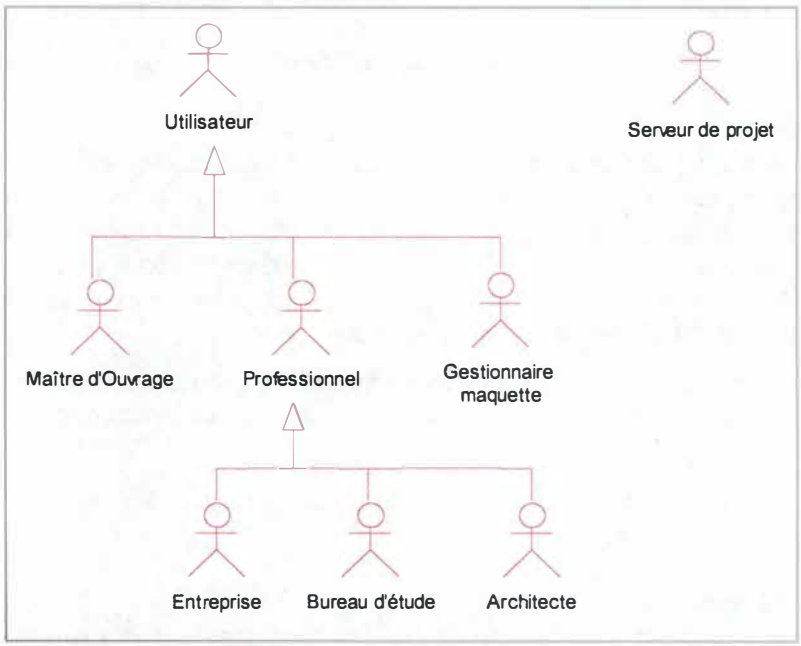


Figure 6.4 : Profil Acteurs.

Nous allons maintenant décrire le rôle de chacun des acteurs dans le système envisagé.

Utilisateur

Il s'agit soit du maître d'ouvrage, soit d'un professionnel participant à la conception ou à la réalisation d'un ouvrage.

Fonction : → Effectuer une demande d'information.
→ Consulter le projet représentant l'ouvrage.
→ Emettre un avis.

Maître d'ouvrage

Il s'agit d'une personne pour le compte de qui est construit l'ouvrage ; il est généralement le futur propriétaire.

Fonction : → Hérite des fonctions de « Utilisateur »
→ Donner son accord sur le projet au terme de chaque phase.

Professionnel

Tout professionnel intervenant dans la conception ou la réalisation de l'ouvrage. Dans cette étude, nous considérons que les différents intervenants proviennent soit d'un bureau d'architecture, soit d'un bureau d'étude, soit de l'entreprise responsable de la construction de l'ouvrage.

Fonction : → Hérite des fonctions de « Utilisateur »
→ Créer et publier une proposition technique.
→ Demander au gestionnaire de la maquette de valider une proposition.

Bureau d'étude

Il est chargé d'une mission dite d'ingénierie, vérifie la faisabilité du projet sur divers points techniques.

Fonction : → Hérite des fonctions de « Professionnel »

Entreprise

L'équipe de réalisation chargée par contrat de la responsabilité de l'ensemble de la construction d'un ouvrage.

Fonction : → Hérite des fonctions de « Professionnel »

Architecte

L'architecte n'est pas un spécialiste comme la plupart de ses partenaires, c'est avant tout un homme de synthèse. La qualité architecturale n'est pas la qualité cumulée de toutes les disciplines associées à la conception ; ce qui compte est l'affirmation d'une intention globale et cohérente. La place des architectes au sein d'un système d'information y est centrale. De plus, l'architecte traite à la fois avec la maîtrise d'ouvrage et avec tous les acteurs de la construction. Il est le trait d'union entre les deux autres parties.

C'est pour cette raison qu'il serait logique de lui confier également le rôle de gestionnaire de la maquette. Cependant, au niveau conceptuel, nous avons préféré créer 2 rôles distincts.

Fonction : → Hérite des fonctions de « Professionnel »
→ Demander au maître de l'ouvrage de marquer son accord sur le projet au terme de chaque phase.

Gestionnaire de la maquette

Le gestionnaire de la maquette est le garant de la cohérence de la maquette de référence. Il a l'obligation de valider ou non les propositions émanant des professionnels et d'intégrer les modifications à la maquette de référence.

C'est également lui qui va initialiser le serveur de projet.

Fonction : → Hérite des fonctions de « Utilisateur »
→ Valider une proposition technique proposée par un professionnel.
→ Mettre à jour la maquette de référence décrivant l'ouvrage.
→ Initialiser le serveur de projet devant accueillir la maquette de référence et les propositions.

Serveur de projet.

Le Serveur de projet est composé de deux parties :

- La maquette de référence, c'est-à-dire l'ensemble des éléments représentant l'ouvrage à réaliser. Ces éléments ont un caractère contractuel.
- Les propositions techniques produites par les professionnels. Ces éléments sont le support à l'élaboration de solutions et ont un caractère informel.

Fonction : → Extraire et transmettre des éléments du projet.
→ Publier une proposition technique.
→ Extraire et transmettre d'une proposition les éléments propres à la stabilité.
→ Extraire et transmettre d'une proposition les éléments propres à l'électricité.
→ Extraire et transmettre d'une proposition les éléments propres à ...
→ Intégrer certains objets d'une proposition dans la maquette de référence.
→ Enregistrer un avis concernant un objet.

7.2.3 Fonctionnalités du système.

Les types d'informations générés au cours d'un projet et les savoir-faire liés à leur production sont très nombreux et peuvent varier suivant la nature et l'organisation du projet. Pour cette raison, nous avons orienté notre étude sur les pratiques de collaboration (exemple : validation, demande d'information) plutôt que sur le terrain du savoir faire (exemple : traitement des appels d'offre, génération de situations de chantier). L'application de ce principe nous a permis d'envisager un système qui s'adapte aux pratiques actuelles et ne soit pas trop contraignant pour les utilisateurs. En effet, cette approche présente une certaine souplesse et garantit d'être effectif dans un maximum de situations et pour un maximum d'acteurs.

Partant de ce constat, nous proposons l'étude d'un système de coopération permettant aux acteurs d'un projet de se coordonner grâce à des fonctionnalités respectant des procédures de partage prédéterminées. Ces procédures de partage que nous qualifions de « requêtes » sont issues des pratiques du secteur de l'ingénierie. Elles constituent une approche simple de la coordination où l'implication des acteurs dans l'utilisation d'un système de partage est faible. Ces requêtes échangées entre acteurs reposent essentiellement sur le suivi des objets d'un projet (consultation, modification, annotation, validation...).

Le système de coopération a pour fonction de centraliser les échanges et de visualiser au travers de ces échanges l'évolution du projet.

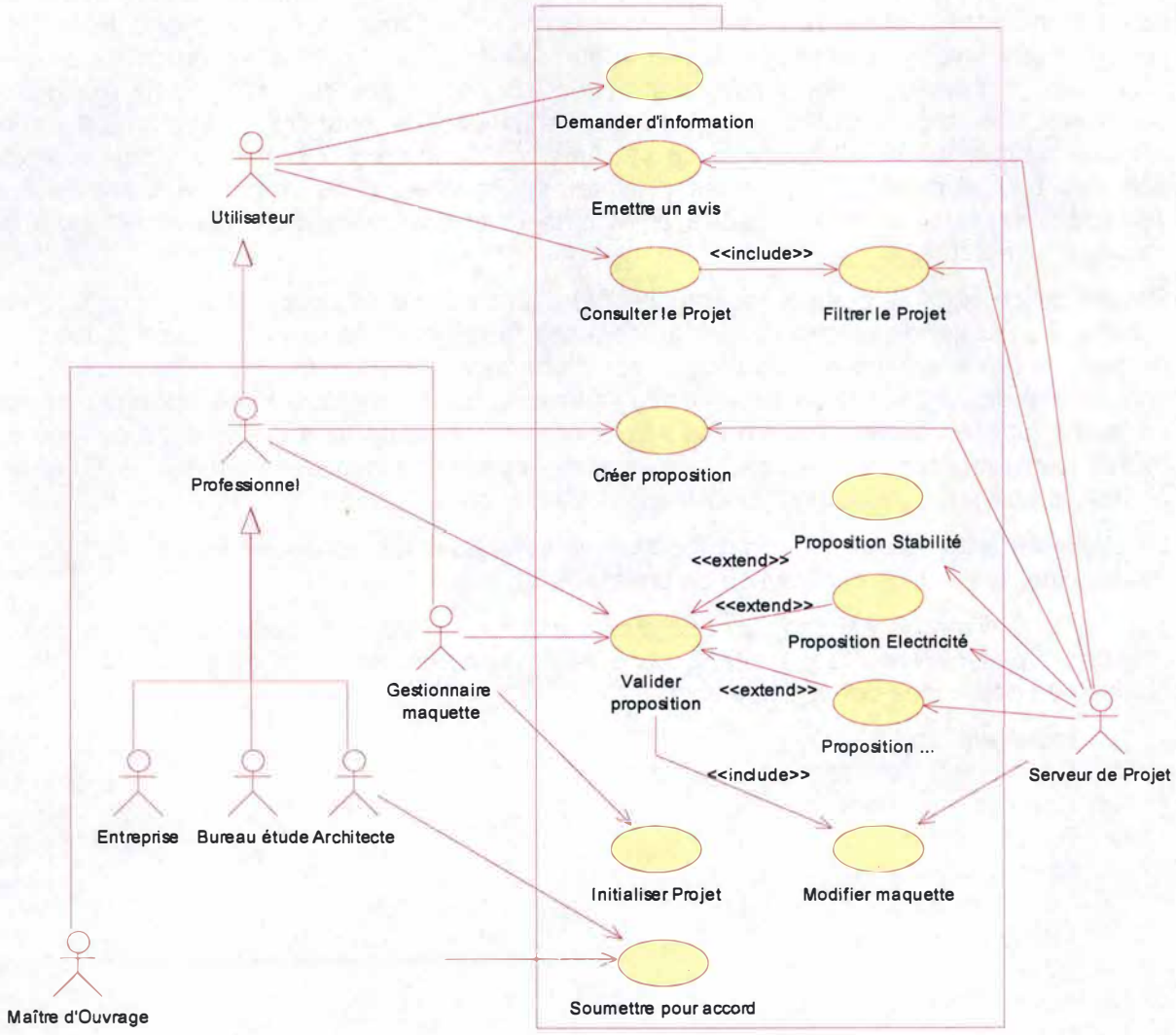
Afin de formaliser les échanges, nous avons défini neuf types de requêtes sur les objets. D'autres types de requêtes existent, nous ne les examinerons pas dans le cadre de ce travail. Les neuf types de requêtes sont :

- Initialiser Projet.
- Demander information.
- Consulter le Projet.
- Filtrer le Projet.
- Créer proposition.
- Valider proposition.
- Modifier maquette.
- Emettre un avis.
- Soumettre pour accord.

Toute requête se termine par l'accomplissement de ce qui a été demandé ou par l'annulation d'une partie (initiateur ou destinataire). Lorsqu'une requête s'adresse à un groupe, il suffit qu'un seul acteur du groupe réponde à la requête pour que celle-ci soit réalisée. Si l'initiateur d'une requête veut obtenir l'assentiment de plusieurs acteurs, il proposera une requête différente pour chacun d'eux. Tant qu'une requête n'est pas terminée, les acteurs concernés sont tenus au courant de son état (exemple : envoyée, reçue, en cours de traitement, ajournée).

Au travers d'un diagramme de cas d'utilisation, nous allons maintenant représenter de manière graphique, les acteurs, les fonctionnalités du système ainsi que leurs interdépendances.

7.2.4 Diagramme des uses cases.



7.2.5 Description des uses cases.

UC1 : Demander information.

Objectif : Un utilisateur introduit une demande d'information ou une demande de précision concernant un quelconque point du projet. Cette demande n'est associée à aucun objet en particulier.

Description : L'initiateur introduit une demande d'information et l'adresse à un destinataire, le destinataire réceptionne la demande d'information et répond à l'initiateur. L'initiateur reçoit et consulte la réponse.
Remarque : une demande peut être envoyée simultanément à plusieurs destinataires.

Pré-condition : Le système est alimenté et fonctionne correctement.
Les utilisateurs font partie de l'équipe projet et ont accès au système.

Post-condition : L'initiateur a reçu la réponse correspondant à la demande d'information qu'il a envoyée.

UC2 : Emettre avis

Objectif : Un utilisateur désire émettre un avis ou une remarque concernant des objets appartenant soit à la maquette de référence, soit à une proposition. Il peut adresser son avis à un ou plusieurs destinataires.

Description : L'utilisateur décide d'émettre un avis, il détermine les objets concernés par cet avis et les destinataires auxquels il veut transmettre l'avis. Le système enregistre l'avis et le transmet au destinataire.

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur dispose des droits nécessaires pour émettre un avis.

Post-condition : L'avis est attaché aux objets qui le concernent et a été transmis aux destinataires désignés par l'initiateur.

UC3 : Consulter le Projet

Objectif : Un utilisateur introduit une demande de consultation concernant des objets du Projet. Ces objets peuvent appartenir soit à la maquette de référence, soit à une proposition à laquelle il a accès.

Description : L'utilisateur introduit une demande de consultation. L'utilisateur à la possibilité de filtrer la maquette de référence ou la proposition qu'il a choisie. Le système permet à l'utilisateur de consulter et d'importer les objets sélectionnés.

Exemple : Le bureau d'étude en stabilité demande pour consulter la proposition (P1) émise par l'architecte. Le bureau d'étude consulte P1 et décide d'importer P1 afin de produire une proposition alternative (P2).

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur dispose des droits nécessaires pour consulter les objets.

Post-condition : L'utilisateur a consulté et/ou importé les objets du Projet correspondant à sa demande.

UC4 : Créer une proposition.

Objectif : Un professionnel peut créer et/ou modifier un certain nombre d'objets et peut rendre ces modifications visibles à d'autres utilisateurs.
Le statut de proposition signifie que les modifications n'ont pas été validées par le gestionnaire de la maquette de référence. Cependant, ces modifications peuvent être rendues visibles à d'autres acteurs afin d'entreprendre une réflexion de co-conception. Les propositions n'ont pas de valeur contractuelle ; seules les données intégrées à la maquette de référence ont valeur contractuelle.

Description : Un professionnel crée des objets et/ou importe des objets existants et les modifie. Ensuite le professionnel choisit les utilisateurs qu'il compte informer de l'existence de cette nouvelle proposition et demande au système d'enregistrer sa proposition et de la rendre accessible pour les utilisateurs qu'il a choisis. Le système envoie un message de notification aux utilisateurs qui ont accès à la proposition.

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur dispose des droits nécessaires pour introduire une demande de validation.
L'utilisateur a créé une proposition et l'a stockée sur le serveur de Projet.

Post-condition : Les modifications effectuées par le professionnel ont été enregistrées comme proposition.
La proposition est accessible aux utilisateurs choisis par l'initiateur de la proposition.
Les utilisateurs qui ont accès à la proposition ont été avertis de l'existence de cette nouvelle proposition.

UC5 : Valider une proposition.

Objectif : Un professionnel soumet une de ses propositions au gestionnaire de la maquette de référence afin de la faire valider et d'intégrer dans la maquette de référence, les modifications qui ont été préalablement validées par le gestionnaire de la maquette de référence.

Description : Lorsqu'une proposition atteint un niveau de définition jugé suffisant par un professionnel (A1), celui-ci peut demander au gestionnaire de la maquette de référence (A2) de la valider, c'est-à-dire de déclarer son accord par rapport aux options prises dans cette proposition. Cela n'empêche d'ailleurs pas A1 de continuer à faire évoluer sa proposition mais il le fait en toute connaissance de cause. La requête s'achève lorsque A2 valide la proposition et intègre les modifications dans la maquette de référence. La validation peut n'être obtenue qu'après plusieurs cycles d'ajournement de validation.

Pré-condition : Le système est alimenté et fonctionne correctement.
La proposition est stockée dans le serveur de Projet.
L'utilisateur dispose des droits nécessaires pour soumettre sa proposition.

Post-condition : Le gestionnaire de la maquette a validé la proposition.
Les modifications ont été intégrées à la maquette de référence.
Tous les utilisateurs ont été avertis de la mise à jour de la maquette de référence.

Le cas d'utilisation « Valider une proposition » a été étendu de manière à pouvoir associer un secteur d'activité spécifique à la demande de validation. De cette manière, seuls les objets de la proposition qui sont liés à cette activité seront intégrés dans la maquette de référence. On peut ainsi limiter le rôle et les responsabilités des acteurs à leurs activités.

UC6 : Modifier la maquette de référence

Objectif : Intégrer dans la maquette de référence, les modifications qui ont été préalablement validées par le gestionnaire de la maquette de référence.

Description : Le gestionnaire de la maquette de référence a validé une proposition et demande au système d'intégrer les objets qui ont été modifiés dans le cadre de la proposition. Le système sauvegarde l'état actuel des objets visés (historique) et intègre les modifications. Ensuite, le système notifie à tous les intervenants que la maquette de référence a été mise à jour et donne à l'utilisateur la possibilité de visualiser les objets modifiés.

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur dispose des droits nécessaires pour modifier la maquette de référence.
La proposition a été validée par le gestionnaire de la maquette de référence.

Post-condition : Les modifications ont été intégrées à la maquette de référence.
Tous les utilisateurs ont été avertis de la mise à jour de la maquette de référence.
Les objets qui ont été remplacés ont été sauvegardés.

UC7 : Filtrer le Projet

Objectif : L'utilisateur peut filtrer le Projet (maquette de référence ou proposition) afin de n'avoir accès qu'à certain types d'objets et ainsi obtenir une vue correspondant à ses attentes.

Description : Le système propose différentes possibilités de filtrage à l'utilisateur ; l'utilisateur choisit les filtres à appliquer et les transmet au serveur de projet. Le serveur de projet transmet les objets sélectionnés à l'utilisateur.

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur dispose des droits nécessaires pour filtrer le Projet.
L'utilisateur a introduit une demande de consultation.

Post-condition : Le système a permis à l'utilisateur de limiter sa consultation aux types d'objets qu'il a choisis.

UC8 : Initialisation d'un Projet

Objectif : Initialisation d'un projet, dans le cadre d'un nouveau projet.

Description : La personne désignée et enregistrée comme gestionnaire de maquette de référence introduit les données administratives relatives au projet et les enregistre comme Projet initial. Une fois enregistrée, le système avertit tous les intervenants que le Projet a été créé et le rend accessible aux différents intervenants.

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur dispose des droits nécessaires pour initialiser le Projet.

Post-condition : Le Projet a été créé.
Le Projet est accessible pour tous les utilisateurs via le serveur de Projet.
Tous les utilisateurs ont été avertis de la création du Projet.

UC9 : Soumettre pour accord

Objectif : À la fin de chaque phase de conception ou de réalisation, l'architecte demande au maître de l'ouvrage de marquer son accord par rapport au projet.

Description : L'architecte soumet la maquette de référence au maître de l'ouvrage afin qu'il puisse la consulter et donner son accord par rapport au projet.

Pré-condition : Le système est alimenté et fonctionne correctement.
L'utilisateur qui demande l'accord du maître de l'ouvrage est un architecte.

Post-condition : Le maître de l'ouvrage a donné son accord concernant la phase du projet qui lui a été soumise.
La maquette de référence et l'accord du maître de l'ouvrage sont archivés.

CHAPITRE 8 : Le niveau logique

Dans ce chapitre, nous allons modéliser de manière plus détaillée une transaction. Cette description consiste d'une part, à représenter la dynamique de la transaction par un diagramme d'activité et, d'autre part, à représenter chacun des messages intervenant dans la transaction par un diagramme de classe.

En ce qui concerne le choix de la transaction à décrire, nous avons opté pour la transaction « Demande de validation Stabilité », dans laquelle un Ingénieur stabilité élabore une proposition technique et souhaite que les informations relatives à cette proposition soient intégrées à la maquette de référence. Avant la mise à jour de la maquette de référence, les modifications doivent être soumises, pour validation, au gestionnaire de la maquette de référence.

Le diagramme d'activité est accompagné d'une description textuelle.

Pour que cette description ne soit pas trop abstraite, commençons par décrire le contexte dans lequel cette transaction va se dérouler.

Au départ, le gestionnaire de la maquette réserve un espace pour le projet et signale aux professionnels qu'il est possible de stocker des propositions sur le serveur de Projet et d'introduire des demandes de validation concernant les propositions.

Ensuite, l'architecte élabore une première maquette présentant clairement la conception de la future construction et la soumet au gestionnaire de la maquette pour qu'il la valide. Le gestionnaire de la maquette la valide et l'enregistre comme maquette de référence.

Les différents utilisateurs sont alors informés qu'il leur est possible de consulter la maquette de référence en lecture. Les bureaux d'études chargés de la conception du bâtiment vont dès lors pouvoir consulter et copier les objets de la maquette de référence dont ils ont besoins pour effectuer leurs études. Prenons par exemple le cas d'un ingénieur en stabilité, il va pouvoir filtrer la maquette de référence pour visualiser et importer uniquement les objets qui présentent un intérêt dans le cadre d'une étude de stabilité (par exemple, les murs porteur, les poutres,...) et il ne lui sera pas nécessaire de ressaisir les informations provenant de l'architecte.

Sur base de ces objets de références, l'ingénieur va pouvoir élaborer une structure pour le bâtiment. Lors de l'élaboration de sa solution, l'ingénieur peut rendre sa proposition visible à d'autres utilisateurs en déposant sa proposition sur le serveur de Projet et en précisant les utilisateurs qu'il autorise à accéder à sa proposition. Les différents acteurs ont alors accès à la proposition en lecture.

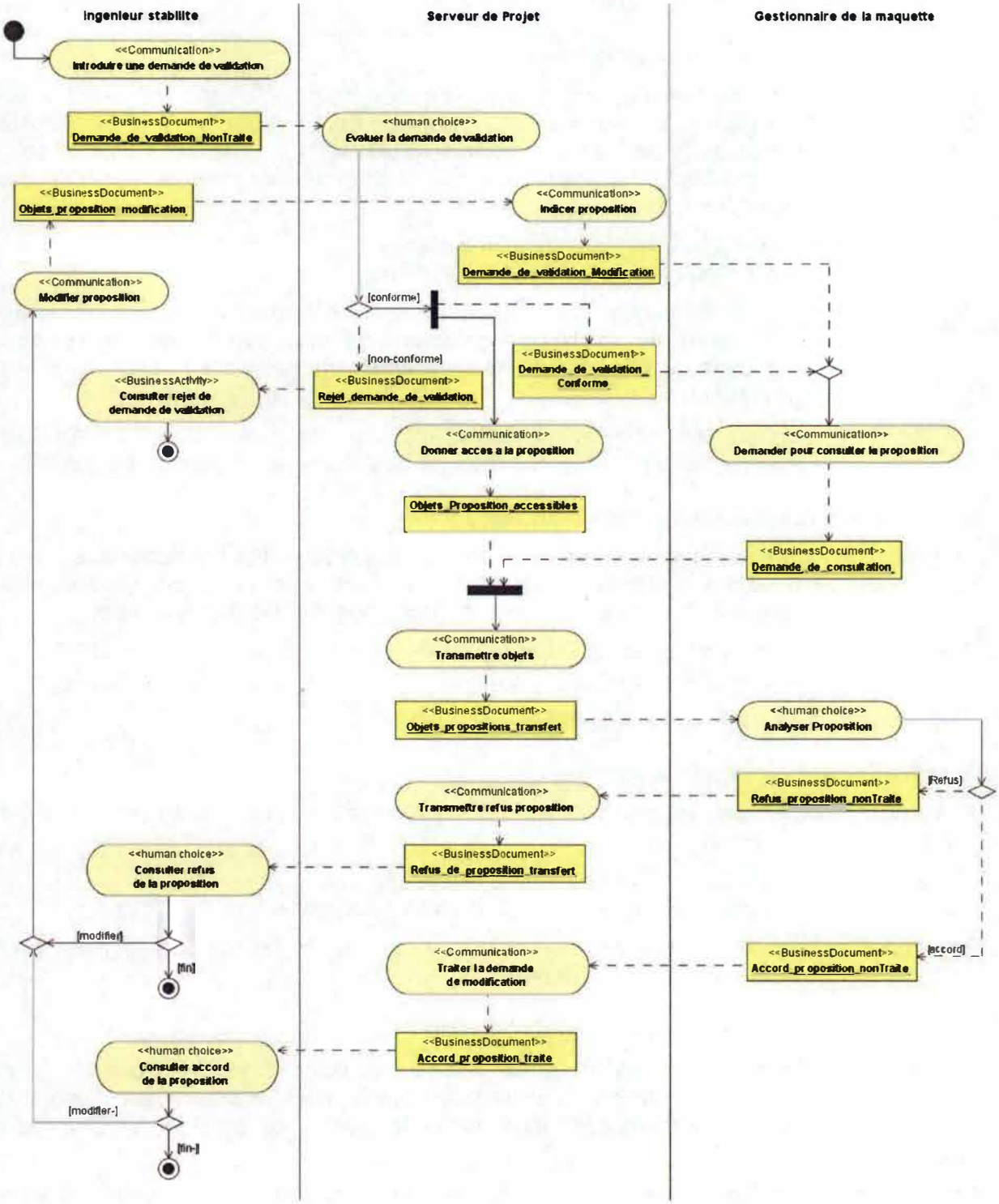
Lorsque l'ingénieur en stabilité juge que la proposition qu'il a produite est suffisamment mature et qu'elle peut être intégrée à la maquette de référence, il peut introduire une « demande de validation Stabilité » pour sa proposition.

La demande de validation de proposition est la première étape de la transaction que nous nous proposons maintenant de décrire.

8.1 Description de la transaction « Demande de validation stabilité»

La description de l'information dynamique liée à cette transaction est représentée dans un diagramme d'activité et est accompagnée d'une description textuelle des activités et des messages.

8.1.1 Diagramme d'activité



8.1.2 Description des activités

Action **« Introduire une demande de validation »** :

L'ingénieur en stabilité élabore une proposition (Stab1) de structure pour le bâtiment et souhaite que les informations relatives à cette proposition soient intégrées à la maquette de référence. Il décide donc d'introduire une demande de validation pour sa proposition (Stab1).

Pré condition : /

Post condition : Envoi d'un message « Demande_de_validation_NonTraite » à destination du serveur de projet.

Action **« Evaluer la demande de validation »** :

Le serveur de projet reçoit un message lui indiquant que l'ingénieur en stabilité désire qu'une de ses propositions (Stab1) soit soumise pour validation au gestionnaire de la maquette de référence. Le serveur de projet vérifie alors que la demande de validation concernant la proposition (Stab1) est conforme c'est-à-dire que la proposition existe et que l'ingénieur stabilité en est bien le propriétaire.

Précondition : Le serveur de projet a reçu un message « Demande_de_validation_NonTraite »

Post condition : Si la demande de validation est conforme, envoi du message « Demande_de_validation_Conforme » à destination du gestionnaire de la maquette de référence et déclenchement de l'action « Donner accès à la proposition »

Si la demande de validation est non-conforme, envoi du message « Rejet_demande_de_validation » à destination de l'ingénieur en stabilité.

Action **« Consulter rejet de demande de validation »** :

L'ingénieur en stabilité reçoit un message lui indiquant que la proposition technique (Stab1) qu'il a soumise n'est pas conforme et que sa demande de validation n'est pas recevable. Cette décision provenant du serveur de projet est accompagnée des motifs du rejet.

Pré condition : L'ingénieur en stabilité a reçu un message « Rejet_demande_de_validation »

Post condition : La transaction se termine.

Action **« Donner accès à la proposition »** :

Le serveur de projet rend les objets de la proposition (Stab1) accessibles au gestionnaire de la maquette de référence.

Pré condition : Le serveur de projet a reçu un message « Demande_de_validation_NonTraite » conforme.

Post condition : Les objets de la proposition sont accessibles en lecture pour le gestionnaire de la maquette de référence.

Action **« Demander pour consulter la proposition »** :

Le gestionnaire de la maquette de référence reçoit un message lui indiquant que l'ingénieur en stabilité a introduit une demande de validation concernant la proposition (Stab1). Le gestionnaire de la maquette de référence demande alors pour consulter la proposition (Stab1).

Pré condition : Le gestionnaire de la maquette de référence a reçu un message « Demande_de_validation_Conforme »

Post condition : Envoi d'un message « Demande_de_consultation » à destination du serveur de projet.

Action **« Transmettre objets »** :

Le serveur de projet reçoit un message lui indiquant que le gestionnaire de la maquette de référence veut consulter les objets relatifs à la proposition (Stab1). Le serveur de projet vérifie si ces objets sont accessibles pour le gestionnaire de la maquette de référence et ensuite, lui communique les objets demandés.

Pré condition : Le serveur de projet a reçu un message « Demande_de_consultation ».
Les objets de la proposition sont accessibles au gestionnaire de la maquette de référence.

Post condition : Envoi d'un message « Objets_proposition_transfert » à destination du gestionnaire de la maquette de référence.

Action **« Analyser proposition »** :

Le gestionnaire de la maquette de référence reçoit un message contenant les objets qui composent la proposition (Stab1). Ensuite, le gestionnaire analyse la proposition et sur base de cette analyse décide soit d'intégrer complètement ou partiellement la proposition, soit de refuser d'intégrer la proposition.

Lorsque le gestionnaire décide d'accepter partiellement la proposition, il doit spécifier les objets qu'il accepte d'intégrer dans la maquette de référence. Ce choix peut être effectué à différents niveaux dans l'arborescence des éléments IFC.

Pré condition : Le gestionnaire de la maquette de référence a reçu un message « Objets_proposition_transfert »

Post condition : Si le gestionnaire de la maquette de référence accepte globalement ou partiellement les modifications, envoi d'un message « Accord_proposition_nonTraite » à destination du serveur de projet.

Si le gestionnaire de la maquette de référence refuse les modifications, envoi d'un message « Refus_proposition_nonTraite » à destination du serveur de projet.

Action **« Transmettre refus proposition »** :

Le serveur de projet reçoit un message lui indiquant que la proposition a été refusée et transmet les informations concernant le refus de la proposition à l'ingénieur en stabilité.

Pré condition : Le serveur de projet a reçu un message « Refus_proposition_nonTraite ».

Post condition : Envoi du message « Refus_de_proposition_transfert » à destination de l'ingénieur stabilité.

Action **« Consulter refus de la proposition »** :

L'ingénieur en stabilité reçoit un message lui indiquant que la proposition technique (Stab1) qu'il a soumise n'est pas compatible avec la maquette de référence. Cette décision est accompagnée des motifs du refus du gestionnaire de la maquette de référence.

Pré condition : L'ingénieur en stabilité a reçu un message « Refus_de_proposition_transfert »

Post condition : Si l'ingénieur en stabilité décide d'adapter sa proposition en fonction des remarques formulées par le gestionnaire de la maquette de référence, il déclenche l'action modifier proposition.

Si l'ingénieur en stabilité estime qu'il ne peut pas adapter sa proposition en fonction des remarques du gestionnaire de la maquette de référence, il peut décider d'abandonner sa proposition. Dans ce cas, la transaction se termine.

Action « **Traiter la demande de modification** » :

Le serveur de projet reçoit un message lui indiquant que la demande de validation a été accordée totalement ou partiellement et lui demandant d'intégrer l'ensemble ou une partie des objets de la proposition (Stab1) dans la maquette de référence. Sur base de ce message, le serveur de projet intègre dans la maquette de référence les objets de la proposition qui sont renseignés dans le message.

Pré condition : Le serveur de projet a reçu un message « Accord_proposition_nonTraite »

Post condition : Le serveur de projet a intégré les objets de la proposition (Stab1) pour lesquels le gestionnaire de la maquette a donné son accord.

Envoi d'un message « Accord_proposition_traite » à destination de l'ingénieur en stabilité.

Action **« Consulter accord de la proposition »** :

L'ingénieur en stabilité reçoit un message lui indiquant que sa demande de validation est accordée partiellement ou totalement. Cette décision est accompagnée d'une part de la description des modifications qui ont été intégrées à la maquette de référence et d'autre part, pour les objets qui n'auraient pas été acceptés, des motifs du refus. En fonction de ce message, l'ingénieur peut décider soit de continuer à enrichir sa proposition, soit de clôturer sa proposition.

Les motifs de refus peuvent se présenter sous différentes formes, par exemple des notations textuelles que l'on peut lier aux objets de la proposition ou encore un fichier de surlignage applicable à une vue particulière du projet (vue en plan, en coupe, ...).

Pré condition : Le serveur de projet a reçu un message « Accord_proposition_traite »

Post condition : Si l'ingénieur en stabilité décide d'adapter sa proposition en fonction des remarques formulées par le gestionnaire de la maquette de référence, il déclenche l'action modifier proposition.

Si l'ingénieur en stabilité décide de clôturer la proposition car il estime que l'adaptation de la maquette est satisfaisante ou qu'il ne peut pas adapter sa proposition en fonction des remarques du gestionnaire de la maquette de référence, alors il peut mettre fin à la transaction.

Action **« Modifier proposition »** :

Suite à un accord de proposition ou à un refus de proposition, l'ingénieur en stabilité modifie les objets de la proposition (Stab1). Une fois la proposition modifiée, l'ingénieur en stabilité communique la nouvelle version de la proposition au serveur de projet.

Pré condition : Suite à un refus de proposition, l'ingénieur en stabilité a décidé de modifier sa proposition.
ou
Suite à un accord de proposition, l'ingénieur en stabilité a décidé d'enrichir la proposition et donc de la modifier.

Post condition : Envoi d'un message « Objets_proposition_modification » à destination du serveur de projet.

Action **« Indicer proposition »** :

Le serveur de projet reçoit un message lui indiquant que la proposition a été modifiée. Le serveur de projet archive la proposition précédente et rend la proposition modifiée accessible en lui affectant un indice incrémenté (Stab2).

Pré condition : Le serveur de projet a reçu un message « Objets_proposition_modification »

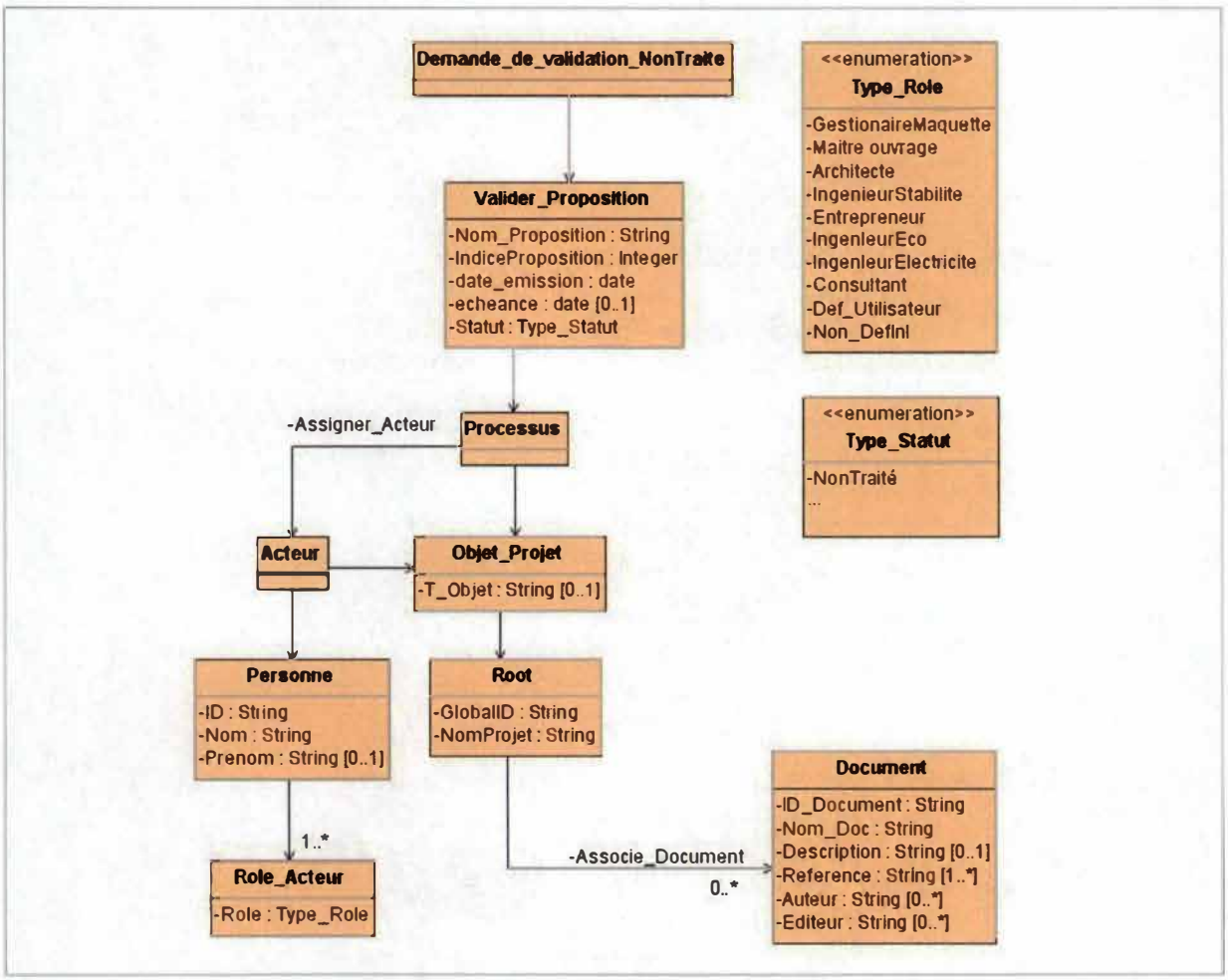
Post condition : La proposition précédente (Stab1) est archivée.
La proposition mise à jour est indicée (Stab2).
Envoi du message « Demande_de_validation_modification » à destination du gestionnaire de la maquette de référence.

8.2 Contenu des messages

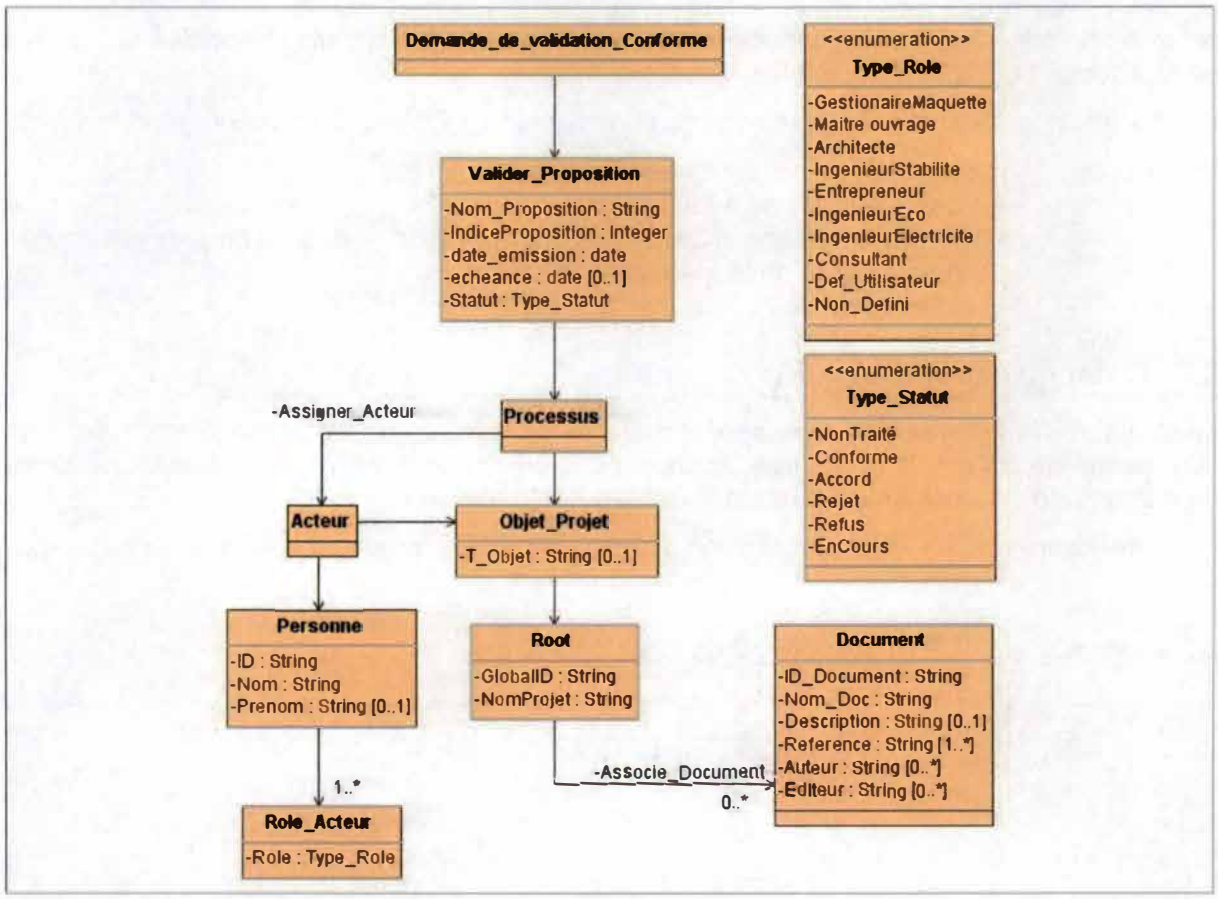
La structure de chacun des messages échangés est ensuite définie explicitement dans un diagramme de classe UML. La spécification est en réalité une simple sélection d'une partie du diagramme de classe représentant le domaine.

Dans le cadre de ce travail, nous nous limiterons à la modélisation des 4 messages suivants :

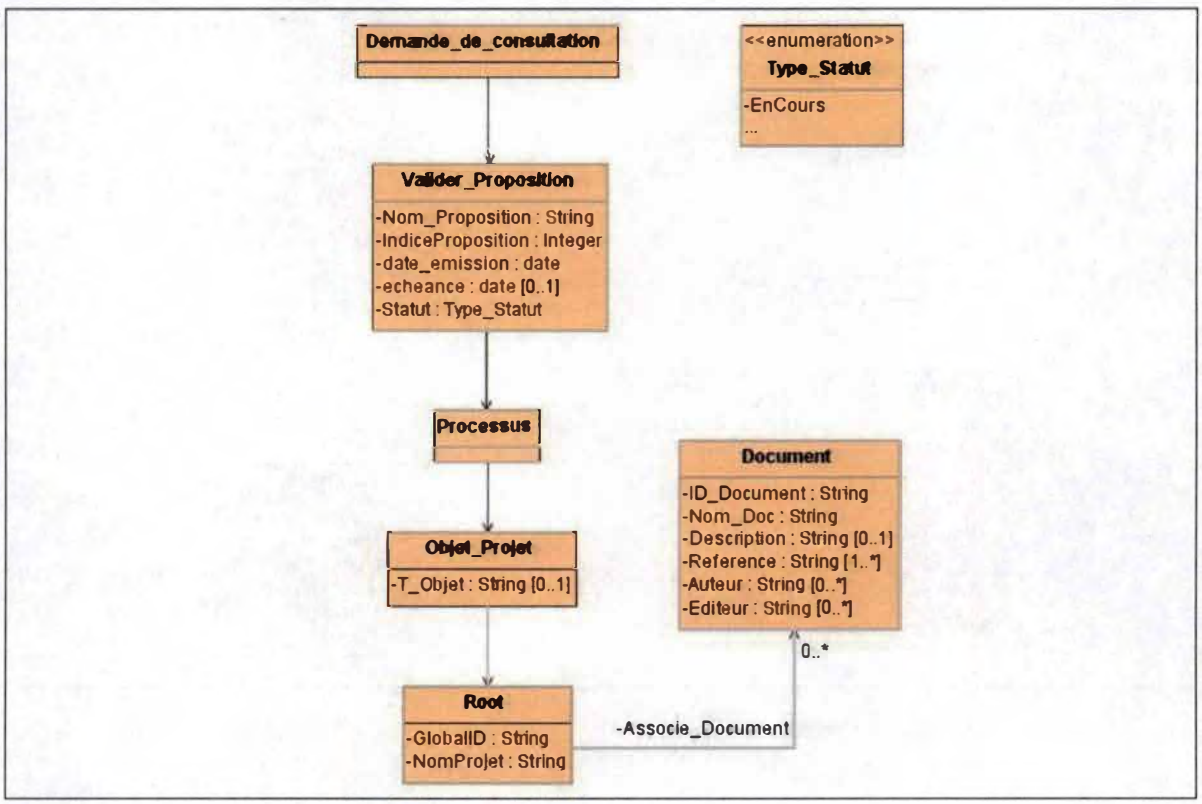
8.2.1 Message : « Demande_de_validation_NonTraite »



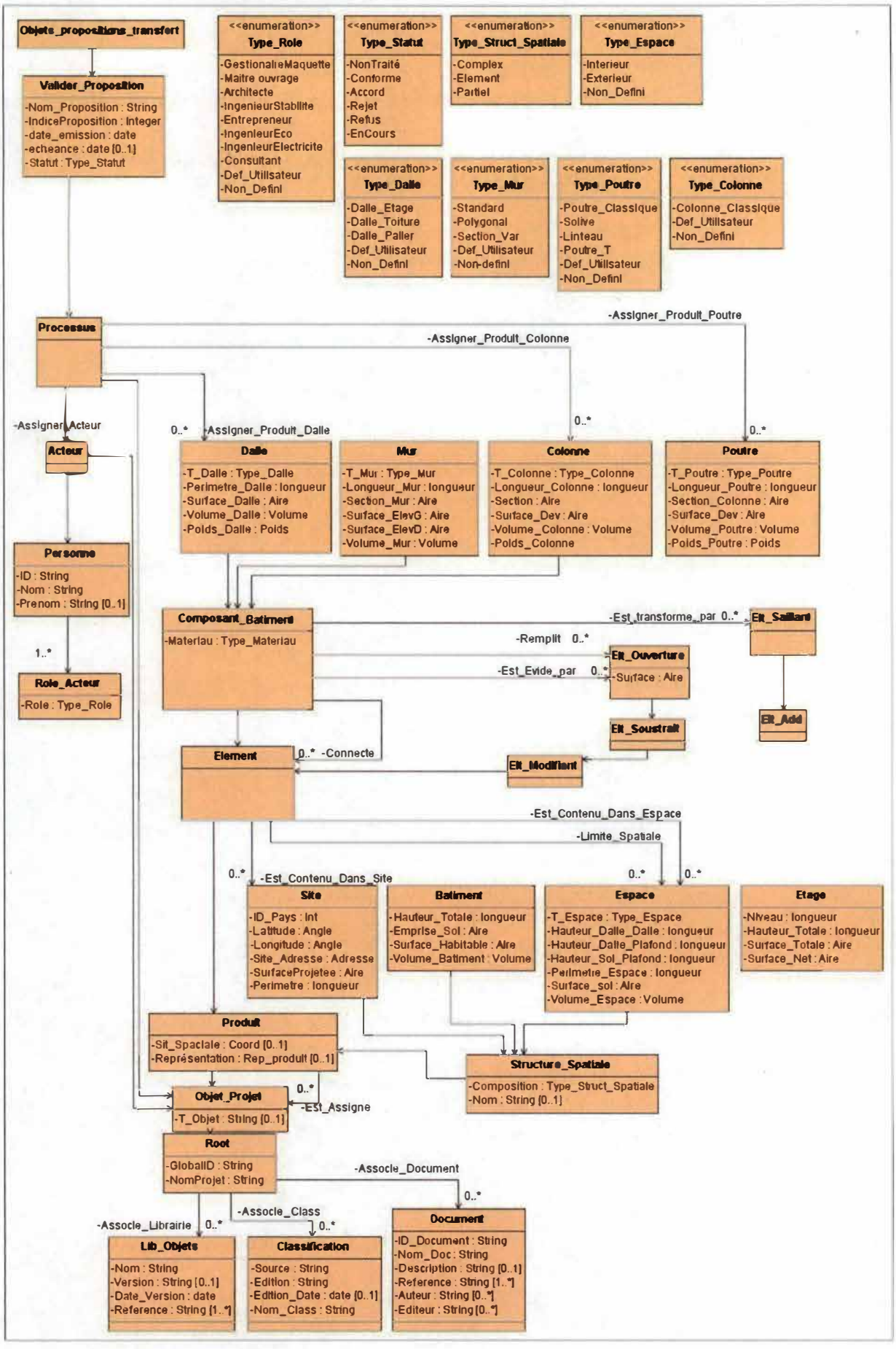
8.2.2 Message « Demande_de_validation_Conforme »



8.2.3 Message « Demande_de_consultation »



8.2.4 Message « Objets_propositions_transfert »



CHAPITRE 9 : Le niveau physique

Dans ce chapitre, nous allons présenter le code associé à la dynamique de la transaction et aux messages.

La traduction de la transaction et des messages est réalisée automatiquement à l'aide du logiciel EFFICIENT.

9.1 Traduction de la dynamique de la transaction en XPDL

Voir « Annexe 1 » sur le CD-Rom joint à ce mémoire.

9.2 Traduction des messages en Schémas XML

Voir « Annexe 2 » sur le CD-Rom joint à ce mémoire.

CONCLUSION

Au terme de ce travail, nous allons tenter d'en dresser le bilan, d'en faire ressortir les idées les plus importantes.

L'objectif de ce mémoire était de formaliser une approche méthodologique à la conception et à la spécification de transactions e-business et ensuite d'appliquer cette méthode au domaine de la construction.

Dans un premier temps, nous avons décrit les outils nécessaires à la compréhension et à la modélisation de transactions e-business. L'approche de conception de transaction e-business qui a été adoptée dans ce travail repose sur deux standards : UML et XML. Le choix de UML et XML reflète la volonté de rester aligné sur les standards internationaux qui se dégagent en matière e-business.

Ensuite, nous avons défini la notion de transaction e-business, et établi un état de l'art concernant la mise en œuvre des solutions d'échange de données. Nous nous sommes particulièrement intéressés à l'initiative ebXML car elle propose une gestion de la séquence des échanges et une composition dynamique du contenu des messages en fonction des exigences des partenaires.

Sur cette base, nous avons défini une méthode de développement de transactions e-business. Cette méthodologie vise à favoriser la compréhension de la sémantique de la transaction et la compréhension de la sémantique des messages avant de passer à leur implémentation. Cette approche permet aux experts métier de formaliser les transactions qu'ils souhaitent mettre en place sans posséder un bagage technique important.
Cette méthode repose sur la production de modèles UML et leur transformation automatique en langage XML.

Nous avons ensuite appliqué la méthodologie de modélisation de transaction au domaine de la construction. Pour cela, nous avons tout d'abord décrit le contexte spécifique de notre étude et nous avons pu constater que ce secteur rencontre de nombreuses difficultés en ce qui concerne le partage de l'information entre les différents corps de métier. Le coût d'un projet de construction en est d'ailleurs affecté de manière significative. De plus à l'heure actuelle, les applications utilisées dans le secteur du bâtiment font montre de faibles dispositions et de capacités limitées en matière d'échanges et de partage d'informations relatives aux projets de construction. Des problèmes importants dans la gestion opérationnelle des projets en découlent.

Un besoin urgent existe donc et un large consensus entre les professionnels du secteur s'est dégagé pour mettre en place des solutions adaptées, notamment le modèle IFC. Nous nous sommes appuyés sur ce modèle afin de structurer l'information descriptive du bâtiment en vue des échanges au travers d'un diagramme de classe.

Les normes d'échanges en cours de réalisation permettent d'envisager l'émergence d'une maquette numérique du projet " bâtiment " mais comme nous l'avons signalé dans le cadre de ce mémoire, la maquette numérique n'est pas suffisante, encore faut-il être capable de gérer les informations contenues dans cette maquette.

C'est dans cette optique de gestion des échanges d'informations que s'inscrit ce travail.
Le système envisagé se base sur des pratiques de collaboration et s'adapte aux pratiques actuelles sans être trop contraignant pour les utilisateurs. En effet, cette approche présente une certaine souplesse et garantit d'être effective dans un maximum de situations et pour un maximum d'acteurs.

Après avoir modélisé les concepts du domaine et les services nécessaires à la gestion de la maquette numérique, nous avons modélisé une transaction décrite par les flux d'informations et les messages échangés entre acteurs. Cette transaction a été développée à titre

d'exemple afin que le lecteur puisse mieux comprendre d'une part la méthodologie de développement de transaction e-business et d'autre part la problématique des échanges d'informations envisagée dans le contexte de la maquette numérique.

Ce travail nous a permis de mettre en évidence des avancées et des perspectives importantes pour l'évolution des méthodologies de travail des acteurs du bâtiment. Les principales perspectives sont les suivantes :

D'un point de vue technique, pour aboutir à la maquette numérique, certaines technologies doivent encore évoluer. Les serveurs Internet doivent encore progresser pour obtenir une rapidité de réponse encore meilleure, notamment pour les projets de grande taille. Dans un même temps, il est utile de doter les serveurs de systèmes d'interrogation évolués (Services Web). Ces services Web permettront, via Internet, de gérer les objets du projet, d'échanger de données et d'accéder de manière sélective aux informations du bâtiment. Ces fonctionnalités devront être intuitives et non contraignantes pour l'utilisateur. C'est dans ce cadre que s'inscrit ce mémoire.

Du point de vue des utilisateurs finaux (les professionnels de la construction). La rationalisation de l'utilisation des outils informatiques a un grand rôle à jouer dans l'amélioration de la productivité et de la qualité. L'adoption de technologies informatiques, ambitieuses mais réalistes, telles que la maquette numérique, vont contribuer à ces améliorations.

Cependant, ces technologiques induisent inévitablement un investissement matériel et temporel de la part des utilisateurs dont les bénéfices ne seront pas perceptibles directement. Il conviendra donc de leur faire prendre conscience des gains qu'ils pourront en tirer dans un avenir proche.

Par ailleurs, on ne peut pas dissocier les technologies et les méthodes de travail. C'est pourquoi, l'apport de la maquette numérique impose de **sensibiliser** et de faire comprendre aux acteurs de la construction, ce qu'est le travail coopératif basé sur le numérique, pour rapidement les préparer à de nouvelles pratiques professionnelles. L'objectif préalable consiste à rassembler tous les acteurs[8], à les sensibiliser aux technologies de l'information afin de travailler ensemble, en coopération et en partageant les informations du projet. Cette approche nécessite un changement des habitudes et des mentalités pour tous les acteurs.

Le problème majeur lié à la maquette numérique sera d'**accompagner le changement** dans la façon d'aborder le projet et de passer d'une culture d'échange de plans à celles du partage d'objets décrivant le bâtiment tout au long de son cycle de vie.

Dans ce contexte, l'animateur du projet Efficient peut s'avérer très utile car il permet aux acteurs de modéliser et de valider de manière coopérative les échanges d'informations. Une telle approche comporte certain nombre d'avantages pour les acteurs du secteur de la construction :

- Cette approche structurante pourrait faire prendre conscience aux acteurs de la construction qu'à l'heure actuelle, il n'existe pour ainsi dire aucune procédure permettant de gérer les échanges d'informations dans le cadre d'un projet.
- La méthode implique les futurs utilisateurs dans l'élaboration du système et diminue ainsi la résistance au changement. Il faut noter que la résistance au changement est très importante dans le domaine de la construction.
- Les transactions sont directement modélisées par les experts métiers, qui sont les mieux placés pour décrire leur travail.
- L'animateur permet, pour chaque transaction, de tester et le cas échéant d'adapter la séquence des messages et le contenu des messages. Les acteurs peuvent donc adapter le système à leurs pratiques.

[8] La maquette numérique n'a de sens fonctionnel et économique, que si toute la chaîne des acteurs y adhère.

- La méthodologie envisagée permet d'adapter les transactions et les messages en fonction d'un projet spécifique, d'un type de marché spécifique ou encore d'un lieu géographique.

Nous sommes convaincus qu'une telle approche favoriserait l'adoption de la maquette numérique dans le secteur de la construction, ce qui permettrait de mieux structurer les messages échangés et leur séquence. On pourrait ainsi améliorer la collaboration, augmenter la productivité et la qualité des services offerts par les différents acteurs.

BIBLIOGRAPHIE

[Arnold 1998] Florian ARNOLD, Gert PODEHL
« Best of Both Worlds – A Mapping from EXPRESS-G to UML »
Research Group for Computer Application in Engineering Design Department of Mechanical and Chemical Engineering University of Kaiserslautern, 1998

[Billon 1999] Roland BILLON
« Comprendre les concepts des IFC, décrire son projet en vue des échanges. »
MEDI@CONSTRUCT, septembre 1999

[Boukhors 2002] Anelyse BOUKHORS, Alexandre KASZYCKY, Jérôme LAPLACE, Sandrine MUNEROT, Laurent POUBLAN.
« XML : La synthèse »
Dunod, Paris, 2002

[b-process 2002] EDI – b-process
« XML et EDI, la vision b-process »
www.b-process.com/ iso_album/annexes_vision_edixml_bp_f1.pdf , 2002

[Bray 2000] T. BRAY, J. PAOLI, C. M. SPERBERG-McQUEEN, E. MALER,
« Extensible Markup Language (XML) 1.0 (Second Edition) »,
http://www.w3.org/TR/REC-xml#dt-doctype , octobre 2000

[Carlson 2001] David CARLSON
« Modélisation d'applications XML avec UML »
Eyrolles – Paris, octobre 2001

[CEFACT 2001] CEFACT
« UN/CEFACT's Modelling Methodology. »
www.unece.org/cefact/docum/download/00bp030.doc , novembre 2001

[Chauvet 2001] J.M. CHAUVET
« Services Web avec SOAP, WSDL, ebXML,... »
Eyrolles – Paris, Octobre 2001

[Dubois 2001] Eric DUBOIS
« Cours de première Licence en Informatique à Horaire Décalé : Conception des systèmes d'information »
Facultés Universitaires Notre Dame de la Paix, Namur, 2001

[Dubois 2002] Eric DUBOIS
« Cour de 3e cycle universitaire "Informatique & Innovation": Conception et développement de transactions e-business »
Centre de Recherche Public Henri Tudor, Luxembourg, 2002

[Gregoire 2004] Bertrand GREGOIRE, Sophie RAMEL, Michael SCHMITT, Pierre Brimont, Eric DUBOIS,
« Efficient : atelier de prototypage de transactions d'e-commerce »
Workshop ECI 2004, INSA, Lyon, 2004

[ebXML 2003] UN/CEFACT et OASIS,
« Electronic Business XML »,
http://www.ebxml.org , Dernière mise à jour, 2003

[EDIFRANCE 2002] EDIFRANCE
« Comprendre XML pour les échanges électroniques professionnels »
Une coédition Gencod EAN France et Edifrance, mai 2002

[Eshuis 2003] Rik ESHUIS, Pierre Brimont, Eric DUBOIS, Bertrand GREGOIRE, Sophie RAMEL,
« Animating ebXML Translations with a Workflow Engine »
Robert Meersman, Zahir Tari, Douglas Schmidt et al., editors, Proc. CoopIS 2003, volume 2888, Lecture Notes in Computer Science, Springer, 2003

[Fannader 2002] Rémy FANNADER
« UML : Séquençage des modèles et plans de route »
Dunod, Paris, 2002

[Ferries 1997] Bernard FERRIES, Jean-Pierre BEDRUNE
« Présentation du modèle de données IFC, Rapprochement avec les modèles français. »
MIDIVALEUR, juin 1997

[Ferries 2003] Bernard FERRIES
« Faciliter l'usage des logiciels de simulation grâce aux échanges d'objets IFC»
LAURENTI
http://www.laurenti.com , janvier 2003

[Girard 2001] Didier GIRARD, Tanguy Crusson
« XML pour l'entreprise »
http://application-servers.com/livresblancs /xml/ , 2001

[Gosselin 2000] Philippe GOSSELIN
« La maîtrise du processus de construction.»
Société Royale Belge des Ingénieurs et des Industriels (SRBII), 2000

[Hancke 2002] Johann-Christian HANCKE,
« XML pour les débutants »,
KnowWare E.U.R.L., Ecuelles, 2002

[Ibrahim 2000] Bertrand IBRAHIM,
« Introduction à XML »,
Université de Genève, CUI,
http://cuisung.unige.ch/xml/Base.html , 2000

[Malcurat 2001] Olivier MALCURAT
« Thèse : Modélisation d'un environnement logiciel d'assistance au travail coopératif.dans le secteur de l'architecture et du BTP »
CRAI (UMR MAP 694), 2001

[Muller 2000] Pierre-Alain MULLER, Nathalie GAERTNER
« Modélisation objet avec UML »
Eyrolles, Paris, Octobre 2000

[OMG 2001] Spécidfications UML
« UML specification v1.4 »
http://www.omg.org , 2001

[Pierra 2000] G. PIERRA
« Spécification de données : Le langage EXPRESS. »
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique.
Laboratoire d'Informatique Scientifique et Industrielle, décembre 2000

[Ramel 2003] Sophie RAMEL
« Support groupe de travail e-business de 20/12/2004 »
Centre de Recherche Public Henri Tudor, Luxembourg, 2003

[Robbe 2002] Maxime ROBBE, Sébastien WALGRAFFE,
« Etude et expérimentation d'outils de publication, de transformation et de validation de document XML »
Facultés Universitaires Notre Dame de la Paix, Namur, 2002

[Schneider 1998] G. SCHNEIDER and J. WINTERS,
« Applying Uses Cases – A practical Guide »
Adisson Wesley, 1998

[Sperberg 2002] C. M. SPERBERG-McQUEEN, H. THOMPSON,
« W3C XML Schema »,
http://www.w3.org/XML/Schema , août 2002

[Swift 2000] SWIFT
« SwiftML design rules, technical specification »
SWIFT, La Hulpe, 2000

[XML 2002] W3C,
« eXtensible Markup Language (XML) »,
http://www.w3.org/XML/ , Dernière mise à jour, 2002

OMG Unified Modeling Language Specification

March 2003
Version 1.5
formal/03-03-01

An Adopted Formal Specification of the Object Management Group, Inc.

Copyright © 2000-2002 Alcatel
Copyright © 1997-2001 Computer Associates International Inc.
Copyright © 1997-2001 Electronic Data Systems Corporation
Copyright © 1997-2001 Hewlett-Packard Company
Copyright © 1997-2001 IBM Corporation
Copyright © 1997-2002, I-Logix
Copyright © 1997-2001 IntelliCorp
Copyright © 2000-2002 Kabira Technologies
Copyright © 2000-2002 Kennedy Carter
Copyright © 1997-2001 Microsoft Corporation
Copyright © 1997-2001 Object Management Group
Copyright © 1997-2001 Oracle Corporation
Copyright © 2000-2002 Project Technology
Copyright © 1997-2001 Ptech Inc.
Copyright © 1997-2001 Rational Software Corporation
Copyright © 1997-2001 Reich Technologies
Copyright © 1997-2001 Softeam
Copyright © 1997-2001 Taskon A/S
Copyright © 2000-2002 Telelogic
Copyright © 1997-2001 Unisys Corporation

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be

required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

Contents

Contents . . . . . **v**

Foreword . . . . . **xxv**

Preface . . . . . **xxvii**

1 UML Summary **1-1**

1.1 Overview 1-1

1.2 Primary Artifacts of the UML 1-2

1.2.1 UML-defining Artifacts 1-2

1.2.2 Development Project Artifacts 1-2

1.3 Motivation to Define the UML 1-3

1.3.1 Why We Model 1-3

1.3.2 Industry Trends in Software 1-3

1.3.3 Prior to Industry Convergence 1-4

1.4 Goals of the UML 1-4

1.5 Scope of the UML 1-6

1.5.1 Outside the Scope of the UML 1-7

1.5.2 Comparing UML to Other Modeling Languages . . . 1-8

1.5.3 Features of the UML 1-9

1.6 UML - Past, Present, and Future 1-11

1.6.1 UML 0.8 - 0.91 1-11

1.6.2 UML Partners 1-12

1.6.3 UML - Present and Future 1-13

2 UML Semantics **2-1**

Part 1 - Background

2.1 Introduction 2-2
2.1.1 Purpose and Scope 2-2
2.1.2 Approach 2-3
2.2 Language Architecture 2-4
2.2.1 Four-layer Metamodel Architecture 2-4
2.2.2 Package Structure 2-6
2.3 Language Formalism 2-7
2.3.1 Levels of Formalism 2-8
2.3.2 Package Specification Structure 2-9
2.3.3 Use of a Constraint Language 2-10
2.3.4 Use of Natural Language 2-10
2.3.5 Naming Conventions and Typography 2-11

Part 2 - Foundation

2.4 Foundation Package 2-11
2.5 Core 2-12
2.5.1 Overview 2-12
2.5.2 Abstract Syntax 2-12
2.5.3 Well-formedness Rules 2-51
2.5.4 Detailed Semantics 2-64
2.6 Extension Mechanisms 2-73
2.6.1 Overview 2-73
2.6.2 Abstract Syntax 2-75
2.6.3 Well-formedness Rules 2-80
2.6.4 Detailed Semantics 2-82
2.6.5 Notes 2-83
2.7 Data Types 2-85
2.7.1 Overview 2-85
2.7.2 Abstract Syntax 2-85

Part 3 - Behavioral Elements

2.8 Behavioral Elements Package 2-92
2.9 Common Behavior 2-93
2.9.1 Overview 2-93
2.9.2 Abstract Syntax 2-94
2.9.3 Well-formedness Rules 2-103
2.9.4 Detailed Semantics 2-109
2.10 Collaborations 2-111
2.10.1 Overview 2-111
2.10.2 Abstract Syntax 2-113
2.10.3 Well-formedness Rules 2-119

2.10.4 Detailed Semantics 2-124
2.10.5 Notes 2-129
2.11 Use Cases 2-129
2.11.1 Overview 2-129
2.11.2 Abstract Syntax 2-130
2.11.3 Well-formedness Rules 2-133
2.11.4 Detailed Semantics 2-135
2.11.5 Notes 2-140
2.12 State Machines 2-140
2.12.1 Overview 2-140
2.12.2 Abstract Syntax 2-141
2.12.3 Well-formedness Rules 2-150
2.12.4 Detailed Semantics 2-154
2.12.5 Notes 2-165
2.13 Activity Graphs 2-170
2.13.1 Overview 2-170
2.13.2 Abstract Syntax 2-170
2.13.3 Well-formedness Rules 2-175
2.13.4 Detailed Semantics 2-178
2.13.5 Notes 2-181
2.14 Actions 2-181
Part 4 - General Mechanisms
2.15 Model Management. 2-181
2.15.1 Overview 2-181
2.15.2 Abstract Syntax 2-182
2.15.3 Well-formedness Rules 2-186
2.15.4 Semantics 2-192
2.15.5 Notes 2-198
Part 5 - Actions
2.16 Action Package 2-199
2.17 Actions Overview 2-200
2.17.1 Action Metamodel 2-200
2.17.2 Design Principles and Rationale 2-201
2.17.3 The Actions 2-204
2.18 Action Conventions 2-207
2.18.1 Chapter Structure. 2-207
2.18.2 Description of a Class 2-208
2.19 Action Foundation. 2-211
2.19.1 Action Specification 2-211

2.19.2 Action Execution Model 2-214
2.19.3 Action Foundation Classes 2-219
2.20 Composite Actions 2-228
2.20.1 Composite Action Specification 2-228
2.20.2 Composite Action Execution 2-234
2.20.3 Composite Action Classes 2-239
2.21 Read and Write Actions 2-252
2.21.1 Object Actions 2-253
2.21.2 Attribute Actions 2-254
2.21.3 Association Actions 2-255
2.21.4 Variable Actions 2-259
2.21.5 Other Actions 2-260
2.21.6 Additional OCL Operations for Read and Write Actions 2-261
2.21.7 Read and Write Action Classes 2-263
2.22 Computation Actions 2-287
2.22.1 Computation actions 2-287
2.22.2 Computation Classes 2-289
2.23 Collection Actions 2-296
2.23.1 General Rules for Collection Actions 2-297
2.23.2 Collection Action Classes 2-298
2.24 Messaging Actions 2-311
2.24.1 Request 2-311
2.24.2 Asynchronous Invocation 2-312
2.24.3 Synchronous invocation 2-313
2.24.4 Request Handling 2-314
2.24.5 Reply Handling 2-315
2.24.6 Procedures 2-315
2.24.7 Performing requests 2-316
2.24.8 Effect Resolution 2-317
2.24.9 Operation Lookup 2-318
2.24.10 Transition Triggering 2-319
2.24.11 Direct Communication among Executions 2-319
2.24.12 Strong Typing 2-320
2.24.13 Transmitting messages 2-320
2.24.14 Return information 2-320
2.24.15 Messaging Classes 2-321
2.24.16 Optional Profile for Resolution of Operations and Signals 2-330
2.25 Jump Actions 2-332

2.25.1 Jumps 2-332
2.25.2 Break and Continue Statements 2-334
2.25.3 Exceptions 2-335
2.25.4 Jumps with Concurrent Executions 2-336
2.25.5 Jump Classes 2-336
2.25.6 Additional Jump Semantics for Actions Defined Elsewhere 2-340
2.25.7 Jump Value Classes 2-342

3 UML Notation Guide 3-1

Part 1 - Background

3.1 Introduction 3-5

Part 2 - Diagram Elements

3.2 Graphs and Their Contents 3-6
3.3 Drawing Paths 3-7
3.4 Invisible Hyperlinks and the Role of Tools 3-7
3.5 Background Information 3-8
3.5.1 Presentation Options 3-8
3.6 String 3-8
3.6.1 Semantics 3-8
3.6.2 Notation 3-8
3.6.3 Presentation Options 3-9
3.6.4 Examples 3-9
3.6.5 Mapping 3-9
3.7 Name 3-9
3.7.1 Semantics 3-9
3.7.2 Notation 3-9
3.7.3 Example 3-10
3.7.4 Mapping 3-10
3.8 Label 3-10
3.8.1 Semantics 3-10
3.8.2 Notation 3-10
3.8.3 Presentation Options 3-11
3.8.4 Example 3-11
3.9 Keywords 3-11
3.10 Expression 3-11
3.10.1 Semantics 3-11
3.10.2 Notation 3-12
3.10.3 Examples 3-12
3.10.4 Mapping 3-12

3.10.5 OCL Expressions 3-12
3.10.6 Selected OCL Notation 3-13
3.10.7 Examples 3-13
3.11 Note 3-13
3.11.1 Semantics 3-13
3.11.2 Notation 3-13
3.11.3 Presentation Options 3-13
3.11.4 Example 3-14
3.11.5 Mapping 3-14
3.12 Type-Instance Correspondence 3-14
Part 3 - Model Management
3.13 Package 3-16
3.13.1 Semantics 3-16
3.13.2 Notation 3-16
3.13.3 Presentation Options 3-17
3.13.4 Style Guidelines 3-17
3.13.5 Example 3-18
3.13.6 Mapping 3-19
3.14 Subsystem 3-19
3.14.1 Semantics 3-19
3.14.2 Notation 3-19
3.14.3 Presentation Options 3-20
3.14.4 Example 3-21
3.14.5 Mapping 3-24
3.15 Model 3-24
3.15.1 Semantics 3-24
3.15.2 Notation 3-24
3.15.3 Presentation Options 3-25
3.15.4 Example 3-25
3.15.5 Mapping 3-26
Part 4 - General Extension Mechanisms
3.16 Constraint and Comment 3-26
3.16.1 Semantics 3-26
3.16.2 Notation 3-27
3.16.3 Example 3-28
3.16.4 Mapping 3-28
3.17 Element Properties 3-29
3.17.1 Semantics 3-29
3.17.2 Notation 3-29

3.17.3 Presentation Options 3-30
3.17.4 Style Guidelines 3-31
3.17.5 Example. 3-31
3.17.6 Mapping 3-31
3.18 Stereotypes 3-31
3.18.1 Semantics 3-31
3.18.2 Notation. 3-31
3.18.3 Examples. 3-32
3.18.4 Mapping 3-33
Part 5 - Static Structure Diagrams
3.19 Class Diagram. 3-34
3.19.1 Semantics 3-34
3.19.2 Notation. 3-34
3.19.3 Mapping 3-34
3.20 Object Diagram. 3-35
3.21 Classifier. 3-35
3.22 Class 3-35
3.22.1 Semantics 3-35
3.22.2 Basic Notation. 3-36
3.22.3 Presentation Options 3-36
3.22.4 Style Guidelines 3-36
3.22.5 Example. 3-37
3.22.6 Mapping 3-37
3.23 Name Compartment 3-38
3.23.1 Notation. 3-38
3.23.2 Mapping 3-38
3.24 List Compartment 3-38
3.24.1 Notation. 3-38
3.24.2 Presentation Options 3-39
3.24.3 Example. 3-40
3.24.4 Mapping 3-41
3.25 Attribute 3-41
3.25.1 Semantics 3-41
3.25.2 Notation. 3-42
3.25.3 Presentation Options 3-43
3.25.4 Style Guidelines 3-44
3.25.5 Example. 3-44
3.25.6 Mapping 3-44
3.26 Operation. 3-44

3.26.1 Semantics 3-44
3.26.2 Notation. 3-44
3.26.3 Presentation Options 3-46
3.26.4 Style Guidelines 3-47
3.26.5 Example. 3-47
3.26.6 Mapping 3-47
3.27 Nested Class Declarations. 3-48
3.27.1 Semantics 3-48
3.27.2 Notation. 3-48
3.27.3 Mapping 3-48
3.28 Type and Implementation Class 3-49
3.28.1 Semantics 3-49
3.28.2 Notation. 3-49
3.28.3 Example. 3-50
3.28.4 Mapping 3-50
3.29 Interfaces. 3-50
3.29.1 Semantics 3-50
3.29.2 Notation. 3-51
3.29.3 Example. 3-51
3.29.4 Mapping 3-52
3.30 Parameterized Class (Template) 3-52
3.30.1 Semantics 3-52
3.30.2 Notation. 3-53
3.30.3 Presentation Options 3-53
3.30.4 Example. 3-54
3.30.5 Mapping 3-54
3.31 Bound Element 3-54
3.31.1 Semantics 3-54
3.31.2 Notation. 3-55
3.31.3 Style Guidelines 3-55
3.31.4 Example. 3-55
3.31.5 Mapping 3-55
3.32 Utility 3-56
3.32.1 Semantics 3-56
3.32.2 Notation. 3-56
3.32.3 Example. 3-56
3.32.4 Mapping 3-56
3.33 Metaclass. 3-57
3.33.1 Semantics 3-57
3.33.2 Notation. 3-57

3.33.3 Mapping 3-57
3.34 Enumeration 3-57
3.34.1 Semantics 3-57
3.34.2 Notation 3-57
3.34.3 Mapping 3-57
3.35 Stereotype Declaration 3-57
3.35.1 Semantics 3-57
3.35.2 Notation 3-58
3.35.3 Mapping 3-61
3.36 Powertype 3-61
3.36.1 Semantics 3-61
3.36.2 Notation 3-61
3.36.3 Mapping 3-61
3.37 Class Pathnames 3-61
3.37.1 Notation 3-61
3.37.2 Example 3-62
3.37.3 Mapping 3-62
3.38 Accessing or Importing a Package 3-62
3.38.1 Semantics 3-62
3.38.2 Notation 3-63
3.38.3 Example 3-64
3.38.4 Mapping 3-64
3.39 Object 3-64
3.39.1 Semantics 3-64
3.39.2 Notation 3-64
3.39.3 Presentation Options 3-65
3.39.4 Style Guidelines 3-66
3.39.5 Variations 3-66
3.39.6 Example 3-66
3.39.7 Mapping 3-66
3.40 Composite Object 3-67
3.40.1 Semantics 3-67
3.40.2 Notation 3-67
3.40.3 Example 3-67
3.40.4 Mapping 3-68
3.41 Association 3-68
3.42 Binary Association 3-68
3.42.1 Semantics 3-68
3.42.2 Notation 3-68

3.42.3 Presentation Options 3-69
3.42.4 Style Guidelines 3-69
3.42.5 Options 3-69
3.42.6 Example. 3-70
3.42.7 Mapping 3-70
3.43 Association End 3-71
3.43.1 Semantics 3-71
3.43.2 Notation. 3-71
3.43.3 Presentation Options 3-73
3.43.4 Style Guidelines 3-74
3.43.5 Example. 3-74
3.43.6 Mapping 3-74
3.44 Multiplicity 3-75
3.44.1 Semantics 3-75
3.44.2 Notation. 3-75
3.44.3 Style Guidelines 3-75
3.44.4 Example. 3-75
3.44.5 Mapping 3-76
3.45 Qualifier 3-76
3.45.1 Semantics 3-76
3.45.2 Notation. 3-76
3.45.3 Presentation Options 3-77
3.45.4 Style Guidelines 3-77
3.45.5 Example. 3-77
3.45.6 Mapping 3-77
3.46 Association Class 3-77
3.46.1 Semantics 3-77
3.46.2 Notation. 3-78
3.46.3 Presentation Options 3-78
3.46.4 Style Guidelines 3-78
3.46.5 Example. 3-78
3.46.6 Mapping 3-79
3.47 N-ary Association 3-79
3.47.1 Semantics 3-79
3.47.2 Notation. 3-79
3.47.3 Style Guidelines 3-79
3.47.4 Example. 3-80
3.47.5 Mapping 3-80
3.48 Composition 3-81
3.48.1 Semantics 3-81

3.48.2 Notation 3-81
3.48.3 Design Guidelines 3-82
3.48.4 Example 3-83
3.48.5 Mapping 3-84
3.49 Link 3-84
3.49.1 Semantics 3-84
3.49.2 Notation 3-84
3.49.3 Example 3-85
3.49.4 Mapping 3-85
3.50 Generalization 3-86
3.50.1 Semantics 3-86
3.50.2 Notation 3-86
3.50.3 Presentation Options 3-87
3.50.4 Example 3-88
3.50.5 Mapping 3-89
3.51 Dependency 3-90
3.51.1 Semantics 3-90
3.51.2 Notation 3-90
3.51.3 Presentation Options 3-91
3.51.4 Example 3-92
3.51.5 Mapping 3-93
3.52 Derived Element 3-93
3.52.1 Semantics 3-93
3.52.2 Notation 3-93
3.52.3 Style Guidelines 3-93
3.53 InstanceOf 3-93
3.53.1 Semantics 3-93
3.53.2 Notation 3-93
3.53.3 Mapping 3-93

Part 6 - Use Case Diagrams

3.54 Use Case Diagram 3-94
3.54.1 Semantics 3-94
3.54.2 Notation 3-94
3.54.3 Example 3-95
3.54.4 Mapping 3-95
3.55 Use Case 3-96
3.55.1 Semantics 3-96
3.55.2 Notation 3-96
3.55.3 Presentation Options 3-96

3.55.4 Style Guidelines 3-96
3.55.5 Mapping 3-97
3.56 Actor 3-97
3.56.1 Semantics 3-97
3.56.2 Notation 3-97
3.56.3 Presentation Options 3-97
3.56.4 Style Guidelines 3-97
3.56.5 Mapping 3-97
3.57 Use Case Relationships 3-97
3.57.1 Semantics 3-97
3.57.2 Notation 3-98
3.57.3 Example 3-99
3.57.4 Mapping 3-99
3.58 Actor Relationships 3-99
3.58.1 Semantics 3-99
3.58.2 Notation 3-99
3.58.3 Example 3-100
3.58.4 Mapping 3-100

Part 7 - Interaction Diagrams

3.59 Collaboration 3-101
3.59.1 Semantics 3-101
3.60 Sequence Diagram 3-102
3.60.1 Semantics 3-102
3.60.2 Notation 3-102
3.60.3 Presentation Options 3-102
3.60.4 Example 3-104
3.60.5 Mapping 3-106
3.61 Object Lifeline 3-108
3.61.1 Semantics 3-108
3.61.2 Notation 3-109
3.61.3 Presentation Options 3-109
3.61.4 Example 3-109
3.61.5 Mapping 3-110
3.62 Activation 3-110
3.62.1 Semantics 3-110
3.62.2 Notation 3-110
3.62.3 Example 3-111
3.62.4 Mapping 3-111
3.63 Message and Stimulus 3-111

3.63.1 Semantics 3-111
3.63.2 Notation. 3-111
3.63.3 Presentation options 3-111
3.63.4 Example. 3-113
3.63.5 Mapping 3-113
3.64 Transition Times 3-113
3.64.1 Semantics 3-113
3.64.2 Notation. 3-113
3.64.3 Presentation Options 3-114
3.64.4 Example. 3-114
3.64.5 Mapping 3-114
Part 8 - Collaboration Diagrams
3.65 Collaboration Diagram 3-114
3.65.1 Semantics 3-114
3.65.2 Notation. 3-114
3.65.3 Example. 3-116
3.65.4 Mapping 3-117
3.66 Pattern Structure 3-117
3.66.1 Semantics 3-117
3.66.2 Notation. 3-118
3.66.3 Mapping 3-121
3.67 Collaboration Contents 3-121
3.67.1 Semantics 3-121
3.67.2 Notation. 3-121
3.67.3 Mapping 3-122
3.68 Interactions 3-123
3.68.1 Semantics 3-123
3.68.2 Notation. 3-123
3.68.3 Mapping 3-124
3.68.4 Example. 3-124
3.69 Collaboration Roles. 3-124
3.69.1 Semantics 3-124
3.69.2 Notation. 3-124
3.69.3 Presentation options 3-125
3.69.4 Example. 3-126
3.69.5 Mapping 3-126
3.70 Multiobject 3-127
3.70.1 Semantics 3-127
3.70.2 Notation. 3-127

3.70.3 Example. 3-128
3.70.4 Mapping 3-128
3.71 Active object 3-128
3.71.1 Semantics 3-128
3.71.2 Notation. 3-128
3.71.3 Example. 3-129
3.71.4 Mapping 3-129
3.72 Message and Stimulus. 3-130
3.72.1 Semantics 3-130
3.72.2 Notation. 3-130
3.72.3 Presentation Options 3-133
3.72.4 Example. 3-133
3.72.5 Mapping 3-133
3.73 Creation/Destruction Markers 3-134
3.73.1 Semantics 3-134
3.73.2 Notation. 3-135
3.73.3 Presentation options 3-135
3.73.4 Example. 3-135
3.73.5 Mapping 3-135

Part 9 - Statechart Diagrams

3.74 Statechart Diagram 3-136
3.74.1 Semantics 3-136
3.74.2 Notation. 3-136
3.74.3 Mapping 3-137
3.75 State 3-137
3.75.1 Semantics 3-137
3.75.2 Notation. 3-138
3.75.3 Example. 3-139
3.75.4 Mapping 3-139
3.76 Composite States. 3-140
3.76.1 Semantics 3-140
3.76.2 Notation. 3-140
3.76.3 Examples. 3-141
3.76.4 Mapping 3-142
3.77 Events 3-142
3.77.1 Semantics 3-142
3.77.2 Notation. 3-143
3.77.3 Example. 3-144
3.77.4 Mapping 3-144

3.78 Simple Transitions. 3-145
3.78.1 Semantics 3-145
3.78.2 Notation. 3-145
3.78.3 Example. 3-146
3.78.4 Mapping 3-146
3.79 Transitions to and from Concurrent States 3-146
3.79.1 Semantics 3-146
3.79.2 Notation. 3-146
3.79.3 Example. 3-147
3.79.4 Mapping 3-147
3.80 Transitions to and from Composite States. 3-147
3.80.1 Semantics 3-147
3.80.2 Notation. 3-147
3.80.3 Presentation Options 3-148
3.80.4 Example. 3-149
3.80.5 Mapping 3-150
3.81 Factored Transition Paths 3-150
3.81.1 Semantics 3-150
3.81.2 Notation. 3-150
3.81.3 Examples. 3-151
3.82 Submachine States 3-152
3.82.1 Semantics 3-152
3.82.2 Notation. 3-152
3.82.3 Example. 3-153
3.82.4 Mapping 3-154
3.83 Synch States 3-154
3.83.1 Semantics 3-154
3.83.2 Notation. 3-154
3.83.3 Example. 3-155
3.83.4 Mapping 3-155

Part 10 - Activity Diagrams

3.84 Activity Diagram. 3-155
3.84.1 Semantics 3-155
3.84.2 Notation. 3-156
3.84.3 Example. 3-157
3.84.4 Mapping 3-158
3.85 Action state 3-158
3.85.1 Semantics 3-158
3.85.2 Notation. 3-158

3.85.3 Presentation options 3-158
3.85.4 Example. 3-158
3.85.5 Mapping 3-158
3.86 Subactivity state 3-159
3.86.1 Semantics 3-159
3.86.2 Notation. 3-159
3.86.3 Example. 3-159
3.86.4 Mapping 3-159
3.87 Decisions. 3-159
3.87.1 Semantics 3-159
3.87.2 Notation. 3-160
3.87.3 Example. 3-160
3.87.4 Mapping 3-160
3.88 Call States 3-161
3.88.1 Semantics 3-161
3.88.2 Notation. 3-161
3.88.3 Example. 3-161
3.88.4 Mapping 3-161
3.89 Swimlanes. 3-161
3.89.1 Semantics 3-161
3.89.2 Notation. 3-162
3.89.3 Example. 3-162
3.89.4 Mapping 3-163
3.90 Action-Object Flow Relationships 3-163
3.90.1 Semantics 3-163
3.90.2 Notation. 3-163
3.90.3 Example. 3-164
3.90.4 Mapping 3-164
3.91 Control Icons. 3-165
3.91.1 Notation. 3-165
3.91.2 Mapping 3-167
3.92 Synch States 3-168
3.93 Dynamic Invocation 3-168
3.93.1 Semantics 3-168
3.93.2 Notation. 3-168
3.93.3 Mapping 3-169
3.94 Conditional Forks 3-169

Part 11 - Implementation Diagrams

3.95 Component Diagram 3-169

3.95.1 Semantics 3-169
3.95.2 Notation. 3-169
3.95.3 Example. 3-170
3.95.4 Mapping 3-171
3.96 Deployment Diagram 3-171
3.96.1 Semantics 3-171
3.96.2 Notation. 3-172
3.96.3 Example. 3-172
3.96.4 Mapping 3-173
3.97 Node 3-173
3.97.1 Semantics 3-173
3.97.2 Notation. 3-173
3.97.3 Example. 3-173
3.97.4 Mapping 3-174
3.98 Component 3-174
3.98.1 Semantics 3-174
3.98.2 Notation. 3-175
3.98.3 Example. 3-175
3.98.4 Mapping 3-176

4 UML Example Profiles. 4-1

Example 1 - UML Profile for Software Development Processes

4.1 Introduction. 4-1
4.2 Summary of Profile 4-2
4.3 Stereotypes and Notation 4-2
4.3.1 Use Case Stereotypes 4-3
4.3.2 Analysis Stereotypes 4-4
4.3.3 Design Stereotypes 4-5
4.3.4 Implementation Stereotypes 4-6
4.3.5 Class Stereotypes. 4-7
4.3.6 Association Stereotypes 4-8
4.4 Well-formedness Rules 4-9
4.4.1 Generalization 4-9
4.4.2 Containment 4-9

Example 2 - UML Profile for Business Modeling

4.5 Introduction. 4-9
4.6 Summary of Profile 4-10
4.7 Stereotypes and Notation 4-10
4.7.1 Use Case Stereotypes 4-11

4.7.2 Organization Stereotypes 4-12
4.7.3 Class Stereotypes 4-13
4.7.4 Association Stereotypes 4-15
4.8 Well-formedness Rules 4-15
4.8.1 Generalization 4-15

5 UML Model Interchange 5-1
5.1 Overview 5-1
5.2 Model Interchange Using XMI 5-23
5.3 Model Interchange Using CORBA IDL 5-24

6 Object Constraint Language Specification 6-1
6.1 Overview 6-2
6.1.1 Why OCL? 6-2
6.1.2 Where to Use OCL 6-3
6.2 Introduction 6-3
6.2.1 Legend 6-3
6.2.2 Example Class Diagram 6-4
6.3 Relation to the UML Metamodel 6-4
6.3.1 Self 6-4
6.3.2 Specifying the UML context 6-5
6.3.3 Invariants 6-5
6.3.4 Pre- and Postconditions 6-6
6.3.5 Package context 6-6
6.3.6 General Expressions 6-7
6.4 Basic Values and Types 6-7
6.4.1 Types from the UML Model 6-8
6.4.2 Enumeration Types 6-8
6.4.3 Let Expressions and «definition» Constraints 6-8
6.4.4 Type Conformance 6-9
6.4.5 Re-typing or Casting 6-10
6.4.6 Precedence Rules 6-10
6.4.7 Use of Infix Operators 6-10
6.4.8 Keywords 6-11
6.4.9 Comment 6-11
6.4.10 Undefined Values 6-11
6.5 Objects and Properties 6-12
6.5.1 Properties 6-12
6.5.2 Properties: Attributes 6-12
6.5.3 Properties: Operations 6-12

6.5.4 Properties: Association Ends and Navigation 6-13
6.5.5 Navigation to Association Classes 6-15
6.5.6 Navigation from Association Classes 6-16
6.5.7 Navigation through Qualified Associations 6-16
6.5.8 Using Pathnames for Packages 6-17
6.5.9 Accessing overridden properties of supertypes 6-17
6.5.10 Predefined properties on All Objects 6-18
6.5.11 Features on Classes Themselves 6-19
6.5.12 Collections 6-19
6.5.13 Collections of Collections 6-21
6.5.14 Collection Type Hierarchy and Type Conformance Rules 6-21
6.5.15 Previous Values in Postconditions 6-21
6.6 Collection Operations 6-22
6.6.1 Select and Reject Operations 6-22
6.6.2 Collect Operation 6-24
6.6.3 ForAll Operation 6-25
6.6.4 Exists Operation 6-26
6.6.5 Iterate Operation 6-26
6.6.6 Iterators in Collection Operations 6-27
6.6.7 Resolving Properties 6-27
6.7 The Standard OCL Package 6-28
6.8 Predefined OCL Types 6-29
6.8.1 Basic Types 6-29
6.8.2 Collection-Related Types 6-36
6.9 Grammar 6-45

UML Standard Elements A-1

Action Language Examples B-1

B.1 The Action Languages B-1
B.2 Presentation of the Examples B-2
B.3 Control Structures B-3
B.4 Object Manipulation B-6
B.5 Messaging Actions B-19
B.6 Complete Example: The FFT B-26
B.6.1 The Fast Fourier Transform B-27
B.6.2 Illustrative Notation B-28
B.6.3 Discussion B-30
B.6.4 Implementation Using Memory Writes B-33

Glossary . . . . . Glossary -1

Index . . . . . Index -1

Foreword

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

The UML represents the culmination of best practices in practical object-oriented modeling. The UML is the product of several years of hard work, in which we focused on bringing about a unification of the methods most used around the world, the adoption of good ideas from many quarters of the industry, and, above all, a concentrated effort to make things simple.

We mean “we” in the most general sense. The three of us started the UML effort at Rational and were its original chief methodologists, but the final product was a team effort among many UML partners under the sponsorship of OMG. All partners came with their own perspectives, areas of concern, and areas of interest; this diversity of experience and viewpoints has enriched and strengthened the final result. We extend our personal thanks to everyone who was a part of making the UML a reality. We would like to thank Rational for giving us the opportunity to work freely so that we might focus on unification, and we want to recognize all the other companies representing the UML partners for seeing the importance of the UML to the industry as a whole and giving their representatives time to work on this project. We must also thank the OMG for providing the framework under which we could bring together many diverse opinions to develop a consensus result. We expect that OMG’s ownership of the UML standard and the public’s free access to it will ensure the widespread use and advancement of UML technology over the coming years.

In an effort that involved so many companies and individuals with so many agendas, one would think that the resulting product would be the software equivalent of a camel: a most dysfunctional-looking animal that appears to have been the work product of an

ill-formed committee of misfits. The UML most decidedly is not a random collection of political compromises. If anything, because of the focus we placed upon creating a complete and formal model, the UML is coherent and has harmony of design.

In this context it is also exciting to point out that the UML was developed alongside, and with the full collaboration, of the OMG's Meta-Object Facility (MOF) team. The MOF, which represents the state of the art in distributed object repository architectures, is OMG's adopted technology for modeling and representing metadata (including the UML metamodel) as CORBA objects. The UML and MOF standards are key building blocks of OMG's development environment for building and deploying distributed object systems.

It is a very real sign of maturity of the industry that the UML exists as a standard. At a time when software is increasingly more complex and more central to the mission of companies and countries, the UML comes at the right time to help organizations deal with this complexity. Already, without a lot of the fanfare or hype sometimes associated with programming languages, the UML is in use in hundreds (if not thousands) of projects around the world, a sign that it is part of the mainstream of engineering software.

Grady Booch

Ivar Jacobson

Jim Rumbaugh

Rational Software Corporation

Preface

About the Object Management Group (OMG)

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

Associated OMG Documents

The CORBA documentation set includes the following:

- CORBA: Common Object Request Broker Architecture and Specification contains the architecture and specifications for the Object Request Broker.
- CORBAservices: Common Object Services Specification contains specifications for the object services.
- CORBAfacilities: Common Facilities Architecture contains information about the design of Common Facilities; it provides the framework for Common Facility specifications.
- Object Management Architecture Guide defines the OMG’s technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. To obtain books in the documentation set, or other OMG publications, refer to the enclosed subscription card or contact the Object Management Group, Inc. at:

OMG Headquarters

250 First Avenue

Needham, MA 02494

Tel: +1-781-444-0404

Fax: +1-781-444-0320

pubs@omg.org

http://www.omg.org

OMG's adoption of the UML specification reduces the degree of confusion within the industry surrounding modeling languages. It settles unproductive arguments about method notations and model interchange mechanisms and allows the industry to focus on higher leverage, more productive activities. Additionally, it enables semantic interchange between visual modeling tools.

Introduction to OMG Modeling

The OMG Modeling documents describe the OMG standards for modeling distributed software architectures and systems along with their CORBA Interfaces. There are two complementary specifications:

- Unified Modeling Language Specification
- Meta-Object Facility Specification

The Unified Modeling Language (UML) Specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. The specification includes the formal definition of a common Object Analysis and Design (OA&D) metamodel, a graphic notation, and a CORBA IDL facility that supports model interchange between OA&D tools and metadata repositories. The UML provides the foundation for specifying and sharing CORBA-based distributed object models.

The Meta-Object Facility (MOF) Specification defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. These interoperable metamodels include the UML metamodel, the MOF meta-metamodel, as well as future OMG adopted technologies that will be specified using metamodels. The MOF provides the infrastructure for implementing CORBA-based design and reuse repositories. The MOF specifies precise mapping rules that enable the CORBA interfaces for metamodels to be automatically generated, thus encouraging consistency in manipulating metadata in all phases of the distributed application development cycle.

Since the UML and MOF are based on a four-layer metamodel architecture it is essential that the metamodels for each facility are architecturally aligned. For a description of the four layer metamodel architecture, please refer to Section 2.2, “Meta-data Architectures,” on page 2-1 in the MOF Specification. In order to achieve architectural alignment considerable effort has been expended so that the UML and MOF share the same core semantics. This alignment allows the MOF to reuse the UML notation for visualizing metamodels. In those areas where semantic differences are required, well-defined mapping rules are provided between the metamodels. The OMG distributed repository architecture, which integrates UML and MOF with CORBA is described in “Resolution of Technical Criteria” in the Preface of the MOF Specification.

As the first adopted technologies specified using a metamodeling approach, the UML and MOF establish a rigorous foundation for OMG’s metamodel architectures. Future metamodel standards should reuse their core semantics and emulate their systematic approach to architecture alignment.

Architectural Alignment of UML, MOF, and CORBA

Introduction

This section explains the architectural alignment of the OA&D Facility (OA&DF) metamodel and the MOF meta-metamodel, and their relationships to the OMA and CORBA object models. When discussing specific models, MOF corresponds to the MOF meta-metamodel also referred to as the MOF Model. The UML is used to refer to the proposed OA&DF metamodel.

As yet, there is not an MOF meta-metamodel standard or an OA&D metamodel standard. However, since each of these specifications has been unified, a proactive approach has been taken towards architectural alignment. Considerable structure sharing between the two specifications has been accomplished. As the OA&DF and MOF technologies evolve, additional alignment work will be addressed by standard OMG processes such as those for Revision Task Forces and subsequent RFPs.

The MOF and OA&DF alignment work has focused on aligning the metamodels and applying the MOF IDL Mapping for generating the CORBA IDL for both the MOF and UML models. This was accomplished by defining the MOF and UML models using the MOF and by generating the IDL interfaces based on the MOF specification. Note that both the MOF and OADF specifications use the UML notation for graphically defining the models.

In terms of abstraction levels and the kinds of meta-metaobjects used, the UML and MOF meta-metamodels are well aligned. There are significant advantages in aligning the OA&DF meta-metamodel with the MOF meta-metamodel. In the case of the MOF, meta-metamodel alignment facilitates interoperability between the OA&DF and the MOF. An example of OA&DF-MOF interoperability is the use of an MOF-compliant repository to store an OA&DF object model.

Alignment of the UML, MOF, and CORBA paves the way for future extensibility of CORBA in key areas such as richer semantics, relationships, and constraints. Likewise the longer-term benefits to UML and MOF include better recognition and addressing of distributed computing issues in developing CORBA-compliant systems.

Motivation

The primary reason for aligning the OA&DF metamodel with the MOF meta-metamodel is to facilitate interoperability between the two facilities using CORBA IDL. When considering interoperability between the OA&DF and the MOF, it is important to consider the difference in scope between the facilities. The MOF goal is to allow interoperability across the application development cycle by supporting the definition of multiple metamodels, whereas the OA&DF focuses on supporting the definition of a single OA&D metamodel. An example of OA&DF-MOF interoperability is the use of an MOF-compliant repository to store and interchange OA&DF object models.

The key motivation to align the MOF and OA&DF with CORBA is to address the requirement of aligning with CORBA and between the two facilities. In addition, the MOF and OA&DF (especially the UML) specifications signify years of modeling and metamodeling experience that are being integrated. As such, some of the key concepts in the UML and MOF are potential candidates to evolve the OMG Core object model and CORBA IDL in the future.

Approach

The UML and MOF are based on a four-layer metamodel architecture, where the MOF meta-metamodel is the meta-metamodel for the UML metamodel. As a result, the UML metamodel may be considered an instance-of the MOF meta-metamodel. This is sometimes referred to as loose metamodeling, where an M_n level model is an instance of an M_{n+1} level model.

Since the MOF and OA&DF have different scopes, and diverge in the area of relationships, we have not been able to apply strict metamodeling. In strict metamodeling, every element of an M_n level model is an instance of exactly one element of M_{n+1} level model. Consequently, there is not a strict isomorphic mapping between all the MOF meta-metamodel elements and the UML meta-metamodel elements. In principle strict metamodeling is difficult (or sometimes impossible to accomplish) as the complexity of new concepts (for example patterns and frameworks) continues to increase. In any case, using a small set of primitive concepts such as those defined in the MOF it is possible to define arbitrarily complex metamodels.

In spite of this, since the two models were designed to be interoperable, the two metamodels are structurally quite similar. The following sections compare the core MOF and UML modeling concepts, and contrast them with the OMA and CORBA/IDL core object models. The issues related to mapping metaclasses that are not isomorphic; for example, Association classes are also discussed.

The following tables are comparison tables that summarize the mappings of similar concepts across the meta-metamodeling layers. Where there is no analog for a concept, it will be identified and discussed in ”Issues Related to UML-MOF Mapping” on page xxxii..

Metamodel Comparison

Most of the metaobjects for the UML core metamodel and the MOF core meta-metamodel can be mapped to each other in a straightforward manner. Similarly, these metaobjects can also be mapped to the OMA and CORBA core object models in a direct way.

The following tables compare the core modeling concepts and the data types for these models.

UML Metamodel	**MOF Meta-metamodel**	**OMA Core Object Model CORBA Object Model**	**CORBA IDL**
Association (n-ary)	Association (binary)		
AssociationClass	NA		
AssociationEnd	AssociationEnd		
Attribute	Attribute	Attribute	Attribute
BehavioralFeature	BehavioralFeature		
Class	Class	Class	Interface (as Class)
Classifier	Classifier		
Constraint	Constraint		
DataType	DataType	Data type	Data type
Dependency (class)	/dependsOn (association)		
Exception	Exception		Exception
Feature	Feature		
GeneralizableElement	GeneralizableElement		
Generalization (class)	generalizes (association)	Generalization	Generalization
Interface	Class (as Interface)	Interface	Interface
ModelElement	ModelElement		
NA	Reference		
NA	Constant		Constant
Namespace	Namespace		~ IR Containers
Operation	Operation	Operation	Operation
Package	Package		Module

UML Metamodel	MOF Meta-metamodel	OMA Core Object Model CORBA Object Model	CORBA IDL
Parameter	Parameter	Parameter	Parameter
StructuralFeature	StructuralFeature		
Type (stereotype)	Class (as Type)	Type	Interface (as Type)

UML Metamodel	MOF Meta-metamodel	CORBA Object Model and IDL
AggregationKind	AggregationKind	
Boolean	CORBA Boolean	Boolean
Enumeration	CORBA Enum	Enum
Expression	NameType	
Integer	CORBA Short, Long, Unsigned Short, Unsigned Long, Double, Octet, Float	Short, Long, Unsigned Short, Unsigned Long, Double, Octet, Float
Multiplicity	MultiplicityType	
Name	NameType	Name
ParameterDirectionKind	DirectionKind	
ScopeKind	ScopeKind	
String	CORBA String, Char	String, Char
VisibilityKind	VisibilityKind	

The MOF supports the full range of CORBA data types as well as additional data types defined using the MOF primitive types. UML supports a subset of CORBA data types in its semantic model but mapping to a subset of specific CORBA types is clearly possible.

The following sections discuss issues related to areas where the mapping between metamodels is not direct.

Issues Related to UML-MOF Mapping

In general, the mapping from the UML meta-metamodel to the MOF meta-metamodel is straightforward.

A review of the previous comparison tables indicates that in most cases the mapping from the UML meta-metamodel to the MOF meta-metamodel is direct. In fact, for most of the core constructs there is a structural equivalency in the mapping.

The key differences are due to different usage scenarios of MOF and UML. The MOF needs to be simpler, directly implementable, and provide a set of CORBA interfaces for manipulating meta objects. The UML is used as a general-purpose modeling

language, with potentially many implementation targets. These differences are commonly observed in repository, meta-CASE, and modeling-tool implementations. The key differences are:

- The MOF only supports binary associations while UML supports higher-order (also referred to as 'N-ary') associations. This tradeoff was made because N-ary relationships are rarely used in metamodeling and the design goal was to keep the MOF interfaces simpler. We have anticipated extending the MOF to support higher order associations in future.
- Associations in the MOF are limited to simple associations and cannot contain features. Association Classes in UML can contain features (such as attributes). The MOF has been defined to be structurally extensible to full-blown association classes in the future by relaxing this constraint. UML Association Classes are modeled as MOF Classes with well-defined multiplicity constraints to ensure shared lifetime of features owned by the association.
- The MOF supports the concept of a Reference which allows direct navigation from one Classifier to another. The UML metamodel uses the Target AssociationEnd's 'name' property as a 'pseudo-attribute' for the same purpose, but navigates to another classifier through an Association. The concept of reference is widely used in object repositories, as well as object and object-relational databases and optimizes navigation across a metamodel.
- Some concepts such as Generalization, Dependency, and Refinement are reified as classes in UML, but implemented as Associations in the MOF. The MOF does not require the richness of UML in these areas.
- The MOF supports the full set of CORBA data types, whereas the UML metamodel does not define data types to this depth. A CORBA-specific implementation of UML is clearly possible by either creating the additional data types needed or by providing appropriate mappings.
- The UML has clearly defined the similarities of the key concepts of Class, Interface, and Type. The MOF and UML both use the Class concept as the most general of these in their respective models. The MOF specification is focused only on specification of metamodels and generation of CORBA interfaces; therefore, it does not deal with implementation concepts such as 'Methods.' UML clearly needs to support these concepts because UML can be used to model conceptual, logical, and implementation models. In this sense, the MOF uses the Class concept to define Types, since MOF Classes do not have any methods, just operations. This is consistent with the definition of UML Type as a stereotype of Class with a constraint that Types cannot contain methods. The MOF Class concept is rich enough to define Interfaces, and in fact the MOF specification itself clearly shows that an MOF Class can be mapped to multiple CORBA Interfaces.

  The previous table shows that the mapping of metadatatypes between the meta-metamodels is also straightforward. Here also there are more MOF meta-metaobjects than there are UML meta-metaobjects. The MOF supports the full range of CORBA data types as well as additional data types defined using the MOF primitive types. UML supports a subset of CORBA data types in its semantic model but maps to specific CORBA types in its corresponding interface model.

Relationship to Other Models

Secondary emphasis was placed on the architectural alignment with CASE Data Interchange Format (CDIF) and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP), both of which have influenced the metamodel architectures. CDIF provided many useful concepts for specifying robust stream-based interchange formats. Similarly, ODP furnished many useful ideas for specifying model viewpoints. The document entitled *Relationship of the UML to the RM-ODP* (ormsc/2001-01-01) satisfies the OMG policy regarding compatibility of submitted technology with the architecture for system distribution defined in RM-ODP.

Document Summary

This document is intended primarily as a precise and self-consistent definition of the UML's semantics and notation. The primary audience of this document consists of the Object Management Group, standards organizations, book authors, trainers, and tool builders. The authors assume familiarity with object-oriented analysis and design methods. The document is not written as an introductory text on building object models for complex systems, although it could be used in conjunction with other materials or instruction. The document will become more approachable to a broader audience as additional books, training courses, and tools that apply to UML become available.

The Unified Modeling Language specification defines compliance to the UML, covers the architectural alignment with other technologies, and is comprised of the following topics:

UML Summary (Chapter 1) - provides an introduction to the UML, discussing motivation and history.

UML Semantics (Chapter 2) - defines the semantics of the Unified Modeling Language. The UML is layered architecturally and organized by packages. Within each package, the model elements are defined in the following terms:

1. Abstract syntax	UML class diagrams are used to present the UML metamodel, its concepts (metaclasses), relationships, and constraints. Definitions of the concepts are included.
2. Well-formedness rules	The rules and constraints on valid models are defined. The rules are expressed in English prose and in a precise Object Constraint Language (OCL). OCL is a specification language that uses logic for specifying invariant properties of systems comprising sets and relationships between sets.
3. Semantics	The semantics of model usage are described in English prose.

UML Notation Guide (Chapter 3) - specifies the graphic syntax for expressing the semantics described by the UML metamodel. Consequently, the UML Notation Guide's chapter should be read in conjunction with the UML Semantics chapter.

UML Example Profiles (Chapter 4) - shows two example profiles, the UML Profile for Software Development Processes and the UML Profile for Business Modeling.

UML Model Interchange (Chapter 5) - specifies the how UML models can be interchanged using XML Metadata Interchange (XMI) and CORBA IDL.

Object Constraint Language Specification (Chapter 6) - defines the Object Constraint Language (OCL) syntax, semantics, and grammar. All OCL features are described in terms of concepts defined in the UML Semantics.

In addition, there is an appendix of Standard Elements that defines standard stereotypes, constraints, and tagged values for UML, and a glossary of terms.

Dependencies between chapters

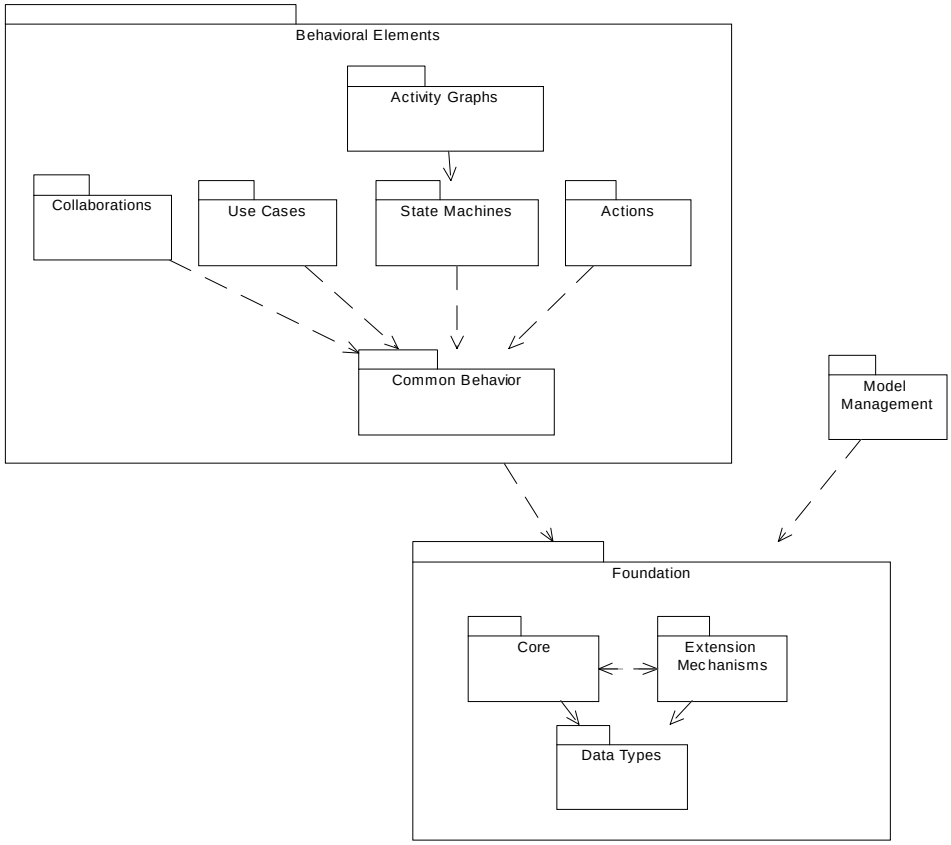
UML Semantics (Chapter 2) can stand on its own, relative to the others, with the exception of the *OCL Specification*. The semantics depends upon OCL for the specification of its well-formedness rules.

The *UML Notation Guide and UML Model Interchange* both depend on the UML Semantics. Specifying these separately permits their evolution in the most flexible way, even though they are not completely independent.

The specifications in the *UML Extensions* chapter depend on both the notation and semantics chapters.

Compliance to the UML

The UML and corresponding facility interface definition are comprehensive. However, these specifications are packaged so that subsets of the UML and facility can be implemented without breaking the integrity of the language. The UML Semantics is packaged as shown in the following figure.



UML Class Diagram Showing Package Structure

This packaging shows the semantic dependencies between the UML model elements in the different packages. The IDL in the facility is packaged almost identically. The notation is also “packaged” along the lines of diagram type. Compliance of the UML is thus defined along the lines of semantics, notation, and IDL.

Even if the compliance points are decomposed into more fundamental units, vendors implementing UML may choose not to fully implement this packaging of definitions, while still faithfully implementing some of the UML definitions. However, vendors who want to precisely declare their compliance to UML should refer to the precise language defined herein, and not loosely say they are “UML compliant.”

Compliance to the UML Semantics

The basic units of compliance are the packages defined in the UML metamodel. The full metamodel includes the corresponding semantic rigor defined in the UML Semantics chapter of this specification.

The class diagram illustrates the package dependencies, which are also summarized in the table below.

Package	Prerequisite Packages
DataTypes	
Core	DataTypes, Extension Mechanisms
Extension Mechanisms	Core, DataTypes
Common Behavior	Foundation
State Machines	Common Behavior, Foundation
Activity Graphs	State Machines, Foundation
Collaborations	Common Behavior, Foundation
Use Cases	Common Behavior, Foundation
Model Management	Foundation
Actions	Common Behavior, Foundation

Complying with a package requires complying with the prerequisite package.

The semantics are defined in an implementation language-independent way. An implementation of the semantics, without consistent interface and implementation choices, does not guarantee tool interoperability. See Chapter 5, “UML Model Interchange.”

In addition to the above packages, compliance to a given metamodel package must load or know about the predefined UML standard elements (i.e., values for all predefined stereotypes, tags, and constraints). These are defined throughout the semantics and notation documents and summarized in the UML Standard Elements appendix. The predefined constraint values must be enforced consistent with their definitions. Having tools know about the standard elements is necessary for the full language and to avoid the definition of user-defined elements that conflict with the standard UML elements. Compliance to the UML Standard Elements and UML Standard Profiles is distinct from the UML Semantics, so not all tools need to know about the UML Standard Elements and UML Standard Profiles.

For any implementation of UML, it is optional that the tool implements the Object Constraint Language. A vendor conforming to OCL support must support the following:

- Validate and store syntactically correct OCL expressions as values for UML data types.
- Be able to perform a full type check on the object constraint expression. This check will test whether all features used in the expression are actually defined in the UML model and used correctly.

All tools conforming to the UML semantics are expected to conform to the following aspects of the semantics:

- abstract syntax; that is, the concepts, valid relationships, and constraints expressed in the corresponding class diagrams,
- well-formedness rules, and
- semantics of model usage.

However, vendors are expected to apply some discretion on how strictly the well-formedness rules are enforced. Tools should be able to report on well-formedness violations, but not necessarily force all models to be well formed. Incomplete models are common during certain phases of the development lifecycle, so they should be permitted.

Compliance to the UML Notation

The UML notation is an essential element of the UML to enable communication between team members. Compliance to the notation is optional, but the semantics are not very meaningful without a consistent way of expressing them.

Notation compliance is defined along the lines of the UML Diagrams types: use case, class, statechart, activity graph, sequence, collaboration, component, and deployment diagrams.

If the notation is implemented, a tool must enforce the underlying semantics and maintain consistency between diagrams if the diagrams share the same underlying model. By this definition, a simple “drawing tool” cannot be compliant to the UML notation.

There are many optional notation adornments. For example, a richly adorned class icon may include an embedded stereotype icon, a list of properties (tagged values and metamodel attributes), constraint expressions, attributes with visibilities indicated, and operations with full signatures. Complying with class diagram support implies the ability to support all of the associated adornments.

Compliance to the notation in the *UML Standard Profiles* is described separately.

Compliance to Model Interchange Using XMI

The UML XMI DTD (document number ad/01-02-16) supports all packages of the UML Interchange Metamodel, which is a MOF translation of the UML Semantics Metamodel. See Model Interchange Using XMI (section 5.2). Each package of the Interchange Metamodel defines a separate compliance option.

Compliance to Model Interchange Using CORBA IDL

A UML CORBAfacility must support the MOF Reflective interfaces. Supporting the reflective interfaces for the Core package is the most basic level of compliance.Support for each additional package is a separate compliance option. In addition, support for each package's specific IDL module defined in UML_1.4_CORBA_IDL.zip (document number ad/01-02-17) constitutes a separate compliance option.

Summary of Compliance Points

Compliance Point	Valid Options
Foundation	no/incomplete, complete, complete including IDL and/or XMI DTD
Common Behavior	no/incomplete, complete, complete including IDL and/or XMI DTD
State Machines	no/incomplete, complete, complete including IDL and/or XMI DTD
Activity Graphs	no/incomplete, complete, complete including IDL and/or XMI DTD
Collaboration	no/incomplete, complete, complete including IDL and/or XMI DTD
Use Cases	no/incomplete, complete, complete including IDL and/or XMI DTD
Model Management	no/incomplete, complete, complete including IDL and/or XMI DTD
UML Profiles	no/incomplete, complete
Use Case diagram	no/incomplete, complete
Class diagram	no/incomplete, complete
Statechart diagram	no/incomplete, complete
Activity Graph diagram	no/incomplete, complete
Sequence diagram	no/incomplete, complete
Collaboration diagram	no/incomplete, complete
Component diagram	no/incomplete, complete
Deployment diagram	no/incomplete, complete
Model Interchange Using XMI	no/incomplete, complete
Model Interchange Using CORBA IDL	no/incomplete, complete, reflective interfaces, package-based interfaces
OCL	no/incomplete, complete

Acknowledgements

The UML was crafted through the dedicated efforts of individuals and companies who find UML strategic to their future. This section acknowledges the efforts of these individuals who contributed to defining UML.

UML Core Team

The following persons were members of the core development team for the UML proposal or served on the first or second UML Revision Task Force:

- Colorado State University: Robert France
- Computer Associates: John Clark
- Concept 5 Technologies: Ed Seidewitz
- Data Access Corporation: Tom Digre
- Enea Data: Karin Palmkvist
- Hewlett-Packard Company: Martin Griss
- IBM Corporation: Steve Brodsky, Steve Cook
- I-Logix: Eran Gery, David Harel
- ICON Computing: Desmond D'Souza
- IntelliCorp and James Martin & Co.: James Odell
- Kabira Technologies: Conrad Bock
- Klasse Objecten: Jos Warmer
- MCI Systemhouse: Joaquin Miller
- OAO Technology Solutions: Ed Seidewitz
- ObjecTime Limited: John Hogg, Bran Selic
- Oracle Corporation: Guus Ramackers
- PLATINUM Technology Inc.: Dilhar DeSilva
- Rational Software: Grady Booch, Ed Eykholt, Ivar Jacobson, Gunnar Overgaard, Jim Rumbaugh
- SAP: Oliver Wiegert
- SOFTEAM: Philippe Desfray
- Sterling Software: John Cheesman, Keith Short
- Sun Microsystems: Peter Walker
- Telelogic: Cris Kobryn, Morgan Björkander
- Taskon: Trygve Reenskaug
- Unisys Corporation: Sridhar Iyengar, GK Khalsa, Don Baisley

UML 1.1 Semantics Task Force

During the final submission phase for UML 1.1, a team was formed to focus on improving the formality of the UML 1.0 semantics, as well as incorporating additional ideas from the partners. Under the leadership of Cris Kobryn, this team was very instrumental in reconciling diverse viewpoints into a consistent set of semantics, as expressed in the revised *UML Semantics*. Other members of this team were Dilhar DeSilva, Martin Griss, Sridhar Iyengar, Eran Gery, James Odell, Gunnar Overgaard, Karin Palmkvist, Guus Ramackers, Bran Selic, and Jos Warmer. Grady Booch, Ivar Jacobson, and Jim Rumbaugh also provided their expertise to the team.

UML Revision Task Force

After the adoption of the UML 1.1 proposal by the OMG membership in November, 1997, the OMG chartered a revision task force (RTF) to deal with bugs, inconsistencies, and other problems that could be handled without greatly expanding the scope of the original proposal. The RTF accepted public comments submitted to the OMG after adoption of the proposal. This document containing UML version 1.3 is the result of the work of the UML RTF. The results have been issued in several preliminary versions with minor changes expected in the final version. If you have a preliminary version of the specification, you can obtain an updated version from the OMG web site at www.omg.org.

Contributors and Supporters

We also acknowledge the contributions, influence, and support of the following individuals.

Jim Amsden, Hernan Astudillo, Colin Atkinson, Dave Bernstein, Philip A. Bernstein, Michael Blaha, Mike Bradley, Ray Buhr, Gary Cernosek, James Cerrato, Brian Cook, Magnus Christerson, Dai Clegg, Peter Coad, Derek Coleman, Ward Cunningham, Raj Datta, Mike Devlin, Philippe Desfray, Bruce Douglass, Nathan Dykman, Staffan Ehnebom, Maria Ericsson, Johannes Ernst, Don Firesmith, Martin Fowler, Adam Frankl, Eric Gamma, Dipayan Gangopadhyay, Garth Gullekson, Rick Hargrove, Tim Harrison, Richard Helm, Brian Henderson-Sellers, Michael Hirsch, Bob Hodges, Glenn Hollowell, Yves Holvoet, Jon Hopkins, John Hsia, Anders Ivner, Ralph Johnson, Stuart Kent, Anneke Kleppe, Philippe Kruchten, Paul Kyzivat, Martin Lang, Grant Larsen, Reed Letsinger, Mary Loomis, Jeff MacKay, Bev Macmaster, Robert Martin, Terrie McDaniel, Jim McGee, Bertrand Meyer, Mike Meier, Randy Messer, Greg Meyers, Fred Mol, Birger Moller-Pedersen, Luis Montero, Paul Moskowitz, Andy Moss, Jan Pachl, Paul Patrick, Woody Pidcock, Bill Premerlani, Jeff Price, Jerri Pries, Terry Quatrani, Mats Rahm, George Reich, Rich Reitman, Rudolf M. Riess, Erick Rivas, Kenny Rubin, Bernhard Rumpe, Jim Rye, Danny Sabbah, Tom Schultz, Gregson Siu, Jeff Sutherland, Dan Tasker, Dave Tropeano, Andy Trice, Dan Uhlar, John Vlissides, Larry Wall, Paul Ward, Diane White, Oliver Wiegert, Alan Wills, Rebecca Wirfs-Brock, Bryan Wood, Ed Yourdon, and Steve Zeigler.

References

[Bock/Odell 94]	C. Bock and J. Odell, “A Foundation For Composition,” *Journal of Object-Oriented Programming*, October 1994.
[Booch et al. 99]	Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.
[Cook 94]	S. Cook and J. Daniels, *Designing Object Systems: Object-oriented Modeling with Syntropy*, Prentice-Hall Object-Oriented Series, 1994.
[D’Souza 99]	D. D’Souza and A. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1999.
[Fowler 97]	M. Fowler with K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.
[Griss 96]	M. Griss, “Domain Engineering And Variability In The Reuse-Driven Software Engineering Business,” *Object Magazine*. December 1996.
[Harel 87]	D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming 8*, (1987), pp. 231-274.
[Harel 96a]	D. Harel and E. Gery, “Executable Object Modeling with Statecharts,” *Proc. 18th Int. Conf. Soft. Eng.*, Berlin, IEEE Press, March, 1996, pp. 246-257.
[Harel 96b]	D. Harel and A. Naamad, “The STATEMATE Semantics of Statecharts,” *ACM Trans. Soft. Eng. Method 5*:4, October 1996.
[Jacobson et al. 99]	Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process*, Addison Wesley, 1999.
[Malan 96]	R. Malan, D. Coleman, R. Letsinger et al, “The Next Generation of Fusion,” Fusion Newsletter, October 1996.
[Martin/Odell 95]	J. Martin and J. Odell, *Object-Oriented Methods, A Foundation*, Prentice Hall, 1995
[Ramackers 95]	Ramackers, G. and Clegg, D., “Object Business Modelling, requirements and approach” in Sutherland, J. and Patel, D. (eds.), *Proceedings of the OOPSLA95 Workshop on Business Object Design and Implementation*, Springer Verlag, publication pending.
[Ramackers 96]	Ramackers, G. and Clegg, D., “Extended Use Cases and Business Objects for BPR,” ObjectWorld UK ‘96, London, June 18-21, 1996.
[Rumbaugh et al. 99]	Jim Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[Selic et al. 94]	B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.
[Warmer et al. 99]	J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.
[UML Web Sites]	OMG UML Resource Page: www.omg.org/uml OMG UML RTF: *www.celigent.com/omg/umlrtf*

UML Summary 1

The UML Summary provides an introduction to the UML, discussing its motivation and history.

Contents

This chapter contains the following topics.

Topic	Page
“Overview”	1-1
“Primary Artifacts of the UML”	1-2
“Motivation to Define the UML”	1-3
“Goals of the UML”	1-4
“Scope of the UML”	1-6
“UML - Past, Present, and Future”	1-11

1.1 Overview

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.

1.2 Primary Artifacts of the UML

What are the primary artifacts of the UML? This can be answered from two different perspectives: the UML definition itself and how it is used to produce project artifacts.

1.2.1 UML-defining Artifacts

To aid the understanding of the artifacts that constitute the Unified Modeling Language itself, this document consists of chapters describing UML Semantics, UML Notation Guide, and UML Standard Profiles.

1.2.2 Development Project Artifacts

The choice of what models and diagrams one creates has a profound influence upon how a problem is attacked and how a corresponding solution is shaped. Abstraction, the focus on relevant details while ignoring others, is a key to learning and communicating. Because of this:

- Every complex system is best approached through a small set of nearly independent views of a model. No single view is sufficient.
- Every model may be expressed at different levels of fidelity.
- The best models are connected to reality.

In terms of the views of a model, the UML defines the following graphical diagrams:

- use case diagram
- class diagram
- behavior diagrams:
  - statechart diagram
  - activity diagram
  - interaction diagrams:
    - sequence diagram
    - collaboration diagram
- implementation diagrams:
  - component diagram
  - deployment diagram

Although other names are sometimes given to these diagrams, this list constitutes the canonical diagram names.

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and supporting tools will provide for a number of derivative views. These diagrams are further described in the UML Notation Guide (Chapter 3 of this specification).

A frequently asked question has been: Why doesn't UML support data-flow diagrams? Simply put, data-flow and other diagram types that were not included in the UML do not fit as cleanly into a consistent object-oriented paradigm. Activity diagrams and collaboration diagrams accomplish much of what people want from DFDs, and then some. Activity diagrams are also useful for modeling workflow.

1.3 Motivation to Define the UML

This section describes several factors motivating the UML and includes why modeling is essential. It highlights a few key trends in the software industry and describes the issues caused by divergence of modeling approaches.

1.3.1 Why We Model

Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building. Good models are essential for communication among project teams and to assure architectural soundness. We build models of complex systems because we cannot comprehend any such system in its entirety. As the complexity of systems increase, so does the importance of good modeling techniques. There are many additional factors of a project's success, but having a rigorous modeling language standard is one essential factor. A modeling language must include:

- Model elements — fundamental modeling concepts and semantics
- Notation — visual rendering of model elements
- Guidelines — idioms of usage within the trade

In the face of increasingly complex systems, visualization and modeling become essential. The UML is a well-defined and widely accepted response to that need. It is the visual modeling language of choice for building object-oriented and component-based systems.

1.3.2 Industry Trends in Software

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software. We look for techniques to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns, and frameworks. We also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, we recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing, and fault tolerance. Development for the worldwide web makes some things simpler, but exacerbates these architectural problems.

Complexity will vary by application domain and process phase. One of the key motivations in the minds of the UML developers was to create a set of semantics and notation that adequately addresses all scales of architectural complexity, across all domains.

1.3.3 Prior to Industry Convergence

Prior to the UML, there was no clear leading modeling language. Users had to choose from among many similar modeling languages with minor differences in overall expressive power. Most of the modeling languages shared a set of commonly accepted concepts that are expressed slightly differently in various languages. This lack of agreement discouraged new users from entering the object technology market and from doing object modeling, without greatly expanding the power of modeling. Users longed for the industry to adopt one, or a very few, broadly supported modeling languages suitable for general-purpose usage.

Some vendors were discouraged from entering the object modeling area because of the need to support many similar, but slightly different, modeling languages. In particular, the supply of add-on tools has been depressed because small vendors cannot afford to support many different formats from many different front-end modeling tools. It is important to the entire object industry to encourage broadly based tools and vendors, as well as niche products that cater to the needs of specialized groups.

The perpetual cost of using and supporting many modeling languages motivated many companies producing or using object technology to endorse and support the development of the UML.

While the UML does not guarantee project success, it does improve many things. For example, it significantly lowers the perpetual cost of training and retooling when changing between projects or organizations. It provides the opportunity for new integration between tools, processes, and domains. But most importantly, it enables developers to focus on delivering business value and gives them a paradigm to accomplish this.

1.4 Goals of the UML

The primary design goals of the UML are as follows:

- Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
- Furnish extensibility and specialization mechanisms to extend the core concepts.
- Support specifications that are independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the object tools market.
- Support higher-level development concepts such as components, collaborations, frameworks and patterns.
- Integrate best practices.

These goals are discussed in detail below.

Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models

It is important that the Object Analysis and Design (OA&D) standard supports a modeling language that can be used “out of the box” to do normal general-purpose modeling tasks. If the standard merely provides a meta-meta-description that requires tailoring to a particular set of modeling concepts, then it will not achieve the purpose of allowing users to exchange models without losing information or without imposing excessive work to map their models to a very abstract form. The UML consolidates a set of core modeling concepts that are generally accepted across many current methods and modeling tools. These concepts are needed in many or most large applications, although not every concept is needed in every part of every application. Specifying a meta-meta-level format for the concepts is not sufficient for model users, because the concepts must be made concrete for real modeling to occur. If the concepts in different application areas were substantially different, then such an approach might work, but the core concepts needed by most application areas are similar and should be supported directly by the standard without the need for another layer.

Furnish extensibility and specialization mechanisms to extend the core concepts

OMG expects that the UML will be tailored as new needs are discovered and for specific domains. At the same time, we do not want to force the common core concepts to be redefined or re-implemented for each tailored area. Therefore, we believe that the extension mechanisms should support deviations from the common case, rather than being required to implement the core modeling concepts themselves. The core concepts should not be changed more than necessary. Users need to be able to

- build models using core concepts without using extension mechanisms for most normal applications,
- add new concepts and notations for issues not covered by the core,
- choose among variant interpretations of existing concepts, when there is no clear consensus, and
- specialize the concepts, notations, and constraints for particular application domains.

Support specifications that are independent of particular programming languages and development processes

The UML must and can support all reasonable programming languages. It also must and can support various methods and processes of building models. The UML can support multiple programming languages and development methods without excessive difficulty.

Provide a formal basis for understanding the modeling language

Because users will use formality to help understand the language, it must be both precise and approachable; a lack of either dimension damages its usefulness. The formalisms must not require excessive levels of indirection or layering, use of low-level mathematical notations distant from the modeling domain, such as set-theoretic notation, or operational definitions that are equivalent to programming an

implementation. The UML provides a formal definition of the static format of the model using a metamodel expressed in UML class diagrams. This is a popular and widely accepted formal approach for specifying the format of a model and directly leads to the implementation of interchange formats. UML expresses well-formedness constraints in precise natural language plus Object Constraint Language expressions. UML expresses the operational meaning of most constructs in precise natural language. The fully formal approach taken to specify languages such as Algol-68 was not approachable enough for most practical usage.

Encourage the growth of the object tools market

By enabling vendors to support a standard modeling language used by most users and tools, the industry benefits. While vendors still can add value in their tool implementations, enabling interoperability is essential. Interoperability requires that models can be exchanged among users and tools without loss of information. This can only occur if the tools agree on the format and meaning of all of the relevant concepts. Using a higher meta-level is no solution unless the mapping to the user-level concepts is included in the standard.

Support higher-level development concepts such as components, collaborations, frameworks, and patterns

Clearly defined semantics of these concepts is essential to reap the full benefit of object-orientation and reuse. Defining these within the holistic context of a modeling language is a unique contribution of the UML.

Integrate best practices

A key motivation behind the development of the UML has been to integrate the best practices in the industry, encompassing widely varying views based on levels of abstraction, domains, architectures, life cycle stages, implementation technologies, etc. The UML is indeed such an integration of best practices.

1.5 Scope of the UML

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.

First and foremost, the Unified Modeling Language fuses the concepts of Booch, OMT, and OOSE. The result is a single, common, and widely usable modeling language for users of these and other methods.

Second, the Unified Modeling Language pushes the envelope of what can be done with existing methods. As an example, the UML authors targeted the modeling of concurrent, distributed systems to assure the UML adequately addresses these domains.

Third, the Unified Modeling Language focuses on a standard modeling language, not a standard process. Although the UML must be applied in the context of a process, it is our experience that different organizations and problem domains require different processes. (For example, the development process for shrink-wrapped software is an interesting one, but building shrink-wrapped software is vastly different from building

hard-real-time avionics systems upon which lives depend.) Therefore, the efforts concentrated first on a common metamodel (which unifies semantics) and second on a common notation (which provides a human rendering of these semantics). The UML authors promote a development process that is use-case driven, architecture centric, and iterative and incremental.

The UML specifies a modeling language that incorporates the object-oriented community's consensus on core modeling concepts. It allows deviations to be expressed in terms of its extension mechanisms. The Unified Modeling Language provides the following:

- Semantics and notation to address a wide variety of contemporary modeling issues in a direct and economical fashion.
- Semantics to address certain expected future modeling issues, specifically related to component technology, distributed computing, frameworks, and executability.
- Extensibility mechanisms so individual projects can extend the metamodel for their application at low cost. We don't want users to directly change the UML metamodel.
- Extensibility mechanisms so that future modeling approaches could be grown on top of the UML.
- Semantics to facilitate model interchange among a variety of tools.
- Semantics to specify the interface to repositories for the sharing and storage of model artifacts.

1.5.1 Outside the Scope of the UML

1.5.1.1 Programming Languages

The UML, a visual modeling language, is not intended to be a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages. The UML is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system, but it does draw the line as you move toward code. For example, complex branches and joins are better expressed in a textual programming language. The UML does have a tight mapping to a family of object languages so that you can get the best of both worlds.

1.5.1.2 Tools

Standardizing a language is necessarily the foundation for tools and process. Tools and their interoperability are very dependent on a solid semantic and notation definition, such as the UML provides. The UML defines a semantic metamodel, not a tool interface, storage, or run-time model, although these should be fairly close to one another.

The UML documents do include some tips to tool vendors on implementation choices, but do not address everything needed. For example, they don't address topics like diagram coloring, user navigation, animation, storage/implementation models, or other features.

1.5.1.3 Process

Many organizations will use the UML as a common language for its project artifacts, but will use the same UML diagram types in the context of different processes. The UML is intentionally process independent, and defining a standard process was not a goal of the UML or OMG's RFP.

The UML authors do recognize the importance of process. The presence of a well-defined and well-managed process is often a key discriminator between hyperproductive projects and unsuccessful ones. The reliance upon heroic programming is not a sustainable business practice. A process

- provides guidance as to the order of a team's activities,
- specifies what artifacts should be developed,
- directs the tasks of individual developers and the team as a whole, and
- offers criteria for monitoring and measuring a project's products and activities.

Processes by their very nature must be tailored to the organization, culture, and problem domain at hand. What works in one context (shrink-wrapped software development, for example) would be a disaster in another (hard-real-time, human-rated systems, for example). The selection of a particular process will vary greatly, depending on such things as problem domain, implementation technology, and skills of the team.

Booch, OMT, OOSE, and many other methods have well-defined processes, and the UML can support most methods. There has been some convergence on development process practices, but there is not yet consensus for standardization. What will likely result is general agreement on best practices and potentially the embracing of a process framework, within which individual processes can be instantiated. Although the UML does not mandate a process, its developers have recognized the value of a use-case driven, architecture-centric, iterative, and incremental process, so were careful to enable (but not require) this with the UML.

1.5.2 Comparing UML to Other Modeling Languages

It should be made clear that the Unified Modeling Language is not a radical departure from Booch, OMT, or OOSE, but rather the legitimate successor to all three. This means that if you are a Booch, OMT, or OOSE user today, your training, experience, and tools will be preserved, because the Unified Modeling Language is a natural evolutionary step. The UML will be equally easy to adopt for users of many other methods, but their authors must decide for themselves whether to embrace the UML concepts and notation underneath their methods.

The Unified Modeling Language is more expressive yet cleaner and more uniform than Booch, OMT, OOSE, and other methods. This means that there is value in moving to the Unified Modeling Language, because it will allow projects to model things they could not have done before. Users of most other methods and modeling languages will gain value by moving to the UML, since it removes the unnecessary differences in notation and terminology that obscure the underlying similarities of most of these approaches.

With respect to other visual modeling languages, including entity-relationship modeling, BPR flow charts, and state-driven languages, the UML should provide improved expressiveness and holistic integrity.

Users of existing methods will experience slight changes in notation, but this should not take much relearning and will bring a clarification of the underlying semantics. If the unification goals have been achieved, UML will be an obvious choice when beginning new projects, especially as the availability of tools, books, and training becomes widespread. Many visual modeling tools support existing notations, such as Booch, OMT, OOSE, or others, as views of an underlying model; when these tools add support for UML (as some already have) users will enjoy the benefit of switching their current models to the UML notation without loss of information.

Existing users of any object method can expect a fairly quick learning curve to achieve the same expressiveness as they previously knew. One can quickly learn and use the basics productively. More advanced techniques, such as the use of stereotypes and properties, will require some study since they enable very expressive and precise models needed only when the problem at hand requires them.

1.5.3 Features of the UML

The goals of the unification efforts were to keep it simple, to cast away elements of existing Booch, OMT, and OOSE that didn't work in practice, to add elements from other methods that were more effective, and to invent new only when an existing solution was not available. Because the UML authors were in effect designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and over-engineering (having an icon for every conceivable modeling element). To that end, they were very careful about adding new things, because they didn't want to make the UML unnecessarily complex. Along the way, however, some things were found that were advantageous to add because they have proven useful in practice in other modeling.

There are several new concepts that are included in UML, including

- extensibility mechanisms (stereotypes, tagged values, and constraints),
- threads and processes,
- distribution and concurrency (e.g., for modeling ActiveX/DCOM and CORBA),
- patterns/collaborations,
- activity diagrams (for business process modeling),
- refinement (to handle relationships between levels of abstraction),

- interfaces and components, and
- a constraint language.

Many of these ideas were present in various individual methods and theories but UML brings them together into a coherent whole. In addition to these major changes, there are many other localized improvements over the Booch, OMT, and OOSE semantics and notation.

The UML is an evolution from Booch, OMT, OOSE, other object-oriented methods, and many other sources. These various sources incorporated many different elements from many authors, including non-OO influences. The UML notation is a melding of graphical syntax from various sources, with a number of symbols removed (because they were confusing, superfluous, or little used) and with a few new symbols added. The ideas in the UML come from the community of ideas developed by many different people in the object-oriented field. The UML developers did not invent most of these ideas; rather, their role was to select and integrate the best ideas from object modeling and computer-science practices. The actual genealogy of the notation and underlying detailed semantics is complicated, so it is discussed here only to provide context, not to represent precise history.

Use-case diagrams are similar in appearance to those in OOSE.

Class diagrams are a melding of OMT, Booch, class diagrams of most other object methods. Stereotypes and their corresponding icons can be defined for various diagrams to support other modeling styles. Stereotypes, constraints, and taggedValues are concepts added in UML that did not previously exist in the major modeling languages.

Statechart diagrams are substantially based on the statecharts of David Harel with minor modifications. Activity graph diagrams, which share much of the same underlying semantics, are similar to the work flow diagrams developed by many sources including many pre-object sources.

Sequence diagrams were found in a variety of object methods under a variety of names (interaction, message trace, and event trace) and date to pre-object days. Collaboration diagrams were adapted from Booch (object diagram), Fusion (object interaction graph), and a number of other sources.

Collaborations are now first-class modeling entities, and often form the basis of patterns.

The implementation diagrams (component and deployment diagrams) are derived from Booch's module and process diagrams, but they are now component-centered, rather than module-centered and are far better interconnected.

Stereotypes are one of the extension mechanisms and extend the semantics of the metamodel. User-defined icons can be associated with given stereotypes for tailoring the UML to specific processes.

Object Constraint Language is used by UML to specify the semantics and is provided as a language for expressions during modeling. OCL is an expression language having its root in the Syntropy method and has been influenced by expression languages in other methods like Catalysis. The informal navigation from OMT has the same intent, where OCL is formalized and more extensive.

Each of these concepts has further predecessors and many other influences. We realize that any brief list of influences is incomplete and we recognize that the UML is the product of a long history of ideas in the computer science and software engineering area.

1.6 UML - Past, Present, and Future

The UML was developed by Rational Software and its partners. Many companies are incorporating the UML as a standard into their development process and products, which cover disciplines such as business modeling, requirements management, analysis & design, programming, and testing.

1.6.1 UML 0.8 - 0.91

1.6.1.1 Precursors to UML

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. Several other techniques influenced these languages, including Entity-Relationship modeling, the Specification & Description Language (SDL, circa 1976, CCITT), and other techniques. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of object methods had trouble finding complete satisfaction in any one modeling language, fueling the “method wars.” By the mid-1990s, new iterations of these methods began to appear, most notably Booch’93, the continued evolution of OMT, and Fusion. These methods began to incorporate each other’s techniques, and a few clearly prominent methods emerged, including the OOSE, OMT-2, and Booch’93 methods. Each of these was a complete method, and was recognized as having certain strengths. In simple terms, OOSE was a use-case oriented approach that provided excellent support business engineering and requirements analysis. OMT-2 was especially expressive for analysis and data-intensive information systems. Booch’93 was particularly expressive during design and construction phases of projects and popular for engineering-intensive applications.

1.6.1.2 Booch, Rumbaugh, and Jacobson Join Forces

The development of UML began in October of 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. Given that the Booch and OMT methods were already independently growing together and were collectively recognized as leading object-oriented methods worldwide, Booch and Rumbaugh joined forces to forge a complete unification of their work. A draft version 0.8 of the

Unified Method, as it was then called, was released in October of 1995. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method. The Objectory name is now used within Rational primarily to describe its UML-compliant process, the Rational Unified Process.

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

As they began their unification, they established four goals to focus their efforts:

1. Enable the modeling of systems (and not just software) using object-oriented concepts.
2. Establish an explicit coupling to conceptual as well as executable artifacts.
3. Address the issues of scale inherent in complex, mission-critical systems.
4. Create a modeling language usable by both humans and machines.

Devising a notation for use in object-oriented analysis and design is not unlike designing a programming language. There are tradeoffs. First, one must bound the problem: Should the notation encompass requirement specification? (Yes, partially.) Should the notation extend to the level of a visual programming language? (No.) Second, one must strike a balance between expressiveness and simplicity: Too simple a notation will limit the breadth of problems that can be solved; too complex a notation will overwhelm the mortal developer. In the case of unifying existing methods, one must also be sensitive to the installed base: Make too many changes, and you will confuse existing users. Resist advancing the notation, and you will miss the opportunity of engaging a much broader set of users. The UML definition strives to make the best tradeoffs in each of these areas.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

1.6.2 UML Partners

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a

joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML definition. Those contributing most to the UML definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML, a modeling language that was well defined, expressive, powerful, and generally applicable.

In January 1997 IBM & ObjecTime; Platinum Technology; Ptech; Taskon & Reich Technologies; and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners.

This document is based on the UML 1.1 release and is the result of a collaborative team effort. The UML Partners have worked hard as a team to define UML. While each partner came in with their own perspective and areas of interest, the result has benefited from each of them and from the diversity of their experiences. The UML Partners contributed a variety of expert perspectives, including, but not limited to, the following: OMG and RM-ODP technology perspectives, business modeling, constraint language, state machine semantics, types, interfaces, components, collaborations, refinement, frameworks, distribution, and metamodel.

1.6.3 UML - Present and Future

The UML is nonproprietary and open to all. It addresses the needs of user and scientific communities, as established by experience with the underlying methods on which it is based. Many methodologists, organizations, and tool vendors have committed to use it. Since the UML builds upon similar semantics and notation from Booch, OMT, OOSE, and other leading methods and has incorporated input from the UML partners and feedback from the general public, widespread adoption of the UML should be straightforward.

There are two aspects of “unified” that the UML achieves: First, it effectively ends many of the differences, often inconsequential, between the modeling languages of previous methods. Secondly, and perhaps more importantly, it unifies the perspectives among many different kinds of systems (business versus software), development phases (requirements analysis, design, and implementation), and internal concepts.

1.6.3.1 Standardization of the UML

Many organizations have already endorsed the UML as their organization’s standard, since it is based on the modeling languages of leading object methods. The UML is ready for widespread use. This document is suitable as the primary source for authors writing books and training materials, as well as developers implementing visual modeling tools. Additional collateral, such as articles, training courses, examples, and books, will soon make the UML very approachable for a wide audience.

The Unified Modeling Language v. 1.1 specification which was added to the list of OMG Adopted Technologies in November 1997. Since then the OMG has assumed responsibility for the further development of the UML standard.

1.6.3.2 Revision of the UML

After adoption of the UML 1.1 specification by the OMG membership in November 1997, the OMG chartered a revision task force (RTF) to accept comments from the general public and to make revisions to the specifications to handle bugs, inconsistencies, ambiguities, and minor omissions that could be handled without a major change in scope from the original specification. The members of the RTF were drawn from the original proposers with a few additional persons. The RTF issued several preliminary reports with the final report containing UML 1.3 scheduled for the second quarter of 1999. It contained a number of changes to the UML metamodel, semantics, and notation, but in the big picture this version should be considered a minor upgrade to the original specification. More substantive changes and expansion in scope requires the full OMG specification and adoption process.

1.6.3.3 Industrialization

Many organizations and vendors worldwide have already embraced the UML. The number of endorsing organizations is expected to grow significantly over time. These organizations will continue to encourage the use of the Unified Modeling Language by making the definition readily available and by encouraging other methodologists, tool vendors, training organizations, and authors to adopt the UML.

The real measure of the UML's success is its use on successful projects and the increasing demand for supporting tools, books, training, and mentoring.

1.6.3.4 Future UML Evolution

Although the UML defines a precise language, it is not a barrier to future improvements in modeling concepts. We have addressed many leading-edge techniques, but expect additional techniques to influence future versions of the UML. Many advanced techniques can be defined using UML as a base. The UML can be extended without redefining the UML core.

The UML, in its current form, is expected to be the basis for many tools, including those for visual modeling, simulation, and development environments. As interesting tool integrations are developed, implementation standards based on the UML will become increasingly available.

The UML has integrated many disparate ideas, so this integration will accelerate the use of object-orientation. Component-based development is an approach worth mentioning. It is synergistic with traditional object-oriented techniques. While reuse based on components is becoming increasingly widespread, this does not mean that component-based techniques will replace object-oriented techniques. There are only subtle differences between the semantics of components and classes.

UML Semantics 2

Note – Change bars mark the differences between UML 1.4 and UML 1.5. Changes based on the ISO version of UML 1.4.1 (formal/03-02-04) are in this font.

Contents

This chapter contains the following sections.

Section Title	Page
Part 1 - Background	
“Introduction”	2-2
“Language Architecture”	2-4
“Language Formalism”	2-7
Part 2 - Foundation	
“Foundation Package”	2-11
“Core”	2-12
“Extension Mechanisms”	2-73
“Data Types”	2-85
Part 3 - Behavioral Elements	
“Behavioral Elements Package”	2-92
“Common Behavior”	2-93
“Collaborations”	2-111
“Use Cases”	2-129

Section Title	Page
“State Machines”	2-140
“Activity Graphs”	2-170
“Actions”	2-181
Part 4 - General Mechanisms	
“Model Management”	2-181
Part 5- Actions	
“Action Package”	2-199
“Actions Overview”	2-200
“Action Conventions”	2-207
“Action Foundation”	2-211
“Composite Actions”	2-228
“Read and Write Actions”	2-252
“Computation Actions”	2-287
“Collection Actions”	2-296
“Messaging Actions”	2-311
“Jump Actions”	2-332

Part 1 - Background

2.1 Introduction

2.1.1 Purpose and Scope

The primary audience for this detailed description consists of the OMG, other standards organizations, tool builders, metamodelers, methodologists, and expert modelers. The authors assume familiarity with metamodeling and advanced object modeling. Readers looking for an introduction to the UML or object modeling should consider another source.

Although the document is meant for advanced readers, it is also meant to be easily understood by its intended audience. Consequently, it is structured and written to increase readability. The structure of the document, like the language, builds on previous concepts to refine and extend the semantics. In addition, the document is written in a ‘semi-formal’ style that combines natural and formal languages in a complementary manner.

This section specifies semantics for structural and behavioral object models. Structural models (also known as static models) emphasize the structure of objects in a system, including their classes, interfaces, attributes and relations. Behavioral models (also known as dynamic models) emphasize the behavior of objects in a system, including their methods, interactions, collaborations, and state histories.

This section provides complete semantics for all modeling notations described in the UML Notation Guide (Chapter 3). This includes support for a wide range of diagram techniques: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, and deployment diagram. The UML Notation Guide includes a summary of the semantics sections that are relevant to each diagram technique.

2.1.2 Approach

This section emphasizes language architecture and formal rigor. The architecture of the UML is based on a four-layer metamodel structure, which consists of the following layers: user objects, model, metamodel, and meta-metamodel. This document is primarily concerned with the metamodel layer, which is an instance of the meta-metamodel layer. For example, Class in the metamodel is an instance of MetaClass in the meta-metamodel. The metamodel architecture of UML is discussed further in Section 2.2, “Language Architecture,” on page 2-4.

The UML metamodel is a logical model and not a physical (or implementation) model. The advantage of a logical metamodel is that it emphasizes declarative semantics, and suppresses implementation details. Implementations that use the logical metamodel must conform to its semantics, and must be able to import and export full as well as partial models. However, tool vendors may construct the logical metamodel in various ways, so they can tune their implementations for reliability and performance. The disadvantage of a logical model is that it lacks the imperative semantics required for accurate and efficient implementation. Consequently, the metamodel is accompanied with implementation notes for tool builders.

UML is also structured within the metamodel layer. The language is decomposed into several logical packages: Foundation, Behavioral Elements, and Model Management. These packages in turn are decomposed into subpackages. For example, the Foundation package consists of the Core, Extension Mechanisms, and Data Types subpackages. The structure of the language is fully described in Section 2.2, “Language Architecture,” on page 2-4.

The metamodel is described in a semi-formal manner using these views:

- Abstract syntax
- Well-formedness rules
- Semantics

The abstract syntax is provided as a model described in a subset of UML, consisting of a UML class diagram and a supporting natural language description. (In this way the UML bootstraps itself in a manner similar to how a compiler is used to compile itself.) The well-formedness rules are provided using a formal language (Object Constraint

Language) and natural language (English). Finally, the semantics are described primarily in natural language, but may include some additional notation, depending on the part of the model being described. The adaptation of formal techniques to specify the language is fully described in Section 2.3, “Language Formalism,” on page 2-7.

In summary, the UML metamodel is described in a combination of graphic notation, natural language, and formal language. We recognize that there are theoretical limits to what one can express about a metamodel using the metamodel itself. However, our experience suggests that this combination strikes a reasonable balance between expressiveness and readability.

2.2 Language Architecture

2.2.1 Four-layer Metamodel Architecture

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture. This architecture is a proven infrastructure for defining the precise semantics required by complex models. There are several other advantages associated with this approach:

- It refines semantic constructs by recursively applying them to successive metalayers.
- It provides an architectural basis for defining future UML metamodel extensions.
- It furnishes an architectural basis for aligning the UML metamodel with other standards based on a four-layer metamodeling architecture, in particular the OMG Meta-Object Facility (MOF).

The generally accepted framework for metamodeling is based on an architecture with four layers:

- meta-metamodel
- metamodel
- model
- user objects

The functions of these layers are summarized in the following table.

.

Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	MetaClass, MetaAttribute, MetaOperation
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	Class, Attribute, Operation, Component
model	An instance of a metamodel. Defines a language to describe an information domain.	StockShare, askPrice, sellLimitOrder, StockQuoteServer
user objects (user data)	An instance of a model. Defines a specific information domain.	<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>

The meta-metamodeling layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel. A meta-metamodel defines a model at a higher level of abstraction than a metamodel, and is typically more compact than the metamodel that it describes. A meta-metamodel can define multiple metamodels, and there can be multiple meta-metamodels associated with each metamodel.

While it is generally desirable that related metamodels and meta-metamodels share common design philosophies and constructs, this is not a strict rule. Each layer needs to maintain its own design integrity. Examples of meta-metaobjects in the meta-metamodeling layer are: MetaClass, MetaAttribute, and MetaOperation.

A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics. Examples of metaobjects in the metamodeling layer are: Class, Attribute, Operation, and Component.

A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain. Examples of objects in the modeling layer are: StockShare, askPrice, sellLimitOrder, and StockQuoteServer.

User objects (a.k.a. user data) are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain. Examples of objects in the user objects layer are: <Acme_Software_Share_98789>, 654.56, sell_limit_order, and <Stock_Quote_Svr_32123>.

2.2.1.1 Architectural Alignment with the MOF Meta-Metamodel

Both the UML and the MOF are based on a four-layer metamodel architecture, where the MOF meta-metamodel is the meta-metamodel for the UML metamodel. Since the MOF and UML have different scopes and differ in their abstraction levels (the UML

metamodel tends to be more of a logical model than the MOF meta-metamodel), they are related by loose metamodeling rather than strict metamodeling.[1] As a result, the UML metamodel is an instance of the MOF meta-metamodel.

Consequently, there is not a strict isomorphic instance-of mapping between all the MOF meta-metamodel elements and the UML metamodel elements. In spite of this, since the two models were designed to be interoperable, the UML Core package metamodel and the MOF meta-metamodel are structurally quite similar.

2.2.2 Package Structure

The complexity of the UML metamodel is managed by organizing it into logical packages. These packages group metaclasses that show strong cohesion with each other and loose coupling with metaclasses in other packages. The metamodel is decomposed into the top-level packages shown in Figure 2-1.

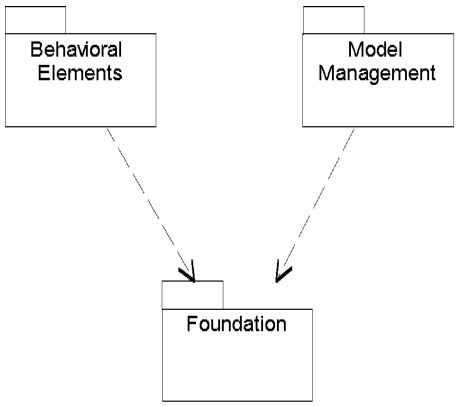


Figure 2-1 Top-Level Packages

The Foundation and Behavioral Elements packages are further decomposed as shown in Figure 2-2 and Figure 2-3.

1. In loose (or "non-strict") metamodeling a M_n level model is an instance of a M_{n+1} level model. In strict metamodeling, every element of a M_n level model is an instance of exactly one element of M_{n+1} level model.

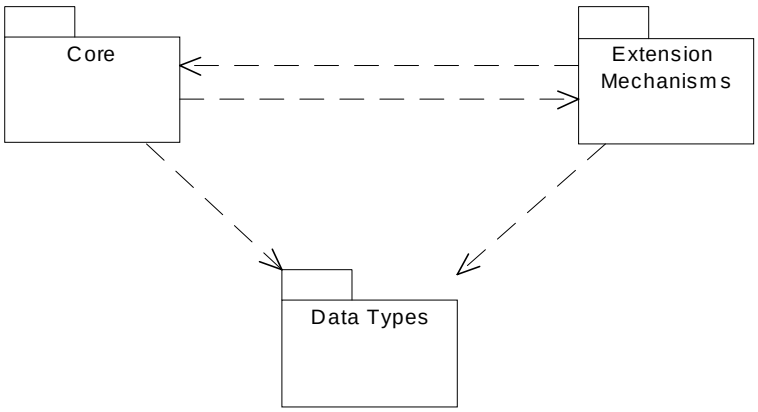


Figure 2-2 Foundation Packages

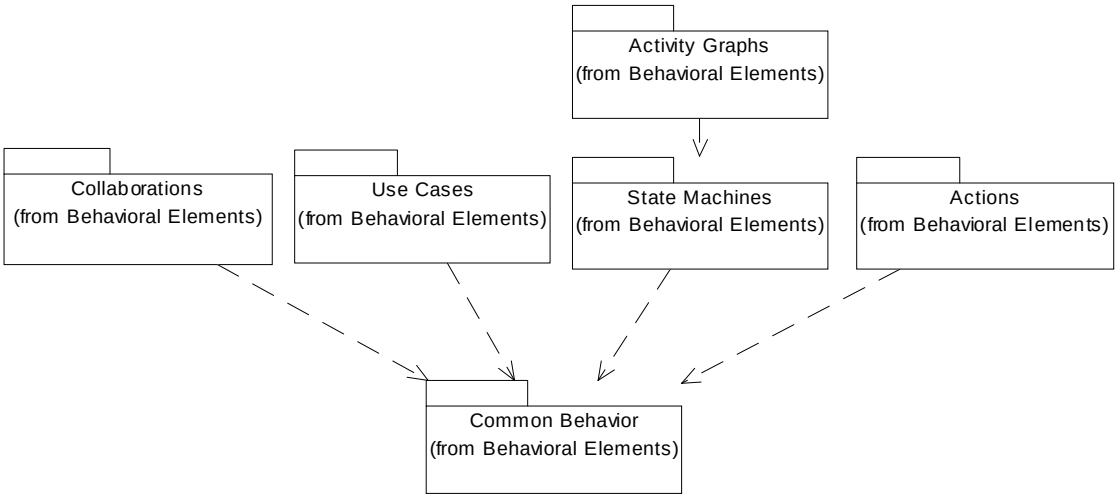


Figure 2-3 Behavioral Elements Packages

The functions and contents of these packages are described in “Part 3 - Behavioral Elements” on page 2-92.

2.3 Language Formalism

This section contains a description of the techniques used to describe UML. The specification adapts formal techniques to improve precision while maintaining readability. The technique describes the UML metamodel in three views using both text and graphic presentations. The benefits of adapting formal techniques include:

- the correctness of the description is improved,
- ambiguities and inconsistencies are reduced,
- the architecture of the metamodel is validated by a complementary technique, and
- the readability of the description is increased.

It is important to note that the current description is not a completely formal specification of the language because to do so would have added significant complexity without clear benefit. In addition, the state of the practice in formal specifications does not yet address some of the more difficult language issues that UML introduces.

The structure of the language is nevertheless given a precise specification, which is required for tool interoperability. The dynamic semantics are described using natural language, although in a precise way so they can easily be understood. Currently, the dynamic semantics are not considered essential for the development of tools; however, this will probably change in the future.

2.3.1 Levels of Formalism

A common technique for specification of languages is to first define the syntax of the language and then to describe its static and dynamic semantics. The syntax defines what constructs exist in the language and how the constructs are built up in terms of other constructs. Sometimes, especially if the language has a graphic syntax, it is important to define the syntax in a notation independent way, that is, to define the abstract syntax of the language. The concrete syntax is then defined by mapping the notation onto the abstract syntax. The syntax is described in the *Abstract Syntax* sections.

The static semantics of a language define how an instance of a construct should be connected to other instances to be meaningful, and the dynamic semantics define the meaning of a well formed construct. The meaning of a description written in the language is defined only if the description is well formed, that is, if it fulfills the rules defined in the static semantics. The static semantics are found in sections headed *Well-formedness Rules*. The dynamic semantics are described under the heading *Semantics*. In some cases, parts of the static semantics are also explained in the *Semantics* section for completeness.

The specification uses a combination of languages - a subset of UML, an object constraint language, and precise natural language to describe the abstract syntax and semantics of the full UML. The description is self-contained; no other sources of information are needed to read the document[2]. Although this is a metacircular description[3], understanding this document is practical since only a small subset of UML constructs are needed to describe its semantics.

---

2. Although a comprehension of the UML's four-layer metamodel architecture and its underlying meta-metamodel is helpful, it is not essential to understand the UML semantics.

In constructing the UML metamodel different techniques have been used to specify language constructs, using some of the capabilities of UML. The main language constructs are reified into metaclasses in the metamodel. Other constructs, in essence being variants of other ones, are defined as stereotypes of metaclasses in the metamodel. This mechanism allows the semantics of the variant construct to be significantly different from the base metaclass. Another more “lightweight” way of defining variants is to use metaattributes. As an example, the aggregation construct is specified by an attribute of the metaclass AssociationEnd, which is used to indicate if an association is an ordinary aggregate, a composite aggregate, or a common association.

2.3.2 Package Specification Structure

This section provides information for each package in the UML metamodel. Each package has one or more of the following subsections.

2.3.2.1 Abstract Syntax

The abstract syntax is presented in a UML class diagram showing the metaclasses defining the constructs and their relationships. The diagram also presents some of the well-formedness rules, mainly the multiplicity requirements of the relationships, and whether or not the instances of a particular sub-construct must be ordered. Finally, a short informal description in natural language describing each construct is supplied. The first paragraph of each of these descriptions is a general presentation of the construct that sets the context, while the following paragraphs give the informal definition of the metaclass specifying the construct in UML. For each metaclass, its attributes are enumerated together with a short explanation. Furthermore, the opposite role names of associations connected to the metaclass are also listed in the same way.

2.3.2.2 Well-formedness Rules

The static semantics of UML metaclasses, except for multiplicity and ordering constraints, are defined as a set of invariants of an instance of the metaclass. (Note that a metaclass is not required to have any invariants.) These invariants have to be satisfied for the construct to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Each invariant is defined by an OCL expression together with an informal explanation of the expression. In many cases, additional operations on the metaclasses are needed for the OCL expressions. These are then defined in a separate subsection after the well-formedness rules for the construct, using the same approach as the abstract syntax: an informal explanation followed by the OCL expression defining the operation.

---

3. In order to understand the description of the UML semantics, you must understand some UML semantics.

The statement ‘No extra well-formedness rules’ means that all current static semantics are expressed in the superclasses together with the multiplicity and type information expressed in the diagrams.

2.3.2.3 Semantics

The meanings of the constructs are defined using natural language. The constructs are grouped into logical chunks that are defined together. Since only concrete metaclasses have a true meaning in the language, only these are described in this section.

2.3.2.4 Standard Elements

Stereotypes of the metaclasses defined previously in the section are listed, with an informal definition in natural language. Well-formedness rules, if any, for the stereotypes are also defined in the same manner as in the *Well-formedness Rules* subsection.

Other kinds of standard elements (constraints and tagged-values) are listed, and are defined in the *Standard Elements* appendix.

2.3.2.5 Notes

This subsection may contain rationales for metamodeling decisions, pragmatics for the use of the constructs, and examples all written in natural language.

2.3.3 Use of a Constraint Language

The specification uses the Object Constraint Language (OCL), as defined in Chapter 6, “*Object Constraint Language Specification*” for expressing well-formedness rules. The following conventions are used to promote readability:

- Self - which can be omitted as a reference to the metaclass defining the context of the invariant, has been kept for clarity.
- In expressions where a collection is iterated, an iterator is used for clarity, even when formally unnecessary. The type of the iterator is usually omitted, but included when it adds to understanding.
- The ‘collect’ operation is left implicit where this is practical.

2.3.4 Use of Natural Language

We strove to be precise in our use of natural language, in this case English. For example, the description of UML semantics includes phrases such as “X provides the ability to…” and “X is a Y.” In each of these cases, the usual English meaning is assumed, although a deeply formal description would demand a specification of the semantics of even these simple phrases.

The following general rules apply:

- When referring to an instance of some metaclass, we often omit the word “instance.” For example, instead of saying “a Class instance” or “an Association instance,” we just say “a Class” or “an Association.” By prefixing it with an “a” or “an,” assume that we mean “an instance of.” In the same way, by saying something like “Elements” we mean “a set (or the set) of instances of the metaclass Element.”
- Every time a word coinciding with the name of some construct in UML is used, that construct is referenced.
- Terms including one of the prefixes sub, super, or meta are written as one word (for example, metamodel, subclass).

2.3.5 Naming Conventions and Typography

In the description of UML, the following conventions have been used:

- When referring to constructs in UML, not their representation In the metamodel, normal text is used.
- Metaclass names that consist of appended nouns/adjectives, initial embedded capitals are used (for example, ‘ModelElement,’ ‘StructuralFeature’).
- Names of metaassociations/association classes are written in the same manner as metaclasses (for example, ‘ElementReference’).
- Initial embedded capital is used for names that consist of appended nouns/adjectives (for example, ‘ownedElement,’ ‘allContents’).
- Boolean metaattribute names always start with ‘is’ (for example, ‘isAbstract’).
- Enumeration types always end with “Kind” (for example, ‘AggregationKind’).
- While referring to metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are always used.
- Names of stereotypes are delimited by guillemets and begin with lowercase (for example, «type»).

Part 2 - Foundation

2.4 Foundation Package

The Foundation package is the language infrastructure that specifies the static structure of models. The Foundation package is decomposed into the following subpackages: Core, Extension Mechanisms, and Data Types. Figure 2-4 illustrates the Foundation Packages. The Core package specifies the basic concepts required for an elementary metamodel and defines an architectural backbone for attaching additional language constructs, such as metaclasses, metaassociations, and metaattributes. The Extension Mechanisms package specifies how model elements are customized and extended with new semantics. The Data Types package defines basic data structures for the language.

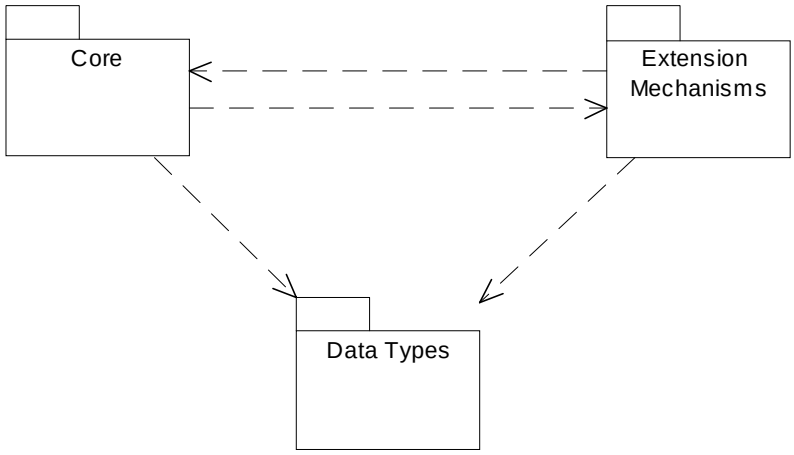


Figure 2-4 Foundation Packages

2.5 Core

2.5.1 Overview

The Core package is the most fundamental of the subpackages that compose the UML Foundation package. It defines the basic abstract and concrete metamodel constructs needed for the development of object models. Abstract constructs are not instantiable and are commonly used to reify key constructs, share structure, and organize the UML metamodel. Concrete metamodel constructs are instantiable and reflect the modeling constructs used by object modelers (cf. metamodelers). Abstract constructs defined in the Core include ModelElement, GeneralizableElement, and Classifier. Concrete constructs specified in the Core include Class, Attribute, Operation, and Association.

The Core package specifies the core constructs required for a basic metamodel and defines an architectural backbone (“skeleton”) for attaching additional language constructs such as metaclasses, metaassociations, and metaattributes. Although the Core package contains sufficient semantics to define the remainder of UML, it is not the UML meta-metamodel. It is the underlying base for the Foundation package, which in turn serves as the infrastructure for the rest of language. In other packages, the Core is extended by adding metaclasses to the backbone using generalizations and associations.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Core package.

2.5.2 Abstract Syntax

The abstract syntax for the Core package is expressed in graphic notation in the following figures. Figure 2-5 on page 2-13 shows the model elements that form the structural backbone of the metamodel. Figure 2-6 on page 2-14 shows the model

elements that define relationships. Figure 2-7 on page 2-15 shows the model elements that define dependencies. Figure 2-8 on page 2-16 shows the various kinds of classifiers. Figure 2-9 on page 2-17 shows auxiliary elements for template parameters, presentation elements, and comments.

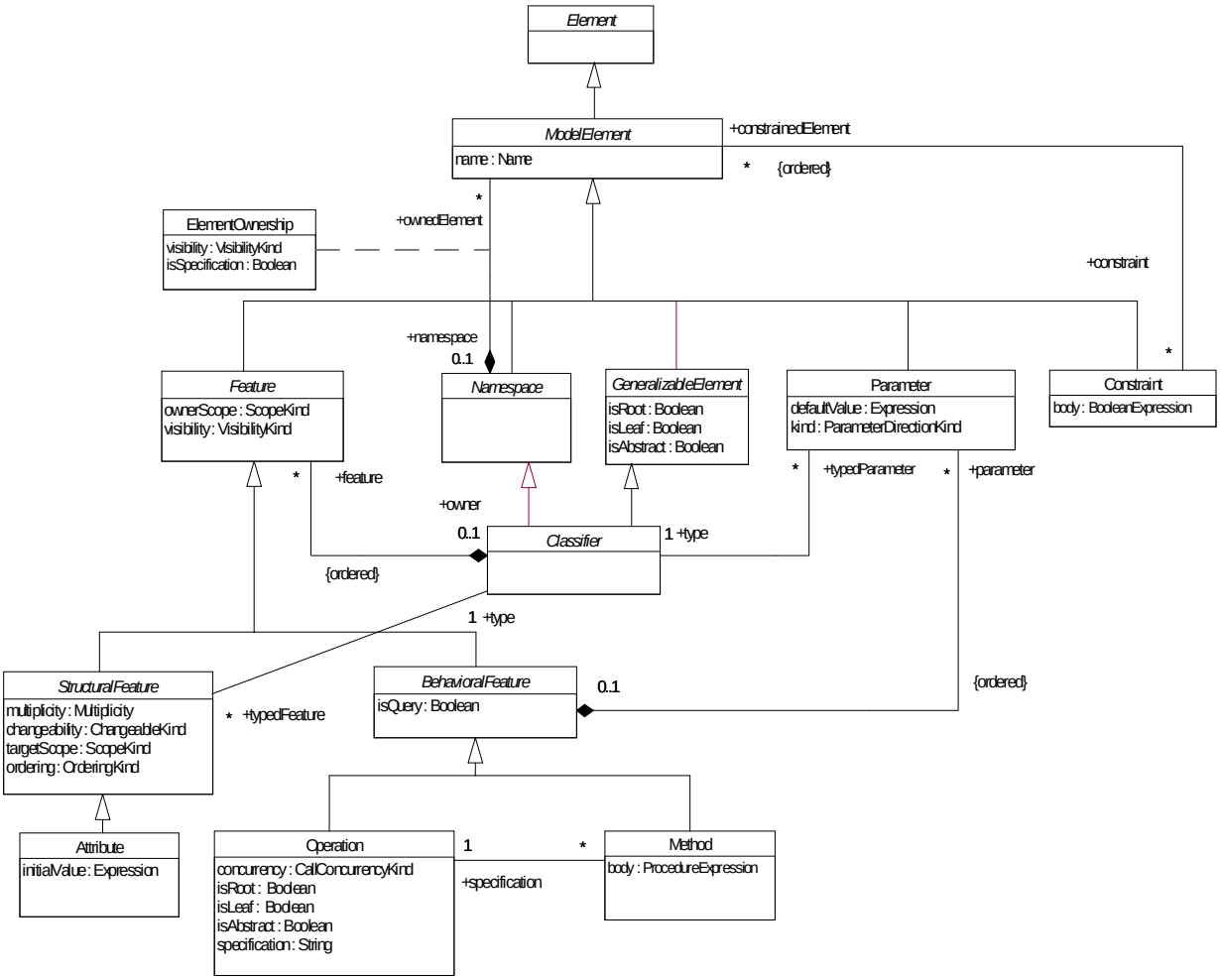


Figure 2-5 Core Package - Backbone

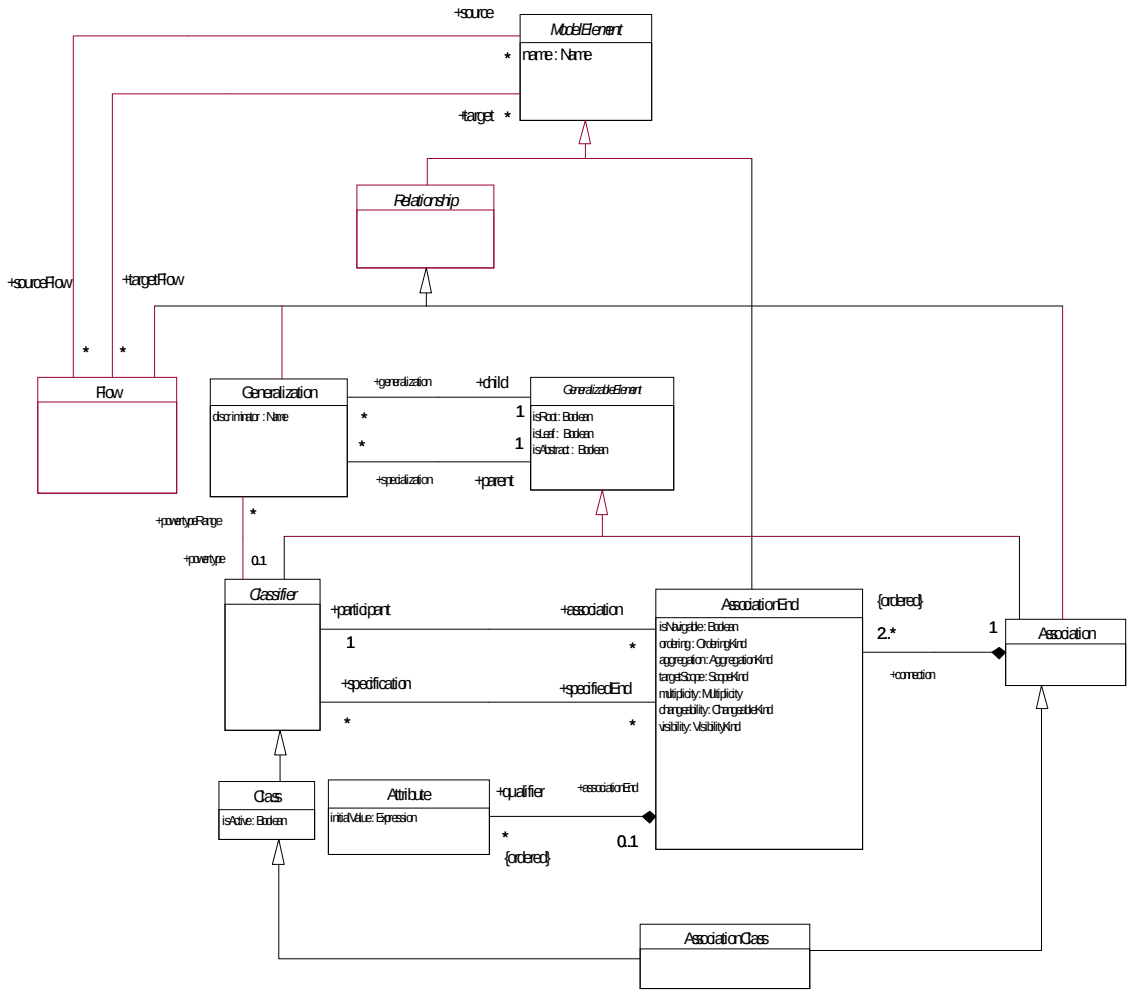


Figure 2-6 Core Package - Relationships

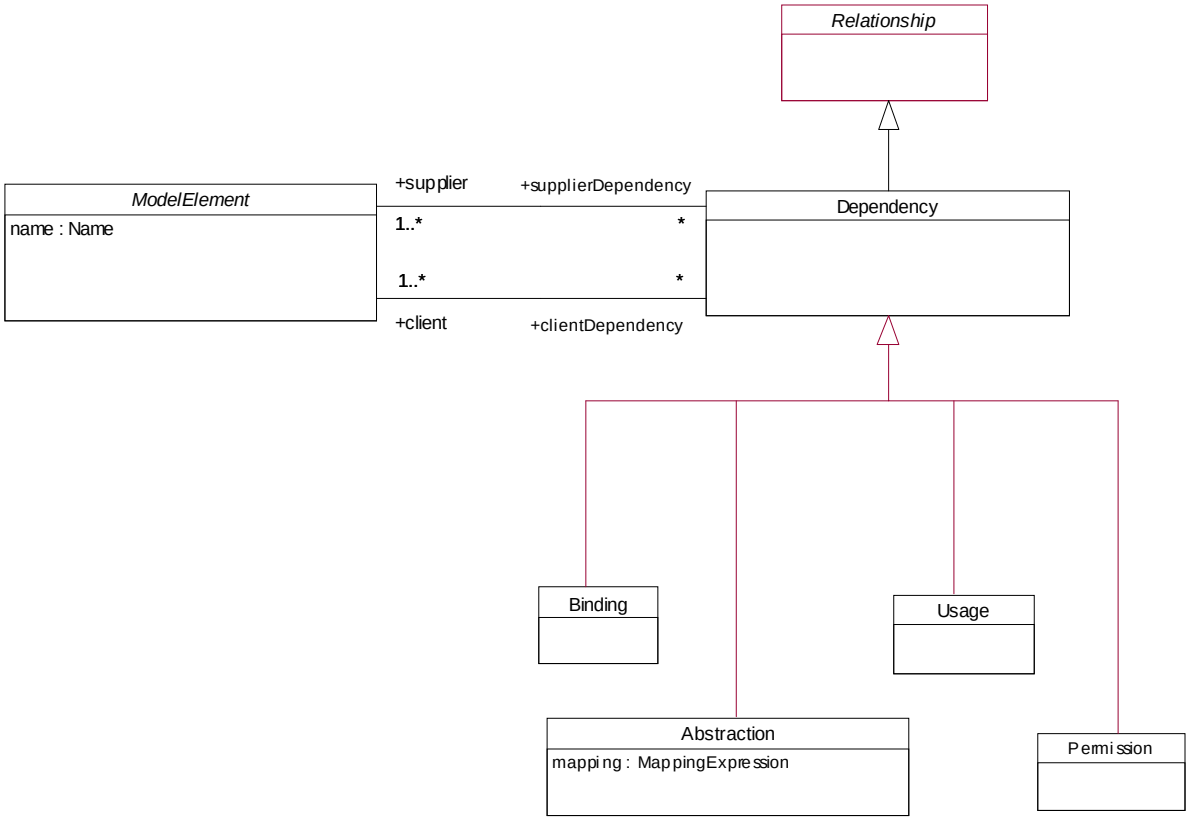


Figure 2-7 Core Package - Dependencies

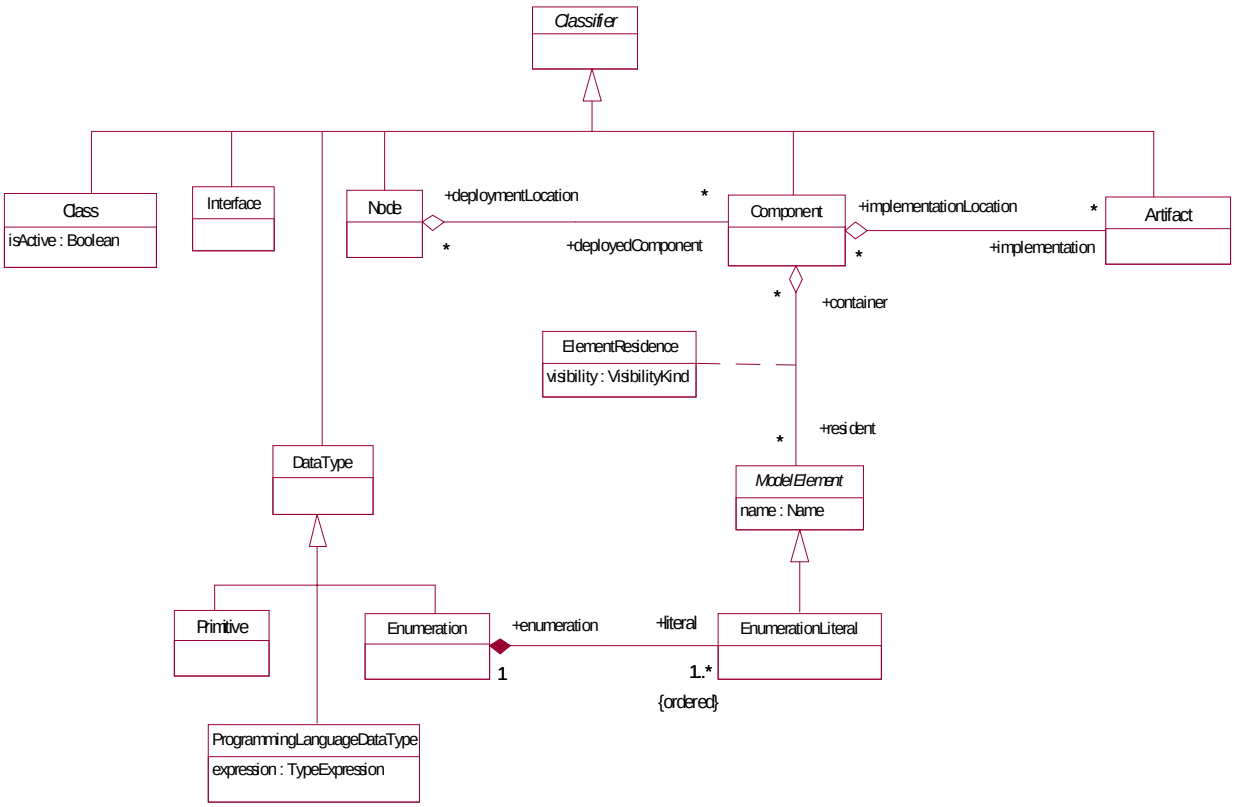


Figure 2-8 Core Package - Classifiers

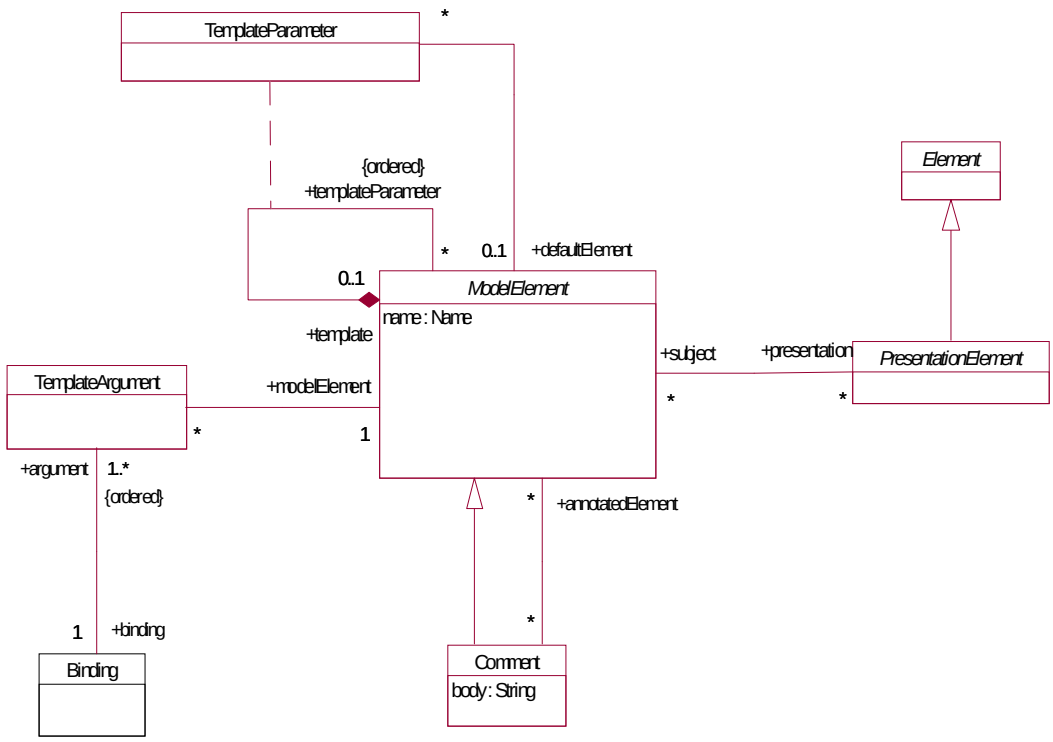


Figure 2-9 Core Package - Auxiliary elements

2.5.2.1 Abstraction

An abstraction is a Dependency relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints.

In the metamodel, an Abstraction is a Dependency in which there is a mapping between the supplier and the client. Depending on the specific stereotype of Abstraction, the mapping may be formal or informal, and it may be unidirectional or bidirectional.

If an Abstraction element has more than one client element, the supplier element maps into the set of client elements as a group. For example, an analysis-level class might be split into several design-level classes. The situation is similar if there is more than one supplier element.

The UML standard stereotyped classes of Abstraction are Derivation, Realization, Refinement, and Trace. (These are the names for the Abstraction class with the stereotypes «derive», «realize», «refine», and «trace», respectively.)

Attributes

mapping	A MappingExpression that states the abstraction relationship between the supplier and the client. In some cases, such as Derivation, it is usually formal and unidirectional; in other cases, such as Trace, it is usually informal and bidirectional. The mapping attribute is optional and may be omitted if the precise relationship between the elements is not specified.

Stereotypes

«derive»	Class	(Name for the stereotyped class is Derivation.) Specifies a derivation relationship among model elements that are usually, but not necessarily, of the same type. A derived dependency specifies that the client may be computed from the supplier. The mapping specifies the computation. The client may be implemented for design reasons, such as efficiency, even though it is logically redundant.
«realize»	Class	(Name for the stereotyped class is Realization.) Specifies a realization relationship between a specification model element or elements (the supplier) and a model element or elements that implement it (the client). The implementation model element is required to support all of the operations or received signals that the specification model element declares. The implementation model element must make or inherit its own declarations of the operations and signal receptions. The mapping specifies the relationship between the two. The mapping may or may not be computable. Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.
«refine»	Class	(Name for the stereotyped class is Refinement.) Specifies refinement relationship between model elements at different semantic levels, such as analysis and design. The mapping specifies the relationship between the two elements or sets of elements. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes.
«trace»	Class	(Name for the stereotyped class is Trace.) Specifies a trace relationship between model elements or sets of model elements that represent the same concept in different models. Traces are mainly used for tracking requirements and changes across models. Since model changes can occur in both directions, the directionality of the dependency can often be ignored. The mapping specifies the relationship between the two, but it is rarely computable and is usually informal.

2.5.2.2 Artifact

An Artifact represents a physical piece of information that is used or produced by a software development process. Examples of Artifacts include models, source files, scripts, and binary executable files. An Artifact may constitute the implementation of a deployable component.

In the metamodel, an Artifact is a Classifier with an optional aggregation association to one or more Components. As a Classifier, Artifacts may have Features that represent properties of the Artifact (e.g., a “read-only” attribute or a “check in” operation).

It should be noted that sometimes Artifacts may need to be linked to Classifiers directly, without introducing a ‘Component.’ For instance, in the context of code generation, the resulting Artifacts (source code files) are never deployed as Components. In that case, a «derive» Dependency can be used between the Classifier(s) and the generated Artifact.

The standard stereotypes of Artifact are «file», the subclasses of «file» («executable», «source», «library», and «document»), and «table». These stereotypes can be further subclassed into implementation and platform specific stereotypes (e.g., «jarFile» for Java archives).

Associations

implementation Location	The deployable Component(s) that are implemented by this Artifact.

Stereotypes

«document»	Class	Denotes a generic file that is not a «source» file or «executable». Subclass of «file».
«executable»	Class	Denotes a program file that can be executed on a computer system. Subclass of «file».
«file»	Class	Denotes a physical file in the context of the system developed.
«library»	Class	Denotes a static or dynamic library file. Subclass of «file».
«source»	Class	Denotes a source file that can be compiled into an executable file. Subclass of «file».
«table»	Class	Denotes a database table.

2.5.2.3 Association

An association defines a semantic relationship between classifiers. The instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once.

In the metamodel, an Association is a declaration of a semantic relationship between Classifiers, such as Classes. An Association has at least two AssociationEnds. Each end is connected to a Classifier - the same Classifier may be connected to more than one AssociationEnd in the same Association. The Association represents a set of connections among instances of the Classifiers. An instance of an Association is a Link, which is a tuple of Instances drawn from the corresponding Classifiers.

Attributes

name	The name of the Association which, in combination with its associated Classifiers, must be unique within the enclosing namespace (usually a Package).

Associations

connection	An Association consists of at least two AssociationEnds, each of which represents a connection of the association to a Classifier. Each AssociationEnd specifies a set of properties that must be fulfilled for the relationship to be valid. The bulk of the structure of an Association is defined by its AssociationEnds. The classifiers belonging to the association are related to the AssociationEnds by the participant rolename association.

Stereotypes

«implicit»	The «implicit» stereotype is applied to an association, specifying that the association is not manifest, but rather is only conceptual.

Standard Constraints

xor Association	The {xor} constraint is applied to a set of associations, specifying that over that set, exactly one is manifest for each associated instance. Xor is an exclusive or (not inclusive or) constraint.

Tagged Values

persistence Association	Persistence denotes the permanence of the state of the association, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

Inherited Features

Association is a GeneralizableElement. The following elements are inherited by a child Association.

connection	The child must have the same number of ends as the parent. Each participant class must be a descendant of the participant class in the same position in the parent. If the Association is an AssociationClass, its class properties (attributes, operations, etc.) are inherited. Various other properties are subject to change in the child. This specification is likely to be further clarified in UML 2.0.

Non-Inherited Features

isRoot isLeaf isAbstract	Not inheritable by their very nature, but they define the generalization structure.
name	Each model element has a unique name.

2.5.2.4 AssociationClass

An association class is an association that is also a class. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not any of the classifiers.

Inherited Features

AssociationClass inherits features as specified in both Class and Association.

In the metamodel, an AssociationClass is a declaration of a semantic relationship between Classifiers, which has a set of features of its own. AssociationClass is a subclass of both Association and Class (i.e., each AssociationClass is both an Association and a Class); therefore, an AssociationClass has both AssociationEnds and Features.

2.5.2.5 AssociationEnd

An association end is an endpoint of an association, which connects the association to a classifier. Each association end is part of one association. The association-ends of each association are ordered.

In the metamodel, an AssociationEnd is part of an Association and specifies the connection of an Association to a Classifier. It has a name and defines a set of properties of the connection (e.g., which Classifier the Instances must conform to, their multiplicity, and if they may be reached from another Instance via this connection).

In the following descriptions when referring to an association end for a binary association, the source end is the other end. The target end is the one whose properties are being discussed.

Attributes

aggregation	When placed on one end (the “target” end), specifies whether the class on the target end is an aggregation with respect to the class on the other end (the “source”end). Only one end can be an aggregation. Possibilities are: • none - The target class is not an aggregate. • aggregate - The target class is an aggregate; therefore, the source class is a part and must have the aggregation value of none. The part may be contained in other aggregates. • composite - The target class is a composite; therefore, the source class is a part and must have the aggregation value of none. The part is strongly owned by the composite and may not be part of any other composite.
changeability	Specifies whether or not links may be created or destroyed after the initialization of objects at the opposite ends. Possibilities are: • changeable - No restrictions on creation and destruction of links. • frozen - No links may be destroyed after the objects at the opposite ends have been initialized, and no new links may be created after the objects that would participate in the new link at the opposite ends have been initialized. • addOnly - No link may be destroyed after the objects at the opposite ends have been initialized. Links may be created anytime.
ordering	When placed on a target end, specifies whether the set of links from the source instance to the target instance is ordered. The ordering must be determined and maintained by Operations that add links. It represents additional information not inherent in the objects or links themselves. Possibilities are: • unordered - The links form a set with no inherent ordering. • ordered - A set of ordered links can be scanned in order. • Other possibilities (such as sorted) may be defined later by declaring additional keywords. As with user-defined stereotypes, this would be a private extension supported by particular editing tools.
isNavigable	When placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible. Specification of each direction across the Association is independent. A value of true means that the association can be navigated by the source class and the target rolename can be used in navigation expressions.
multiplicity	When placed on a target end, specifies the number of target instances that may be associated with a single source instance across the given Association.

name	(Inherited from ModelElement) The rolename of the end. When placed on a target end, provides a name for traversing from a source instance across the association to the target instance or set of target instances. It represents a pseudo-attribute of the source classifier (i.e., it may be used in the same way as an Attribute) and must be unique with respect to Attributes and other pseudo-attributes of the source Classifier.
targetScope	Specifies whether the target value is an instance or a classifier. Possibilities are: • instance. An instance value is part of each link. This is the default. • classifier. A classifier itself is part of each link. Normally this would be fixed at modeling time and need not be stored separately at run time.
visibility	Specifies the visibility of the association end from the viewpoint of the classifier on the other end. Possibilities are: • public - Other classifiers may navigate the association and use the rolename in expressions, similar to the use of a public attribute. • protected - Descendants of the source classifier may navigate the association and use the rolename in expressions, similar to the use of a protected attribute. • private - Only the source classifier may navigate the association and use the rolename in expressions, similar to the use of a private attribute. • package - Classifiers in the same package (or a nested subpackage, to any level) as the association declaration may navigate the association and use the rolename in expressions.

Associations

qualifier	An optional list of qualifier Attributes for the end. If the list is empty, then the Association is not qualified.
specification	Designates zero or more Classifiers that specify the Operations that may be applied to an Instance accessed by the AssociationEnd across the Association. These determine the minimum interface that must be realized by the actual Classifier attached to the end to support the intent of the Association. May be an Interface or another Classifier. These classifiers do not indicate the classes of the participants in a link, merely the operations that may be applied when traversing a link.
participant	Designates the Classifier participating in the Association at the given end. A link of the Association contains a reference to an instance of the class (including a descendant of the given class or a class that realizes a given interface) in the given position in the link.
(unnamed composite end)	Designates the Association that owns the AssociationEnd.

Stereotypes

«association»	Class	Specifies a real association (default and redundant, but may be included for emphasis).
«global»	Class	Specifies that the target is a global value that is known to all elements rather than an actual association.
«local»	Class	Specifies that the relationship represents a local variable within a procedure rather than an actual association.
«parameter»	Class	Specifies that the relationship represents an operation, method, or procedure parameter rather than an actual association.
«self»	Class	Specifies that the relationship represents a reference to the object that owns an operation or action rather than an actual association.

2.5.2.6 Attribute

An attribute is a named slot within a classifier that describes a range of values that instances of the classifier may hold.

In the metamodel, an Attribute is a named piece of the declared state of a Classifier, particularly the range of values that Instances of the Classifier may hold.

Attributes

initialValue	An Expression specifying the value of the attribute upon initialization. It is meant to be evaluated at the time the object is initialized. (Note that an explicit constructor may supersede an initial value.)

Associations

associationEnd	Designates the optional AssociationEnd that owns a qualifier attribute. Note that an attribute may be part of an AssociationEnd (in which case it is a qualifier) or part of a Classifier (by inheritance from Feature, in which case it is a feature) but not both. If the value is empty, the attribute is not a qualifier attribute.

2.5.2.7 BehavioralFeature

A behavioral feature refers to a dynamic feature of a model element, such as an operation or method.

In the metamodel, a BehavioralFeature specifies a behavioral aspect of a Classifier. All different kinds of behavioral aspects of a Classifier, such as Operation and Method, are subclasses of BehavioralFeature. BehavioralFeature is an abstract metaclass.

Attributes

isQuery	Specifies whether an execution of the Feature leaves the state of the system unchanged. True indicates that the state is unchanged; false indicates that side-effects may occur.
name	(Inherited from ModelElement) The name of the Feature. The entire signature of the Feature (name and parameter list) must be unique within its containing Classifier.

Associations

parameter	An ordered list of Parameters for the Operation. To call the Operation, the caller must supply a list of values compatible with the types of the Parameters.

Stereotypes

«create»	Class	Specifies that the designated feature creates an instance of the classifier to which the feature is attached. May be promoted to the Classifier containing the feature.
«destroy»	Class	Specifies that the designated feature destroys an instance of the classifier to which the feature is attached. May be promoted to the classifier containing the feature.

2.5.2.8 Binding

A binding is a relationship between a template (as supplier) and a model element generated from the template (as client). It includes a list of arguments that match the template parameters. The template is a form that is cloned and modified by substitution to yield an implicit model fragment that behaves as if it were a direct part of the model. A Binding must have one supplier and one client; unlike a general Dependency, the supplier and client may not be sets.

In the metamodel, a Binding is a Dependency where the supplier is the template and the client is the instantiation of the template that performs the substitution of parameters of a template. A Binding has a list of arguments that replace the parameters of the supplier to yield the client. The client is fully specified by the binding of the supplier's parameters and does not add any information of its own. An element may participate as a supplier in multiple Binding relationships to different clients. An element may participate in only one Binding relationship as a client.

Associations

argument	An ordered list of arguments. Each argument is a TemplateArgument element. The model element attached to the TemplateArgument by the modelElement association replaces the corresponding supplier parameter in the supplier definition, and the result represents the definition of the client as if it had been defined directly.

2.5.2.9 Class

A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.

In the metamodel, a Class describes a set of Objects sharing a collection of Features, including Operations, Attributes, and Methods that are common to the set of Objects. Furthermore, a Class may realize zero or more Interfaces; this means that its full descriptor (see “Inheritance” on page 2-69 for the definition) must contain every Operation from every realized Interface (it may contain additional operations as well).

A Class defines the data structure of Objects, although some Classes may be abstract (i.e., no Objects can be created directly from them). Each Object instantiated from a Class contains its own set of values corresponding to the StructuralFeatures declared in the full descriptor. Objects do not contain values corresponding to BehavioralFeatures or class-scope Attributes; all Objects of a Class share the definitions of the BehavioralFeatures from the Class, and they all have access to the single value stored for each class-scope attribute.

Attributes

isActive	Specifies whether an Object of the Class maintains its own thread of control. If true, then an Object has its own thread of control and runs concurrently with other active Objects. Such a class is informally called an *active class*. If false, then Operations run in the address space and under the control of the active Object that controls the caller. Such a class is informally called a *passive class*.

Stereotypes

«auxiliary»	Class	Specifies a class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. The class that the auxiliary supports may be defined explicitly using a Focus class or implicitly by a dependency relationship. Auxiliary classes are typically used together with Focus classes, and are particularly useful for specifying the secondary business logic or control flow of components during design. See also: «focus».
«focus»	Class	Specifies a class that defines the core logic or control flow for one or more auxiliary classes that support it. Support classes may be defined explicitly using Auxiliary classes or implicitly by dependency relationships. Focus classes are typically used together with one or more Auxiliary classes, and are particularly useful for specifying the core business logic or control flow of components during design. See also: «auxiliary».

***Stereotypes** (continued)*

«implementation»	Class	Specifies the implementation of a class in some programming language (e.g., C++, Smalltalk, Java) in which an instance may not have more than one class. This is in contrast to Class, for which an instance may have multiple classes at one time and may gain or lose classes over time, and an object (a child of instance) may dynamically have multiple classes. An Implementation class is said to *realize* a Type if it provides all of the operations defined for the Type with the same behavior as specified for the Type's operations. An Implementation Class may realize a number of different Types. Note that the physical attributes and associations of the Implementation class do not have to be the same as those of any Type it realizes and that the Implementation Class may provide methods for its operations in terms of its physical attributes and associations. See also: «type».
«type»	Class	Specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type may not contain any methods, maintain its own thread of control, or be nested. However, it may have attributes and associations. The associations of a Type are defined solely for the purpose of specifying the behavior of the type's operations and do not represent the implementation of state data. Although an object may have at most one Implementation Class, it may conform to multiple different Types. See also: «implementation».

Inherited Features

Class is a GeneralizableElement. The following elements are inherited by a child classifier, in addition to those specified under its parent, Classifier.

isActive	The child may be active when the parent is passive, but not vice versa. In most cases, they are the same.

2.5.2.10 Classifier

A classifier is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, component, artifact, and others that are defined in other metamodel packages.

In the metamodel, a Classifier declares a collection of Features, such as Attributes, Methods, and Operations. It has a name, which is unique in the Namespace enclosing the Classifier. Classifier is an abstract metaclass.

Classifier is a child of GeneralizableElement and Namespace. As a GeneralizableElement, it may inherit Features and participation in Associations (in addition to things inherited as a ModelElement). It also inherits ownership of StateMachines, Collaborations, etc.

As a Namespace, a Classifier may declare other Classifiers nested in its scope. Nested Classifiers may be accessed by other Classifiers only if the nested Classifiers have adequate visibility. There are no data value or state consequences of nested Classifiers (i.e., it is not an aggregation or composition).

Associations

feature	An ordered list of Features, like Attribute, Operation, Method owned by the Classifier.
association	Denotes the AssociationEnd of an Association in which the Classifier participates at the given end. This is the inverse of the participant association from AssociationEnd. A link of the association contains a reference to an instance of the class in the given position.
powertypeRange	Designates zero or more Generalizations for which the Classifier is a powertype. If the cardinality is zero, then the Classifier is not a powertype; if the cardinality is greater than zero, then the Classifier is a powertype over the set of Generalizations designated by the association, and the child elements of the Generalizations are the instances of the Classifier as a powertype. A Classifier that is a powertype can be marked with the «powertype» stereotype.
specifiedEnd	Indicates an AssociationEnd for which the given Classifier specifies operations that may be applied to instances obtained by traversing the association from the other end. (This relationship does not define the structure of the association, merely operations that may be applied on traversing it.)

Stereotypes

«metaclass»	Class	Specifies that the instances of the classifier are classes.
«powertype»	Class	Specifies that the classifier is a metaclass whose instances are siblings marked by the same discriminator. For example, the metaclass TreeSpecies might be a power type for the subclasses of Tree that represent different species, such as AppleTree, BananaTree, and CherryTree.
«process»	Class	Specifies a classifier that represents a heavy-weight flow of control.
«thread»	Class	Specifies a classifier that represents a flow of control.
«utility»	Class	Specifies a classifier that has no instances, but rather denotes a named collection of non-member attributes and operations, all of which are class-scoped.

Tagged Values

persistence	Persistence denotes the permanence of the state of the classifier, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).
semantics Classifier	Semantics is the specification of the meaning of the classifier.

Inherited Features

Classifier is a GeneralizableElement. The following elements are inherited by a child classifier. Note that inheritance makes the inherited elements part of the (virtual) full descriptor of the classifier, but it does not change its actual data structure. See the explanation for the meaning of each kind of inheritance.

associationEnd	The child class inherits participation in all associations of its parent, subject to all the same properties.
constraint	Constraints on the parent apply to the child.
feature	Attributes of the parent are part of the full descriptor of the child and may not be declared again or overridden. Operations of the parent are part of the full descriptor of the child but may be overridden; a redeclaration may change its hierarchy location (isRoot, isLeaf, isAbstract) but may not change its specification or parameter structure. The concurrency level may be loosened (e.g., from guarded to concurrent). An overriding operation may link to a different Method. An overriding operation can have isQuery=true when the parent had a false value, but not vice versa (in other words, once a side-effect, always a side-effect). Methods of the parent are part of the full descriptor of the child but may be overridden. An overriding method can set the isQuery status, change its hierarchy structure, but may not change its parameter structure. It may link to a different operation that overrides the operation of the parent method.
generalization specialization	These are implicitly inherited, in the sense that they define ancestors and descendants, but not explicitly inherited, as they are the arcs in the generalization graph. They establish the generalization structure itself as a directed graph, into which the child classifier fits.
ownedElement	The namespace of the parent is available to the child, except for private access.

Non-Inherited Features

The following elements are not inherited by a child classifier.

isRoot isLeaf isAbstract	By their very nature, these are not inherited
name	Each classifier has its own unique name
parameter	Template structure is not inherited. Each classifier must declare its own templace structure, if any. A nontemplate can be child of a template and vice versa.
powertypeRange	A powertype corresponds to a particular node in the generalization hierarchy, so it is not inherited.

2.5.2.11 Comment

A comment is an annotation attached to a model element or a set of model elements. It has no semantic force but may contain information useful to the modeler.

Attributes

body	A string that is the comment.

Associations

annotatedElement	A ModelElement or set of ModelElements described by the Comment.

Stereotypes

«requirement»	Class	Specifies a desired feature, property, or behavior of an element as part of a system.
«responsibility»	Class	Specifies a contract or an obligation of an element in its relationship to other elements.

2.5.2.12 Component

A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.

A component is typically specified by one or more classifiers that reside on the component. A subset of these classifiers explicitly define the component's external interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component. A component may be implemented by one or more artifacts, such as binary, executable, or script files. A component may be deployed on a node.

Components may be specified in both design models (e.g., using static structure diagrams) and in implementation models (e.g., using implementation diagrams). When they are specified as part of a design model components need not be allocated to nodes, nor do they need to have any associated implementation artifacts.

In the metamodel, a Component is a child of Classifier. It does not have its own Features, but instead acts as a container for other Classifiers that have Features. A Component is specified by the Interfaces is exposes and the Classifiers that reside on it. The visibility attribute of the ElementResidence association defines whether a resident element is visible outside the Component: an external Interface of a Component has visibility value ‘public.’ A Component may be implemented by one or more Artifacts, and may be deployed on a Node.

Associations

deploymentLocation	The set of Nodes the Component is residing on.
resident	(Association class ElementResidence) The set of model elements that specify the component. The visibility attribute shows the external visibility of the element outside the component: an external Interface of a Component has visibility = ‘public’ for its ElementResidence association.
implementation	The set of Artifacts that implement the Component. For a Component, these Artifacts are generally «executable».

Inherited Features

The following elements are inherited by a child Component, in addition to those specified under Classifier.

(none)

Non-Inherited Features

deploymentLocation	The set of locations may differ. Often it is more restrictive on the child.
resident	The set of resident elements may differ. Often it is more restrictive on the child and contains additional elements.
implementation	The set of Artifacts that implement the child Component will usually differ.

2.5.2.13 Constraint

A constraint is a semantic condition or restriction expressed in text.

In the metamodel, a Constraint is a BooleanExpression on an associated ModelElement(s), which must be true for the model to be well formed. This restriction can be stated in natural language, or in different kinds of languages with a well defined

semantics. Certain Constraints are predefined in the UML, others may be user defined. Note that a Constraint is an assertion, not an executable mechanism. It indicates a restriction that must be enforced by correct design of a system.

Attributes

body	A BooleanExpression that must be true when evaluated for an instance of a system to be well formed.

Associations

constrainedElement	A ModelElement or list of ModelElements affected by the Constraint. If the constrained element is a Stereotype, then the constraint applies to all ModelElements that use the stereotype.

Stereotypes

«invariant» Class	Specifies a constraint that must be attached to a set of classifiers or relationships. It indicates that the conditions of the constraint must hold over time (for the time period of concern in the particular containing element) for the classifiers or relationships and their instances.
«postcondition» Class	Specifies a constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold after the invocation of the operation.
«precondition» Class	Specifies a constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold for the invocation of the operation.
«stateInvariant» Class	Specifies a constraint that must be attached to a state vertex in a state machine that has a classifier for a context. The stereotype indicates that the constraint holds for instances of the classifier when an instance is in that state.

2.5.2.14 DataType

A data type is a type whose values have no identity (i.e., they are pure values). Data types include primitive built-in types (such as integer and string) as well as definable enumeration types (such as the predefined enumeration type boolean whose literals are false and true).

In the metamodel, a DataType defines a special kind of Classifier in which Operations are all pure functions (i.e., they can return DataValues but they cannot change DataValues, because they have no identity). For example, an “add” operation on a number with another number as an argument yields a third number as a result; the target and argument are unchanged.

Inherited Features

DataType inherits features as specified in Classifier.

2.5.2.15 Dependency

A term of convenience for a Relationship other than Association, Generalization, Flow, or metarelationship (such as the relationship between a Classifier and one of its Instances).

A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements.

In the metamodel, a Dependency is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier (i.e., the client element requires the presence and knowledge of the supplier element).

The kinds of Dependency are Abstraction, Binding, Permission, and Usage. Various stereotypes of those elements are predefined.

Associations

client	The element that is affected by the supplier element. In some cases (such as a Trace Abstraction) the direction is unimportant and serves only to distinguish the two elements.
supplier	Inverse of client. Designates the element that is unaffected by a change. In a two-way relationship (such as some Refinement Abstractions) this would be the more general element. In an undirected situation, such as a Trace Abstraction, the choice of client and supplier may be irrelevant.

2.5.2.16 Element

An element is an atomic constituent of a model.

In the metamodel, an Element is the top metaclass in the metaclass hierarchy. It has two subclasses: ModelElement and PresentationElement. Element is an abstract metaclass.

Tagged Values

documentation	Element	Documentation is a comment, description, or explanation of the element to which it is attached.

2.5.2.17 ElementOwnership

Element ownership defines the visibility of a ModelElement contained in a Namespace.

In the metamodel, ElementOwnership reifies the relationship between ModelElement and Namespace denoting the ownership of a ModelElement by a Namespace and its visibility outside the Namespace. See Section 2.5.2.27, “ModelElement,” on page 2-41.

Attributes

isSpecification	Specifies whether the ownedElement is part of the specification for the containing namespace (in cases where specification is distinguished from the realization). Otherwise the ownedElement is part of the realization. In cases in which the distinction is not made, the value is false by default.
visibility	Specifies whether the ModelElement can be seen and referenced by other ModelElements. Possibilities: • public - Any outside ModelElement can see the ModelElement. • protected - Any descendent of the ModelElement can see the ModelElement. • private - Only the ModelElement itself, or elements nested within it can see the ModelElement. • package - ModelElements declared in the same package (or a nested subpackage, to any level) as the given ModelElement can see the ModelElement. Note that use of an element in another Package may also be subject to access or import of its Package as described in Model Management; see Permission.

2.5.2.18 ElementResidence

Association class between Component and ModelElement that defines the set of ModelElements that specify a Component. See Component::resident. Shows that the component supports the element. The visibility attribute of ElementResidence defines the visibility of a resident element outside the component: an external Interface of a Component has visibility = ‘public’ for its ElementResidence association.

Attributes

visibility	Specifies whether a ModelElement that resides in a Component is visible externally. Possible values for ElementResidence visibility are: • public - Any resident ModelElement with public visibility is part of the Component’s external Interface and can be used by other elements, if they have permission to access or import the Component. • private - The ModelElement is internal to the Component and cannot be used by external elements. • protected - The ModelElement is only visible to Descendant Components. Note: the visibility values ‘package’ does not apply to Element Residence visibility. The Component and its residents have ElementOwnership associations with visibility values to the Package that contains them.

2.5.2.19 Enumeration

In the metamodel, Enumeration defines a kind of DataType whose range is a list of predefined values, called enumeration literals.

Enumeration literals can be copied, stored as values, and passed as arguments. They are ordered within their enumeration datatype. An enumeration literal can be compared for an exact match or to a range within its enumeration datatype. There is no other algebra defined on them (e.g., they cannot be added or subtracted).

The run-time instances of a Primitive datatype are Values. Each such value corresponds to exactly one EnumerationLiteral defined as part of the Enumeration type itself.

An Enumeration may have operations, but they must be pure functions (this is the rule for all DataType elements).

Associations

literal	An ordered set of EnumationLiteral elements, each specifying a possible value of an instance of the enumeration element.

2.5.2.20 EnumerationLiteral

An EnumerationLiteral defines an element of the run-time extension of an Enumeration data type. It has no relevant substructure, that is, it is atomic. The enumeration literals of a particular Enumeration datatype are ordered.

It has a name (inherited from ModelElement) that can be used to identify it within its enumeration dataype.

Note that an EnumerationLiteral is a ModelElement and may appear in (M1) models to define the structure of an Enumeration. In a run-time (M0) system, enumeration literals are DataValues in many-to-one correspondence to EnumerationLiterals that they represent. (This is a subtle but necessary distinction between M1 and M0 entities.)

The run-time values corresponding to enumeration literals can be compared for equality and for relative ordering or inclusion in a range of enumeration literals.

Associations

enumeration	The enumeration classifier of which this enumeration literal is an instance.

2.5.2.21 Feature

A feature is a property, like operation or attribute, which is encapsulated within a Classifier.

In the metamodel, a Feature declares a behavioral or structural characteristic of an Instance of a Classifier or of the Classifier itself. Feature is an abstract metaclass.

Attributes

name	(Inherited from ModelElement) The name used to identify the Feature within the Classifier or Instance. It must be unique across inheritance of names from ancestors including names of outgoing AssociationEnd. (See more specific rules for the exact details. Attributes, discriminators, and opposite association ends must have unique names in the set of inherited names. There may be multiple declarations of the same operation. Multiple operations may have the same name but different signatures; see the rules for precise details.)
ownerScope	Specifies whether Feature appears in each Instance of the Classifier or whether there is just a single instance of the Feature for the entire Classifier. Possibilities are: • instance - Each Instance of the Classifier holds its own value for the Feature. • classifier - There is just one value of the Feature for the entire Classifier.
visibility	Specifies whether the Feature can be used by other Classifiers. Visibilities of nested Classifiers combine so that the most restrictive visibility is the result. Possibilities: • public - Any outside Classifier with visibility to the Classifier can use the Feature. • protected - Any descendent of the Classifier can use the Feature. • private - Only the Classifier itself can use the Feature. • package - Any Classifier declared in the same package (or a nested subpackage, to any level) as the owner of the Feature can use the Feature.

Associations

owner	The Classifier declaring the Feature. Note that an Attribute may be owned by a Classifier (in which case it is a feature) or an AssociationEnd (in which case it is a qualifier) but not both.

2.5.2.22 Flow

A flow is a relationship between two versions of an object or between an object and a copy of it.

In the metamodel, a Flow is a child of Relationship. A Flow is a directed relationship from a source or sources to a target or targets.

Predefined stereotypes of Flow are «become» and «copy». Become relates one version of an object to another with a different value, state, or location. Copy relates an object to another object that starts as a copy of it.

Stereotypes

«become»	Class	Specifies a Flow relationship, source and target of which represent the same instance at different points in time, but each with potentially different values, state instance, and roles. A Become flow relationship from A to B means that instance A becomes B with possibly new values, state instance, and roles at a different moment in time/space.
«copy»	Class	Specifies a Flow relationship, the source and target of which are different instances, but each with the same values, state instance, and roles (but a distinct identity). A Copy flow relationship from A to B means that B is an exact copy of A. Future changes in A are not necessarily reflected in B.

2.5.2.23 *GeneralizableElement*

A generalizable element is a model element that may participate in a generalization relationship.

In the metamodel, a GeneralizableElement can be a generalization of other GeneralizableElements (i.e., all Features defined in and all ModelElements contained in the ancestors are also present in the GeneralizableElement). GeneralizableElement is an abstract metaclass.

Attributes

isAbstract	Specifies whether the GeneralizableElement may not have a direct instance. True indicates that an instance of the GeneralizableElement must be an instance of a child of the GeneralizableElement. False indicates that there may be an instance of the GeneralizableElement that is not an instance of a child. An abstract GeneralizableElement is not instantiable since it does not contain all necessary information. That is, it may not have a direct instance. It may have an indirect instance, and a model at a higher level of abstraction may include instances of an abstract type, with the understanding that in a fully expanded concrete snapshot, such instances would have concrete types that are descendants of the abstract types.
isLeaf	Specifies whether the GeneralizableElement is a GeneralizableElement with no descendents. True indicates that it may not have descendents, false indicates that it may have descendents (whether or not it actually has any descendents at the moment).
isRoot	Specifies whether the GeneralizableElement is a root GeneralizableElement with no ancestors. True indicates that it may not have ancestors, false indicates that it may have ancestors (whether or not it actually has any ancestors at the moment).

Associations

generalization	Designates a Generalization whose parent GeneralizableElement is the immediate ancestor of the current GeneralizableElement.
specialization	Designates a Generalization whose child GeneralizableElement is the immediate descendent of the current GeneralizableElement.

Inherited Features

The following elements are inherited by a child GenerizableElement.

constraint	All constraints on the parent apply to the child.

Non-Inherited Features

isRoot isLeaf isAbstract	Not inheritable by their very nature, but they define the generalization structure. IsRoot may be true only if there are no parents. IsLeaf may be true only if there are no children.
name	Each model element has a unique name.

2.5.2.24 Generalization

A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information.

In the metamodel, a Generalization is a directed inheritance relationship, uniting a GeneralizableElement with a more general GeneralizableElement in a hierarchy. Generalization is a subtyping relationship (i.e., an Instance of the more general GeneralizableElement may be substituted by an Instance of the more specific GeneralizableElement). See Inheritance for the consequences of Generalization relationships.

Attributes

discriminator	Designates the partition to which the Generalization link belongs. All of the Generalization links that share a given parent GeneralizableElement are divided into disjoint sets (that is, partitions) by their discriminator names. Each partition (a set of links sharing a discriminator name) represents an orthogonal dimension of specialization of the parent GeneralizableElement. The discriminator need not be unique. The empty string is also considered as a partition name, therefore all Generalization links have a discriminator. If the set of Generalization links that have the same parent all have the same name, then the children in the Generalization links are GeneralizableElements that specialize the parent, and an instance of any of them is a legal instance of the parent. Otherwise an indirect instance of the parent must be a (direct or indirect) instance of at least one element from each of the partitions.

Associations

child	Designates a GeneralizableElement that is the specialized version of the parent GeneralizableElement.
parent	Designates a GeneralizableElement that is the generalized version of the child GeneralizableElement.
powertype	Designates a Classifier that serves as a powertype for the child element along the dimension of generalization expressed by the Generalization. The child element is therefore an instance of the powertype element.

Stereotypes

«implementation» Class	Specifies that the child inherits the implementation of the parent (its attributes, operations and methods) but does not make public the supplier's interfaces nor guarantee to support them, thereby violating substitutability. This is private inheritance and is usually used only for programming implementation purposes.

Standard Constraints

complete Generalization	Specifies a constraint applied to a set of generalizations with the same discriminator and the same parent, indicating that any instance of the parent must be an instance of at least one child within the set of generalizations. If a parent has a single discriminator, the set of its child generalizations being complete implies that the parent is abstract. The connotation of declaring a set of generalizations complete is that all of the children with the given discriminator have been declared and that additional ones are not expected (in other words, the set of generalizations is closed), and designs may assume with some confidence that the set of children is fixed. If a new child is nevertheless added in the future, existing models may be adversely affected and may require modification.

***Standard Constraints** (continued)*

disjoint Generalization	Specifies a constraint applied to a set of generalizations, indicating that instance of the parent may be an instance of no more than one of the given children within the set of generalizations. This is the default semantics of generalization.
incomplete Generalization	Specifies a constraint applied to a set of generalizations with the same discriminator, indicating that an instance of the parent need not be an instance of a child within the set (but there is no guarantee that such an instance will actually exist). Being incomplete implies that the parent is concrete. The connotation of declaring a set of generalizations incomplete is that all of the children with the given discriminator have not necessarily been declared and that additional ones might be added, therefore users should not count on the set of children being fixed.
overlapping Generalization	Specifies a constraint applied to a set of generalizations, indicating that an instance of one child may be simultaneously an instance of another child in the set (but there is no guarantee that such an instance will actually exist).

2.5.2.25 Interface

An interface is a named set of operations that characterize the behavior of an element.

In the metamodel, an Interface contains a set of Operations that together define a service offered by a Classifier realizing the Interface. A Classifier may offer several services, which means that it may realize several Interfaces, and several Classifiers may realize the same Interface.

Interfaces are GeneralizableElements.

Interfaces may not have Attributes, Associations, or Methods. An Interface may participate in an Association provided the Interface cannot see the Association; that is, a Classifier (other than an Interface) may have an Association to an Interface that is navigable from the Classifier but not from the Interface.

Inherited Features

Interface inherits features as specified in Classifier.

2.5.2.26 Method

A method is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation.

In the metamodel, a Method is a declaration of a named piece of behavior in a Classifier and realizes one (directly) or a set (indirectly) of Operations of the Classifier.

There may be at most one method for a particular classifier (as owner of the method) and operation (as specification of the method) pairing.

Attributes

body	The implementation of the Method as a ProcedureExpression.

Associations

specification	Designates an Operation that the Method implements. The Operation must be owned by the Classifier that owns the Method or be inherited by it. The signatures of the Operation and Method must match.

2.5.2.27 ModelElement

A model element is an element that is an abstraction drawn from the system being modeled. Contrast with view element, which is an element whose purpose is to provide a presentation of information for human comprehension.

In the metamodel, a ModelElement is a named entity in a Model. It is the base for all modeling metaclasses in the UML (even though it is not displayed explicitly as such on diagrams for ElementOwnership, ElementResidence, ElementImport, TemplateParameter, TemplateArgument, and Argument). All other modeling metaclasses are either direct or indirect subclasses of ModelElement.

Each ModelElement can be regarded as a template. A template has a set of templateParameters that denotes which of the parts of a ModelElement are the template parameters. A ModelElement is a template when there is at least one template parameter. If it is not a template, a ModelElement cannot have template parameters. However, such embedded parameters are not usually complete and need not satisfy well-formedness rules. It is the arguments supplied when the template is instantiated that must be well formed.

Partially instantiated templates are allowed. This is the case when there are arguments provided for some, but not all templateParameters. A partially instantiated template is still a template, since it still has parameters.

Attributes

name	An identifier for the ModelElement within its containing Namespace.

Associations

asArgument	Indicates zero or more TemplateArgument for which the model element is an argument in a template binding.
clientDependency	Inverse of client. Designates a set of Dependency in which the ModelElement is a client.
constraint	A set of Constraints affecting the element.
implementationLocation	The component that an implemented model element resides in.

namespace	Designates the Namespace that contains the ModelElement. Every ModelElement except a root element must belong to exactly one Namespace or else be a composite part of another ModelElement (which is a kind of virtual namespace). The pathname of Namespace or ModelElement names starting from the root package provides a unique designation for every ModelElement. The association attribute visibility specifies the visibility of the element outside its namespace (see ElementOwnership).
presentation	A set of PresentationElements that present a view of the ModelElement.
supplierDependency	Inverse of supplier. Designates a set of Dependency in which the ModelElement is a supplier.
templateParameter	(association class TemplateParameter) A composite aggregation ordered list of parameters. Each parameter is a dummy ModelElement designated as a placeholder for a real ModelElement to be substituted during a binding of the template (see Binding). The real model element must be of the same kind (or a descendant kind) as the dummy ModelElement. The properties of the dummy ModelElement are ignored, except the name of the dummy element is used as the name of the template parameter. The association class TemplateParameter may be associated with a default ModelElement of the same kind as the dummy ModelElement. In the case of a Binding that does not supply an argument corresponding to the parameter, the value of the default ModelElement is used. If a Binding lacks an argument and there is no default ModelElement, the construct is invalid. Note that the template parameter element lacks structure. For example, a parameter that is a Class lacks Features; they are found in the actual argument.

Note that if a ModelElement has at least one templateParameter, then it is a template, otherwise it is an ordinary element.

Tagged Values

derived	A true value indicates that the model element can be completely derived from other model elements and is therefore logically redundant. In an analysis model, the element may be included to define a useful name or concept. In a design model, the usual intent is that the element should exist in the implementation to avoid the need for recomputation.

Inherited Features

ModelElement is not a GeneralizableElement but some of its descendants are. The following elements are inherited by children of elements that are GeneralizableElements.

constraint	The child is subject to all constraints of the parent.
presentation	The child is, by default, presented the same as the parent, but the presentation may be overridden.
stereotype	If a model element is classified by a stereotype, then its children are also classified by the stereotype. They may use the tags defined on the stereotype and they are subject to constraints imposed by the stereotype.
taggedValue	If a tag is defined to apply to a model element (for example, because it is classified by a stereotype defining the tag), then the tag applies to children of the model element..

Non-Inherited Features

clientDependency supplierDependency	A general inheritance rule is not possible
deploymentLocation	The set of locations may differ. Often it is more restrictive on the child.
implementationLocation	The child may be implemented differently from the parent.
name	Each model element has its own name. Names are not inherited.
namespace	The child and the parent may be in different namespaces.
templateParameter	A parent and child may have different template structure.

2.5.2.28 Namespace

A namespace is a part of a model that contains a set of ModelElements each of whose names designates a unique element within the namespace.

In the metamodel, a Namespace is a ModelElement that can own other ModelElements, like Associations and Classifiers. The name of each owned ModelElement must be unique within the Namespace. Moreover, each contained ModelElement is owned by at most one Namespace. The concrete subclasses of Namespace have additional constraints on which kind of elements may be contained. Namespace is an abstract metaclass.

Note that explicit parts of a model element, such as the features of a Classifier, are not modeled as owned elements in a namespace. A namespace is used for unstructured contents such as the contents of a package or a class declared inside the scope of another class.

Associations

ownedElement	(association class ElementOwnership) A set of ModelElements owned by the Namespace. Its visibility attribute states whether the element is visible outside the namespace.

2.5.2.29 Node

A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well, and upon which components may be deployed.

In the metamodel, a Node is a subclass of Classifier. It is associated with a set of Components that are deployed on the Node.

Associations

deployedComponent	The set of Components deployed on the Node.

Inherited Features

The following elements are inherited by a child Node, in addition to those specified under Classifier.

(none)

Non-Inherited Features

resident	The set of resident elements may differ. Often it is more restrictive on the child.

2.5.2.30 Operation

An operation is a service that can be requested from an object to effect behavior. An operation has a signature, which describes the actual parameters that are possible (including possible return values).

In the metamodel, an Operation is a BehavioralFeature that can be applied to the Instances of the Classifier that contains the Operation.

Attributes

concurrency	Specifies the semantics of concurrent calls to the same passive instance (i.e., an Instance originating from a Classifier with isActive=false). Active instances control access to their own Operations so this property is usually (although not required in UML) set to sequential. Possibilities include: • sequential - Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed. • guarded - Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence. The others are blocked until the performance of the first Operation is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks. Guarded Operations must perform correctly (or block themselves) in the case of a simultaneous sequential Operation or guarded semantics cannot be claimed. • concurrent - Multiple calls from concurrent threads may occur simultaneously to one Instance (on any concurrent Operation). All of them may proceed concurrently with correct semantics. Concurrent Operations must perform correctly in the case of a simultaneous sequential or guarded Operation or concurrent semantics cannot be claimed.
isAbstract	If true, then the operation does not have an implementation, and one must be supplied by a descendant. If false, the operation must have an implementation in the class or inherited from an ancestor.
isLeaf	If true, then the implementation of the operation may not be overriden by a descendant class. If false, then the implementation of the operation may be overridden by a descendant class (but it need not be overridden).
isRoot	If true, then the class must not inherit a declaration of the same operation. If false, then the class may (but need not) inherit a declaration of the same operation. (But the declaration must match in any case; a class may not modify an inherited operation declaration.)

Tagged Values

semantics	Operation	Semantics is the specification of the meaning of the operation.

2.5.2.31 Parameter

A parameter is an unbound variable that can be changed, passed, or returned. A parameter may include a name, type, and direction of communication. Parameters are used in the specification of operations, messages and events, templates, etc.

In the metamodel, a Parameter is a declaration of an argument to be passed to, or returned from, an Operation, a Signal, etc.

Attributes

defaultValue	An Expression whose evaluation yields a value to be used when no argument is supplied for the Parameter.
kind	Specifies what kind of a Parameter is required. Possibilities are: • in - An input Parameter (may not be modified). • out - An output Parameter (may be modified to communicate information to the caller). • inout - An input Parameter that may be modified. • return -A return value of a call.
name	(Inherited from ModelElement) The name of the Parameter, which must be unique within its containing Parameter list.

Associations

type	Designates a Classifier to which an argument value must conform.

2.5.2.32 Permission

Permission is a kind of dependency. It grants a model element permission to access elements in another namespace.

In the metamodel, Permission in a Dependency between a client ModelElement and a supplier ModelElement. The client receives permission to reference the supplier's contents. The supplier must be a Namespace.

The predefined stereotypes of Permission are access, import, and friend.

In the case of the access and import stereotypes, the client is granted permission to reference elements in the supplier namespace with public visibility. In the case of the import stereotype, the public names in the supplier namespace are added to the client namespace. An element may also access any protected contents of an ancestor namespace. An element may also access any contents (public, protected, private, or package) of its own namespace or a containing namespace.

In the case of the friend stereotype, the client is granted permission to reference elements in the supplier namespace, regardless of visibility.

Stereotypes

«access»	Class	Access is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target namespace are accessible to the namespace of the source package.
«friend»	Class	Friend is a stereotyped permission dependency whose source is a model element, such as an operation, class, or package, and whose target is a model element in a different package, such as an operation, class, or package. A friend relationship grants the source access to the target regardless of the declared visibility. It extends the visibility of the supplier so that the client can see into the supplier.
«import»	Class	Import is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target package are added to the namespace of the source package.

2.5.2.33 *PresentationElement*

A presentation element is a textual or graphical presentation of one or more model elements.

In the metamodel, a PresentationElement is an Element which presents a set of ModelElements to a reader. It is the base for all metaclasses used for presentation. All other metaclasses with this purpose are either direct or indirect subclasses of PresentationElement. PresentationElement is an abstract metaclass. The subclasses of this class are proper to a graphic editor tool and are not specified here. It is a stub for their future definition.

2.5.2.34 *Primitive*

A Primitive defines a predefined DataType, without any relevant UML substructure (i.e., it has no UML parts). A primitive datatype may have an algebra and operations defined outside of UML, for example, mathematically. Primitive datatypes used in UML itself include Integer, UnlimitedInteger, and String.

The run-time instances of a Primitive dataype are DataValues. The values are in many-to-one correspondence to mathemetical elements defined outside of UML (for example, the various integers).

2.5.2.35 *ProgrammingLanguageDataType*

A data type is a type whose values have no identity (i.e., they are pure values). A programming language data type is a data type specified according to the semantics of a particular programming language, using constructs available in that language. There are a wide variety of programming languages and many of them include type constructs not included as UML classifiers. In some cases, it is important to represent those constructs such that their exact form in the programming language is available. The ProgrammingLanguagedData type captures such programming language types in a language-dependent fashion. They are represented by the name of the language and a string characterizing them, subject to interpretation by the particular language. Because

they are dependent on particular languages, they are not portable among languages (except by agreement among the languages) and they do not map into other UML classifiers. Their semantics is therefore opaque within UML except by special interpretation by a profile intended for the particular language.

Note that many or most programming language types can be directly represented using other UML classifiers, and such representation makes available deeper semantic analysis.

A ProgrammingLanguageDataType may omit its name. Two ProgrammingLanguageDataType elements without names are not considered equivalent.

Attributes

expression	An expression for the ProgrammingLanguageDataType expressed in its particular programming language.

Inherited Features

ProgrammingLanguageDataType is meant to define language-dependent constructs for which inheritance properties are undefined in UML.

2.5.2.36 Relationship

A relationship is a connection among model elements.

In the metamodel, Relationship is a term of convenience without any specific semantics. It is abstract.

Children of Relationship are Association, Dependency, Flow, and Generalization.

2.5.2.37 StructuralFeature

A structural feature refers to a static feature of a model element, such as an attribute.

In the metamodel, a StructuralFeature declares a structural aspect of an Instance of a Classifier, such as an Attribute. For example, it specifies the multiplicity and changeability of the StructuralFeature. StructuralFeature is an abstract metaclass.

Attributes

changeability	Whether the value may be modified after the object is initialized. Possibilities are: • changeable - No restrictions on modification. • frozen - No values may be added or removed after the object is initialized. • addOnly - Values may be added anytime. No values may be removed after the object is initialized.
multiplicity	The possible number of data values for the feature that may be held by an instance. The cardinality of the set of values is an implicit part of the feature. In the common case in which the multiplicity is 1..1, then the feature is a scalar (i.e., it holds exactly one value).
ordering	Specifies whether the set of instances is ordered. The ordering must be determined and maintained by Operations that add values to the feature. This property is only relevant if the multiplicity is greater than one. Possibilities are: • unordered - The instances form a set with no inherent ordering. • ordered - A set of ordered instances can be scanned in order. • Other possibilities (such as sorted) may be defined later by declaring additional keywords. As with user-defined stereotypes, this would be a private extension supported by particular editing tools.
targetScope	Specifies whether the targets are ordinary Instances or Classifiers. Possibilities are: • instance - Each value contains a reference to an Instance of the target Classifier. This is the setting for a normal Attribute. • classifier - Each value contains a reference to the target Classifier itself. This represents a way to store meta-information.

Associations

type	Designates the classifier whose instances are values of the feature. Must be a Class, Interface, or DataType. The actual type may be a descendant of the declared type or (for an Interface) a Class that realizes the declared type.

Tagged Values

persistence Attribute	Persistence denotes the permanence of the state of the feature, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

2.5.2.38 *TemplateArgument*

Reifies the relationship between a Binding and one of its arguments (a ModelElement).

Associations

binding	The Binding that owns the template argument.
modelElement	The actual argument for the subject Binding.

2.5.2.39 TemplateParameter

Defines the relationship between a template (a ModelElement) and its parameter (a ModelElement). A ModelElement with at least one templateParameter association is a template (by definition).

In the metamodel, TemplateParameter reifies the relationship between a ModelElement that is a template and a ModelElement that is a dummy placeholder for a template argument. See Section 2.5.2.27, “ModelElement,” on page 2-41, association templateParameter, for details.

Associations

defaultElement	An optional default value ModelElement. In case of a Binding of the template ModelElement in the reified TemplateParameter class association, the defaultElement is used as the argument of the bound element if no argument is supplied for the corresponding template parameter. If no argument is supplied and there is no default value, the model is ill formed.

2.5.2.40 Usage

A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. The relationship is not a mere historical artifact, but an ongoing need; therefore, two elements related by usage must be in the same model.

In the metamodel, a Usage is a Dependency in which the client requires the presence of the supplier. How the client uses the supplier, such as a class calling an operation of another class, a method having an argument of another class, and a method from a class instantiating another class, is defined in the description of the particular Usage stereotype.

Various stereotypes of Usage are predefined, but the set is open-ended and may be added to.

Stereotypes

«call»	Class	Call is a stereotyped usage dependency whose source is an operation and whose target is an operation. The relationship may also be subsumed to the class containing an operation, with the meaning that there exists an operation in the class to which the dependency applies. A call dependency specifies that the source operation or an operation in the source class invokes the target operation or an operation in the target class. A call dependency may connect a source operation to any target operation that is within scope including, but not limited to, operations of the enclosing classifier and operations of other visible classifiers.
«create»	Class	Create is a stereotyped usage dependency denoting that the client classifier creates instances of the supplier classifier.
«instantiate»	Class	A stereotyped usage dependency among classifiers indicating that operations on the client create instances of the supplier.
«send»	Class	Send is a stereotyped usage dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.

2.5.3 Well-formedness Rules

The following well-formedness rules apply to the Core package.

2.5.3.1 Association

[1] The AssociationEnds must have a unique name within the Association.

```
self.allConnections->forAll( r1, r2 | r1.name = r2.name implies r1 = r2 )
```

[2] At most one AssociationEnd may be an aggregation or composition.

```
self.allConnections->select(aggregation <#none)->size <= 1
```

[3] If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition.

```
self.allConnections->size >=3 implies
    self.allConnections->forall(aggregation = #none)
```

[4] The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association, or be Classifiers with public visibility in other Namespaces to which the Namespace of the Association has “access” Permissions.

```
self.allConnections->forAll(r | self.namespace.allContents->includes (r.participant) ) or
    self.allConnections->forAll(r | self.namespace.allContents->excludes (r.participant) implies
    self.namespace.clientDependency->exists (d |
        d.oclIsTypeOf(Permission) and
        d.stereotype.name = 'access' and
        d.supplier.oclAsType(Namespace).ownedElement->select (e |
                e.elementOwnership.visibility =
                    #public)->includes (r.participant)  or
            d.supplier.oclAsType(GeneralizableElement).
                allParents.oclAsType(Namespace).ownedElement->select (e |
                    e. elementOwnership.visibility =
                        #public)->includes (r.participant) or
            d.supplier.oclAsType(Package).allImportedElements->select (e  |
                e. elementImport.visibility =
                    #public) ->includes (r.participant) ) )
```

Additional operations

[1] The operation allConnections results in the set of all AssociationEnds of the Association.

```
allConnections : Set(AssociationEnd);
allConnections = self.connection
```

2.5.3.2 AssociationClass

[1] The names of the AssociationEnds and the StructuralFeatures do not overlap.

```
self.allConnections->forAll( ar |
    self.allFeatures->forAll( f |
    f.oclIsKindOf(StructuralFeature) implies ar.name <> f.name ))
```

[2] An AssociationClass cannot be defined between itself and something else.

```
self.allConnections->forAll(ar | ar.participant <> self)
```

Additional operations

[1] The operation allConnections results in the set of all AssociationEnds of the AssociationClass, including all connections defined by its parent (transitive closure).

```
allConnections : Set(AssociationEnd);
allConnections = self.connection->union(self.parent->select
    (s | s.oclIsKindOf(Association))->collect (a : Association |
        a.allConnections))->asSet
```

2.5.3.3 AssociationEnd

[1] The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end.

```
(self.participant.oclIsKindOf (Interface) or
self.participant.oclIsKingOf (DataType)) implies
    self.association.connection->select
        (ae | ae <> self)->forAll(ae | ae.isNavigable = #false)
```

[2] An Instance may not belong by composition to more than one composite Instance.

```
self.aggregation = #composite implies self.multiplicity.upperbound = 1
```

Additional operations

[1] The operation upperbound returns the maximum upperbound value across all potential ranges of a multiplicity.

```
upperbound( ) : UnlimitedInteger;
upperbound =
    self.range->exists(r : MultiplicityRange | r.upper = result) and
    self.range->forall(r : MultiplicityRange | r.upper <= result)
```

2.5.3.4 Attribute

[1] Qualifier attributes have multiplicity of 1..1

```
self.associationEnd->notEmpty() implies self.multiplicity.is(1,1)
```

2.5.3.5 BehavioralFeature

[1] All Parameters should have a unique name.

```
self.parameter->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)
```

[2] The type of the Parameters should be included in the Namespace of the Classifier.

```
self.parameter->forAll( p |
    self.owner.namespace.allContents->includes (p.type) )
```

Additional operations

[1] The operation hasSameSignature checks if the argument has the same signature as the instance itself.

```
hasSameSignature ( b : BehavioralFeature ) : Boolean;
hasSameSignature (b) =
    (self.name = b.name) and
    (self.parameter->size = b.parameter->size) and
    Sequence{ 1..(self.parameter->size) }->forAll( index : Integer |
        b.parameter->at(index).type =
            self.parameter->at(index).type and
        b.parameter->at(index).kind =
            self.parameter->at(index).kind
    )
```

[2] The operation matchesSignature checks if the argument has a signature that would clash with the signature of the instance itself (and therefore must be unique). Mismatches in kind or any differences in return parameters do not cause a mismatch:

```
matchesSignature ( b : BehavioralFeature ) : Boolean;
matchesSignature (b) =
    (self.name = b.name) and
    (self.parameter->size = b.parameter->size) and
    Sequence{ 1..(self.parameter->size) }->forAll( index : Integer |
        b.parameter->at(index).type =
            self.parameter->at(index).type or
        (b.parameter->at(index).kind = return and
            self.parameter->at(index).kind = return)
    )
```

2.5.3.6 Binding

[1] The client ModelElement must conform to the type of the supplier ModelElement in a Binding.

```
self.client.oclIsKindOf(self.supplier)
```

[2] Each argument ModelElement of the supplier must have the same type (or a descendant of the type) of the corresponding supplier parameter ModelElement in a Binding.

```
let range : Set(Integer) = [1..self.arguments->size()] in
range->forAll(index |
    arguments->at(index).oclIsKindOf(
        supplier.templateParameter->at(index).oclType
```

[3] The number of arguments must equal the number of parameters.

```
self.arguments->size() = self.supplier.templateParameter->size()
```

[4] A Binding has one client and one supplier.

```
(self.client->size = 1) and (self.supplier->size = 1)
```

[5] A ModelElement may participate in at most one Binding as a client.

```
Binding.allInstances->forAll
  [b1, b2 | (b1 <> b2) implies (b1.client <> b2.client)]
```

2.5.3.7 Class

[1] If a Class is concrete, all the Operations of the Class should have a realizing Method in the full descriptor.

```
not self.isAbstract implies self.allOperations->forAll (op |
self.allMethods->exists (m | m.specification->includes(op)))
```

[2] A Class can only contain Classes, Associations, Generalizations, UseCases, Constraints, Dependencies, Collaborations, DataTypes, and Interfaces as a Namespace.

```
self.allContents->forAll(c | )
    c.oclIsKindOf(Class         )or
    c.oclIsKindOf(Association   )   or
    c.oclIsKindOf(Generalization)or
    c.oclIsKindOf(UseCase               ) or
    c.oclIsKindOf(Constraint    )or
    c.oclIsKindOf(Dependency    )or
    c.oclIsKindOf(Collaboration )or
    c.oclIsKindOf(DataType ) or
    c.oclIsKindOf(Interface     )
```

2.5.3.8 Classifier

[1] No BehavioralFeature of the same kind may match the same signature in a Classifier.

```
self.feature->forAll(f, g |
(
    (
        (f.oclIsKindOf(Operation) and g.oclIsKindOf(Operation)) or
        (f.oclIsKindOf(Method   ) and g.oclIsKindOf(Method   )) or
        (f.oclIsKindOf(Reception) and g.oclIsKindOf(Reception))
    ) and
    f.oclAsType(BehavioralFeature).matchesSignature(g)
)
implies f = g)
```

[2] No Attributes may have the same name within a Classifier

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( p, q |
    p.name = q.name implies p = q )
```

[3] No opposite AssociationEnds may have the same name within a Classifier.

```
self.allOppositeAssociationEnds->forAll ( p, q | p.name = q.name implies p = q )
```

[4] The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier.

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( a |
    not self.allOppositeAssociationEnds->union (self.allContents)->collect ( q |
        q.name )->includes (a.name) )
```

[5] The name of an opposite AssociationEnd may not be the same as the name of an Attribute or a ModelElement contained in the Classifier.

```
self.oppositeAssociationEnds->forAll ( o |
    not self.allAttributes->union (self.allContents)->collect ( q |
        q.name )->includes (o.name) )
```

[6] For each Operation in a specification realized by the Classifier, the Classifier must have a matching Operation.

```
self.specification.allOperations->forAll (interOp |
    self.allOperations->exists( op | op.hasMatchingSignature (interOp) ) )
```

[7] All of the generalizations in the range of a powertype have the same discriminator.

```
self.powertypeRange->forAll
    (g1, g2 | g1.discriminator = g2.discriminator)
```

[8] Discriminator names must be distinct from attribute names and opposite AssociationEnd names.

```
self.allDiscriminators->intersection (self.allAttributes.name->union
(self.allOppositeAssociationEnds.name))->isEmpty
```

Additional operations

[1] The operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features.

```
allFeatures : Set(Feature);
allFeatures = self.feature->union(
                    self.parent.oclAsType(Classifier).allFeatures)
```

[2] The operation allOperations results in a Set containing all Operations of the Classifier itself and all its inherited Operations.

```
allOperations : Set(Operation);
allOperations = self.allFeatures->select(f | f.oclIsKindOf(Operation))
```

[3] The operation allMethods results in a Set containing all Methods of the Classifier itself and all its inherited Methods.

```
allMethods : set(Method);
allMethods = self.allFeatures->select(f | f.oclIsKindOf(Method))
```

[4] The operation allAttributes results in a Set containing all Attributes of the Classifier itself and all its inherited Attributes.

```
allAttributes : set(Attribute);
allAttributes = self.allFeatures->select(f | f.oclIsKindOf(Attribute))
```

[5] The operation associations results in a Set containing all Associations of the Classifier itself.

```
associations : set(Association);
associations = self.association.association->asSet
```

[6] The operation allAssociations results in a Set containing all Associations of the Classifier itself and all its inherited Associations.

```
allAssociations : set(Association);
allAssociations = self.associations->union (
                        self.parent.oclAsType(Classifier).allAssociations)
```

[7] The operation oppositeAssociationEnds results in a set of all AssociationEnds that are opposite to the Classifier.

```
oppositeAssociationEnds : Set (AssociationEnd);
oppositeAssociationEnds =
    self.associations->select ( a | a.connection->select ( ae |
        ae.participant = self ).size = 1 )->collect ( a |
            a.connection->
                select ( ae | ae.participant <> self ) )->union (
    self.associations->select ( a | a.connection->select ( ae |
        ae.participant = self ).size > 1 )->collect ( a |
            a.connection) )
```

[8] The operation allOppositeAssociationEnds results in a set of all AssociationEnds, including the inherited ones, that are opposite to the Classifier.

```
allOppositeAssociationEnds : Set (AssociationEnd);
allOppositeAssociationEnds = self.oppositeAssociationEnds->union (
        self.parent.allOppositeAssociationEnds )
```

[9] The operation specification yields the set of Classifiers that the current Classifier realizes.

```
specification: Set(Classifier)
specification = self.clientDependency->
    select(d |
        d.oclIsKindOf(Abstraction)
        and d.stereotype.name = "realization"
        and d.supplier.oclIsKindOf(Classifier))
    .supplier.oclAsType(Classifier)
```

[10] The operation allContents returns a Set containing all ModelElements contained in the Classifier together with the contents inherited from its parents.

```
allContents : Set(ModelElement);
allContents = self.contents->union(
    self.parent.allContents->select(e |
        e.elementOwnership.visibility = #public or
        e.elementOwnership.visibility = #protected))
```

[11] The operation allDiscriminators results in a Set containing all Discriminators of the Generalizations from which the Classifier is descended itself and all its inherited Features.

```
allDiscriminators : Set(Name);
allDiscriminators = self.generalization.discriminator->union(
                                self.parent.oclAsType(Classifier).allDiscriminators)
```

2.5.3.9 Comment

No extra well-formedness rules.

2.5.3.10 Component

[1] A Component may only contain other Components in its namespace.

```
self.allContents->forAll( c | c.oclIsKindOf(Component))
```

[2] A Component does not have any Features.

```
self.feature->isEmpty
```

[3] A Component may only have as residents DataTypes, Interfaces, Classes, Associations, Dependencies, Constraints, Signals, DataValues and Objects.

```
self.allResidentElements->forAll( re |
    re.oclIsKindOf(DataType) or
    re.oclIsKindOf(Interface) or
    re.oclIsKindOf(Class) or
    re.oclIsKindOf(Association) or
    re.oclIsKindOf(Dependency) or
    re.oclIsKindOf(Constraint) or
    re.oclIsKindOf(Signal) or
    re.oclIsKindOf(DataValue) or
    re.oclIsKindOf(Object) )
```

Additional operations

[1] The operation allResidentElements results in a Set containing all ModelElements resident in a Component or one of its ancestors.

```
allResidentElements : set(ModelElement)
    allResidentElements = self.resident->union(
    self.parent.oclAsType(Component).allResidentElements->select( re |
    re.elementResidence.visibility = #public or
re.elementResidence.visibility = #protected))
```

2.5.3.11 Constraint

[1] A Constraint cannot be applied to itself.

```
not self.constrainedElement->includes (self)
```

2.5.3.12 DataType

[1] A DataType can only contain Operations, which all must be queries.

```
self.allFeatures->forAll(f |
                    f.oclIsKindOf(Operation) and f.oclAsType(Operation).isQuery)
```

[2] A DataType cannot contain any other ModelElements.

```
self.allContents->isEmpty
```

2.5.3.13 Dependency

No extra well-formedness rules.

2.5.3.14 Element

No extra well-formedness rules.

2.5.3.15 ElementOwnership

No additional well-formedness rules.

2.5.3.16 ElementResidence

No additional well-formedness rules.

2.5.3.17 Enumeration

No additional well-formedness rules.

2.5.3.18 EnumerationLiteral

No additional well-formedness rules.

2.5.3.19 Feature

No extra well-formedness rules.

2.5.3.20 GeneralizableElement

[1] A root cannot have any Generalizations.

```
self.isRoot implies self.generalization->isEmpty
```

[2] No GeneralizableElement can have a parent Generalization to an element that is a leaf.

```
self.parent->forAll(s | not s.isLeaf)
```

[3] Circular inheritance is not allowed.

```
not self.allParents->includes(self)
```

[4] The parent must be included in the Namespace of the GeneralizableElement.

```
self.generalization->forAll(g |
    self.namespace.allContents->includes(g.parent) )
```

[5] A GeneralizableElement may only be a child of GeneralizableElement of the same kind.

```
self.generalization.parent->forAll(p | self.oclIsKindOf(p))
```

Additional Operations

[1] The operation parent returns a Set containing all direct parents.

```
parent : Set(GeneralizableElement);
parent = self.generalization.parent
```

[2] The operation allParents returns a Set containing all the Generalizable Elements inherited by this GeneralizableElement (the transitive closure), excluding the GeneralizableElement itself.

```
allParents : Set(GeneralizableElement);
allParents = self.parent->union(self.parent.allParents)
```

2.5.3.21 Generalization

No extra well-formedness rules.

2.5.3.22 ImplementationClass (stereotype of Class)

[1] All direct instances of an implementation class must not have any other Classifiers that are implementation classes.

```
self.instance.forall(i | i.classifier.forall(c |
    c.stereotype.name = "implementationClass" implies c = self))
```

[2] A parent of an implementation class must be an implementation class.

```
self.parent->forAll(stereotype.name="implementationClass")
```

2.5.3.23 Interface

[1] An Interface can only contain Operations.

```
self.allFeatures->forAll(f |
    f.oclIsKindOf(Operation) or f.oclIsKindOf(Reception))
```

[2] An Interface cannot contain any ModelElements.

```
self.allContents->isEmpty
```

[3] All Features defined in an Interface are public.

```
self.allFeatures->forAll ( f | f.visibility = #public )
```

2.5.3.24 Method

[1] If the realized Operation is a query, then so is the Method.

```
self.specification->isQuery implies self.isQuery
```

[2] The signature of the Method should be the same as the signature of the realized Operation.

```
self.hasSameSignature (self. specification)
```

[3] The visibility of the Method should be the same as for the realized Operation.

```
self.visibility = self.specification.visibility
```

[4] The realized Operation must be a feature (possibly inherited) of the same Classifier as the Method.

```
self.owner.allOperations->includes(self.specification)
```

[5] If the realized Operation has been overridden one or more times in the ancestors of the owner of the Method, then the Method must realize the latest overriding (that is, all other Operations with the same signature must be owned by ancestors of the owner of the realized Operation).

```
self.specification.owner.allOperations->includesAll
    (self.owner.allOperations->select(op |
        self.hasSameSignature(op)))
```

[6] There may be at most one method for a given classifier (as owner of the method) and operation (as specification of the method) pair.

```
self.owner.allMethods->select(operation = self.operation)->size = 1
```

2.5.3.25 ModelElement

That part of the model owned by a template is not subject to all well-formedness rules. A template is not directly usable in a well formed model. The results of binding a template are subject to well-formedness rules.

```
(not expressed in OCL)
```

Additional operations

[1] The operation supplier results in a Set containing all direct suppliers of the ModelElement.

```
supplier : Set(ModelElement);
supplier = self.clientDependency.supplier
```

[2] The operation allSuppliers results in a Set containing all the ModelElements that are suppliers of this ModelElement, including the suppliers of these Model Elements. This is the transitive closure.

```
allSuppliers : Set(ModelElement);
allSuppliers = self.supplier->union(self.supplier.allSuppliers)
```

[3] The operation “model” results in the set of Models to which the ModelElement belongs.

```
model : Set(Model);
model = self.namespace->union(self.namespace.allSurroundingNamespaces)
            ->select( ns|
                    ns.oclIsKindOf (Model))
```

[4] A ModelElement is a template when it has parameters.

```
isTemplate : Boolean;
isTemplate = (self.templateParameter->notEmpty)
```

[5] A ModelElement is an instantiated template when it is related to a template by a Binding relationship.

```
isInstantiated : Boolean;
isInstantiated = self.clientDependency->select(
    oclIsKindOf(Binding))->notEmpty
```

[6] The templateArguments are the arguments of an instantiated template, which substitute for template parameters.

```
templateArguments : Set(ModelElement);
templateArguments = self.clientDependency->
            select(oclIsKindOf(Binding)).oclAsType(Binding).argument
```

2.5.3.26 Namespace

[1] If a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace.

```
self.allContents->forAll(me1, me2 : ModelElement |
    ( not me1.oclIsKindOf (Association) and not me2.oclIsKindOf (Association) and
        me1.name <> '' and me2.name <> '' and me1.name = me2.name
    ) implies
        me1 = me2 )
```

[2] All Associations must have a unique combination of name and associated Classifiers in the Namespace.

```
self.allContents -> select(oclIsKindOf(Association))->
    forAll(a1, a2 |
        a1.name = a2.name and
        a1.connection.participant = a2.connection.participant
        implies a1 = a2)
```

Additional operations

[1] The operation contents results in a Set containing all ModelElements contained by the Namespace.

```
contents : Set(ModelElement)
contents = self.ownedElement -> union(self.namespace, contents)
```

[2] The operation allContents results in a Set containing all ModelElements contained by the Namespace.

```
allContents : Set(ModelElement);
allContents = self.contents
```

[3] The operation allVisibleElements results in a Set containing all ModelElements visible outside of the Namespace.

```
allVisibleElements : Set(ModelElement)
allVisibleElements = self.allContents -> select(e |
    e.elementOwnership.visibility = #public)
```

[4] The operation allSurroundingNamespaces results in a Set containing all surrounding Namespaces.

```
allSurroundingNamespaces : Set(Namespace)
allSurroundingNamespaces =
self.namespace->union(self.namespace.allSurroundingNamespaces)
```

2.5.3.27 Node

No extra well-formedness rules.

2.5.3.28 Operation

No additional well-formedness rules.

2.5.3.29 Parameter

No additional well-formedness rules.

2.5.3.30 PresentationElement

No extra well-formedness rules.

2.5.3.31 Primitive

No additional well-formedness rules.

2.5.3.32 StructuralFeature

[1] The connected type should be included in the owner's Namespace.

```
self.owner.namespace.allContents->includes(self.type)
```

[2] The type of a StructuralFeature must be a Class, DataType, or Interface.

```
self.type.oclIsKindOf(Class) or
self.type.oclIsKindOf(DataType) or
self.type.oclIsKindOf(Interface)
```

2.5.3.33 Trace

A trace is an Abstraction with the «trace» stereotype. These are the additional constraints due to the stereotype.

[1] The client ModelElements of a Trace must all be from the same Model.

```
self.client->forAll(e1, e2 | e1.model = e2.model)
```

[2] The supplier ModelElements of a Trace must all be from the same Model.

```
self.supplier->forAll(e1, e2 | e1.model = e2.model)
```

[3] The client and supplier ModelElements must be from two different Models.

```
self.client.model <> self.supplier.model
```

[4] The client and supplier ModelElements must all be from models of the same system.

```
self.client.model.intersection(self.supplier.model) <> Set{}
```

2.5.3.34 Type (stereotype of Class)

[1] A Type may not have any Methods.

```
not self.feature->exists(oclIsKindOf(Method))
```

[2] The parent of a type must be a type.

```
self.parent->forAll(stereotype.name = "type")
```

2.5.3.35 Usage

No extra well-formedness rules.

2.5.4 Detailed Semantics

This section provides a description of the dynamic semantics of the elements in the Core. It is structured based on the major constructs in the core, such as interface, class, and association.

2.5.4.1 Association

An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association ends, each specifying a connected classifier and a set of properties that must be fulfilled for the relationship to be valid. The multiplicity property of an association end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance at each of the other ends, with single qualifier values at all qualified ends. A multiplicity is a range of nonnegative integers. When multiplicity is enforced is a semantic variation point. For example, implementations might allow violations of minimum multiplicity during object initialization.

The association end also states whether or not a link may be traversed towards the object on that end from the objects on the other ends of the link (isNavigable). Navigability does not apply to getting an end object from a link object, that is, a link of an association class, because once a link object is obtained, the navigation has already taken place.

The visibility of an association end specifies whether procedures and actions owned by other classifiers can navigate links of the association. Navigation is constrained by the visibility of the end being read. The options are relative to the classifiers at the other ends of the association. The association end may limit navigation of links to

- procedures and actions owned by all classifiers (public),
- classifiers at the other ends and their children (protected),
- classifiers at the other ends and not their children (private), or
- classifiers in the same package, or a nested subpackage, to any level (package).

Visibility does not apply to getting an end object from a link object, that is, a link of an association class, because once a link object is visible to a procedure or action, all its ends are visible.

An association end also specifies whether or not links may be created or destroyed after the initialization of objects at the opposite ends. The association end may state

- that no constraints exist (changeable),
- that a link may not be destroyed after the objects at the opposite ends have been initialized, and that new links may not be created after the objects that would participate in the new link at the opposite ends have been initialized (frozen), or,
- that a link may not be destroyed after the objects at the opposite ends have been initialized (addOnly).

Note that the semantics of frozen requires that objects participating in links with two or more frozen ends cannot have links created unless all the linked objects are being initialized. Changeability constraints affect when links may be created or destroyed, not whether links themselves are mutable. Links are not mutable once they are created, except that they can be destroyed and reordered. Qualifier values, end objects, and link classifier, if any, of a link cannot be changed once a link is created. Changeability constraints also do not affect the modifiability of the objects that are attached to the links, or the classifiers participating in the association.

The ordering attribute of association end states that if the instances related to a single instance at each of the other ends, with single qualifier values at all qualified ends, have an ordering that must be preserved, the order of insertion of new links must be specified by operations that add or modify links. Note that sorting is a performance optimization and is not an example of a logically ordered association, because the ordering information in a sort does not add any information.

In UML, Associations can be of three different kinds: 1) ordinary association, 2) composite aggregate, and 3) shareable aggregate. Since the aggregate construct can have several different meanings depending on the application area, UML gives a more precise meaning to two of these constructs (i.e., association and composite aggregate) and leaves the shareable aggregate more loosely defined in between.

An association may represent an aggregation (i.e., a whole/part relationship). In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations. Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. This means that the composite object is responsible for the creation and destruction of the parts. In implementation terms, it is responsible for their memory allocation. If a composite object is destroyed, it must destroy all of its parts. It may remove a part and give it to another composite object, which then assumes responsibility for it. If the multiplicity from a part to composite is zero-to-one, the composite may remove the part, and the part may assume responsibility for itself, otherwise it may not live apart from a composite.

A consequence of these rules is that a composite implies propagation semantics (i.e., some of the dynamic semantics of the whole is propagated to its parts). For example, if the whole is copied or destroyed, then so are the parts as well (because a part may belong to at most one composite).

A classifier on the composite end of an association may have parts that are classifiers and associations. At the instance level, an instance of a part element is considered “part of” the instance of a composite element. If an association is part of a composite and it connects two classes that are also part of the same composite, then a link of the association will connect objects that are part of the same composite object of which the link is part.

A shareable aggregation denotes weak ownership (i.e., the part may be included in several aggregates) and its owner may also change over time. However, the semantics of a shareable aggregation does not imply deletion of the parts when an aggregate referencing it is deleted. Both kinds of aggregations define a transitive, antisymmetric relationship (i.e., the instances form a directed, non-cyclic graph). Composition instances form a strict tree (or rather a forest).

A qualifier declares a partition of the set of associated instances with respect to an instance at the qualified end (the qualified instance is at the end to which the qualifier is attached). A qualifier instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. In the common case in which the multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and designates at most one associated object. In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets, each selected by a given qualifier instance. In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences. In the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value.

Note that the multiplicity of a qualifier is given assuming that the qualifier value is supplied. The “raw” multiplicity without the qualifier is assumed to be 0..*. This is not fully general but it is almost always adequate, as a situation in which the raw multiplicity is 1 would best be modeled without a qualifier.

Note also that a qualified multiplicity whose lower bound is zero indicates that a given qualifier value may be absent, while a lower bound of 1 indicates that any possible qualifier value must be present. The latter is reasonable only for qualifiers with a finite number of values (such as enumerated values or integer ranges) that represent full tables indexed by some finite range of values.

2.5.4.2 AssociationClass

An association may be refined to have its own set of features (i.e., features that do not belong to any of the connected classifiers) but rather to the association itself. Such an association is called an association class. It will be both an association, connecting a set of classifiers, and a class, and as such have features and be included in other associations. The semantics of such an association is a combination of the semantics of an ordinary association and of a class.

The AssociationClass construct can be expressed in a few different ways in the metamodel (e.g., as a subclass of Class, as a subclass of Association, or as a subclass of Classifier). Since an AssociationClass is a construct being both an association (having a set of association-ends) and a class (declaring a set of features), the most accurate way of expressing it is as a subclass of both Association and Class. In this way, AssociationClass will have all the properties of the other two constructs. Moreover, if new kinds of associations containing features (e.g., AssociationDataType) are to be included in UML, these are easily added as subclasses of Association and the other Classifier.

The terms child, subtype, and subclass are synonyms and mean that an instance of a classifier being a subtype of another classifier can always be used where an instance of the latter classifier is expected. The neutral terms *parent* and *child*, with the transitive closures *ancestor* and *descendant*, are the preferred terms in this document.

2.5.4.3 Class

The purpose of a class is to declare a collection of methods, operations, and attributes that fully describe the structure and behavior of objects. All objects instantiated from a class will have attribute values matching the attributes of the full class descriptor and support the operations found in the full class descriptor. Some classes may not be directly instantiated. These classes are said to be abstract and exist only for other classes to inherit and reuse the features declared by them. No object may be a direct instance of an abstract class, although an object may be an indirect instance of one through a subclass that is non-abstract.

When a class is instantiated to create a new object, a new instance is created, which is initialized containing an attribute value for each attribute found in the full class descriptor. The object is also initialized with a connection to the list of methods in the full class descriptor.

Note – An actual implementation behaves as if there were a full class descriptor, but many clever optimizations are possible in practice.

Finally, the identity of the new object is returned to the creator. The identity of every instance in a well formed system is unique and automatic.

A class can have generalizations to other classes. This means that the full class descriptor of a class is derived by inheritance from its own segment declaration and those of its ancestors. Generalization between classes implies substitutability (i.e., an instance of a class may be used whenever an instance of a superclass is expected). If the class is specified as a root, it cannot be a subclass of other classes. Similarly, if it is specified as a leaf, no other class can be a subclass of the class.

Each attribute declared in a class has a visibility and a type. Visibility limits availability of the attribute to procedures and actions of any class (public), inside the class and its subclasses (protected), any classes within the containing package (package), or only inside the class (private). The targetScope of the attribute declares whether its value should be an instance (of a child) of that type or if it should be (a child of) the type itself. There are two alternatives for the ownerScope of an attribute:

- it may state that each object created by the class (or by its subclasses) has its own value of the attribute, or
- that the value is owned by the class itself.

An attribute also declares how many attribute values should be connected to each owner (multiplicity). When multiplicity is enforced is a semantic variation point. For example, implementations might allow violations of minimum multiplicity during object initialization. An attribute also declares what the initial values should be, and if these attribute values may be changed:

- none - no constraints exists,
- frozen - values cannot be added or removed after the object has been initialized, or
- addOnly - new values may be added anytime. Values cannot be removed after the object has been initialized.

For each operation, the operation name, the types of the parameters, and the return type(s) are specified, as well as its visibility (see above). An operation may also include a specification of the effects of its invocation. The specification can be done in several different ways (e.g., with pre- and post-conditions, pseudo-code, or just plain text). Each operation declares if it is applicable to the instances, the class, or to the class itself (ownerScope). Furthermore, the operation states whether or not its application will modify the state of the object (isQuery). The operation also states whether or not the operation may be realized by a different method in a subclass (isPolymorphic). A method realizing an operation has the same signature as the operation and a procedure implementing the specification of the operation. Methods in descendents override and replace methods inherited from ancestors (see Section 2.5.4.4, “Inheritance,” on page 2-69). Each method implements an operation declared in the class or inherited from an ancestor. The same operation may be declared more than once in a full class descriptor, but their descriptions must all match,

except that the generalization properties (isRoot, IsAbstract, isLeaf) may vary, and a child operation may strengthen query properties (the child may be a query even though the parent is not). The specification of the method must match the specification of its matching operation, as defined above for operations. Furthermore, if the isQuery attribute of an operation is true, then it must also be true in any realizing method. However, if it is false in the operation, it may still be true in the method if the method does not actually modify the state to carry out the behavior required by the operation (this can only be true if the operation does not inherently modify state). The visibility of a method must match its operation.

Classes may have associations to each other. This implies that objects created by the associated classes are semantically connected (i.e., that links exist between the objects, according to the requirements of the associations). See *Association* on the next page. Associations are inherited by subclasses.

A class may realize a set of interfaces. This means that each operation found in the full descriptor for any realized interface must be present in the full class descriptor with the same specification (see Semantics section Section 2.5.4.4, “Inheritance,” on page 2-69). The relationship between interface and class is not necessarily one-to-one; a class may offer several interfaces and one interface may be offered by more than one class. The same operation may be defined in multiple interfaces that a class supports; if their specifications are identical, then there is no conflict; otherwise, the model is ill formed. Moreover, a class may contain additional operations besides those found in its interfaces.

A class acts as the namespace for various kinds of contained elements defined within its scope, including classes, interfaces, and associations (note that this is purely a scoping construction and does not imply anything about aggregation), the contained classifiers can be used as ordinary classifiers in the container class. If a class inherits another class, the contents of the ancestor are available to its descendents if the visibility of an element is public or protected; however, if the visibility is private, then the element is not visible and therefore not available in the descendant.

2.5.4.4 Inheritance

To understand inheritance, it is first necessary to understand the concept of a full descriptor and a segment descriptor. A full descriptor is the full description needed to describe an object or other instance (see Section 2.5.4.5, “Instantiation,” on page2-70). It contains a description of all of the attributes, associations, and operations that the object contains. In a pre-object-oriented language, the full descriptor of a data structure was declared directly in its entirety. In an object-oriented language, the description of an object is built out of incremental segments that are combined using inheritance to produce a full descriptor for an object. The segments are the modeling elements that are actually declared in a model. They include elements such as class and other generalizable elements. Each generalizable element contains a list of features and other relationships that it adds to what it inherits from its ancestors. The mechanism of inheritance defines how full descriptors are produced from a set of segments connected by generalization. The full descriptors are implicit, but they define the structure of actual instances.

Each kind of generalizable element has a set of inheritable features. For any model element, these include constraints. For classifiers, these include features (attributes, operations, signal receptions, and methods) and participation in associations. The ancestors of a generalizable element are its parents (if any) together with all of their ancestors (with duplicates removed). For a Namespace (such as a Package or a Class with nested declarations), the public or protected contents of the Namespace are available to descendants of the Namespace.

If a generalizable element has no parent, then its full descriptor is the same as its segment descriptor. If a generalizable element has one or more parents, then its full descriptor contains the union of the features from its own segment descriptor and the segment descriptors of all of its ancestors. For a classifier, no attribute, operation, or signal with the same signature may be declared in more than one of the segments (in other words, they may not be redefined). A method may be declared in more than one segment. The way methods override each other is a semantic variation point. The constraints on the full descriptor are the union of the constraints on the segment itself and all of its ancestors. If any of them are inconsistent, then the model is ill formed.

In any full descriptor for a classifier, each method must have a corresponding operation. In a concrete classifier, each operation in its full descriptor must have a corresponding method in the full descriptor.

The purpose of the full descriptor is explained under Section 2.5.4.5, “Instantiation,” on page 2-70.

2.5.4.5 Instantiation

The purpose of a model is to describe the possible states of a system and their behavior. The state of a system comprises objects, values, and links. Each object is described by a full class descriptor. The class corresponding to this descriptor is the direct class of the object. If an object is not completely described by a single class (multiple classification), then any class in the minimal set of unrelated (by generalization) classes whose union completely describes the object is a direct class of the object. Similarly each link has a direct association and each value has a direct data type. Each of these instances is said to be a direct instance of the classifier from which its full descriptor was derived. An instance is an indirect instance of the classifier or any of its ancestors.

The data content of an object comprises one value for each attribute in its full class descriptor (and nothing more). The value must be consistent with the type of the attribute. The data content of a link comprises a tuple containing a list of instances, one that is an indirect instance of each participant classifier in the full association descriptor. The instances and links must obey any constraints on the full descriptors of which they are instances (including both explicit constraints and built-in constraints such as multiplicity).

The state of a system is a valid system instance if every instance in it is a direct instance of some element in the system model and if all of the constraints imposed by the model are satisfied by the instances.

The behavioral parts of UML describe the valid sequences of valid system instances that may occur as a result of both external and internal behavioral effects.

2.5.4.6 Interface

The purpose of an interface is to collect a set of operations that constitute a coherent service offered by classifiers. Interfaces provide a way to partition and characterize groups of operations. An interface is only a collection of operations with a name. It cannot be directly instantiated. Instantiable classifiers, such as class or use case, may use interfaces for specifying different services offered by their instances. Several classifiers may realize the same interface. All of them must contain at least the operations matching those contained in the interface. The specification of an operation contains the signature of the operation (i.e., its name, the types of the parameters, and the return type). An interface does not imply any internal structure of the realizing classifier. For example, it does not define which algorithm to use for realizing an operation. An operation may, however, include a specification of the effects of its invocation. The specification can be done in several different ways (e.g., with pre and post-conditions, pseudo-code, or just plain text).

Each operation declares if it applies to the instances of the classifier declaring it or to the classifier itself (e.g., a constructor on a class (ownerScope)). Furthermore, the operation states whether or not its application will modify the state of the instance (isQuery). The operation also states whether or not all the classes must have the same realization of the operation (isPolymorphic).

An interface can be a child of other interfaces denoted by generalizations. This means that a classifier offering the interface must provide not only the operations declared in the interface but also those declared in the ancestors of the interface. If the interface is specified as a root, it cannot be a child of other interfaces. Similarly, if it is specified as a leaf, no other interface can be a child of the interface.

2.5.4.7 Operation

Operation is a conceptual construct, while Method is the implementation construct. Their common features, such as having a signature, are expressed in the BehavioralFeature metaclass, and the specific semantics of the Operation. The Method constructs are defined in the corresponding subclasses of BehavioralFeature.

2.5.4.8 PresentationElement

The responsibility of presentation element is to provide a textual and graphical projection of a collection of model elements. In this context, projection means that the presentation element represents a human readable notation for the corresponding model elements. The notation for UML can be found in Chapter 3 of this document.

Presentation elements and model elements must be kept in agreement, but the mechanisms for doing this are design issues for model editing tools.

2.5.4.9 Template

A template is a parameterized model element that cannot be used directly in a model. Instead, it may be used to generate other model elements using the Binding relationship; those generated model elements can be used in normal relationships with other elements.

A template represents the parameterization of a model element, such as a class or an operation, although conceptually any model element may be used (but not all may be useful). The template element is attached by composite aggregation to an ordered list of parameter elements. Each parameter element has a name that represents a parameter name within the template element. Any use of the name within the scope of the template element represents an unbound parameter that is to be replaced by an actual value in a Binding of the template. For example, a parameter may represent the type of an attribute of a class (for a class template). The corresponding attribute would have an association to the template parameter as its type.

Note that the scope of the template includes all of the elements recursively owned by it through composite aggregation. For example, a parameterized class template owns its attributes, operations, and so on. Neither the parameterized elements nor its contents may be used directly in a model without binding.

A template element has the templateParameter association to a list of ModelElements that serve as its parameters. To avoid introducing metamodel (M2) elements in an ordinary (M1) model, the model contains a representative of each parameter element, rather than the type of the parameter element. For example, a frequent kind of parameter is a class. Instead of including the metaclass Class in the (M1) ordinary model, a dummy class must be declared whose name is the name of the parameter. This dummy element is meaningful only within the template (it may not be used within the wider model) and it has no features (such as attributes and operations), because the features are part of an actual element that is supplied when the template is bound. Because a template parameter is only a dummy that lacks internal structure, it may violate well-formedness constraints of elements of its kind; the actual elements supplied during binding must satisfy ordinary well-formedness constraints.

Note also that when the template is bound, the bound element does not show the explicit structure of an element of its kind; it is a stub. Its semantics and well-formedness rules must be evaluated as if the actual substitutions of actual elements for parameters had been made; but the expansions are not explicitly shown in a canonical model as they are regarded as derived.

A template element is therefore effectively isolated from the directly-usable part of the model and is indirectly connected to its ultimate instances through Binding associations to bound elements. The bound elements may be used in ordinary models in places where the model element underlying the template could be used.

2.5.4.10 Miscellaneous

A constraint is a Boolean expression over one or several elements that must always be true. A constraint can be specified in several different ways (e.g., using natural language or a constraint language).

A dependency specifies that the semantics of a set of model elements requires the presence of another set of model elements. This implies that if the source is somehow modified, the dependents probably must be modified. The reason for the dependency can be specified in several different ways (e.g., using natural language or an algorithm) but is often implicit.

A Usage or Binding dependency can be established only between elements in the same model, since the semantics of a model cannot be dependent on the semantics of another model. If a connection is to be established between elements in different models, a Trace or Refinement should be used. Refinement can connect elements in different or same models.

Whenever the supplier element of a dependency changes, the client element is potentially invalidated. After such invalidation, a check should be performed followed by possible changes to the derived client element. Such a check should be performed after which action can be taken to change the derived element to validate it again. The semantics of this validation and change is outside the scope of UML.

A data type is a special kind of classifier, similar to a class, but whose instances are primitive values (not objects). For example, the integers and strings are usually treated as primitive values. A primitive value does not have an identity, so two occurrences of the same value cannot be differentiated. Usually, it is used for specification of the type of an attribute. An enumeration type is a user-definable type comprising a finite number of values.

2.6 Extension Mechanisms

2.6.1 Overview

The Extension Mechanisms package is the subpackage that specifies how specific UML model elements are customized and extended with new semantics by using stereotypes, constraints, tag definitions, and tagged values. A coherent set of such extensions, defined for specific purposes, constitutes a UML *profile* (see Section 2.15, “Model Management,” on page 2-181).

The UML provides a rich set of modeling concepts and notations that have been carefully designed to meet the needs of typical software modeling projects. However, users may sometimes require additional features beyond those defined in the UML standard. These needs are met in UML by its built-in extension mechanisms that enable new kinds of modeling elements to be added to the modeler’s repertoire as well as to attach free-form information to modeling elements. The principal extension mechanism is the concept of Stereotype. It provides a way of defining virtual subclasses of UML metaclasses with new metaattributes and additional semantics.

A fundamental constraint on all extensions defined using the profile extension mechanism is that extensions must be strictly additive to the standard UML semantics. This means that such extensions must not conflict with or contradict the standard semantics. In effect, these extension mechanisms are a means for *refining* the standard semantics of UML and do not support arbitrary semantic extension. They allow the modeler to add new modeling elements to UML for use in creating UML models for

process-specific or implementation language-specific domains (for example, supporting code generation for a certain language and infrastructure). It should be noted that stereotypes and tags are used in the definition of UML itself. They are used to define standard model elements that are not considered complex enough to be defined directly as UML metaclasses.

Stereotypes are themselves metaclasses in UML. Consequently, the user of a UML tool can define stereotypes; for example, a new stereotype «persistent» could be defined that can be attached to classes. Many users will not define new stereotypes, but will only apply them during modeling; for example, the stereotype "«persistent»" can be attached to the class "Invoice" by the modeler. A tool could use this as an indicator that a database table definition needs to be generated.

A *profile* is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged definitions, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

A *stereotype* is a model element that defines additional values (based on tag definitions), additional constraints, and optionally a new graphical representation. All model elements that are branded by one or more particular stereotypes receive these values and constraints in addition to the attributes, associations, and superclasses that the element has in the standard UML. Stereotypes augment the classification mechanism based on the built in UML metamodel class hierarchy; therefore, names of new stereotypes must not clash with the names of predefined UML metamodel elements or standard elements.

Tag definitions specify new kinds of properties that may be attached to model elements. The actual properties of individual model elements are specified using *Tagged Values*. These may either be simple datatype values or references to other model elements. Tag definitions can be compared to metaattribute definitions while tagged values correspond to values attached to model elements. They may be used to represent properties such as management information (author, due date, status), code generation information (optimizationLevel, containerClass).

Constraints can also be attached to any model element to refine its semantics. Constraints attached to a stereotype must be observed by all model elements branded by that stereotype. If the rules are specified formally in a profile (for example, by using OCL for the expression of constraints), then a modeling tool may be able to interpret the rules and aid the modeler in enforcing them when applying the profile.

Although it is outside the scope and intent of the UML specification, it is also possible to extend the UML metamodel by explicitly adding new metaclasses and other meta constructs. This capability depends on the use of tools and repositories that support the OMG Meta Object Facility (MOF). Profiles are sometimes referred to as the 'lightweight' built-in extension mechanisms of UML, in contrast with the 'heavyweight' extensibility mechanism as defined by the MOF specification. This is because there are restrictions on how UML profiles can extend the UML metamodel. These restrictions are intended to ensure that any extensions defined by a UML profile are purely additive. Such restrictions do not apply in the MOF context where, in

principle, any metamodel can be defined. (Consequently, every profile definition could also be expressed as an MOF metamodel, but not all MOF metamodels based on UML can be expressed as proper UML profiles.)

From a pragmatic viewpoint, when modeling tools are used to specify lightweight extensions, they should fully support UML extension mechanisms (including a default graphical notation for extended elements) and the XMI that they produce must be compatible with the predefined XMI for UML DTDs. (Note that this is expected to be less readable than a dedicated XMI format based on an MOF metamodel.)

When defining profiles, modelers should be careful to base their extensions on the most semantically similar constructs in the UML metamodel. Failure to observe this can easily result in semantically incorrect or semantically redundant language extensions. When capturing the extended semantics of a domain in the definition of a profile (with the purpose of enabling tool support for the domain), modelers should also be careful not to focus exclusively on defining stereotypes. In most cases a combination of stereotypes and predefined standard model elements will be most effective. Examples of standard or common model elements in a profile definition are standard classes that the user is intended to reuse or subclass, or a set of standard Templates that the user may apply.

Several profile-related standard stereotypes and tags are defined in the Model Management package and chapter, including «profile», «modelLibrary», «appliedProfile», and {applicableSubset}.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Extension Mechanisms package.

2.6.2 Abstract Syntax

The abstract syntax for the Extension Mechanisms package is expressed in graphic notation in Figure 2-10.

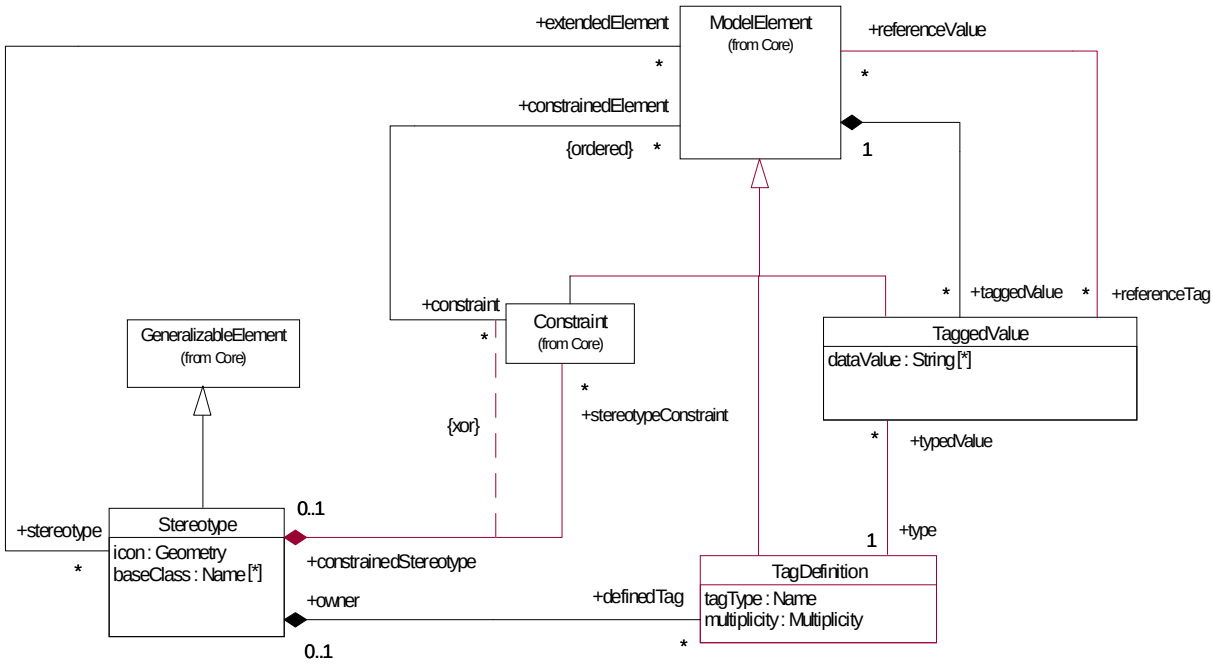


Figure 2-10 Extension Mechanisms

2.6.2.1 *Constraint (as extended)*

The constraint concept allows new semantics to be specified linguistically for a model element. The specification is written as an expression in a designated constraint language. The language can be specially designed for writing constraints (such as OCL), a programming language, mathematical notation, or natural language. If constraints are to be enforced by a model editor tool, then the tool must understand the syntax and semantics of the constraint language. Because the choice of language is arbitrary, constraints are an extension mechanism.

In the metamodel, a constraint directly attached to a model element describes semantic restrictions that this model element must obey. Constraints attached to a Stereotype apply to each model element that bears that stereotype. Note that, for the case of constraints attached to stereotype definitions, the scope of the constraint is the UML metamodel and not the model in which it is defined. This allows the definition of well-formedness rules for stereotypes in the same manner as the well-formedness rules of other metamodel elements.

Attributes

body	A boolean expression defining the constraint. Expressions are written as strings in a designated language. For the model to be well formed, the expression must always yield a true value when evaluated for instances of the constrained elements at any time when the system is stable; that is, not during the execution of an atomic operation. When a constraint is attached to a stereotype, the lexical scope of that constraint is the UML metamodel rather than the M1 model in which the constraint is defined. This means that there is no need to explicitly import the UML metamodel.

Associations

constrainedElement	An ordered list of elements subject to the constraint.
constrainedStereotype	A stereotype to which the constraint applies. This constraint will automatically apply to all model elements branded by that stereotype.

Any one Constraint must have one or more constrainedElement links, or one constrainedStereotype link, but not both.

2.6.2.2 ModelElement (as extended)

Any model element may have arbitrary tagged values and constraints (subject to these making sense). A model element may also have one or more stereotypes. In the latter case, the base class of the stereotype must match the metaclass of that model element (such as Class, Association, Dependency) or one of its subclasses. The presence of a stereotype may impose implicit constraints on the modeling element and may require the presence of specific tagged values.

Associations

constraint	A constraint that must be satisfied by the model element. A model element may have a set of constraints. The constraint is to be evaluated when the system is stable; that is, not in the middle of an atomic operation.
stereotype	Designates the stereotypes that further qualify the UML metaclass (the base class or one of its subclasses) of the modeling element. The stereotype does not conflict with or contradict the standard semantics of the metaclass to which it applies, but may specify additional constraints and tag definitions. All constraints and tag definitions on a stereotype apply to the model elements that are branded by the stereotype. The stereotype acts as a virtual metaclass describing the model element.
taggedValue	An arbitrary property attached to the model element based on an associated tag definition. The interpretation of the tagged value is outside the scope of the UML metamodel.

2.6.2.3 Stereotype

The stereotype concept provides a way of branding (classifying) model elements so that they behave in some respects as if they were instances of new virtual metamodel constructs. These model elements have the same structure (attributes, associations, operations) as similar non-stereotyped model elements of the same kind. The stereotype may specify additional constraints and tag definitions that apply to model elements. In addition, a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure.

In the metamodel, the Stereotype metaclass is a subclass of GeneralizableElement. Tag definitions and constraints attached to a stereotype apply to all model elements branded by that stereotype. A stereotype may also specify a geometrical icon to be used for presenting elements with the stereotype.

If a stereotype is a subclass of another stereotype, then it inherits all of the constraints and tagged values from its stereotype supertype and it must apply to the same kind of base class. A stereotype keeps track of the base class to which it may be applied. Stereotypes are typically grouped in a Profile package.

Attributes

baseClass	Specifies the names of one or more UML modeling elements to which the stereotype applies, such as Class, Association, Refinement, Constraint. This is the name of a metaclass; that is, a class from the UML metamodel itself rather than a user model class.
icon	The geometrical description for an icon to be used to present an image of a model element branded by the stereotype.

Associations

extendedElement	Designates the model elements affected by the stereotype. Each one must be a model element of the kind specified by the baseClass attribute.
definedTag	Specifies a set of tag definitions, each of which specifies tagged values that a model element branded by the stereotype can have.
stereotypeConstraint	Designates constraints that apply to all model elements branded by this stereotype. These constraints are defined in the scope of the full UML metamodel.

2.6.2.4 TagDefinition

A tag definition specifies the tagged values that can be attached to a kind of model element. Among other things, tag definitions can be used to define the virtual meta attributes of the stereotype to which they are attached. Some of these meta attributes may be references to other metamodel elements and, in effect, can be used to specify new one-way meta references. *However, this latter feature should be used with discretion since it can easily be misused to define new semantics that are more than just refinement of the original UML metamodel.*

Tag definitions should be defined in conjunction with a stereotype since that allows them to be used in a more disciplined manner (stereotypes are constrained by the semantics of their base class). However, primarily for reasons of compatibility with models defined on the basis of UML 1.3, it is still possible to have tag definitions that are not associated with any stereotype.

Attributes

multiplicity	Specifies the number of data values that tagged values based on this tag must have, or, the number of model elements that can be associated to the related tagged values.
tagType	In the general case, where the tag type is a data type, this specifies the range of values of the tagged values associated with the tag definition. In the special case, where the tag type refers to a metaclass that is not a datatype, the tag value references model elements that are instances of the metaclass.

Associations

typedValue	The tagged values that conform to this tag definition.
owner	The stereotype to which this tag definition belongs.

2.6.2.5 *TaggedValue*

A tagged value allows information to be attached to any model element in conformance with its tag definition. Although a tagged value, being an instance of a kind of ModelElement, automatically inherits the *name* attribute, the name that is actually used in the tagged value is the name of the associated tag definition. The interpretation of tagged values is intentionally beyond the scope of UML semantics. It must be determined by user or tool conventions that may be specified in a profile in which the tagged value is defined. It is expected that various model analysis tools will define tag definitions to supply information needed for their operations beyond the basis semantics of UML. Such information could include code generation options, model management information, or user-specified semantics.

Any tagged value must have one or more reference value links or one or more data values, but not both.

Attributes

dataValue	Specifies the set of values that are part of the tagged value. The type of this value must conform to the type specified in the *tagType* attribute of the associated tag definition. The number of values that can be specified is defined by the *multiplicity* attribute of the associated tag definition.

Associations

type	Specifies the tag definition that defines the name, meaning, and type of the tagged value.
referenceValue	Specifies the model elements that this tagged value references. These elements are model-level instances of the metaclass or stereotype specified by the *tagType* attribute of the corresponding tag definition. The number of references is defined by the *multiplicity* attribute of the associated tag definition.

2.6.3 Well-formedness Rules

The following well-formedness rules apply to the Extension Mechanisms package.

2.6.3.1 Constraint

[1] A Constraint attached to a stereotype must not conflict with constraints on any inherited stereotype, or associated with the base class.

-- cannot be specified with OCL, level M2 not accessible

[2] A constraint attached to a stereotyped model element (either directly or through another stereotype) must not conflict with any constraints on the associated stereotype, nor with the class (the base class) of the model element.

-- cannot be specified with OCL, level M2 not accessible

[3] A constraint attached to a stereotype will apply to all model elements branded by that stereotype and must not conflict with any constraints on the attached branding stereotype, nor with the class (the base class) of the model element.

-- cannot be specified with OCL, level M2 not accessible

2.6.3.2 ModelElement

[1] Tags associated with a model element (directly via a property list or indirectly via a stereotype) must not clash with any meta attributes associated with the model element.

-- cannot be specified with OCL, level M2 not accessible

[2] A model element must have at most one tagged value with a given tag name.

```
self.taggedValue->forAll(t1, t2 : TaggedValue |
    t1.type.name = t2.type.name implies t1 = t2)
```

[3] A stereotype cannot extend itself.

```
self.stereotype->excludes(self)
```

2.6.3.3 Stereotype

[1] Stereotype names must not clash with any base class names.

```
Stereotype.allInstances->forAll(st | st.baseClass <> self.name)
```

[2] The base class name must be provided

```
Set {self.baseClass}->notEmpty
```

[3] Tag names attached to a stereotype must not clash with M2 meta-attribute namespace of the appropriate base class element, nor with tag definition names of any inherited stereotype

```
--  cannot be specified with OCL, level M2 not accessible
```

[4] The base class of a stereotype must be the same or a subclass of the base class of parent stereotypes.

```
-- cannot be specified with OCL, level M2 not accessible
```

[5] All stereotype definitions must be contained either directly or transitively in a profile package.

```
findProfile(self)->notEmpty
```

Additional Operations

[1] The find profile operation returns either the single-element set containing profile package in which the model element is defined or an empty set if the element is not contained in any profile.

```
findProfile (me : ModelElement) : Set (Package)
    if (me.namespace->notEmpty) then
        if (me.namespace.oclIsKindOf(Package) and
            me.namespace.stereotype->notEmpty) and
            me.namespace.stereotype->exists(s|s.name = profile) then
            result = me.namespace
        else -- go up to the next level of namespace
            result = findProfile (me.namespace)
    else
        result = me.namespace -- return empty set
```

2.6.3.4 TagDefinition

[1] The type associated with a tag definition is either the name of a UML metaclass, including elements of the DataType package, or an instance of the DataType metaclass or one of its descendants.

```
-- cannot be specified with OCL, level M2 not accessible
```

[2] All tag definitions must be contained either directly or transitively in a profile package.

```
findProfile(self)->notEmpty
```

2.6.3.5 TaggedValue

[1] The data value of a tagged value is exclusive to the “referenceValue” association.

```
if (self.referenceValue->size > 0)
    then (self.dataValue->size = 0)
    else (self.dataValue->size > 0)
endif
```

[2] The data value of a tagged value must conform to the data type specified by the “tagType” attribute of the tag definition.

-- cannot be specified with OCL (requires an OCL function that converts a string name into a corresponding metatype)

[3] The model elements associated with a tagged value by the “referenceValue” association must be instances of the metaclass specified by the “tagType” attribute of the tag definition.

-- cannot be specified with OCL (requires an OCL function that converts a string name into a corresponding metatype)

2.6.4 Detailed Semantics

The various extension mechanisms defined in this chapter represent extensions to the modeling language UML that affect the structure and semantics of models produced by the user.

Within a model, any user-level model element may have a set of links to stereotypes, and a set of tagged values conformant to existing tag definitions. The constraints defined for the stereotype specify restrictions on the instantiation of the model. An instance of a user-level model element must satisfy all of the constraints on its model element for the model to be well formed. Evaluation of constraints is to be performed when the relevant portion of the system is “stable,” that is, after the completion of any internal operations when it is waiting for external events. In general, constraints are written in any language that can adequately specify the desired constraints, such as OCL, C++, or natural language. The interpretation of the constraints must be specified by the constraint language.

A stereotype refers to a base class, which is a class in the UML metamodel (not a user-level modeling element) such as Class, Association, Refinement, etc. A stereotype may be a subclass of one or more existing stereotypes. In that case, it inherits their constraints and tag definitions and may add additional ones of its own. In principle, a stereotype inherits the base class value of its parent, if one exists (this is expressed as a constraint on these values). The modeler may refine this to any subclass of that base class. For instance, if a stereotype *s* with a base class *b* is defined, then a stereotype *t* that has *s* as its superclass has either *b* or any *subclass of b* as its base class value. If a stereotype has multiple superclasses, then all of these superclasses must be derived from a single common superclass. In that case, the base class of the subclass is equivalent to the most specific parent stereotype, or a subclass of that. For instance, if a stereotype *s* has supertypes *t* and *u* with base classes “Classifier” and “Class” respectively, then the base class of *s* is “Class” or any subclass of “Class” in UML.

If a model element is branded by an attached stereotype, then the UML base class of the model element must be the base class specified by the stereotype or one of the subclasses of that base class. Any constraints on the stereotype are implicitly attached to the model element. Any tag definitions belonging to the stereotype will serve as specifications for tagged values associated to the model element. If the stereotype is a subclass of one or more stereotypes, then any constraints or tag definitions from those stereotypes also apply to the model element (because they are inherited by this stereotype). If there are any conflicts among the multiple constraints and tag definitions (inherited or directly specified), then the model is ill formed, as is the case with general specialization hierarchies.

2.6.5 Notes

Backward compatibility of profiles with UML 1.3 has been addressed by maintaining the basic UML 1.3 extension features while adding new features that can be optionally exploited. There are two areas where backward compatibility has been carefully considered. First, although it is generally recommended that tags should be defined in the context of a stereotype, they may still be defined independently as was the case with UML 1.3. Second, although it is generally recommended that tag definitions should be typed, they may still be defined with type declared *String*; that is, they are effectively not typed.

UML 1.4 compliant tools are expected to make use of the ability to type tags, and to provide conversion utilities for models based on earlier versions of UML. It is important to note, however, that older models that contain tags declared to be of type *String* should still work correctly, since *String* continues to be a standard UML datatype.

The following are some typical examples of stereotypes and tag definitions:

A stereotype of Class with an associated tag definition

Stereotype	Base Class	Parent	Tags	Constraints	Description
persistent	Class	N/A	storageMode	none	Classes of this stereotype are persistent and may be stored in a variety of different modes.

Tag	Stereotype	Type	Multiplicity	Description
storageMode	persistent	StorageProfile::StorageEnum (an enumeration: {table, file, object})	*	identifies the storage mode

A stereotype of Class with an associated tag definition

Stereotype	Base Class	Parent	Tags	Constraints	Description
persistent	Class	N/A	isPersistent	none	Classes of this stereotype may be persistent, depending on the value of the “isPersistent” tag.

Tag	Stereotype	Type	Multiplicity	Description
isPersistent	persistent	UML::Datatypes::Boolean	1	indicates whether the class is persistent or not

A stereotype of Class with an associated tag definition

Stereotype	Base Class	Parent	Tags	Constraints	Description
persistent	Class	N/A	primaryKeyClass	none	Classes of this stereotype have a reference to indicate the primary key specification.

Tag	Stereotype	Type	Multiplicity	Description
primaryKeyClass	persistent	*reference to* UML::Foundation::Class	1	Identifies the M1 class that serves as the primary key.

A stereotype of Class with an associated tag definition

Stereotype	Base Class	Stereotype Parent	Tags	Constraints	Description
workflow	ActionState	N/A	resources	none	action states of this stereotype represent workflow actions.

A tag defined independently of a stereotype

Tag	Stereotype	Type	Multiplicity	Description
debugMode	N/A	DebugProfile::DebugDomain (an enumeration with three possible choices: {on, off, trace})	1	Used to set the desired debug mode for a model post-processor.

A tag defined independently of a stereotype

Tag	Stereotype	Type	Multiplicity	Description
aliasNames	N/A	UML::Datatypes::String	*	Reuses the standard String datatype at the M1 level.

2.7 Data Types

2.7.1 Overview

The Data Types package is the subpackage that specifies the different data types that are used to define UML. This chapter has a simpler structure than the other packages, since it is assumed that the semantics of these basic concepts are well known.

2.7.2 Abstract Syntax

The abstract syntax for the Data Types package is expressed in graphic notation in Figure 2-11 and Figure 2-12.

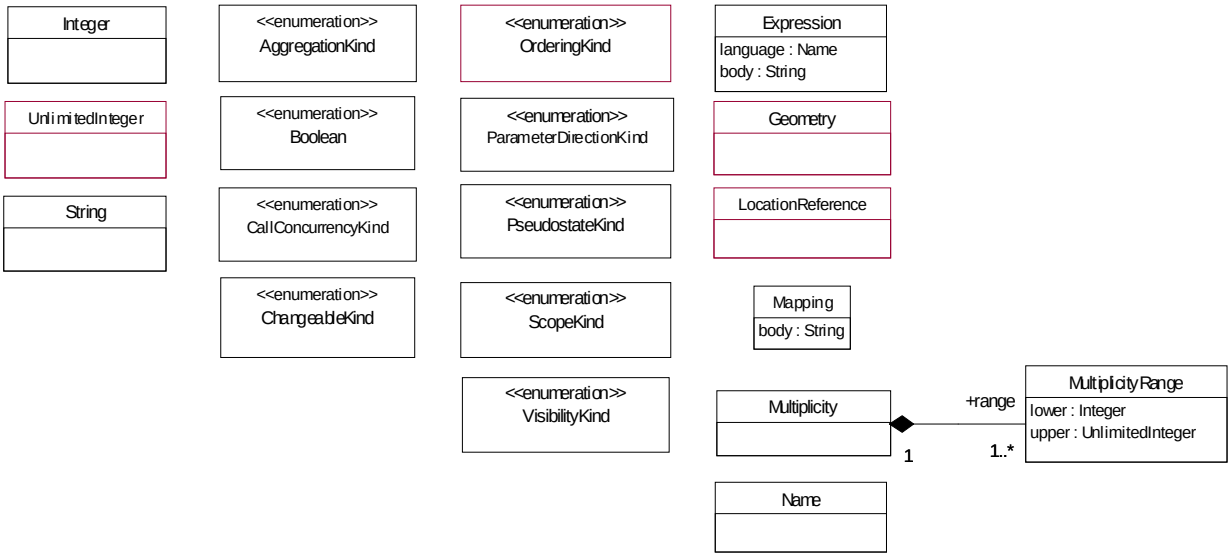


Figure 2-11 Data Types Package - Main

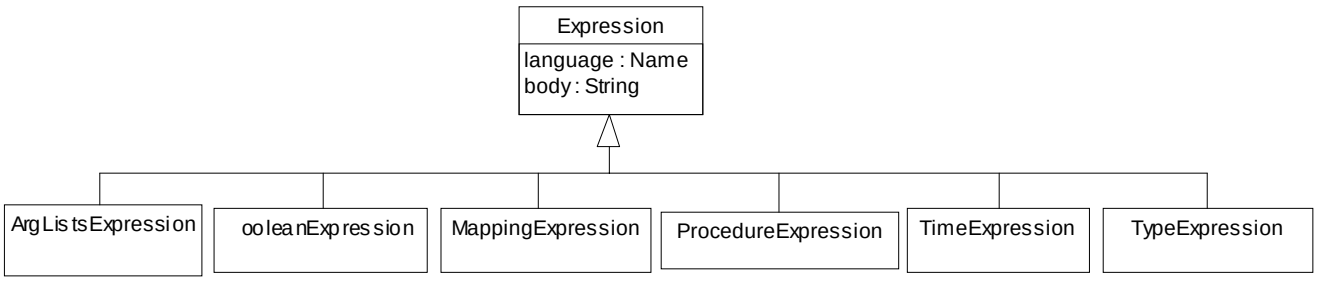


Figure 2-12 Data Types Package - Expressions

In the metamodel, the data types are used for declaring the types of the class attributes. They appear as strings in the diagrams and not with a separate ‘data type’ icon. In this way, the sizes of the diagrams are reduced. However, each occurrence of a particular name of a data type denotes the same data type.

Note that these data types are the data types used for defining UML and not the data types to be used by a user of UML. The latter data types will be instances of the DataType metaclass defined in the metamodel.

2.7.3.1 AggregationKind

An enumeration that denotes what kind of aggregation an Association is. When placed on a target end, specifies the relationship of the target end to the source end. AggregationKind defines an enumeration whose values are:

none	The end is not an aggregate.
aggregate	The end is an aggregate; therefore, the other end is a part and must have the aggregation value of none. The part may be contained in other aggregates.
composite	The end is a composite; therefore, the other end is a part and must have the aggregation value of none. The part is strongly owned by the composite and may not be part of any other composite.

2.7.3.2 ArgListsExpression

In the metamodel, ArgListsExpression defines a statement which will result in a set of object lists when it is evaluated.

2.7.3.3 Boolean

In the metamodel, Boolean defines an enumeration that denotes a logicial condition. Its enumeration literals are:

true	The Boolean condition is satisfied.
false	The Boolean condition is not satisfied.

2.7.3.4 BooleanExpression

In the metamodel, BooleanExpression defines a statement that will evaluate to an instance of Boolean when it is evaluated.

2.7.3.5 CallConcurrencyKind

An enumeration that denotes the semantics of multiple concurrent calls to the same passive instance (i.e., an Instance originating from a Classifier with isActive=false). It is an enumeration with the values.

sequential	Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed.
guarded	Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence. The others are blocked until the performance of the first Operation is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks. Guarded Operations must perform correctly (or block themselves) in the case of a simultaneous sequential Operation or guarded semantics cannot be claimed.
concurrent	Multiple calls from concurrent threads may occur simultaneously to one Instance (on any concurrent Operation). All of them may proceed concurrently with correct semantics. Concurrent Operations must perform correctly in the case of a simultaneous sequential or guarded Operation or concurrent semantics cannot be claimed.

2.7.3.6 ChangeableKind

In the metamodel, ChangeableKind defines an enumeration that denotes how an AttributeLink or LinkEnd may be modified. Its values are:

changeable	No restrictions on modification.
frozen	The value may not be changed from the source end after the creation and initialization of the source object. Operations on the other end may change a value.
addOnly	If the multiplicity is not fixed, values may be added at any time from the source object, but once created a value may not be removed from the source end. Operations on the other end may change a value.

2.7.3.7 Expression

In the metamodel, an Expression defines a statement which will evaluate to a (possibly empty) set of instances when executed in a context. An Expression does not modify the environment in which it is evaluated. An expression contains an expression string and the name of an interpretation language with which to evaluate the string.

Attributes

language	Names the language in which the expression body is represented. The interpretation of the expression depends on the language. If the language name is omitted, no interpretation for the expression can be assumed by UML.
body	The text of the expression expressed in the given language.

Predefined language names include the following:

OCL	The Object Constraint Language (see Chapter 6, “Object Constraint Language Specification”).
	(The empty string) This represents a natural-language statement. As such, it is obviously intended for human information rather than formal specification.

In general, a language name should be spelled and capitalized exactly as it appears in the document defining the language. For example, use COBOL, not Cobol; use Ada, not ADA; use PostScript, not Postscript. In other words, spell it correctly.

2.7.3.8 Geometry

An uninterpreted type used to describe the geometrical shape of icons, such as those that may be attached to stereotypes. The details of this specification are not currently part of UML and must therefore be supplied by the implementation of a model editing tool, with the understanding that they will likely be tool-specific. This type is therefore not actually defined in the metamodel but is used only as the type of attributes.

2.7.3.9 Integer

In the metamodel, Integer is a classifier element that is an instance of Primitive, representing the predefined type of integers. An instance of Integer an element in the (infinite) set of integers (…-2, -1, 0, 1, 2…).

2.7.3.10 LocationReference

Designates a position within a behavior sequences for the insertion of an extension use case. May be a line or range of lines in code, or a state or set of states in a state machine, or some other means in a different kind of specification.

2.7.3.11 Mapping

In the metamodel, a Mapping is an expression that is used for mapping ModelElements. For exchange purposes, it should be represented as a String.

Attributes

body	A string describing the mapping. The format of the mapping is currently unspecified in UML.

2.7.3.12 MappingExpression

An expression that evaluates to a mapping.

2.7.3.13 Multiplicity

In the metamodel, a Multiplicity defines a non-empty set of non-negative integers. A set which only contains zero ({0}) is not considered a valid Multiplicity. Every Multiplicity has at least one corresponding String representation.

Additional operations

[1] The allows operation takes an integer as input. It checks if a given integer cardinality is allowed by a multiplicity.

```
allows(i : Integer) : Boolean;
allows(i) = self.range->exists(r : MultiplicityRange |r.contains(i))
```

[2] The operation compatibleWith takes another multiplicity as input. It checks if one multiplicity is compatible with another.

```
compatibleWith(other : Multiplicity) : Boolean;
compatibleWith(other) = Integer.allInstances()->
    forAll(i : Integer | self.allows(i) implies other.allows(i))
```

[3] The operation lowerbound returns the lowest lower bound of the ranges in a multiplicity.

```
lowerbound( ) : Integer;
lowerbound = self.range->exists(r : MultiplicityRange |r.lower = result)
        and self.range->forall(r : MultiplicityRange |r.lower <= result)
```

[4] The operation upperbound returns the highest upper bound of the ranges in a multiplicity.

```
upperbound( ) : UnlimitedInteger;
upperbound = self.range->exists(r : MultiplicityRange |r.upper = result)
            and self.range->forall(r : MultiplicityRange |r.upper <= result)
```

[5] The is operation determines if the upper and lower bound of the ranges are the ones given.

```
is(lowerbound : integer, upperbound : unlimitedInteger) : Boolean
is(lowerbound, upperbound) = (lowerbound = self.lowerbound and
                                upperbound = self.upperbound)
```

2.7.3.14 MultiplicityRange

In the metamodel, a MultiplicityRange defines a range of integers. The upper bound of the range cannot be below the lower bound. The lower bound must be a nonnegative integer. The upper bound must be a nonnegative integer or the special value *unlimited*, which indicates there is no upper bound on the range.

Additional operations

[1] The operation contains takes an integer as input and checks if a given integer is within the range specified by a multiplicity range.

```
contains(i : Integer) : Boolean;
contains(i) = (self.lower<=i and i<=self.upper)
```

2.7.3.15 Name

In the metamodel, a Name defines a token which is used for naming ModelElements. A name is represented as a String.

2.7.3.16 OrderingKind

Defines an enumeration that specifies how the elements of a set are arranged. Used in conjunction with elements that have a multiplicity in cases when the multiplicity value is greater than one. The ordering must be determined and maintained by operations that modify the set. The intent is that the set of enumeration literals be open for new values to be added by tools for purposes of design, code generation, etc. For example, a value of sorted might be used for a design specification. Values are:

unordered	The elements of the set have no inherent ordering.
ordered	The elements of the set have a sequential ordering.
	Other possibilities (such as sorted) may be defined later by declaring additional keywords. As with user-defined stereotypes, this would be a private extension supported by particular editing tools.

2.7.3.17 ParameterDirectionKind

In the metamodel, ParameterDirectionKind defines an enumeration that denotes if a Parameter is used for supplying an argument and/or for returning a value. The enumeration values are:

in	An input Parameter (may not be modified).
out	An output Parameter (may be modified to communicate information to the caller).
inout	An input Parameter that may be modified.
return	A return value of a call.

2.7.3.18 ProcedureExpression

In the metamodel, ProcedureExpression defines a statement that will result in a change to the values of its environment when it is evaluated.

2.7.3.19 PseudostateKind

In the metamodel, PseudostateKind defines an enumeration that discriminates the kind of Pseudostate. See Section 2.7.3.19, “PseudostateKind,” on page 2-90 for details. The enumeration values are listed below.

choice	Splits an incoming transition into several disjoint outgoing transitions. Each outgoing transition has a guard condition that is evaluated after prior actions on the incoming path have been completed. At least one outgoing transition must be enabled or the model is ill formed.
deepHistory	When reached as the target of a transition, restores the full state configuration that was active just before the enclosing composite state was last exited.
fork	Splits an incoming transition into several concurrent outgoing transitions. All of the transitions fire together.
initial	The default target of a transition to the enclosing composite state.
join	Merges transitions from concurrent regions into a single outgoing transition. All the transitions fire together.
junction	Chains together transitions into a single run-to-completion path. May have multiple input and/or output transitions. Each complete path involving a junction is logically independent and only one such path fires at one time. May be used to construct branches and merges.
shallowHistory	When reached as the target of a transition, restores the state within the enclosing composite state that was active just before the enclosing state was last exited. Does not restore any substates of the last active state.

2.7.3.20 ScopeKind

In the metamodel, ScopeKind defines an enumeration that denotes whether a feature belongs to individual instances or an entire classifier. Its values are:

instance	The feature pertains to Instances of a Classifier. For example, it is a distinct Attribute in each Instance or an Operation that works on an Instance.
classifier	The feature pertains to an entire Classifier. For example, it is an Attribute shared by the entire Classifier or an Operation that works on the Classifier, such as a creation operation.

2.7.3.21 String

In the metamodel, String is a classifier element that is an instance of Primitive. An instance of String defines a piece of text.

2.7.3.22 TimeExpression

In the metamodel, TimeExpression defines a statement that will define the time of occurrence of an event. The specific format of time expressions is not specified here and is subject to implementation considerations.

2.7.3.23 TypeExpression

In the metamodel, TypeExpression is the encoding of a programming language type in the interpretation language. It is used within a ProgrammingLanguageDataType.

2.7.3.24 UnlimitedInteger

In the metamodel, UnlimitedInteger is a classifier element that is an instance of Primitive. It defines a data type whose range is the nonnegative integers augmented by the special value “unlimited.” It is used for the upper bound of multiplicities.

Additional operations

[1] The operation <= determines whether an unlimited integer is less than or equal to another.

```
<= (ui2 : unlimitedInteger) : Boolean;
<= (ui2) = (ui2 = #unlimited or
              (self <> #unlimited
                  and self.oclAsType(Integer)
                          <= ui2.oclAsType(Integer)))
```

2.7.3.25 VisibilityKind

In the metamodel, VisibilityKind defines an enumeration that denotes how the element to which it refers is seen outside the enclosing name space. Its values are:

public	Other elements may see and use the target element.
protected	Descendants of the source element may see and use the target element.
private	Only the source element may see and use the target element.
package	Elements declared in the same package as the target element may see and use the target element.

Part 3 - Behavioral Elements

This Behavioral Elements package is the language superstructure that specifies the dynamic behavior or models. The Behavioral Elements package is decomposed into the following subpackages: Common Behavior, Collaborations, Use Cases, State Machines, Activity Graphs, and Actions.

2.8 Behavioral Elements Package

Common Behavior specifies the core concepts required for behavioral elements. The Collaborations package specifies a behavioral context for using model elements to accomplish a particular task. The Use Case package specifies behavior using actors and use cases. The State Machines package defines behavior using finite-state transition

systems. The Activity Graphs package defines a special case of a state machine that is used to model processes. The Actions package defines behavior using a detailed model of computation.

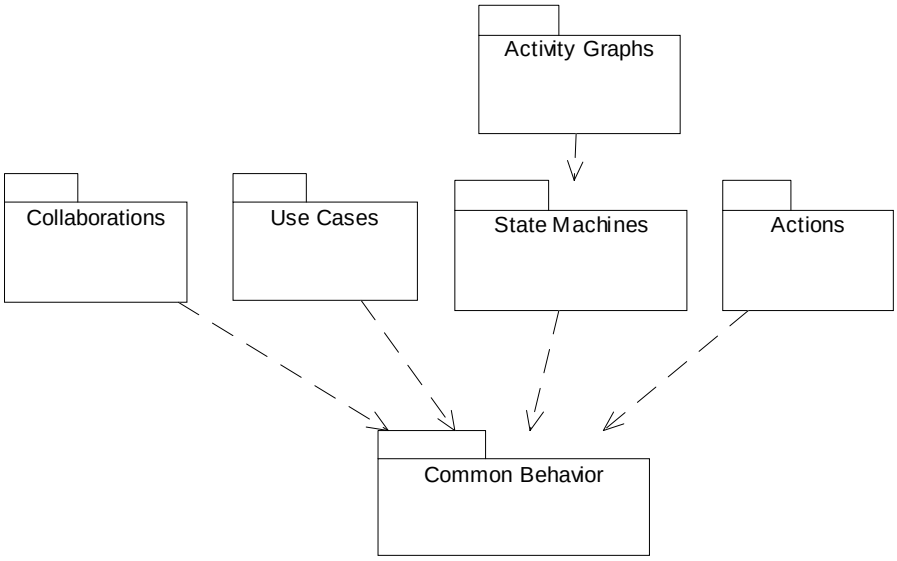


Figure 2-13 Behavioral Elements Package

2.9 Common Behavior

2.9.1 Overview

The Common Behavior package is the most fundamental of the subpackages that compose the Behavioral Elements package. It specifies the core concepts required for dynamic elements and provides the infrastructure to support Collaborations, State Machines, Use Cases, and Actions.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Common Behavior package.

2.9.2 Abstract Syntax

The abstract syntax for the Common Behavior package is expressed in graphic notation in the following figures. Figure 2-14 shows the model elements that define Signals and Receptions.

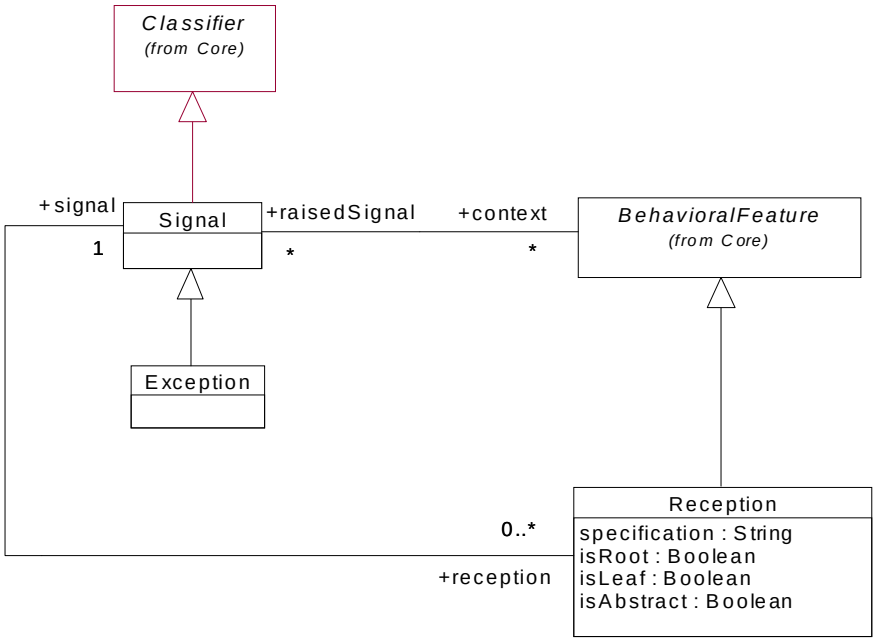


Figure 2-14 Common Behavior - Signals

Figure 2-15 on page 2-95 illustrates the Procedure model element and its support for Expressions and Methods.

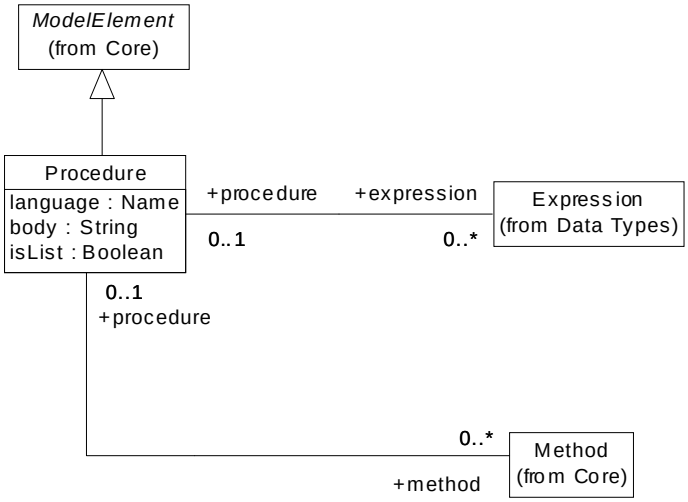


Figure 2-15 Common Behavior - Procedure

Figure 2-16 on page 2-96 shows the model elements that define Instances and Links.

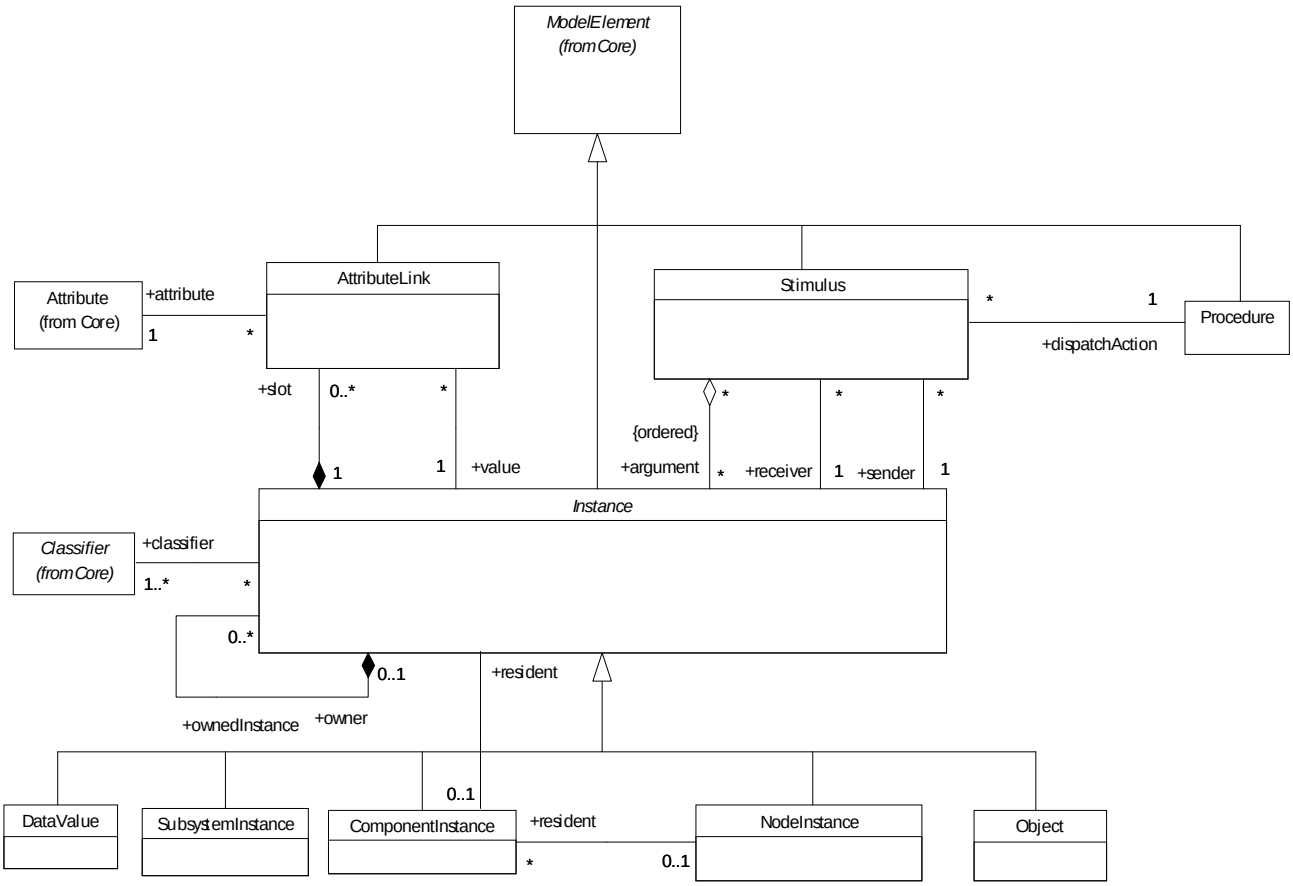


Figure 2-16 Common Behavior - Instances

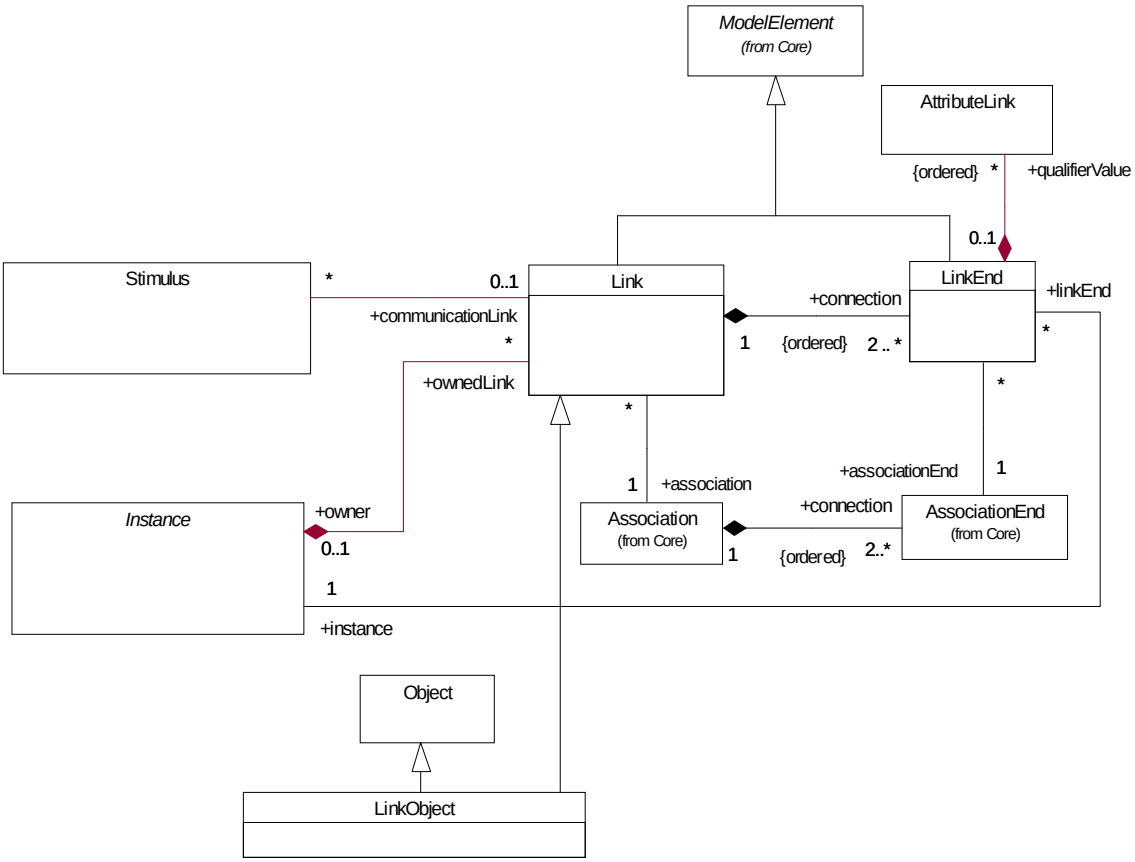


Figure 2-17 Common Behavior - Links

The following metaclasses are contained in the Common Behavior package.

2.9.2.1 *AttributeLink*

An attribute link is a named slot in an instance, which holds the value of an attribute.

In the metamodel, AttributeLink is a piece of the state of an Instance and holds the value of an Attribute.

Associations

value	The Instance, which is the value of the AttributeLink.
attribute	The Attribute from which the AttributeLink originates.

2.9.2.2 ComponentInstance

A component instance is an instance of a component that resides on a node instance. A component instance may have a state.

In the metamodel, a ComponentInstance is an Instance that originates from a Component. It may be associated with a set of Instance, and may reside on a NodeInstance.

Associations

resident	A collection of Instances that exist inside the ComponentInstance.

2.9.2.3 DataValue

A data value is an instance with no identity.

In the metamodel, DataValue is a child of Instance that cannot change its state (i.e., all Operations that are applicable to it are pure functions or queries). DataValues are typically used as attribute values.

2.9.2.4 Exception

An exception is a signal raised by behavioral features typically in case of execution faults.

In the metamodel, Exception is derived from Signal. An Exception is associated with the BehavioralFeatures that raise it.

Associations

context	(Inherited from Signal) The set of BehavioralFeatures that raise the exception.

2.9.2.5 Instance

The instance construct defines an entity to which a set of operations can be applied and which has a state that stores the effects of the operations.

In the metamodel, Instance is connected to at least one Classifier that declares its structure and behavior. It has a set of attribute values and is connected to a set of Links, both sets matching the definitions of its Classifiers. The two sets implement the current state of the Instance. An Instance may also own other Instances or Links.

Instance is an abstract metaclass.

Associations

slot	The set of AttributeLinks that holds the attribute values of the Instance.
linkEnd	The set of LinkEnds of the connected Links that are attached to the Instance.
classifier	The set of Classifiers that declare the structure of the Instance.
ownedInstance	The set of Instances that are owned by the Instance.
ownedLink	The set of Links that are owned by the Instance.
owner	Specifies the Instance that owns the Instance.

Standard Constraints

destroyed *Association*	Destroyed is a constraint applied to an instance, specifying that the instance is destroyed during the execution.
new *Association*	New is a constraint applied to an instance, specifying that the instance is created during the execution.
transient *Association*	Transient is a constraint applied to an instance, specifying that the instance is created and destroyed during the execution.

Tagged Values

persistent Association	Persistence denotes the permanence of the state of the instance, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

2.9.2.6 Link

The link construct is a connection between instances.

In the metamodel, Link is an instance of an Association. It has a set of LinkEnds that matches the set of AssociationEnds of the Association. A Link defines a connection between Instances.

Associations

association	The Association that is the declaration of the link.
connection	The tuple of LinkEnds that constitute the Link.
owner	Specifies the Instance that owns the Link.

Standard Constraints

destroyed *Association*	Destroyed is a constraint applied to a link, specifying that the link is destroyed during the execution.
new *Association*	New is a constraint applied to a link, specifying that the link is created during the execution.
transient *Association*	Transient is a constraint applied to a link, specifying that the link is created and destroyed during the execution.

2.9.2.7 LinkEnd

A link end is an end point of a link.

In the metamodel, LinkEnd is the part of a Link that connects to an Instance. It corresponds to an AssociationEnd of the Link's Association.

Associations

associationEnd	The AssociationEnd that is the declaration of the LinkEnd..
instance	The Instance connected to the LinkEnd.
qualifierValue	The AttributeLinks that hold the values of the Qualifier associated with the corresponding AssociationEnd.

Standard Constraints

association *Association*	Association is a constraint applied to a link-end, specifying that the corresponding instance is visible via association.
global *Association*	Global is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is in a global scope relative to the link.
local *Association*	Local is a constraint applied to link-end, specifying that the corresponding instance is visible because it is in a local scope relative to the link.
parameter *Association*	Parameter is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is a parameter relative to the link.
self *Association*	Self is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is the dispatcher of a request.

2.9.2.8 LinkObject

A link object is a link with its own set of attribute values and to which a set of operations may be applied.

In the metamodel, LinkObject is a connection between a set of Instances, where the connection itself may have a set of attribute values and to which a set of Operations may be applied. It is a child of both Object and Link.

2.9.2.9 NodeInstance

A node instance is an instance of a node. A collection of component instances may reside on the node instance.

In the metamodel, NodeInstance is an Instance that originates from a Node. Each ComponentInstance that resides on a NodeInstance must be an instance of a Component that resides on the corresponding Node.

Associations

resident	A collection of ComponentInstances that reside on the NodeInstances.

2.9.2.10 Object

An object is an instance that originates from a class.

In the metamodel, Object is a subclass of Instance and it originates from at least one Class. The set of Classes may be modified dynamically, which means that the set of features of the Object may change during its life-time.

2.9.2.11 Procedure

A procedure is a coordinated set of actions that models a computation, such as an algorithm. It can also be used without actions to express a procedure in a textual language.

In the metamodel, Procedure is a subclass of ModelElement. It can be linked to a Method or Expression to model how the method is carried out or the expression is evaluated.

Attributes

language	Names the language in which the body attribute is written.
body	The text of the procedure written in the given language.
isList	Determines whether the arguments to the procedure are passed as attributes of a single object, or are passed separately. See descriptions in Actions.

Associations

expression	An expression the value of which is calculated by the procedure. Used to provide a detailed action model for an expression.
method	A method which is performed by the procedure. Used to provide a detailed action model for a method.

2.9.2.12 Reception

A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal. The reception designates a signal and specifies the expected behavioral response. A reception is a summary of expected behavior. The details of handling a signal are specified by a state machine.

In the metamodel, Reception is a child of BehavioralFeature and declares that the Classifier containing the feature reacts to the signal designated by the reception feature. The isPolymorphic attribute specifies whether the behavior is polymorphic or not; a true value indicates that the behavior is not always the same and may be affected by state or subclassing. The specification indicates the expected response to the Signal.

Attributes

isAbstract	If true, then the reception does not have an implementation, and one must be supplied by a descendant. If false, the reception must have an implementation in the classifier or inherited from an ancestor.
isLeaf	If true, then the implementation of the reception may not be overridden by a descendant classifier. If false, then the implementation of the reception may be overridden by a descendant classifier (but it need not be overridden).
isRoot	If true, then the classifier must not inherit a declaration of the same reception. If false, then the classifier may (but need not) inherit a declaration of the same reception. (But the declaration must match in any case; a classifier may not modify an inherited declaration of a reception.)
specification	A description of the effects of the classifier receiving a Signal, stated by a String.

Associations

signal	The Signal that the Classifier is prepared to handle.

2.9.2.13 Signal

A signal is a specification of an asynchronous stimulus communicated between instances. The receiving instance handles the signal by a state machine. Signal is a generalizable element and is defined independently of the classes handling the signal. A reception is a declaration that a class handles a signal, but the actual handling is specified by a state machine.

In the metamodel, Signal is a child to Classifier, with the parameters expressed as Attributes. A Signal is always asynchronous. A Signal is associated with the BehavioralFeatures that raise it.

Associations

context	The set of BehavioralFeatures that raise the signal.
reception	A set of Receptions that indicates Classes prepared to handle the signal.

2.9.2.14 Stimulus

A stimulus reifies a communication between two instances.

In the metamodel, Stimulus is a communication (i.e., a Signal sent to an Instance, or an invocation of an Operation). It can also be a request to create an Instance, or to destroy an Instance. It has a sender, a receiver, and may have a set of actual arguments, all being Instances.

Associations

argument	The sequence of Instances being the arguments of the Stimulus.
communicationLink	The Link, which is used for communication.
dispatchAction	The procedure that caused the Stimulus to be dispatched when it was executed.
receiver	The Instance that receives the Stimulus.
sender	The Instance that sends the Stimulus.

2.9.2.15 SubsystemInstance

A subsystem instance is an instance of a subsystem. It is the runtime representation of a subsystem, hence it can be connected to links corresponding to associations of the subsystem. Its task is to handle incoming communication by re-directing stimuli to the appropriate receiver inside the subsystem.

In the metamodel, SubsystemInstance is a subclass of Instance.

2.9.3 Well-formedness Rules

The following well-formedness rules apply to the Common Behavior package.

2.9.3.1 AttributeLink

[1] The type of the Instance must match the type of the Attribute.

```
self.value.classifier->union (
    self.value.classifier.allParents)->includes (
        self.attribute.type)
```

2.9.3.2 ComponentInstance

[1] A ComponentInstance originates from exactly one Component.

```
self.classifier->size = 1
and
self.classifier.oclIsKindOf (Component)
```

[2] A ComponentInstance may only own ComponentInstances.

```
self.contents->forAll (c | c.oclIsKindOf(ComponentInstance))
```

2.9.3.3 DataValue

[1] A DataValue originates from exactly one Classifier, which is a DataType.

```
(self.classifier->size = 1)
and
self.classifier.oclIsKindOf(DataType)
```

[2] A DataValue has no AttributeLinks.

```
self.slot->isEmpty
```

[3] A DataValue may not contain any Instances.

```
self.contents->isEmpty
```

2.9.3.4 Exception

No extra well-formedness rules.

2.9.3.5 Instance

[1] The AttributeLinks match the declarations in the Classifiers.

```
self.slot->forAll ( al |
    self.classifier->exists ( c |
        c.allAttributes->includes ( al.attribute ) ) )
```

[2] The Links matches the declarations in the Classifiers.

```
self.allLinks->forAll ( l |
    self.classifier->exists ( c |
        c.allAssociations->includes ( l.association ) ) )
```

[3] If two Operations have the same signature, they must be the same.

```
self.classifier->forAll ( c1, c2 |
    c1.allOperations->forAll ( op1 |
        c2.allOperations->forAll ( op2 |
            op1.hasSameSignature (op2)  implies op1 = op2 ) ) )
```

[3] There are no name conflicts between the AttributeLinks and opposite LinkEnds.

```
self.slot->forAll( al |
    not self.allOppositeLinkEnds->exists( le | le.name = al.name ) )
and
self.allOppositeLinkEnds->forAll( le |
    not self.slot->exists( al | le.name = al.name ) )
```

[4] For each Association in which an Instance is involved, the number of opposite LinkEnds must match the multiplicity of the AssociationEnd.

```
self.classifier.allOppositeAssociationEnds->forAll ( ae |
    ae.multiplicity.multiplicityRange->exists ( mr |
        self.selectedLinkEnds (ae)->size >= mr.lower and
        (mr.upper = 'unlimited' or
            (mr.upper <> 'unlimited' and
                self.selectedLinkEnds (ae)->size <=
                mr.upper.oclAsType (Integer) ) ) ) )
```

[5] The number of associated AttributeLinks must match the multiplicity of the Attribute.

```
self.classifier.allAttributes->forAll ( a |
    a.multiplicity.multiplicityRange->exists ( mr |
        self.selectedAttributeLinks (a)->size >= mr.lower and
        (mr.upper = 'unlimited' or
            (mr.upper <> 'unlimited' and
                self.selectedLinkEnds (a)->size <=
                mr.upper.oclAsType (Integer) ) ) ) )
```

Additional operations

[1] The operation allLinks results in a set containing all Links of the Instance itself.

```
allLinks : set(Link);
allLinks = self.linkEnd.link
```

[2] The operation allOppositeLinkEnds results in a set containing all LinkEnds of Links connected to the Instance with another LinkEnd.

```
allOppositeLinkEnds : set(LinkEnd);
allOppositeLinkEnds = self.allLinks.connection->select (le |
    le.instance <> self)
```

[3] The operation selectedLinkEnds results in a set containing all opposite LinkEnds corresponding to a given AssociationEnd.

```
selectedLinkEnds (ae : AssociationEnd) : set(LinkEnd);
selectedLinkEnds (ae) = self.allOppositeLinkEnds->select (le |
    le.associationEnd = ae)
```

[4] The operation selectedAttributeLinks results in a set containing all AttributeLinks corresponding to a given Attribute.

```
selectedAttributeLinks (ae : Attribute) : set(AttributeLink);
selectedAttributeLinks (a) = self.slot->select (s |
    s.attribute = a)
```

[5] The operation contents results in a Set containing all ModelElements contained by the Instance.

```
contents: Set(ModelElement);
contents = self.ownedInstance->union(self.ownedLink)
```

2.9.3.6 Link

[1] The set of LinkEnds must match the set of AssociationEnds of the Association.

```
Sequence {1..self.connection->size}->forAll ( i |
    self.connection->at (i).associationEnd =
    self.association.connection->at (i) )
```

[2] There are not two Links of the same Association that connect the same set of Instances in the same way.

```
self.association.link->forAll ( l |
    Sequence {1..self.connection->size}->forAll ( i |
        self.connection->at (i).instance =
        l.connection->at (i).instance )
            implies self = l )
```

2.9.3.7 LinkEnd

[1] The type of the Instance must match the type of the AssociationEnd.

```
self.instance.classifier->union (
    self.instance.classifier.allParents)->includes (
        self.associationEnd.type)
```

2.9.3.8 LinkObject

[1] One of the Classifiers must be the same as the Association.

```
self.classifier->includes(self.association)
```

[2] The Association must be a kind of AssociationClass.

```
self.association.oclIsKindOf(AssociationClass)
```

2.9.3.9 NodeInstance

[1] A NodeInstance must have only one Classifier as its origin, and it must be a Node.

```
self.classifier->forAll ( c | c.oclIsKindOf(Node))
and
self.classifier->size = 1
```

[2] Each ComponentInstance that resides on a NodeInstance must be an instance of a Component that resides on the corresponding Node.

```
self.resident->forAll(n |
    self.classifier.resident->includes(n.classifier))
```

[3] A NodeInstance may not contain any Instances.

```
self.contents->isEmpty
```

2.9.3.10 Object

[1] Each of the Classifiers must be a kind of Class or ClassifierInState.

```
self.classifier->forAll ( c | c.oclIsKindOf(Class) or
    (c.oclIsKindOf(ClassifierInState) and
    c.oclAsType(ClassifierInState).type.oclIsKindOf(Class)))
```

[2] An Object may only own Objects, DataValues, Links, UseCaseInstances, CollaborationInstances, and Stimuli.

```
self.contents->forAll(c |
    c.oclIsKindOf(Object) or
    c.oclIsKindOf(DataValue) or
    c.oclIsKindOf(Link) or
    c.oclIsKindOf(UseCaseInstance) or
    c.oclIsKindOf(CollaborationInstance) or
    c.oclIsKindOf(Stimuli))
```

2.9.3.11 Procedure

A procedure is a coordinated set of actions that models a computation, such as an algorithm. It can also be used without actions to express a procedure in a textual language.

In the metamodel, Procedure is a subclass of ModelElement. It can be linked to a Method or Expression to model how the method is carried out or the expression is evaluated.

Attributes

language	Names the language in which the body attribute is written. This language name should follow the conventions for language names in UML, as described for the language attribute of Expression.
body	The text of the procedure written in the given language..
isList	Determines whether the arguments tothe procedure are passed as attributes of a single object, or are passed separately. See description in Actions.

Associations

expression	An expression the value of which is calculated by the procedure. Used to provide a detailed action model for an expression.
method	A method which is performed by the procedure. Used to provide a detailed action model for a method.

2.9.3.12 Reception

[1] A Reception cannot be a query.

not self.isQuery

2.9.3.13 Signal

[1] A Signal may not contain any ModelElements.

self.contents->isEmpty

2.9.3.14 Stimulus

[1] The number of arguments must match the number of arguments of the procedure.

self.dispatchAction.argument->size = self.argument->size

2.9.3.15 SubsystemInstance

[1] A SubsystemInstance may only own Objects, DataValues, Links, UseCaseInstances, CollaborationInstances, SubsystemInstances, and Stimuli.

```
self.contents->forAll ( c |
        c.oclIsKindOf(Object) or
        c.oclIsKindOf(DataValue) or
        c.oclIsKindOf(Link) or
        c.oclIsKindOf(UseCaseInstance) or
        c.oclIsKindOf(CollaborationInstance) or
        c.oclIsKindOf(SubsystemInstance) or
        c.oclIsKindOf(Stimulus) )
```

[2] A SubsystemInstance originates from a Subsystem.

```
self.classifier.oclIsKindOf(Subsystem)
```

2.9.4 Detailed Semantics

This section provides a description of the semantics of the elements in the Common Behavior package.

2.9.4.1 Object and DataValue

An object is an instance that originates from a class, it is structured and behaves according to its class. All objects originating from the same class are structured in the same way, although each of them has its own set of attribute links. Each attribute link references an instance, usually a data value. The number of attribute links with the same name fulfills the multiplicity of the corresponding attribute in the class. The set may be modified according to the specification in the corresponding attribute. For example, each referenced instance must originate from (a specialization of) the type of the attribute, and attribute links may be added or removed according to the changeable property of the attribute.

An object may have multiple classes (i.e., it may originate from several classes). In this case, the object will have all the features declared in all of these classes, both the structural and the behavioral ones. Moreover, the set of classes (i.e., the set of features that the object conforms to) may vary over time. New classes may be added to the object and old ones may be detached. This means that the features of the new classes are dynamically added to the object, and the features declared in a class which is removed from the object are dynamically removed from the object. No name clashes between attributes links and opposite link ends are allowed, and each operation which is applicable to the object should have a unique signature.

Another kind of instance is data value, which is an instance with no identity. Moreover, a data value cannot change its state; all operations that are applicable to a data value are queries and do not cause any side effects. Since it is not possible to differentiate between two data values that appear to be the same, it becomes more of a philosophical

issue whether there are several data values representing the same value or just one for each value-it is not possible to tell. In addition, a data value cannot change its data type.

An instance may contain other instances as a result of a (namespace) containment between their classifiers. Namespace rules imply that an instance contained in another instance has access to all names that are accessible to its container instance.

Subsystem instances are further discussed in Model Management.

2.9.4.2 Link

A link is a connection between instances. Each link is an instance of an association, i.e. a link connects instances of (specializations of) the associated classifiers. In the context of an instance, an opposite end defines the set of instances connected to the instance via links of the same association and each instance is attached to its link via a link-end originating from the same association-end. However, to be able to use a particular opposite end, the corresponding link end attached to the instance must be navigable. An instance may use its opposite ends to access the associated instances. An instance can communicate with the instances of its opposite ends and also use references to them as arguments or reply values in communications.

A link object is a special kind of link, which at the same time is also an object. Since an object may change its classes this is also true for a link object. However, one of the classes must always be an association class.

2.9.4.3 Signal, Exception and Stimulus

Several kinds of requests exist between instances (e.g., sending a signal and invoking an operation). The former is used to trigger a reaction in the receiver in an asynchronous way and without a reply, while the latter applies an operation to an instance, which can be either done synchronously or asynchronously and may require a reply from the receiver to the sender. Other kinds of requests are used for example to create a new instance or to delete an already existing instance. When an instance communicates with another instance a stimulus is passed between the two instances. Each stimulus has a sender instance and a receiver instance, and possibly a sequence of arguments according to the specifying signal or operation. The stimulus uses a link between the sender and the receiver for communication. This link may be missing if the receiver is an argument inside the current activation, a local or global variable, or if the stimulus is sent to the sender instance itself. Moreover, a stimulus is dispatched by an action (e.g., a call action or a send action). The action specifies the request made by the stimulus, like the operation to be invoked or the signal event to be raised, as well as how the actual arguments of the stimulus are determined.

A signal may be attached to a classifier, which means that instances of the classifier will be able to receive that signal. This is facilitated by declaring a reception by the classifier. An exception is a special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with

the receiver of the exception, which may be the sender itself. Unlike other signals, the receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified as the target of the send action.

The reception of a stimulus originating from a call action by an instance causes the invocation of an operation on the receiver. The receiver executes the method that is found in the full descriptor of the class that corresponds to the operation. The reception of a stimulus originating from a signal by an instance may cause a transition and subsequent effects as specified by the state machine for the classifier of the recipient. This form of behavior is described in the State Machines package. Note that the invoked behavior is described by methods and state machine transitions. Operations and receptions merely declare that a classifier accepts a given operation invocation or signal but they do not specify the implementation.

2.10 Collaborations

2.10.1 Overview

The Collaborations package is a subpackage of the Behavioral Elements package. The package uses constructs defined in the Foundation package and the Common Behavior packages.

The Collaborations package provides the means to define *Collaborations* and *CollaborationInstanceSets*. The main constructs used in a Collaboration include ClassifierRole, AssociationRole, Interaction, and Message while Instance, Stimulus, and Link are used in a CollaborationInstanceSet.

The description of cooperating Instances involves two aspects: 1) the structural description of the participants, and 2) the description of their communication patterns. The structure of the participants that play the roles in the performance of a specific task and their relationships is called a *Collaboration*. The communication pattern performed by Instances playing the roles to accomplish the task is called an *Interaction*. The behavior is implemented by ensembles of Instances that exchange Stimuli within an overall Interaction. To understand the mechanisms used in a design, it is important to see only those Instances and their Interactions that are involved in accomplishing a purpose or a related set of purposes, projected from the larger system of which they are a part.

A Collaboration includes a set of ClassifierRoles and AssociationRoles that define the participants needed for a given set of purposes. Instances conforming to the ClassifierRoles play the roles defined by the ClassifierRoles, while Links between the Instances conform to AssociationRoles of the Collaboration. ClassifierRoles and AssociationRoles define a usage of Instances and Links, and the Classifiers and Associations declare all required properties of these Instances and Links.

An Interaction is defined in the context of a Collaboration. It specifies the communication patterns between the roles in the Collaboration. More precisely, it contains a set of partially ordered Messages, each specifying one communication; for example, what Signal to be sent or what Operation to be invoked, as well as the roles to be played by the sender and the receiver, respectively.

A CollaborationInstanceSet references a collection of Instances that jointly perform the task specified by the CollaborationInstanceSet's Collaboration. These Instances play the roles defined by the ClassifierRoles of the Collaboration; that is, the Instances have all the properties stated by (the Instances conform to) the ClassifierRoles. The Stimuli sent between the Instances when performing the task are participating in the InteractionInstanceSet of the CollaborationInstanceSet. These Stimuli conform to the Messages in one of the Interactions of the Collaboration. Since an Instance can participate in several CollaborationInstanceSets at the same time, all its communications are not necessarily referenced by only one InteractionInstanceSet. They can be interleaved.

A parameterized Collaboration represents a design construct that can be used repeatedly in different designs. The participants in the Collaboration, including the Classifiers and Relationships, can be parameters of the generic Collaboration. The parameters are bound to particular ModelElements in each instantiation of generic Collaboration. Such a parameterized Collaboration can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects). Whereas most Collaborations can be anonymous because they are attached to a named ModelElement, Collaboration patterns are free standing design constructs that must have names.

A Collaboration may be expressed at different levels of granularity. A coarse-grained Collaboration may be refined to produce another Collaboration that has a finer granularity.

Collaborations can be used for expressing several different things, like how use cases are realized, actor structures of ROOM, OOram role models, and collaborations as defined in Catalysis. They are also used for setting up the context of Interactions and for defining the mapping between the specification part and the realization part of a Subsystem.

A Collaboration may be attached to an Operation or a Classifier, like a UseCase, to describe the realization of the Operation or of the Classifier; that is, what roles Instances play to perform the behavior specified by the Operation or the UseCase. A Collaboration that describes a Classifier, like a UseCase, references Classifiers and Associations in general, while a Collaboration describing an Operation includes the arguments and the local variables of the Operation, as well as ordinary Associations attached to the Classifier owning the Operation. The Interactions defined within the Collaboration specify the communication pattern between the Instances when they perform the behavior specified in the Operation or the UseCase. A Collaboration may also be attached to a Classifier to define the static structure of it; that is, the roles played by the Attributes, the Parameters, etc.

A ClassifierRole or an AssociationRole has one or a collection of Classifiers or Associations as its base. The same Classifier or Association can appear as the base of roles in several Collaborations and several times in the same Collaboration, each time in a different role. In each appearance it is specified which of the properties of the Classifier or the Association are needed in the particular usage. These properties constitute a subset of all the properties of that Classifier or Association.

A Collaboration is a GeneralizableElement. This implies that a Collaboration may specify a task that is a specialization of another Collaboration's task.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Collaborations package.

2.10.2 Abstract Syntax

The abstract syntax for the Collaborations package is expressed in graphic notation in Figure 2-18 through Figure 2-20.

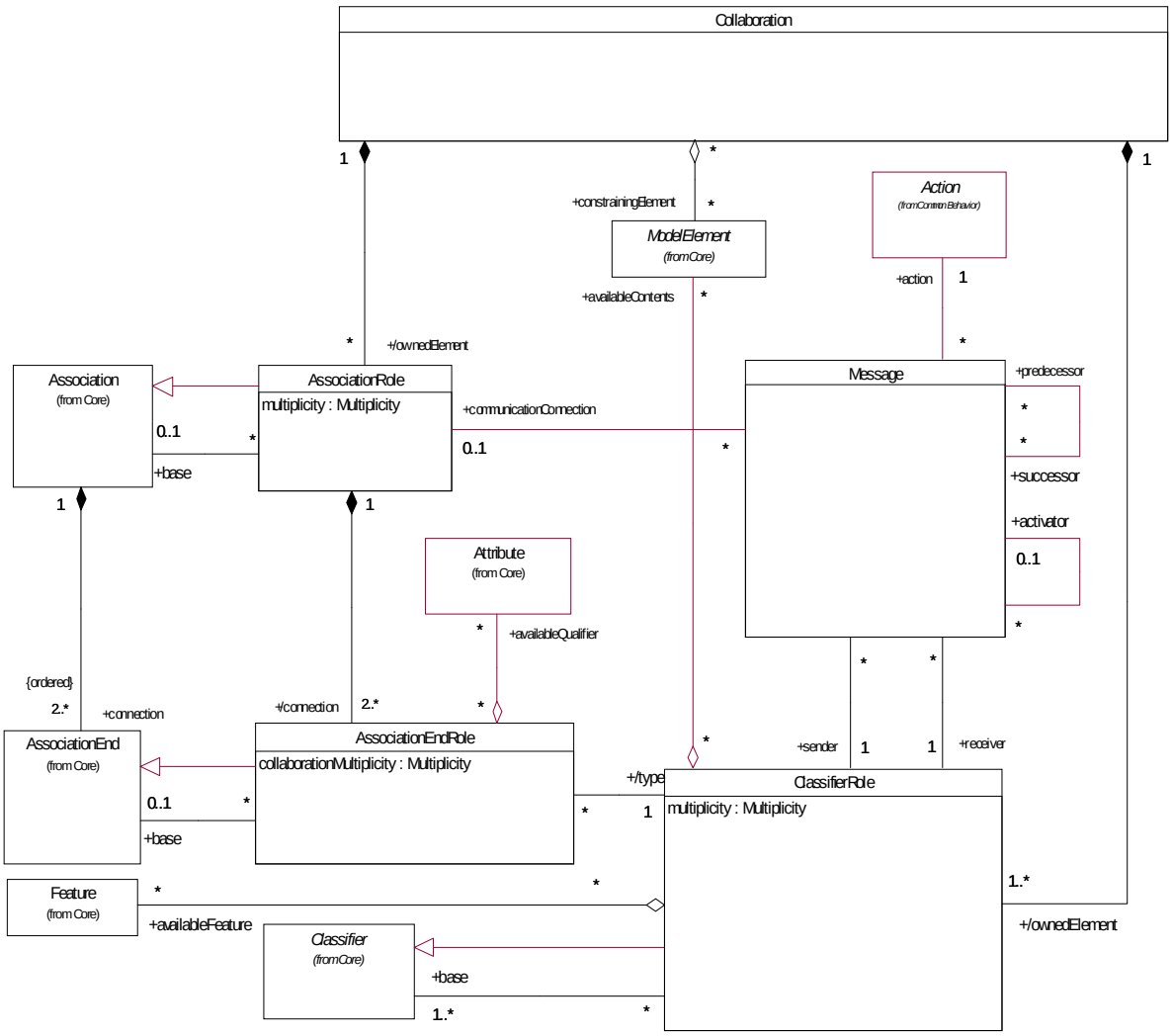


Figure 2-18 Collaborations - Roles

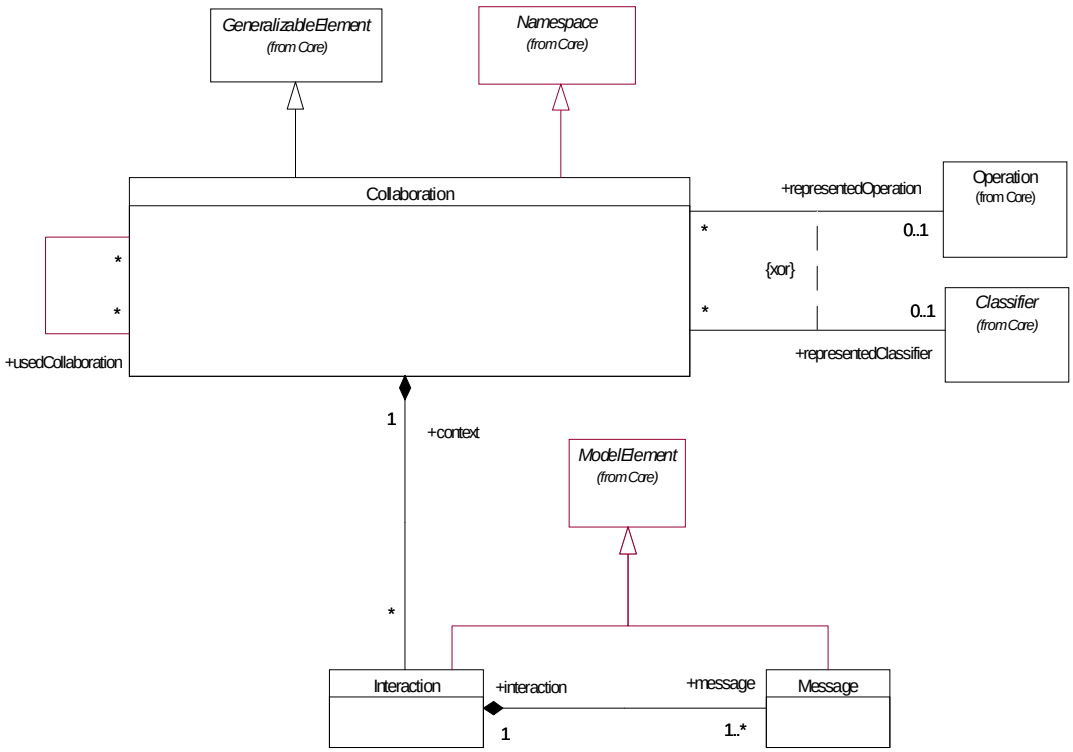


Figure 2-19 Collaborations - Interactions

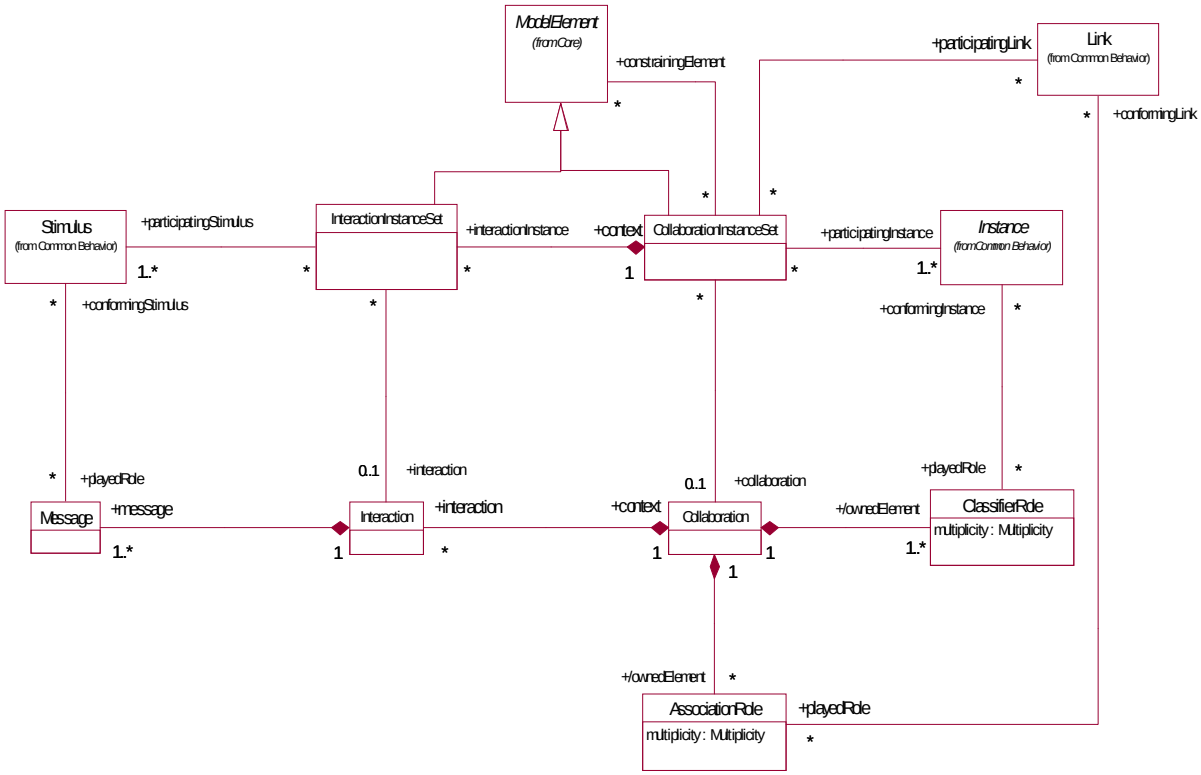


Figure 2-20 Collaborations - Instances

2.10.2.1 AssociationEndRole

An association-end role specifies an endpoint of an association as used in a collaboration.

In the metamodel, an AssociationEndRole is part of an AssociationRole and specifies the connection of an AssociationRole to a ClassifierRole. It is related to the AssociationEnd, declaring the corresponding part in an Association.

Attributes

collaborationMultiplicity	The number of LinkEnds playing this role in a Collaboration.

Associations

availableQualifier	The subset of Qualifiers that are used in the Collaboration.
base	The AssociationEnd of which the AssociationEndRole is a projection.

2.10.2.2 AssociationRole

An association role is a specific usage of an association needed in a collaboration.

In the metamodel, an AssociationRole specifies a restricted view of an Association used in a Collaboration. An AssociationRole is a composition of a set of AssociationEndRoles corresponding to the AssociationEnds of its base Association.

Attributes

multiplicity	The number of Links playing this role in a Collaboration.

Associations

base	The Association of which the AssociationRole is a view.
conformingLink	The collection of Links that conforms to the AssociationRole.

2.10.2.3 ClassifierRole

A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.

In the metamodel, a ClassifierRole specifies one participant of a Collaboration; that is, a role Instances conform to. A ClassifierRole defines a set of Features, which is a subset of those available in the base Classifiers, as well as a subset of ModelElements contained in the base Classifiers, that are used in the role. The ClassifierRole may be connected to a set of AssociationRoles via AssociationEndRoles. As ClassifierRole is a kind of Classifier, a Generalization relationship may be defined between two ClassifierRoles. The child role is a specialization of the parent; that is, the Features and the contents of the child includes the Features and contents of the parent.

Attributes

multiplicity	The number of Instances playing this role in a Collaboration.

Associations

availableContents	The subset of ModelElements contained in the base Classifier, which is used in the Collaboration.
availableFeature	The subset of Features of the base Classifier, which is used in the Collaboration.
base	The Classifiers, which the ClassifierRole is a view of.
conformingInstance	The collection of Instances that conforms to the ClassifierRole.

2.10.2.4 Collaboration

A collaboration describes how an operation or a classifier, like a use case, is realized by a set of classifiers and associations used in a specific way. The collaboration defines a set of roles to be played by instances and links, as well as a set of interactions that define the communication between the instances when they play the roles.

In the metamodel, a Collaboration contains a set of ClassifierRoles and AssociationRoles, which represent the Classifiers and Associations that take part in the realization of the associated Classifier or Operation. The Collaboration may also contain a set of Interactions that are used for describing the behavior performed by Instances conforming to the participating ClassifierRoles.

A Collaboration specifies a view (restriction, slice, projection) of a model of Classifiers. The projection describes the required relationships between Instances that conform to the participating ClassifierRoles, as well as the required subsets of the Features and contained ModelElements of these Classifiers. Several Collaborations may describe different projections of the same set of Classifiers. Hence, a Classifier can be a base for several ClassifierRoles.

A Collaboration may also reference a set of ModelElements, usually Classifiers and Generalizations, needed for expressing structural requirements, such as Generalizations required between the Classifiers themselves to fulfill the intent of the Collaboration.

A Collaboration is a GeneralizableElement, which implies that one Collaboration may specify a task that is a specialization of the task of another Collaboration.

Associations

constrainingElement	The ModelElements that add extra constraints, like Generalization and Constraint, on the ModelElements participating in the Collaboration.
interaction	The set of Interactions that are defined within the Collaboration.
ownedElement	(Inherited from Namespace) The set of roles defined by the Collaboration. These are ClassifierRoles and AssociationRoles.
representedClassifier	The Classifier the Collaboration is a realization of. (Used if the Collaboration represents a Classifier.)
representedOperation	The Operation the Collaboration is a realization of. (Used if the Collaboration represents an Operation.)
usedCollaboration	Collaborations that are used when defining the source Collaboration.

2.10.2.5 CollaborationInstanceSet

A collaboration instance set references a set of instances that jointly collaborate in performing the particular task specified by the collaboration of the collaboration instance. The instances in the collaboration instance set play the roles defined in the collaboration.

In the metamodel, a CollaborationInstanceSet references a set of Instances and Links that play the roles defined by the ClassifierRoles and AssociationRoles of the CollaborationInstanceSet's Collaboration.

A CollaborationInstanceSet contains an InteractionInstanceSet, which references the set of Stimuli that are interchanged between the Instances of the CollaborationInstanceSet and corresponds to the Messages of an Interaction in the CollaborationInstanceSet's Collaboration.

Associations

constrainingElement	The ModelElements that add extra constraints, like Generalization and Constraint, on the ModelElements participating in the Collaboration.
collaboration	The Collaboration, which declares the roles that the Instances that participate in the CollaborationInstanceSet play.
interactionInstanceSet	The InteractionInstanceSet that references the Stimuli passed between the Instances when performing the task of the CollaborationInstanceSet's Collaboration.
participatingInstance	The set of Instances that participate in the CollaborationInstanceSet.
participatingLink	The set of Links that participate in the CollaborationInstanceSet.

2.10.2.6 Interaction

An interaction specifies the communication between instances performing a specific task. Each interaction is defined in the context of a collaboration.

In the metamodel, an Interaction contains a set of Messages specifying the communication between a set of Instances conforming to the ClassifierRoles of the owning Collaboration.

Associations

context	The Collaboration that defines the context of the Interaction.
message	The Messages that specify the communication in the Interaction.

2.10.2.7 InteractionInstanceSet

An interaction instance set is the set of stimuli that participate in a collaboration instance set.

In the metamodel, an InteractionInstanceSet references a collection of Stimuli that conform to the Messages of the InteractionInstanceSet's Interaction.

Associations

context	The CollaborationInstanceSet that defines the context of the InteractionInstanceSet.
participating-Stimulus	The Stimuli that participate in the performance of the CollaborationInstanceSet.
interaction	The Interaction that defines the interaction pattern that the stimuli conforms to.

2.10.2.8 Message

A message defines a particular communication between instances that is specified in an interaction.

In the metamodel, a Message defines one specific kind of communication in an Interaction. A communication can be raising a Signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication, but also the roles of the sender and the receiver, the dispatching Action, and the role played by the communication Link. Furthermore, the Message defines the relative sequencing of Messages within the Interaction.

Associations

action	The Action that causes a Stimulus to be sent according to the Message.
activator	The Message that invokes the behavior causing the dispatching of the current Message.
communicationConnection	The AssociationRole played by the Links used in the communications specified by the Message.
conformingStimulus	The collection of Stimuli that conforms to the Message.
interaction	The Interaction of which the Message is a part.
receiver	The role of the Instance that receives the communication and reacts to it.
predecessor	The set of Messages whose completion enables the execution of the current Message. All of them must be completed before execution begins.
sender	The role of the Instance that invokes the communication and possibly receives a response.

2.10.3 Well-formedness Rules

The following well-formedness rules apply to the Collaborations package.

2.10.3.1 AssociationEndRole

[1] The type of the ClassifierRole must conform to the type of the base AssociationEnd.

```
self.type.base = self.base.type
or
self.type.base.allParents->includes (self.base.type)
```

[2] The type must be a kind of ClassifierRole.

```
self.type.oclIsKindOf (ClassifierRole)
```

[3] The qualifiers used in the AssociationEndRole must be a subset of those in the base AssociationEnd.

```
self.base.qualifier->includesAll (self.availableQualifier)
```

[4] In a Collaboration an Association may only be used for traversal if it is allowed by the base Association.

```
self.isNavigable implies self.base.isNavigable
```

[5] An AssociationEndRole is not a role of another AssociationEndRole.

```
not self.base.oclIsKindOf (AssociationEndRole)
```

2.10.3.2 AssociationRole

[1] The AssociationEndRoles must conform to the AssociationEnds of the base Association.

```
Sequence{ 1..(self.connection->size) }->forAll (index |
    self.connection->at(index).base =
    self.base.connection->at(index))
```

[2] The endpoints must be a kind of AssociationEndRoles.

```
self.connection->forAll( r | r.oclIsKindOf (AssociationEndRole) )
```

[3] An AssociationEnd is not a role of another AssociationEnd.

```
not self.base.oclIsKindOf (AssociationEnd)
```

2.10.3.3 ClassifierRole

[1] The AssociationRoles connected to the ClassifierRole must match a subset of the Associations connected to the base Classifiers.

```
self.allAssociations->forAll( ar |
    self.base.allAssociations->exists ( a | ar.base = a ) )
```

[2] The Features and contents of the ClassifierRole must be subsets of those of the base Classifiers.

```
self.base.allFeatures->includesAll (self.allAvailableFeatures)
and
self.base.allContents->includesAll (self.allAvailableContents)
```

[3] A ClassifierRole does not have any Features of its own.

```
self.allFeatures->isEmpty
```

[4] A ClassifierRole is not a role of another ClassifierRole.

```
not self.base.oclIsKindOf (ClassifierRole)
```

Additional operations

[1] The operation *allAvailableFeatures* results in the set of all Features contained in the ClassifierRole together with those contained in the parents.

```
allAvailableFeatures : Set(Feature);
allAvailableFeatures = self.availableFeature->union
    (self.parent.allAvailableFeatures)
```

[2] The operation *allAvailableContents* results in the set of all ModelElements contained in the ClassifierRole together with those contained in the parents.

```
allAvailableContents : Set(ModelElement);
allAvailableContents = self.availableContents->union
    (self.parent.allAvailableContents)
```

2.10.3.4 Collaboration

[1] All Classifiers and Associations of the ClassifierRoles and AssociationRoles in the Collaboration must be included in the namespace owning the Collaboration.

```
self.allContents->forAll ( e |
    (e.oclIsKindOf (ClassifierRole) implies
        self.namespace.allContents->includes (
            e.oclAsType(ClassifierRole).base) )
    and
    (e.oclIsKindOf (AssociationRole) implies
        self.namespace.allContents->includes (
            e.oclAsType(AssociationRole).base) ))
```

[2] All the constraining ModelElements must be included in the namespace owning the Collaboration.

```
self.constrainingElement->forAll ( ce |
    self.namespace.allContents->includes (ce) )
```

[3] If a ClassifierRole or an AssociationRole does not have a name, then it should be the only one with a particular base.

```
self.allContents->forAll ( p |
    (p.oclIsKindOf (ClassifierRole) implies
        p.name = '' implies
            self.allContents->forAll ( q |
                q.oclIsKindOf(ClassifierRole) implies
                    (p.oclAsType(ClassifierRole).base =
                        q.oclAsType(ClassifierRole).base implies
                            p = q) ) )
    and
    (p.oclIsKindOf (AssociationRole) implies
        p.name = '' implies
            self.allContents->forAll ( q |
                q.oclIsKindOf(AssociationRole) implies
                    (p.oclAsType(AssociationRole).base =
                        q.oclAsType(AssociationRole).base implies
                            p = q) ) )
)
```

[4] A Collaboration may only contain ClassifierRoles and AssociationRoles, the Generalizations and the Constraints between them, and Actions used in the Collaboration's Interactions.

```
self.allContents->forAll ( p |
    p.oclIsKindOf (ClassifierRole) or
    p.oclIsKindOf (AssociationRole) or
    p.oclIsKindOf (Generalization) or
    p.oclIsKindOf (Action) or
    p.oclIsKindOf (Constraint) )
```

[5] An Action contained in a Collaboration must be connected to a Message; that is, be the dispatching Action of the Message, in an Interaction of the Collaboration.

```
self.allContents->forAll ( p |
    p.oclIsKindOf (Action) implies
    self.interaction->exists ( i : Interaction |
        i.messages->exists ( m : Message | m.action = p ) ) )
```

[6] A role with the same name as one of the roles in a parent of the Collaboration must be a child (a specialization) of that role.

```
self.contents->forAll ( c |
    self.parent.allContents->forall ( p |
        c.name = p.name implies c.allParents->include (p) ))
```

Additional operations

[1] The operation *allContents* results in the set of all ModelElements contained in the Collaboration together with those contained in the parents except those that have been specialized.

```
allContents : Set(ModelElement);
allContents = self.contents->union (
    self.parent.allContents->reject ( e |
        self.contents.name->include (e.name) ))
```

2.10.3.5 CollaborationInstanceSet

[1] The Interaction of the CollaborationInstanceSet's InteractionInstanceSet must be defined within the CollaborationInstanceSet's Collaboration.

```
self.collaboration.interaction->includes (
    self.interactionInstanceSet.interaction)
```

2.10.3.6 Interaction

[1] All Signals being sent must be included in the namespace owning the Collaboration in which the Interaction is defined.

```
self.message->forAll ( m |
    m.action.oclIsKindOf(SendAction) implies
        self.context.namespace.allContents->includes (
            m.action->oclAsType (SendAction).signal) )
```

2.10.3.7 InteractionInstanceSet

No extra well-formedness rules.

2.10.3.8 Message

[1] The sender and the receiver must participate in the Collaboration, which defines the context of the Interaction.

```
self.interaction.context.ownedElement->includes (self.sender)
and
self.interaction.context.ownedElement->includes (self.receiver)
```

[2] The predecessors and the activator must be contained in the same Interaction.

```
self.predecessor->forAll ( p | p.interaction = self.interaction )
and
self.activator->forAll ( a | a.interaction = self.interaction )
```

[3] The predecessors must have the same activator as the Message.

```
self.allPredecessors->forAll ( p | p.activator = self.activator )
```

[4] A Message cannot be the predecessor of itself.

```
not self.allPredecessors->includes (self)
```

[5] The communicationLink of the Message must be an AssociationRole in the context of the Message's Interaction.

```
self.interaction.context.ownedElement->includes (
    self.communicationConnection)
```

[6] The sender and the receiver roles must be connected by the AssociationRole, which acts as the communication connection.

```
self.communicationConnection->size > 0 implies
    self.communicationConnection.connection->exists (ar |
        ar.type = self.sender)
    and
    self.communicationConnection.connection->exists (ar |
        ar.type = self.receiver)
```

Additional operations

[1] The operation allPredecessors results in the set of all Messages that precede the current one.

```
allPredecessors : Set(Message);
allPredecessors = self.predecessor->union
    (self.predecessor.allPredecessors)
```

2.10.4 Detailed Semantics

This section provides a description of the semantics of the elements in the Collaborations package. It is divided into two parts: Collaboration and Interaction. The description of behavior involves two aspects: 1) the structural description of the participants, and 2) the description of their communication patterns. The structure of Instances playing roles in a behavior and their relationships is described by a c*ollaboration*. The communication pattern performed by Instances playing the roles to accomplish a specific purpose is specified by an *interaction*.

2.10.4.1 Collaboration

Behavior is implemented by ensembles of instances that exchange stimuli to accomplish a task. To understand the mechanisms used in a design, it is important to see only those instances and their interactions that are involved in accomplishing the task or a related set of tasks, projected from the larger system of which they are parts of, and might be used for other purposes as well. Such a static construct is called a *collaboration*.

A collaboration defines an ensemble of participants that are needed for a given set of purposes. The participants define roles that instances and links play when interacting with each other. The roles to be played by the instances are modeled as classifier roles, and by the links as association roles. Classifier roles and association roles define a usage of instances and links, while the classifiers and associations specify all required properties of these instances and links. This means that the structure of an ensemble of interlinked instances conforms to the roles in a collaboration as they collaborate to achieve a given task. Reasoning about the behavior of an ensemble of instances can therefore be done in the context of the collaboration as well as in the context of the instances.

A collaboration can be used for specification of how an operation or a classifier, like a use case, is realized by an ensemble of classifiers and associations. Together, the classifiers and their associations participating in the collaboration meet the requirements of the realized operation or classifier. The collaboration defines a context in which the behavior of the realized element can be specified.

A collaboration specifies what properties instances must have to be able to take part in the collaboration; that is, a role in the collaboration specifies the required set of features a conforming instance must have. Furthermore, the collaboration also states what associations must exist between the participants, as well as what classifiers a participant, like a subsystem, must contain. Neither all features nor all contents of the participating classifiers and not all associations between these classifiers are always required in a particular collaboration. Because of this, a collaboration is defined in terms of classifier roles. A classifier role is a description of the features required in a particular collaboration; that is, a classifier role can be seen as a projection of a classifier, which is called the base of the classifier role. (In fact, since an instance can originate from multiple classifiers at the same time (multiple classification), a classifier role can have several base classifiers.) However, instances of different classifiers can play the role defined by the classifier role, as long as they have all the required properties. Several classifier roles may have the same base classifier, even in the same collaboration, but their features and contained elements may be different subsets of the features and contained elements of the classifier. These classifier roles specify different roles played by (possibly different) instances of the same classifier.

A collaboration may be attached to an operation or a classifier, like a use case, to describe the context in which their behavior occurs; that is, what roles instances play to perform the behavior specified by the operation or the use case. A collaboration used in this way describes the realization of the operation or the classifier. A collaboration that describes for example a use case, references classifiers and associations in general, while a collaboration describing an operation includes only the parameters and the local variables of the operation, as well as ordinary associations attached to the classifier owning the operation. The interactions defined within the collaboration (see below) specify the communication pattern between the instances when they perform the behavior specified in the operation or the use case. A collaboration may also be attached to a class to define its static structure; that is, how its attributes, parameters etc. cooperate with each other.

In a collaboration, the association roles define what associations are needed between the classifiers in this context. Each association role represents the usage of an association in the collaboration, and it is defined between the classifier roles that represent the associated classifiers. The represented association is called the base association of the association role. As the association roles specify a particular usage of an association in a specific collaboration, all constraints expressed by the association ends are not necessarily required to be fulfilled in the specified usage. The multiplicity of the association end may be reduced in the collaboration; that is, the upper and the lower bounds of the association end roles may be inside the bounds of the corresponding end of the base association, as it might be that only a subset of the associated instances participate in the collaboration instance set. Similarly, an association may be traversed in some, but perhaps not all, of the allowed directions in the specific collaboration; that is, the value of the isNavigable property of an

association end role may be false even if the value of that property of the base association end is true. (However, the opposite is not true; that is, an association may not be used for traversal in a direction that is not allowed according to the isNavigable properties of the association ends.) The changeability and ordering of an association end may be strengthened in an association end role; that is, in a particular usage the end is used in a more restricted way than is defined by the association. Furthermore, if an association has a collection of qualifiers (see the Core), some of them may be used in a specific collaboration. An association end role may therefore include a subset of the qualifiers defined by the corresponding association end of the base association.

A collaboration instance set references a collection of instances that play the roles defined in the collaboration instance set's collaboration. An instance participating in a collaboration instance set plays a specific role; that is, conforms to a classifier role, in the collaboration. The number of instances that should play one specific role in a collaboration is specified by the classifier role's multiplicity. Different instances may play the same role but in different collaboration instance sets. Since all these instances play the same role, they must all conform to the classifier role specifying the role. Thus, they are normally instances of one of the base classifier of the classifier role, or one of their descendants. The only requirement on conforming instances is that they must offer operations according to the classifier role, as well as support attribute links corresponding to the attributes specified by the classifier role, and links corresponding to the association roles connected to the classifier role. They may, therefore, be instances of any classifier meeting this requirement. The instances may, of course, have more attribute links than required by the classifier role, which for example would be the case if they originate from a classifier being a child of a base classifier. Moreover, a conforming instance may also support more attribute links than required if it originates from multiple classifiers (multiple classification). Finally, one instance may play different roles in different collaboration instance sets of the same collaboration. In fact, the instance may play multiple roles in the same collaboration instance set.

Collaborations (but not collaboration instance sets) may have generalization relationships to other collaborations. This means that one collaboration can specify a specialization of another collaboration's task. This implies that all the roles of the parent collaboration are also available in the child collaboration; the child collaboration may, of course, also contain new roles. The former roles may possibly be specialized with new features; that is, the role defined in the parent is replaced in the child by a role with the same name as the parent role. The role in the child must reference the same collection of features and the same collection of contained elements as the role in the parent, and may also reference some additional features and additional contained elements. In this way it is possible to specialize a collaboration both by adding new roles and by replacing existing roles with specializations of them. The specialized role, that is, a role with a generalization relationship to the replaced role, may both reference new features and replace (override) features of its parent. Note that the base classifiers of the specialized roles are not necessarily specializations of the base classifiers of the parent's roles; it is enough that they contain all the required features.

How the instances referenced by a collaboration instance set should interact to jointly perform the behavior of the classifier realized by the collaboration is specified with a set of interactions (see below). The collaboration thus specifies the context in which

these interactions are performed. If the collaboration represents an operation, the context includes things like parameters, attributes, and classifiers contained in the classifier owning the operation. The interactions then specify how the arguments, the attribute values, the instances etc. will cooperate to perform the behavior specified by the operation. If the collaboration is a specialization of another collaboration, all communications specified by the parent collaboration are also included in the child, as the child collaboration includes all the roles of the parent. However, new messages may be inserted into these sequences of communication, since the child may include specializations of the parent's roles as well as new roles. The child may of course also include completely new interactions that do not exist in the parent.

Two or more collaborations may be composed to form a new collaboration. For example, when refining a superordinate use case into a set of subordinate use cases, the collaborations specifying each of the subordinate use cases may be composed into one collaboration, which will be a (simple) refinement of the superordinate collaboration. The composition is done by observing that at least one instance must participate in both sets of collaborating instances. This instance has to conform to one classifier role in each collaboration. In the composite collaboration these two classifier roles are merged into a new one, which will contain all features included in either of the two original classifier roles. The new classifier role will, of course, be able to fulfill the requirements of both of the previous collaborations, so the instance participating in both of the two sets of collaborating instances will conform to the new classifier role.

A parameterized collaboration represents a design construct that can be used repeatedly in different designs. The participants in the collaboration, including the classifiers and relationships, can be parameters of the generic collaboration. The parameters are bound to particular model elements in each instantiation of generic collaboration. Such a parameterized collaboration can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects). Whereas most collaborations can be anonymous because they are attached to a named model element, collaboration patterns are free standing design constructs that must have names.

A collaboration may be a specification of a template. There will not be any instances of such a collaboration template, but it can be used for generating ordinary collaborations, which may be instantiated. Collaboration templates may have parameters that act like placeholders in the template. Usually, these parameters would be used as base classifiers and associations, but other kinds of model elements can also be defined as parameters in the collaboration, like operation or signal. In a collaboration generated from the template these parameters are refined by other model elements that make the collaboration instantiable.

Moreover, a collaboration may also contain a set of constraining model elements, like constraints and generalizations perhaps together with some extra classifiers. These constraining model elements do not participate in the collaboration themselves, but are used for expressing the extra constraints on the participating elements in the collaboration that cannot be covered by the participating roles themselves. For example, in a collaboration template it might be required that the base classifiers of two roles must have a common ancestor, or one role must be a subclass of another one.

These kinds of requirements cannot be expressed with association roles, as the association roles express the required links between participating instances. An extra set of model elements may therefore be included in the collaboration.

2.10.4.2 Interaction

An interaction is defined in the context of a collaboration. It specifies the communication patterns between its roles. More precisely, it contains a set of partially ordered messages, each specifying one communication, such as what signal to be sent or what operation to be invoked, as well as the roles to be played by the sender and the receiver, respectively.

The purpose of an interaction is to specify the communication between an ensemble of interacting instances performing a specific task. An interaction is defined within a collaboration; that is, the collaboration defines the context in which the interaction takes place. The instances performing the communication specified by the interaction are included in a collaboration instance set; that is, they conform to the classifier roles of the collaboration instance set's collaboration.

An interaction specifies the sending of a set of stimuli. These are partially ordered based on which execution thread they belong to. Within each thread the stimuli are sent in a sequential order while stimuli of different threads may be sent in parallel or in an arbitrary order.

An interaction instance set references the collection of stimuli that constitute the actual communication between the collection of instances. These instances are the collection of instances that participate in the collaboration instance set owning the interaction instance set. Hence, the interaction instance set includes those stimuli that the instances communicate when performing the task of the collaboration instance set. The stimuli of an interaction instance set match the messages of the interaction instance set's interaction.

A message is a specification of a communication. It specifies the roles of the sender and the receiver instances, as well as which association role specifies the communication link. The message is connected to an action, which specifies the statement that, when executed, causes the communication specified by the message to take place. If the action is a call action or a send action, the signal to be sent or the operation to be invoked in the communication is stated by the action. The action also contains the argument expressions that, when executed, will determine the actual arguments being transmitted in the communication. Moreover, any conditions or iterations of the communication are also specified by the action. Apart from send action and call action, the action connected to a message can also be of other kinds, like create action and destroy action. In these cases, the communication will not raise a signal or invoke an operation, but cause a new instance to be created or an already existing instance to be destroyed. In the case of a create action, the receiver specified by the message is the role to be played by the instance, which is created when the action is performed.

The stimuli being sent when an action is executed conforms to a message, implying that the sender and receiver instances of the stimuli are in conformance with the sender and the receiver roles specified by the message. Furthermore, the action dispatching the stimulus is the same as the action attached to the message. If the action connected to the message is a create action or destroy action, the receiver role of the message specifies the role to be played by the instance, or was played by the instance, respectively.

The interaction specifies the activator and predecessors of each message. The activator is the message that invoked the procedure that in turn invokes the current message. Every message except the initial messages of an interaction thus has an activator. The predecessors are the set of messages that must be completed before the current message may be executed. The first message in a procedure of course has no predecessors. If a message has more than one predecessor, it represents the joining of two threads of control. If a message has more than one successor (the inverse of predecessor), it indicates a fork of control into multiple threads. Thus, the predecessor's relationship imposes a partial ordering on the messages within a procedure, whereas the activator relationship imposes a tree on the activation of operations. Messages may be executed concurrently subject to the sequential constraints imposed by the predecessors and activator relationship.

2.10.5 Notes

In UML, the term Pattern is a synonym for a collaboration template that describes the structure of a design pattern. This definition is not as powerful as the term is used in other contexts. In general, design patterns involve many nonstructural aspects, such as heuristics for their use and lists of advantages and disadvantages. Such aspects are not modeled by UML and may be represented as text or tables.

2.11 Use Cases

2.11.1 Overview

The Use Cases package is a subpackage of the Behavioral Elements package. It specifies the concepts used for definition of the functionality of an entity like a system. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

The elements in the Use Cases package are primarily used to define the behavior of an entity, like a system or a subsystem, without specifying its internal structure. The key elements in this package are UseCase and Actor. Instances of use cases and instances of actors interact when the services of the entity are used. How a use case is realized in terms of cooperating objects, defined by classes inside the entity, can be specified with a Collaboration. A use case of an entity may be refined to a set of use cases of the elements contained in the entity. How these subordinate use cases interact can also be expressed in a Collaboration. The specification of the functionality of the system itself

is usually expressed in a separate use-case model; that is, a Model stereotyped «useCaseModel» (see Section 4.3, “Stereotypes and Notation,” on page 4-2). The use cases and actors in the use-case model are equivalent to those of the top-level package.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Use Cases package.

2.11.2 Abstract Syntax

The abstract syntax for the Use Cases package is expressed in graphic notation in Figure 2-21 on page 2-130.

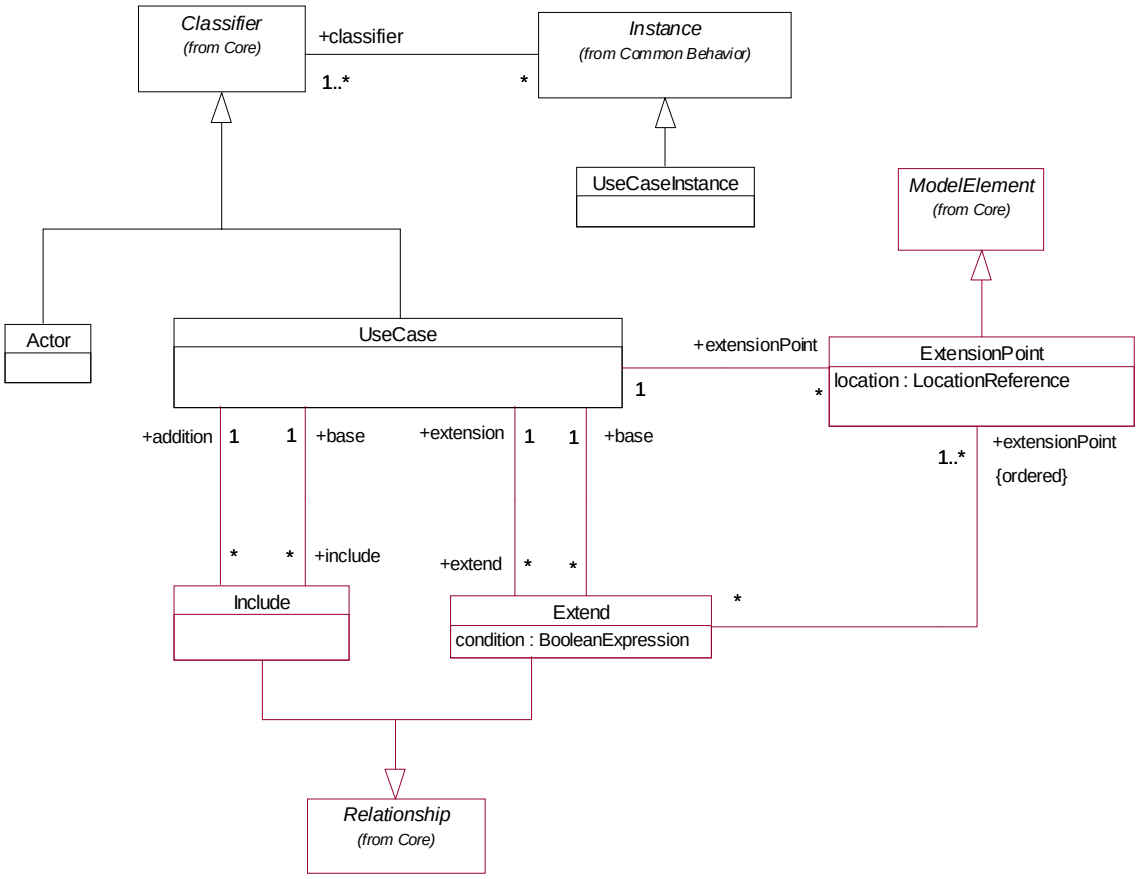


Figure 2-21 Use Cases

The following metaclasses are contained in the Use Cases package.

2.11.2.1 Actor

An *actor* defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.

In the metamodel, Actor is a subclass of Classifier. An Actor has a Name and may communicate with a set of UseCases, and, at realization level, with Classifiers taking part in the realization of these UseCases. An Actor may also have a set of Interfaces, each describing how other elements may communicate with the Actor.

An Actor may have generalization relationships to other Actors. This means that the child Actor will be able to play the same roles as the parent Actor, that is, communicate with the same set of UseCases, as the parent Actor.

2.11.2.2 Extend

An *extend* relationship defines that instances of a use case may be augmented with some additional behavior defined in an extending use case.

In the metamodel, an Extend relationship is a directed relationship implying that a UseCaseInstance of the base UseCase may be augmented with the structure and behavior defined in the extending UseCase. The relationship consists of a condition, which must be fulfilled if the extension is to take place, and a sequence of references to extension points in the base UseCase where the additional behavior fragments are to be inserted.

Attributes

condition	An expression specifying the condition that must be fulfilled if the extension is to take place.

Associations

base	The UseCase to be extended.
extension	The UseCase specifying the extending behavior.
extensionPoint	A sequence of extension-points in the base UseCase specifying where the additions are to be inserted.

2.11.2.3 ExtensionPoint

An extension point references one or a collection of locations in a use case where the use case may be extended.

In the metamodel, an ExtensionPoint has a name and one or a collection of descriptions of locations in the behavior of the owning use case, where a piece of behavior may be inserted into the owning use case.

Attributes

location	A reference to one location or a collection of locations where an extension to the behavior of the use case may be inserted.

2.11.2.4 Include

An include relationship defines that a use case contains the behavior defined in another use case.

In the metamodel, an Include relationship is a directed relationship between two UseCases implying that the behavior in the addition UseCase is inserted into the behavior of the base UseCase. The base UseCase may only depend on the result of performing the behavior defined in the addition UseCase, but not on the structure; that is, on the existence of specific attributes and operations, of the addition UseCase.

Associations

addition	The UseCase specifying the additional behavior.
base	The UseCase that is to include the addition.

2.11.2.5 UseCase

The use case construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity.

In the metamodel, UseCase is a subclass of Classifier, specifying the sequences of actions performed by an instance of the UseCase. The actions include changes of the state and communications with the environment of the UseCase. The sequences can be described using many different techniques, like Operation and Methods, ActivityGraphs, and StateMachines.

There may be Associations between UseCases and the Actors of the UseCases. Such an Association states that an instance of the UseCase and a user playing one of the roles of the Actor communicate. UseCases may be related to other UseCases by Extend, Include, and Generalization relationships. An Include relationship means that a UseCase includes the behavior described in another UseCase, while an Extend relationship implies that a UseCase may extend the behavior described in another UseCase, ruled by a condition. Generalization between UseCases means that the child is a more specific form of the parent. The child inherits all Features and Associations of the parent, and may add new Features and Associations.

The realization of a UseCase may be specified by a set of Collaborations; that is, the Collaborations define how Instances in the system interact to perform the sequences of the UseCase.

Associations

extend	A collection of Extend relationships to UseCases that the UseCase extends.
extensionPoint	Defines a collection of ExtensionPoints where the UseCase may be extended.
include	A collection of Include relationships to UseCases that the UseCase includes.

2.11.2.6 UseCaseInstance

A use case instance is the performance of a sequence of actions specified in a use case.

In the metamodel, UseCaseInstance is a subclass of Instance. Each method performed by a UseCaseInstance is performed as an atomic transaction; that is, it is not interrupted by any other UseCaseInstance.

An explicitly described UseCaseInstance is called a scenario.

2.11.3 Well-formedness Rules

The following well-formedness rules apply to the Use Cases package.

2.11.3.1 Actor

[1] Actors can only have Associations to UseCases, Subsystems, and Classes and these Associations are binary.

```
self.associations->forAll(a |
    a.connection->size = 2 and
    a.allConnections->exists(r | r.type.oclIsKindOf(Actor)) and
    a.allConnections->exists(r |
        r.type.oclIsKindOf(UseCase) or
        r.type.oclIsKindOf(Subsystem) or
        r.type.oclIsKindOf(Class)))
```

[2] Actors cannot contain any Classifiers.

```
self.contents->isEmpty
```

2.11.3.2 Extend

[1] The referenced ExtensionPoints must be included in set of ExtensionPoint in the target UseCase.

```
self.base.allExtensionPoints -> includesAll (self.extensionPoint)
```

2.11.3.3 ExtensionPoint

[1] The name must not be the empty string.

```
not self.name = ''
```

2.11.3.4 Include

No extra well-formedness rules.

2.11.3.5 UseCase

[1] UseCases can only have binary Associations.

```
self.associations->forAll(a | a.connection->size = 2)
```

[2] UseCases cannot have Associations to UseCases specifying the same entity.

```
self.associations->forAll(a |
    a.allConnections->forAll(s, o|
        (s.type.specificationPath->isEmpty and
        o.type.specificationPath->isEmpty )
    or
        (not s.type.specificationPath->includesAll(
            o.type.specificationPath) and
        not o.type.specificationPath->includesAll(
            s.type.specificationPath))
    ))
```

[3] A UseCase cannot contain any Classifiers.

```
self.contents->isEmpty
```

[4] The names of the ExtensionPoints must be unique within the UseCase.

```
self.allExtensionPoints -> forAll (x, y |
    x.name = y.name implies x = y )
```

Additional operations

[1] The operation specificationPath results in a set containing all surrounding Namespaces that are not instances of Package.

```
specificationPath : Set(Namespace)
specificationPath = self.allSurroundingNamespaces->select(n |
    n.oclIsKindOf(Subsystem) or n.oclIsKindOf(Class))
```

[2] The operation allExtensionPoints results in a set containing all ExtensionPoints of the UseCase

```
allExtensionPoints : Set(ExtensionPoint)
allExtensionPoints = self.allSupertypes.extensionPoint -> union (
                                self.extensionPoint)
```

2.11.3.6 UseCaseInstance

[1] The Classifier of a UseCaseInstance must be a UseCase.

```
self.classifier->forAll ( c | c.oclIsKindOf (UseCase) )
```

[2] A UseCaseInstance may not contain any Instances.

```
self.contents->isEmpty
```

2.11.4 Detailed Semantics

This section provides a description of the semantics of the elements in the Use Cases package, and its relationship to other elements in the Behavioral Elements package.

2.11.4.1 Actor

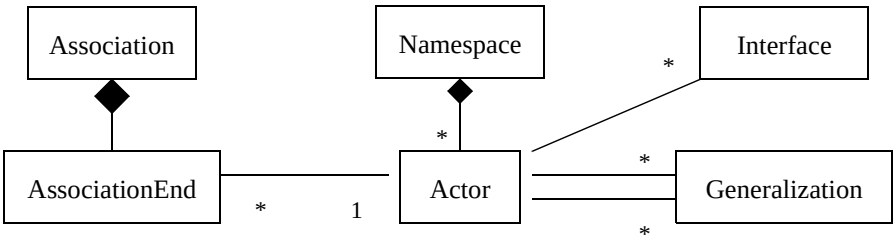


Figure 2-22 Actor Illustration

Actors model parties outside an entity, such as a system, a subsystem, or a class that interact with the entity. Each actor defines a coherent set of roles users of the entity can play when interacting with the entity. Every time a specific user interacts with the entity, it is playing one such role. An instance of an actor is a specific user interacting with the entity. Any instance that conforms to an actor can act as an instance of the actor. If the entity is a system, the actors represent both human users and other systems. Some of the actors of a lower level subsystem or a class may coincide with actors of the system, while others appear inside the system. The roles defined by the latter kind of actors are played by instances of classifiers in other packages or subsystems; in the latter case the classifier may belong to either the specification part or the realization part of the subsystem.

Since an actor is outside the entity, its internal structure is not defined but only its external view as seen from the entity. Actor instances communicate with the entity by sending and receiving message instances to and from use case instances and, at realization level, to and from objects. This is expressed by associations between the actor and the use case or the class. Furthermore, interfaces can be connected to an actor, defining how other elements may interact with the actor.

Two or more actors may have commonalities; that is, communicate with the same set of use cases in the same way. The commonality is expressed with generalizations to another (possibly abstract) actor, which models the common role(s). An instance of a child can always be used where an instance of the parent is expected.

2.11.4.2 UseCase

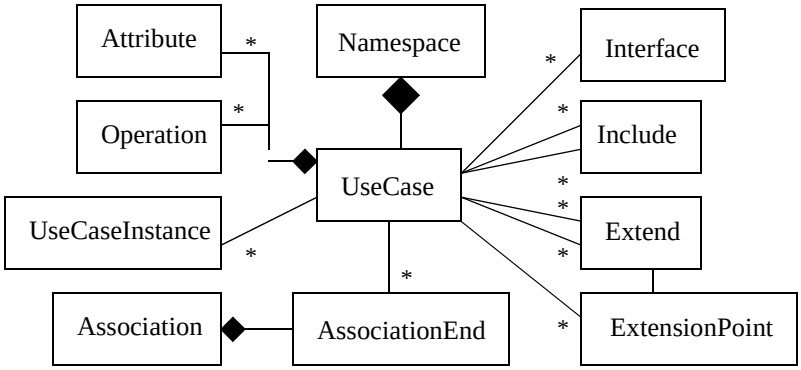


Figure 2-23 UseCase Illustration

In the following text, the term entity is used when referring to a system, a subsystem, or a class and the terms model element and element denote a subsystem or a class.

The purpose of a use case is to define a piece of behavior of an entity without revealing the internal structure of the entity. The entity specified in this way may be a system or any model element that contains behavior, like a subsystem or a class, in a model of a system. Each use case specifies a service the entity provides to its users; that is, a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again. A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity. A use case also includes possible variants of this sequence (for example, alternative sequences, exceptional behavior, error handling, etc.). The complete set of use cases specifies all different ways to use the entity; that is, all behavior of the entity is expressed by its use cases. These use cases can be grouped into packages for convenience.

From a pragmatic point of view, use cases can be used both for specification of the (external) requirements on an entity and for specification of the functionality offered by an (already realized) entity. Moreover, the use cases also indirectly state the requirements the specified entity poses on its users; that is, how they should interact so the entity will be able to perform its services.

Since users of use cases always are external to the specified entity, they are represented by actors of the entity. Thus, if the specified entity is a system or a subsystem at the topmost level, the users of its use cases are modeled by the actors of the system. Those

actors of a lower level subsystem or a class that are internal to the system are often not explicitly defined. Instead, the use cases relate directly to model elements conforming to these implicit actors; that is, whose instances play the roles of these actors in interaction with the use cases. These model elements are contained in other packages or subsystems, where in the subsystem case they may be contained in the specification part or the realization part. The distinction between actor and conforming element like this is often neglected; thus, they are both referred to by the term actor.

There may be associations between use cases and actors, meaning that the instances of the use case and the actor communicate with each other. One actor may communicate with several use cases of an entity; that is, the actor may request several services of the entity, and one use case communicates with one or several actors when providing its service. Note that two use cases specifying the same entity cannot communicate with each other since each of them individually describes a complete usage of the entity. Moreover, use cases always use signals when communicating with actors outside the system, while they may use other communication semantics when communicating with elements inside the system.

The interaction between actors and use cases can be defined with interfaces. An interface of a use case defines a subset of the entire interaction defined in the use case. Different interfaces offered by the same use case need not be disjoint.

A use case can be described in plain text, using operations and methods together with attributes, in activity graphs, by a state machine, or by other behavior description techniques, such as preconditions and postconditions. The interaction between a use case and its actors can also be presented in collaboration diagrams for specification of the interactions between the entity containing the use case and the entity's environment.

A use-case instance is a performance of a use case, initiated by a message instance from an instance of an actor. As a response, the use-case instance performs a sequence of actions as specified by the use case, like communicating with actor instances, not necessarily only the initiating one. The actor instances may send new message instances to the use-case instance and the interaction continues until the instance has responded to all input and does not expect any more input, when it ends. Each method performed by a use-case instance is performed as an atomic transaction; that is, it is not interrupted by any other use-case instance.

In the case where subsystems are used to model the system's containment hierarchy, the system can be specified with use cases at all levels, as use cases can be used to specify subsystems and classes. A use case specifying one model element is then refined into a set of smaller use cases, each specifying a service of a model element contained in the first one. The use case of the whole may be referred to as superordinate to its refining use cases, which, correspondingly, may be called subordinate in relation to the first one. The functionality specified by each superordinate use case is completely traceable to its subordinate use cases. Note, though, that the structure of the container element is not revealed by the use cases, since they only specify the functionality offered by the element. The subordinate use cases of a specific superordinate use case cooperate to perform the superordinate one. Their cooperation is specified by collaborations and may be presented in collaboration diagrams. A specific subordinate use case may appear in several collaborations; that is

play a role in the performances of several superordinate use cases. In each such collaboration, other roles specify the cooperation with this specific subordinate use case. These roles are the roles played by the actors of that subordinate use case. Some of these actors may be the actors of the superordinate use case, as each actor of a superordinate use case appears as an actor of at least one of the subordinate use cases. Furthermore, the interfaces of a superordinate use case are traceable to the interfaces of those subordinate use cases that communicate with actors that are also actors of the superordinate use case.

The environment of subordinate use cases is the model element containing the model elements specified by these use cases. Thus, from a bottom-up perspective, an interaction between subordinate use cases results in a superordinate use case, that is, a use case of the container element.

Use cases of classes are mapped onto operations of the classes, since a service of a class in essence is the invocation of the operations of the class. Some use cases may consist of the application of only one operation, while others may involve a set of operations, usually in a well-defined sequence. One operation may be needed in several of the services of the class, and will therefore appear in several use cases of the class.

The realization of a use case depends on the kind of model element it specifies. For example, since the use cases of a class are specified by means of operations of the class, they are realized by the corresponding methods, while the use cases of a subsystem are realized by the elements contained in the subsystem. Since a subsystem does not have any behavior of its own, all services offered by a subsystem must be a composition of services offered by elements contained in the subsystem (i.e., eventually by classes). These elements will collaborate and jointly perform the behavior of the specified use case. One or a set of collaborations describes how the realization of a use case is made. Hence, collaborations are used for specification of both the refinement and the realization of a use case in terms of subordinate use cases.

The usage of use cases at all levels imply not only a uniform way of specification of functionality at all levels, but also a powerful technique for tracing requirements at the system package level down to operations of the classes. The propagation of the effect of modifying a single operation at the class level all the way up to the behavior of the system package is managed in the same way.

Commonalities between use cases can be expressed in three different ways: with generalization, include, and extend relationships. A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participates in all relationships of the parent use case. The child use case may also define new behavior sequences, as well as add additional behavior into and specialize existing behavior of the inherited ones. One use case may have several parent use cases and one use case may be a parent to several other use cases.

An include relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. When a use-case instance reaches the location where the behavior of an another use case is to be included, it performs all the behavior described by the included use case and then continues according to its original use case. This

means that although there may be several paths through the included use case due to (e.g., conditional statements), all of them must end in such a way that the use-case instance can continue according to the original use case. One use case may be included in several other use cases and one use case may include several other use cases. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior, which may easily be reused in several use cases. Moreover, the base use case may only be dependent on the results of performing the included behavior and not on structure, like Attributes and Associations, of the included use case.

An extend relationship defines that a use case may be augmented with some additional behavior defined in another use case. One use case may extend several use cases and one use case may be extended by several use cases. The base use case may not be dependent of the addition of the extending use case. The extend relationship contains a condition and references a sequence of extension points in the target use case. The condition must be satisfied if the extension is to take place, and the references to the extension points define the locations in the base use case where the additions are to be made. Once an instance of a use case is to perform some behavior referenced by an extension point of its use case, and the extension point is the first one in an extends relationship's sequence of references to extension points, the condition of the relationship is evaluated. If the condition is fulfilled, the sequence obeyed by the use-case instance is extended to include the sequence of the extending use case. The different parts of the extending use case are inserted at the locations defined by the sequence of extension points in the relationship -- one part at each referenced extension point. Note that the condition is only evaluated once: at the first referenced extension point, and if it is fulfilled all of the extending use case is inserted in the original sequence. An extension point may define one location or a set of locations in the behavior defined by the use case. However, if an extend relationship references a sequence of extension points, only the first one may define a set of locations. All other ones must define exactly one location each. Which of the locations of the first extension point to use is determined by where the extension is triggered. This is not possible for the other ones. In other words, once the extension has been triggered, all locations where to add the different part of the extending use case must be uniquely defined. Hence, all extension points, except for the first one, referenced by an extend relationship must define single locations. The description of the location references by an extension point can be made in several different ways, like textual description of where in the behavior the addition should be made, pre-or post conditions, or using the name of a state in a state machine.

Note that the three kinds of relationships described above can only exist between use cases specifying the same entity. The reason for this is that the use cases of one entity specify the behavior of that entity alone; that is, all use-case instances are performed entirely within that entity. If a use case would have a generalization, include, or extend relationship to a use case of another entity, the resulting use-case instances would involve both entities, resulting in a contradiction. However, generalization, include, and extend relationships can be defined from use cases specifying one entity to use cases of another one if the first entity has a generalization to the second one, since the contents of both entities are available in the first entity. However, the contents of the second entity must be at least protected, so they become available inside the child entity.

As a first step when developing a system, the dynamic requirements of the system as a whole can be expressed with use cases. The entity being specified is then the whole system, and the result is a separate model called a use-case model, that is, a model with the stereotype «useCaseModel». Next, the realization of the requirements is expressed with a model containing a system package, probably a package hierarchy, and at the bottom a set of classes. If the system package, that is, a package with the stereotype «topLevelPackage» is a subsystem, its abstract behavior is naturally the same as that of the system. Thus, if use cases are used for the specification part of the system package, these use cases are equivalent to those in the use-case model of the system; that is, they express the same behavior but possibly slightly differently structured. In other words, all services specified by the use cases of a system package, and only those, define the services offered by the system. Furthermore, if several models are used for modeling the realization of a system (for example, an analysis model and a design model), the set of use cases of all system packages and the use cases of the use-case model must be equivalent.

2.11.5 Notes

A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors. This ensures that the use cases are complete specifications and not just fragments.

2.12 State Machines

2.12.1 Overview

The State Machine package is a subpackage of the Behavioral Elements package. It specifies a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. These concepts are based on concepts defined in the Foundation package as well as concepts defined in the Common Behavior package. This enables integration with the other subpackages in Behavioral Elements.

The state machine formalism described in this section is an object-based variant of Harel statecharts. It incorporates several concepts similar to those defined in ROOMcharts, a variant of statechart defined in the ROOM modeling language. The major differences relative to classical Harel statecharts are described on page 2.12.5.42-169.

State machines can be used to specify behavior of various elements that are being modeled. For example, they can be used to model the behavior of individual entities (e.g., class instances) or to define the interactions (e.g., collaborations) between entities.

In addition, the state machine formalism provides the semantic foundation for activity graphs. This means that activity graphs are simply a special form of state machines.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the State Machines package. Activity graphs are described in Section 2.13, “Activity Graphs,” on page 2-170.

2.12.2 Abstract Syntax

The abstract syntax for state machines is expressed graphically in Figure 2-24 on page 2-141, which covers all the basic concepts of state machine graphs such as states and transitions. Figure 2-25 on page 2-142 describes the abstract syntax of events that can trigger state machine behavior. The specifications of the concepts defined in these two diagrams are listed in alphabetical order following the figures.

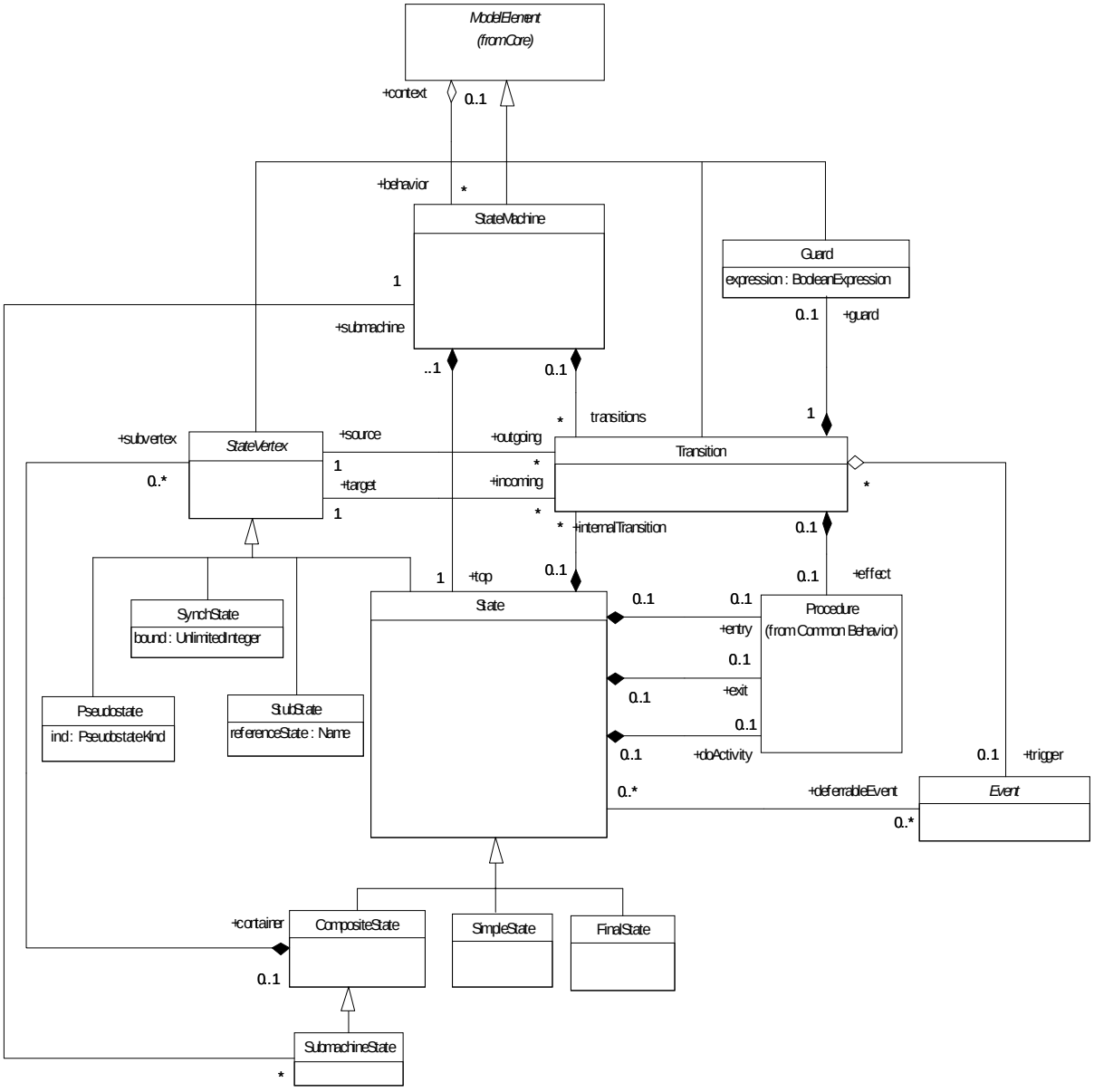


Figure 2-24 State Machines - Main

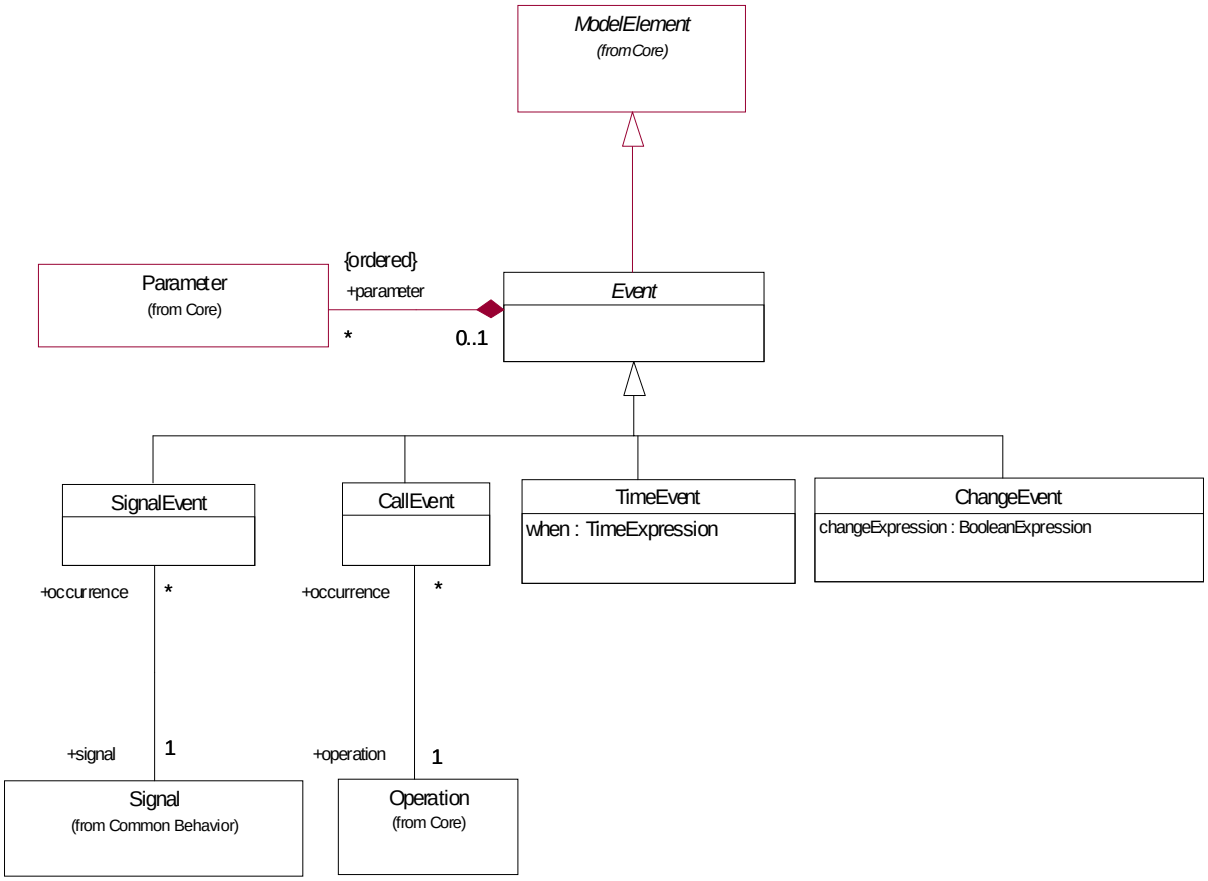


Figure 2-25 State Machines - Events

2.12.2.1 CallEvent

A call event represents the *reception* of a request to synchronously invoke a specific operation. (Note that a call event instance is distinct from the call action that caused it.) The expected result is the execution of a sequence of actions which characterize the operation behavior at a particular state.

Two special cases of CallEvent are the object creation event and the object destruction event.

Associations

operation	Designates the operation whose invocation raised the call event

Stereotypes

«create»	Class	Create is a stereotyped call event denoting that the instance receiving that event has just been created. For state machines, it triggers the initial transition at the topmost level of the state machine (and is the only kind of trigger that may be applied to an initial transition).
«destroy»	Class	Destroy is a stereotyped call event denoting that the instance receiving the event is being destroyed.

2.12.2.2 ChangeEvent

A change event models an event that occurs when an explicit boolean expression becomes true as a result of a change in value of one or more attributes or associations. A change event is raised implicitly and is *not* the result of some explicit change event action.

The change event should not be confused with a guard. A guard is only evaluated at the time an event is dispatched whereas, conceptually, the boolean expression associated with a change event is evaluated continuously until it becomes true. The event that is generated remains until it is consumed even if the boolean expression changes to false after that.

Attributes

changeExpression	The boolean expression that specifies the change event.

2.12.2.3 CompositeState

A composite state is a state that contains other state vertices (states, pseudostates, etc.). The association between the composite and the contained vertices is a composition association. Hence, a state vertex can be a part of at most one composite state.

Any state enclosed within a composite state is called a *substate* of that composite state. It is called a *direct substate* when it is not contained by any other state; otherwise it is referred to as a *transitively nested substate*.

CompositeState is a child of State.

Associations

subvertex	The set of state vertices that are owned by this composite state.

Attributes

isConcurrent	A boolean value that specifies the decomposition semantics. If this attribute is true, then the composite state is decomposed directly into two or more orthogonal conjunctive components called *regions* (usually associated with concurrent execution). If this attribute is false, then there are no direct orthogonal components in the composite.
isRegion	A derived boolean value that indicates whether a CompositeState is a substate of a concurrent state. If it is true, then this composite state is a direct substate of a concurrent state.

2.12.2.4 Event

An event is a specification of a type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration.

Strictly speaking, the term “event” is used to refer to the type and not to an instance of the type. However, on occasion, where the meaning is clear from the context, the term is also used to refer to an event instance.

Event is a child of ModelElement.

Associations

parameter	The list of parameters defined by the event.

2.12.2.5 FinalState

A special kind of state signifying that the enclosing composite state is completed. If the enclosing state is the top state, then it means that the entire state machine has completed.

A final state cannot have any outgoing transitions.

FinalState is a child of State.

2.12.2.6 Guard

A guard is a boolean expression that is attached to a transition as a fine-grained control over its firing. The guard is evaluated when an event instance is dispatched by the state machine. If the guard is true at that time, the transition is enabled, otherwise, it is disabled.

Guards should be pure expressions without side effects. Guard expressions with side effects are ill formed.

Guard is a child of ModelElement.

Attributes

expression	The boolean expression that specifies the guard.

2.12.2.7 PseudoState

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. They are used, typically, to connect multiple transitions into more complex state transitions paths. For example, by combining a transition entering a fork pseudostate with a set of transitions exiting the fork pseudostate, we get a compound transition that leads to a set of concurrent target states.

The following pseudostate kinds are defined:

- An *initial* pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a composite state.
- *deepHistory* is used as a shorthand notation that represents the most recent active configuration of the composite state that directly contains this pseudostate; that is, the state configuration that was active when the composite state was last exited. A composite state can have at most one deep history vertex. A transition may originate from the history connector to the *default* deep history state. This transition is taken in case the composite state had never been active before.
- *shallowHistory* is a shorthand notation that represents the most recent active substate of its containing state (but *not* the substates of that substate). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. A transition may originate from the history connector to the *initial* shallow history state. This transition is taken in case the composite state had never been active before.
- *join* vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards.
- *fork* vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices. The segments outgoing from a fork vertex must not have guards.
- *junction* vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as an *merge*). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a *static conditional branch*. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.) Static conditional branches are distinct from dynamic conditional branches that are realized by choice vertices (described below).

- *choice* vertices which, when reached, result in the dynamic evaluation of the guards of its outgoing transitions. This realizes a *dynamic conditional branch*. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill formed. (To avoid this, it is recommended to define one outgoing transition with the predefined "else" guard for every choice vertex.) Choice vertices should be distinguished from static branch points that are based on junction points (described above).

PseudoState is a child of StateVertex.

Attributes

kind	Determines the precise type of the PseudoState and can be one of: *initial, deepHistory, shallowHistory, join, fork, junction,* or *choice*.

2.12.2.8 SignalEvent

A signal event represents the *reception* of a particular (asynchronous) signal. A signal event instance should not be confused with the action (e.g., send action) that generated it. SignalEvent is a child of Event.

Associations

signal	The specific signal that is associated with this event.

2.12.2.9 SimpleState

A SimpleState is a state that does not have substates. It is a child of State.

2.12.2.10 State

A state is an abstract metaclass that models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (i.e., the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed). State is a child of StateVertex.

Associations

deferrableEvent	A list of events that are candidates to be retained by the state machine if they trigger no transitions out of the state (not consumed).
entry	An optional procedure that is executed whenever this state is entered regardless of the transition taken to reach the state. If defined, entry actions are always executed to completion prior to any internal activity or transitions performed within the state.
exit	An optional procedure that is executed whenever this state is exited regardless of which transition was taken out of the state. If defined, exit actions are always executed to completion only after all internal activities and transition actions have completed execution.
doActivity	An optional activity that is executed while being in the state. The execution starts when this state is entered, and stops either by itself, or when the state is exited, whichever comes first.
internalTransition	A set of transitions that, if triggered, occur without exiting or entering this state. Thus, they do not cause a state change. This means that the entry or exit condition of the State will not be invoked. These transitions can be taken even if the state machine is in one or more regions nested within this state.

2.12.2.11 StateMachine

A state machine is a specification that describes all possible behaviors of some dynamic model element. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of event instances. During this traversal, the state machine executes a series of actions associated with various elements of the state machine.

StateMachine has a composition relationship to State, which represents the top-level state, and a set of transitions. This means that a state machine owns its transitions and its top state. All remaining states are transitively owned through the state containment hierarchy rooted in the top state. The association to ModelElement provides the context of the state machine. A common case of the context relation is where a state machine is used to specify the lifecycle of a classifier.

Associations

context	An association to the model element that whose behavior is specified by this state machine. A model element may have more than one state machine (although one is sufficient for most purposes). Each state machine is optionally owned by one model element.
top	Designates the top-level state that is the root of the state containment hierarchy. There is exactly one state in every state machine that is the top state.

transition	The set of transitions owned by the state machine. Note that internal transitions are owned by their containing states and not by the state machine.

2.12.2.12 StateVertex

A StateVertex is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

StateVertex is a child of ModelElement.

Associations

outgoing	Specifies the transitions departing from the vertex.
incoming	Specifies the transitions entering the vertex.
container	The composite state that contains this state vertex.

2.12.2.13 StubState

A stub state can appear within a submachine state and represents an actual subvertex contained within the referenced state machine. It can serve as a source or destination of transitions that connect a state vertex in the containing state machine with a subvertex in the referenced state machine.

StubState is a child of State.

Associations

referenceState	Designates the referenced state as a pathname (a name formed by the concatenation of the name of a state and the successive names of all states that contain it, up to the top state).

2.12.2.14 SubmachineState

A submachine state is a syntactical convenience that facilitates reuse and modularity. It is a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. The state machine that is inserted is called the *referenced* state machine while the state machine that contains the submachine state is called the *containing* state machine. The same state machine may be referenced more than once in the context of a single containing state machine. In effect, a submachine state represents a “call” to a state machine “subroutine” with one or more entry and exit points.

The entry and exit points are specified by stub states. SubmachineState is a child of State.

Associations

submachine	The state machine that is to be substituted in place of the submachine state.

2.12.2.15 SynchState

A synch state is a vertex used for synchronizing the concurrent regions of a state machine. It is different from a state in the sense that it is not mapped to a boolean value (active, not active), but an integer. A synch sate is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states. SynchState is a child of StateVertex.

Attributes

bound	A positive integer or the value “unlimited” specifying the maximal count of the SynchState. The count is the difference between the number of times the incoming and outgoing transitions of the synch state are fired

2.12.2.16 TimeEvent

A TimeEvent models the expiration of a specific deadline. Note that the time of occurrence of a time event instance (i.e., the expiration of the deadline) is the same as the time of its reception. However, it is important to note that there may be a variable delay between the time of reception and the time of dispatching (e.g., due to queueing delays).

The expression specifying the deadline may be relative or absolute. If the time expression is relative and no explicit starting time is defined, then it is relative to the time of entry into the source state of the transition triggered by the event. In the latter case, the time event instance is generated only if the state machine is still in that state when the deadline expires.

Attributes

when	Specifies the corresponding time deadline

2.12.2.17 Transition

A transition is a directed relationship between a source state vertex and a target state vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to a particular event instance. Transition is a child of ModelElement.

Associations

trigger	Specifies the event that fires the transition. There can be at most one trigger per transition
guard	A boolean predicate that provides a fine-grained control over the firing of the transition. It must be true for the transition to be fired. It is evaluated at the time the event is dispatched. There can be at most one guard per transition.
effect	Specifies an optional procedure to be performed when the transition fires.
source	Designates the originating state vertex (state or pseudostate) of the transition.
target	Designates the target state vertex that is reached when the transition is taken.

2.12.3 Well-formedness Rules

The following well-formedness rules apply to the State Machines package.

2.12.3.1 CompositeState

[1] A composite state can have at most one initial vertex.

```
self.subvertex->select (v | v.oclIsKindOf(Pseudostate))->
    select(p : Pseudostate | p.kind = #initial)->size <= 1
```

[2] A composite state can have at most one deep history vertex.

```
self.subvertex->select (v | v.oclIsKindOf(Pseudostate))->
    select(p : Pseudostate | p.kind = #deepHistory)->size <= 1
```

[3] A composite state can have at most one shallow history vertex.

```
self.subvertex->select(v | v.oclIsKindOf(Pseudostate))->
    select(p : Pseudostate | p.kind = #shallowHistory)->size <= 1
```

[4] There have to be at least two composite substates in a concurrent composite state.

```
(self.isConcurrent) implies
    (self.subvertex->select
        (v | v.oclIsKindOf(CompositeState))->size >= 2)
```

[5] A concurrent state can only have composite states as substates.

```
(self.isConcurrent) implies
    self.subvertex->forAll(s | s.oclIsKindOf(CompositeState))
```

[6] The substates of a composite state are part of only that composite state.

```
self.subvertex->forAll(s | (s.container->size = 1) and (s.container = self))
```

2.12.3.2 FinalState

[1] A final state cannot have any outgoing transitions.

```
self.outgoing->size = 0
```

2.12.3.3 Guard

[1] A guard should not have side effects.

```
self.transition->stateMachine->notEmpty implies
    post: (self.transition.stateMachine->context =
    self.transition.stateMachine->context@pre)
```

2.12.3.4 PseudoState

[1] An initial vertex can have at most one outgoing transition and no incoming transitions.

```
(self.kind = #initial) implies
    ((self.outgoing->size <= 1) and (self.incoming->isEmpty))
```

[2] History vertices can have at most one outgoing transition.

```
((self.kind = #deepHistory) or (self.kind = #shallowHistory)) implies
    (self.outgoing->size <= 1)
```

[3] A join vertex must have at least two incoming transitions and exactly one outgoing transition.

```
(self.kind = #join) implies
    ((self.outgoing->size = 1) and (self.incoming->size >= 2))
```

[4] All transitions incoming a join vertex must originate in different regions of a concurrent state.

```
(self.kind = #join
    and not oclIsKindOf(self.stateMachine, ActivityGraph))
implies
    self.incoming->forAll (t1, t2 | t1<>t2 implies
        (self.stateMachine.LCA(t1.source, t2.source).
            container.isConcurrent)
```

[5] A fork vertex must have at least two outgoing transitions and exactly one incoming transition.

```
(self.kind = #fork) implies
    ((self.incoming->size = 1) and (self.outgoing->size >= 2))
```

[6] All transitions outgoing a fork vertex must target states in different regions of a concurrent state.

```
(self.kind = #fork
    and not oclIsKindOf(self.stateMachine, ActivityGraph)) implies
    self.outgoing->forAll (t1, t2 | t1<>t2 implies
        (self.stateMachine.LCA(t1.target, t2.target).
            container.isConcurrent)
```

[7] A junction vertex must have at least one incoming and one outgoing transition.

```
(self.kind = #junction) implies
    ((self.incoming->size >= 1) and (self.outgoing->size >= 1))
```

[8] A choice vertex must have at least one incoming and one outgoing transition.

```
(self.kind = #choice) implies
    ((self.incoming->size >= 1) and (self.outgoing->size >= 1))
```

2.12.3.5 StateMachine

[1] A StateMachine is aggregated within either a classifier or a behavioral feature.

```
self.context.notEmpty implies
    (self.context.oclIsKindOf(BehavioralFeature) or
        self.context.oclIsKindOf(Classifier))
```

[2] A top state is always a composite.

```
self.top.oclIsTypeOf(CompositeState)
```

[3] A top state cannot have any containing states.

```
self.top.container->isEmpty
```

[4] The top state cannot be the source of a transition.

```
(self.top.outgoing->isEmpty)
```

[5] If a StateMachine describes a behavioral feature, it contains no triggers of type CallEvent, apart from the trigger on the initial transition (see OCL for Transition [8]).

```
self.context.oclIsKindOf(BehavioralFeature) implies
self.transitions->reject(
    source.oclIsKindOf(Pseudostate) and
            source.oclAsType(Pseudostate).kind= #initial).trigger->isEmpty
```

Additional Operations

[1] The operation LCA(s1,s2) returns the state which is the least common ancestor of states s1 and s2.

```
context StateMachine::LCA (s1 : State, s2 : State) :
    CompositeState
result = if ancestor (s1, s2) then
        s1
    else if ancestor (s2, s1) then
        s2
    else (LCA (s1.container, s2.container))
```

[2] The query ancestor(s1, s2) checks whether s2 is an ancestor state of state s1.

```
context StateMachine::ancestor (s1 : State, s2 : State) : Boolean
result = if (s2 = s1) then
        true
    else if (s1.container->isEmpty) then
        true
    else if (s2.container->isEmpty) then
        false
    else (ancestor (s1, s2.container)
```

2.12.3.6 SynchState

[1] The value of the bound attribute must be a positive integer, or unlimited.

```
(self.bound > 0) or (self.bound = unlimited)
```

[2] All incoming transitions to a SynchState must come from the same region and all outgoing transitions from a SynchState must go to the same region.

2.12.3.7 SubmachineState

[1] Only stub states allowed as substates of a submachine state.

```
self.subvertex->forAll (s | s.oclIsTypeOf(StubState))
```

[2] Submachine states are never concurrent.

```
self.isConcurrent = false
```

2.12.3.8 Transition

[1] A fork segment should not have guards or triggers.

```
(self.source.oclIsKindOf(Pseudostate)
    and not oclIsKindOf(self.stateMachine, ActivityGraph)) implies
    ((self.source.oclAsType(Pseudostate).kind = #fork) implies
        ((self.guard->isEmpty) and (self.trigger->isEmpty)))
```

[2] A join segment should not have guards or triggers.

```
self.target.oclIsKindOf(Pseudostate) implies
    ((self.target.oclAsType(Pseudostate).kind = #join) implies
        ((self.guard->isEmpty) and (self.trigger->isEmpty)))
```

[3] A fork segment should always target a state.

```
(self.stateMachine->notEmpty
    and not oclIsKindOf(self.stateMachine, ActivityGraph)) implies
self.source.oclIsKindOf(Pseudostate) implies
    ((self.source.oclAsType(Pseudostate).kind = #fork) implies
            (self.target.oclIsKindOf(State)))
```

[4] A join segment should always originate from a state.

```
(self.stateMachine->notEmpty
    and not oclIsKindOf(self.stateMachine, ActivityGraph)) implies
self.target.oclIsKindOf(Pseudostate) implies
    ((self.target.oclAsType(Pseudostate).kind = #join) implies
            (self.source.oclIsKindOf(State)))
```

[5] Transitions outgoing pseudostates may not have a trigger

```
self.source.oclIsKindOf(Pseudostate)
    implies  (self.trigger->isEmpty))
```

[6] An initial transition at the topmost level either has no trigger or it has a trigger with the stereotype “create.”

```
self.source.oclIsKindOf(Pseudostate) implies
    (self.source.oclAsType(Pseudostate).kind = #initial) implies
        (self.source.container = self.stateMachine.top) implies
            ((self.trigger->isEmpty) or
             (self.trigger.stereotype.name = 'create'))
```

2.12.4 Detailed Semantics

This section describes the execution semantics of state machines. For convenience, the semantics are described in terms of the operations of a hypothetical machine that implements a state machine specification. This is for reference purposes only. Individual realizations are free to choose any form that achieves the same semantics.

In the general case, the key components of this hypothetical machine are:

- an *event queue* which holds incoming event instances until they are dispatched
- an *event dispatcher mechanism* that selects and de-queues event instances from the event queue for processing
- an *event processor* which processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question. Because of that, this component is simply referred to as the “state machine” in the following text.

Although the intent is to define the semantics of state machines very precisely, there are a number of semantic variation points to allow for different semantic interpretations that might be required in different domains of application. These are clearly identified in the text.

The basic semantics of events, states, transitions, etc. are discussed first in separate subsections under the appropriate headings. The operation of the state machine as a whole are then described in the state machine subsection.

2.12.4.1 Event

Event instances are generated as a result of some action either within the system or in the environment surrounding the system. An event is then conveyed to one or more targets. The means by which event instances are transported to their destination depend on the type of action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is practically instantaneous and completely reliable while in others it may involve variable transmission delays, loss of events, reordering, or duplication. No specific assumptions are made in this regard. This provides full flexibility for modeling different types of communication facilities.

An event is *received* when it is placed on the event queue of its target. An event is *dispatched* when it is dequeued from the event queue and delivered to the state machine for processing. At this point, it is referred to as the *current event*. Finally, it is *consumed* when event processing is completed. A consumed event is no longer available for processing. No assumptions are made about the time intervals between event reception, event dispatching, and consumption. This leaves open the possibility of different semantic models such as zero-time semantics.

Any parameter values associated with the current event are available to all actions directly caused by that event (transition actions, entry actions, etc.).

Event generalization may be defined explicitly by a signal taxonomy in the case of signal events, or implicitly defined by event expressions, as in time events.

2.12.4.2 State

Active states

A state can be active or inactive during execution. A state becomes *active* when it is entered as a result of some transition, and becomes *inactive* if it is exited as a result of a transition. A state can be exited and entered as a result of the same transition (e.g., self transition).

State entry and exit

Whenever a state is entered, it executes its entry action *before* any other action is executed. Conversely, whenever a state is exited, it executes its exit action as the final step prior to leaving the state.

If defined, the activity associated with a state is forked as a concurrent activity at the instant when the entry action of the state is completed. Upon exit, the activity is terminated before the exit action is executed.

Activity in state (do-activity)

The activity represents the execution of a sequence of actions, that occurs while the state machine is in the corresponding state. The activity starts executing upon entering the state, following the entry action. If the activity completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition (see below) the state will be exited. If the state is exited as a result of the firing of an outgoing transition before the completion of the activity, the activity is aborted prior to its completion.

Deferred events

A state may specify a set of event types that may be *deferred* in that state. An event instance that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event queue while another non-deferred message is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.

2.12.4.3 CompositeState

Active state configurations

When dealing with composite and concurrent states, the simple term “current state” can be quite confusing. In a hierarchical state machine more than one state can be active at once. If the state machine is in a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active. Furthermore, since some of the composite states in this hierarchy may be concurrent, the current active “state” is actually represented by a tree of states starting with the single top state at the root down to individual simple states at the leaves. We refer to such a state tree as a *state configuration*.

Except during transition execution, the following invariants always apply to state configurations:

- If a composite state is active and not concurrent, exactly one of its substates is active.
- If the composite state is active and concurrent, all of its substates (regions) are active.

Entering a non-concurrent composite state

Upon entering a composite state, the following cases are differentiated:

- *Default entry:* Graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state. In this case, the default transition is taken. If there is a guard on the transition it must be enabled (true). (A disabled initial transition is an ill-defined execution state and its handling is not defined.) The entry action of the state is executed before the action associated with the initial transition.

- *Explicit entry:* If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate.
- *Shallow history entry:* If the transition terminates on a shallow history pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the *default history state* is entered. This is the substate that is target of the transition originating from the history pseudostate. (If no such transition is specified, the situation is illegal and its handling is not defined.) If the active substate determined by history is a composite state, then it proceeds with its default entry.
- *Deep history entry:* The rule here is the same as for shallow history except that the rule is applied recursively to all levels in the active state configuration below this one.

Entering a concurrent composite state

Whenever a concurrent composite state is entered, each one of its concurrent substates (regions) is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of a fork), these regions are entered explicitly and the others by default.

Exiting non-concurrent state

When exiting from a composite state, the active substate is exited recursively. This means that the exit actions are executed in sequence starting with the innermost active state in the current state configuration.

Exiting a concurrent state

When exiting from a concurrent state, each of its regions is exited. After that, the exit actions of the regions are executed.

Deferred events

An event that is deferred in a composite state is automatically deferred in all directly or transitively nested substates.

2.12.4.4 FinalState

When the final state is active, its containing composite state is *completed*, which means that it satisfies the completion condition. If the containing state is the top state, the entire state machine terminates, implying the termination of the entity associated with the state machine. If the state machine specifies the behavior of a classifier, it implies the “termination” of that instance.

2.12.4.5 SubmachineState

A submachine state is a convenience that does not introduce any additional dynamic semantics. It is semantically equivalent to a composite state and may have entry and exit actions, internal transitions, and activities.

2.12.4.6 Transitions

High-level transitions

Transitions originating from the boundary of composite states are called *high-level* or *group* transitions. If triggered, they result in exiting of all the substates of the composite state executing their exit actions starting with the innermost states in the active state configuration. Note that in terms of execution semantics, a high-level transition does not add specialized semantics, but rather reflects the semantics of exiting a composite state.

Compound transitions

A *compound transition* is a derived semantic concept, represents a "semantically complete" path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states. The transition execution semantics described below, refer to compound transitions.

In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates that define path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). For self-transitions, the same state acts as both the source and the destination set. A (simple) transition connecting two states is therefore a special common case of a compound transition.

The tail of a compound transition may have multiple transitions originating from a set of mutually orthogonal concurrent regions that are joined by a join point.

The head of a compound transition may have multiple transitions originating from a fork pseudostate targeted to a set of mutually orthogonal concurrent regions.

In a compound transition multiple outgoing transitions may emanate from a common *junction* point. In that case, only one of the outgoing transition whose guard is true is taken. If multiple transitions have guards that are true, a transition from this set is chosen. The algorithm for selecting such a transition is not specified. Note that in this case, the guards are evaluated before the compound transition is taken.

In a compound transition where multiple outgoing transitions emanate from a common *choice* point, the outgoing transition whose guard is true *at the time the choice point is reached*, will be taken. If multiple transitions have guards that are true, one transition from this set is chosen. The algorithm for selecting this transition is not specified. If no guards are true after the choice point has been reached, the model is ill formed.

Internal transitions

An internal transition executes without exiting or re-entering the state in which it is defined. This is true even if the state machine is in a nested state within this state.

Completion transitions and completion events

A *completion transition* is a transition without an explicit trigger, although it may have a guard defined. When all transition and entry actions and activities in the currently active state are completed, a *completion event* instance is generated. This event is the implicit trigger for a completion transition. The completion event is dispatched before any other queued events and has no associated parameters. For instance, a completion transition emanating from a concurrent composite state will be taken automatically as soon as all the concurrent regions have reached their final state.

If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions.

Enabled (compound) transitions

A transition is *enabled* if and only if:

- All of its source states are in the active state configuration.
- The trigger of the transition is satisfied by the current event. An event instance *satisfies* a trigger if it matches the event specified by the trigger. In case of signal events, since signals are generalized concepts, a signal event satisfies a signal event associated with the same signal or a generalization of thereof.
- If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice point in which all guard conditions are true (transitions without guards are treated as if their guards are always true).

Since more than one transition may be enabled by the same event instance, being enabled is a necessary but not sufficient condition for the firing of a transition.

Guards

In a simple transition with a guard, the guard is evaluated before the transition is triggered.

In compound transitions involving multiple guards, all guards are evaluated before a transition is triggered, unless there are choice points along one or more of the paths. The order in which the guards are evaluated is not defined.

If there are choice points in a compound transition, only guards that precede the choice point are evaluated according to the above rule. Guards downstream of a choice point are evaluated if and when the choice point is reached (using the same rule as above). In other words, for guard evaluation, a choice point has the same effect as a state.

Guards should not include expressions causing side effects. Models that violate this are considered ill formed.

Transition execution sequence

Every transition, except for internal transitions, causes exiting of a source state, and entering of the target state. These two states, which may be composite, are designated as the *main source* and the *main target* of a transition.

The *least common ancestor* (LCA) state of a transition is the lowest composite state that contains all the explicit source states and explicit target states of the compound transition. In case of junction segments, only the states related to the dynamically selected path are considered explicit targets (bypassed branches are not considered).

If the LCA is not a concurrent state, the main source is a direct substate of the least common ancestor that contains the explicit source states, and the main target is a substate of the least common ancestor that contains the explicit target states. In case where the LCA is a concurrent state, the main source and main target are the concurrent state itself. The reason is that if a concurrent region is exited, it forces exit of the entire concurrent state.

Examples:

1. The common simple case: A transition t between two simple states s1 and s2, in a composite state s.

   Here least common ancestor of t is s, the main source is s1 and the main target is s2.

2. A more esoteric case: An unstructured transition from one region to another.

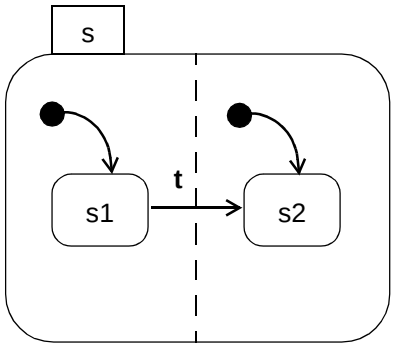


Figure 2-26 Unstructured transition among regions

   Here least common ancestor of t is s, the main source is s and the main target is s, since s is a concurrent state as specified above.

Once a transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source state is properly exited.
- Actions are executed in sequence following their linear order along the segments of the transition: The closer the action to the source state, the earlier it is executed.

- If a choice point is encountered, the guards following that choice point are evaluated dynamically and a path whose guards are true is selected. Entry and exit actions are executed for states entered and exited by the transition into the choice point.
- The main target state is properly entered.

2.12.4.7 StateMachine

Event processing - run-to-completion step

Events are dispatched and processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.

The semantics of event processing is based on the *run-to-completion* assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed.

Run-to-completion may be implemented in various ways. For active classes, it may be realized by an event-loop running in its own concurrent thread, and that reads events from a queue. For passive classes it may be implemented as a monitor.

The processing of a single event by a state machine is known as an *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all actions (but not necessarily activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion* step *is* the passage between two state configurations of the state machine.

The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step.

When an event instance is dispatched, it may result in one or more transitions being enabled for firing. If no transition is enabled and the event is not in the deferred event list of the current state configuration, the event is discarded and the run-to-completion step is completed.

In the presence of concurrent states it is possible to fire multiple transitions as a result of the same event — as many as one transition in each concurrent state in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. Which of the enabled transitions actually fire is determined by the transition selection algorithm described below. The order in which selected transitions fire is not defined.

Each orthogonal region in the active state configuration that is not decomposed into concurrent regions (i.e., “bottom-level” region) can fire at most one transition as a result of the current event. When all orthogonal regions have finished executing the transition, the current event instance is fully consumed, and the run-to-completion step is completed.

During a transition, a number of actions may be executed. If these actions are synchronous, then the transition step is not completed until the invoked objects complete their own run-to-completion steps.

An event instance can arrive at a state machine that is blocked in the middle of a run-to-completion step from some other object within the same thread, in a circular fashion. This event instance can be treated by orthogonal components of the state machine that are not frozen along transitions at that time.

Run-to-completion and concurrency

It is possible to define state machine semantics by allowing the run-to-completion steps to be applied concurrently to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration.

In case of active objects, where each object has its own thread of execution, it is very important to clearly distinguish the notion of run to completion from the concept of thread pre-emption. Namely, run-to-completion event handling is performed by a thread that, in principle, *can* be pre-empted and its execution suspended in favor of another thread executing on the same processing node. (This is determined by the scheduling policy of the underlying thread environment — no assumptions are made about this policy.) When the suspended thread is assigned processor time again, it resumes its event processing from the point of pre-emption and, eventually, completes its event processing.

Conflicting transitions

It was already noted that it is possible for more than one transition to be enabled within a state machine. If that happens, then such transitions may be in *conflict* with each other. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step.

Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed.

An internal transition in a state conflicts only with transitions that cause an exit from that state.

Firing priorities

In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an *implicit* priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.

The priority of a transition is defined based on its source state. The priority of joined transitions is based on the priority of the transition with the most transitively nested source state.

In general, if t1 is a transition whose source state is s1, and t2 has source s2, then:

- If s1 is a direct or transitively nested substate of s2, then t1 has higher priority than t2.
- If s1 and s2 are not in the same state configuration, then there is no priority difference between t1 and t2.

Transition selection algorithm

The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions:

- All transitions in the set are enabled.
- There are no conflicting transitions within the set.
- There is no transition outside the set that has higher priority than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out).

This can be easily implemented by a greedy selection algorithm, with a straightforward traversal of the active state configuration. States in the active state configuration are traversed starting with the innermost nested simple states and working outwards toward the top state. For each state at a given level, all originating transitions are evaluated to determine if they are enabled. This traversal guarantees that the priority principle is not violated. The only non-trivial issue is resolving transition conflicts across orthogonal states on all levels. This is resolved by terminating the search in each orthogonal state once a transition inside any one of its components is fired.

2.12.4.8 Synch States

Synch states provide a means of synchronizing the execution of two concurrent regions. Specifically, a synch state has incoming transitions from a fork (or forks) in one region, the *source* region, and outgoing transitions to a join (or joins) in another, the *target* region. These forks and joins are called *synchronization* forks and joins. The

synch state itself is contained by the least common ancestor of the two regions being synchronized. The synchronized regions do not need to be siblings in state decomposition, but they must have a common ancestor state.

When the source region reaches a synchronization fork, the target states of that fork become active, including the synch state. Activation of the synch state is an indication that the source region has completed some activity. This region can continue performing its remaining activities in parallel. When the target region reaches the corresponding synchronization join, it is prevented from continuing unless all the states leading into the synchronization join are active, including the synch states.

A synch state may have multiple incoming and outgoing transitions, used for multiple synchronization points in each region. Alternatively, it may have single incoming and outgoing transitions where the incoming transition is fired multiple times before the outgoing one is fired. To support these applications, synch states keep count of the difference between the number of times their incoming and outgoing transitions are fired. When an incoming transition is fired, the count is incremented by one, unless its value is equal to the value defined in the *bound* attribute. In that case, the count is not incremented. When an outgoing transition is fired, the count is decremented by one. An outgoing transition may fire only if the count is greater than zero, which prevents the count from becoming negative. The count is automatically set to zero when its container state is exited.

The bound attribute is for limiting the number of times outgoing transitions fire from a synch state. For a state, to realize the equivalent of a binary semaphore, the bound should be set to one. In this case multiple incoming transitions may fire before the outgoing transition does, whereupon the outgoing transition can only fire once.

2.12.4.9 StubStates

Stub states are pseudostates signifying either entry points to or exit points from a submachine. Since a submachine is encapsulated and represented as a submachine state, multi-level (“deep”) transitions may logically connect states in separate state machines. This is facilitated by stub state, representing real states in a referenced machine to or form transitions in the referencing machine are incoming/outgoing. stub states are therefore can only be defined within a submachine state, and are the only potential subvertices of a submachine state.

2.12.5 Notes

2.12.5.1 Protocol State Machines

One application area of state machines is in specifying object protocols, also known as object life cycles. A 'protocol state machine' for a class defines the order (i.e. sequence) in which the operations of that Class can be invoked. The behavior of each of these operations is defined by an associated method, rather than through actions on transitions.

A transition in a protocol state machine has as its trigger a call event that references an operation of the class, and no actions. Such a transition indicates that if the call event occurs when an object of the class is in the source state of the transition and the guard on the transition is true, then the method associated with the operation of the call event will be executed (if one exists), and the object will enter the target state. Semantically, the invocation of the method does not lead to a new call event being raised.

If a call event arrives when the state machine is not in an appropriate state to handle the event, the event is discarded, conform the general RTC semantics. Strictly speaking, from the caller's point of view this means that the call is completed. If instead the semantics are required that the caller should 'hang' (potentially infinitely) if the receiver's state machine is not able to process the call event immediately, then the event must be deferred explicitly. This can be done for all call events in a protocol state machine by deferring them at a superstate level.

In any practical application, a protocol state machine is made up exclusively of 'protocol' transitions, and the entry and exit actions of its states are empty (i.e. no action specifications exist other than for the methods). However, formally it is not prohibited to mix this kind of transition with transitions with explicit actions (as it does not seem worth the effort to prohibit this, and there may be some applications that might benefit from 'mixing').

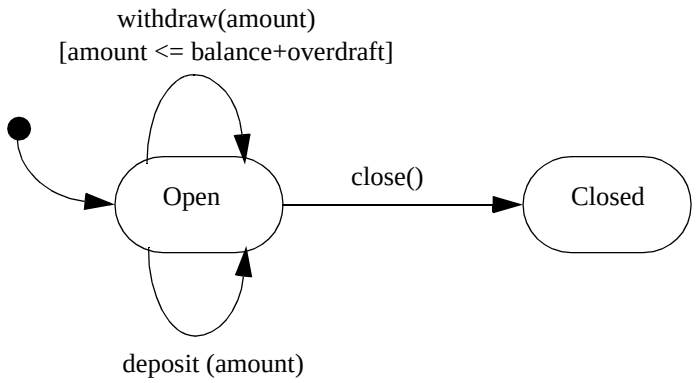


Figure 2-27 Example of a Protocol State Machine for a Class ‘Account’.

2.12.5.2 Example: Modeling Class Behavior

In the software that is implemented as a result of a state modeling design, the state machine may or may not be actually visible in the (generated or hand-crafted) code. The state machine will not be visible if there is some kind of run-time system that supports state machine behavior. In the more general case, however, the software code will contain specific statements that implement the state machine behavior.

A C++ example is shown below:

```
class bankAccount {
    private:
        int balance;
    public:
        void deposit (amount) {
            if (balance > 0)
                balance = balance + amount' // no change
            else
                balance = balance + amount - 1; // transaction fee
            }
    void withdrawal (amount) {
            if (balance>0)
                balance = balance - amount;
            }
    }
```

In the above example, the class has an abstract state manifested by the balance attribute, controlling the behavior of the class. This is modeled by the state machine in Figure 2-28.

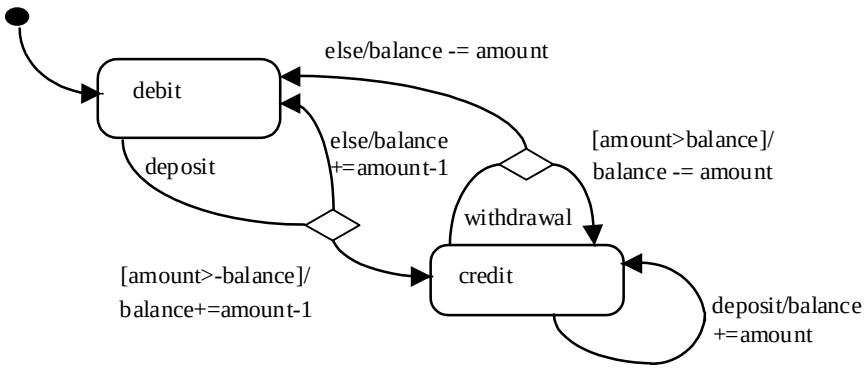


Figure 2-28 State Machine for Modeling Class Behavior

2.12.5.3 Example: State machine refinement

Note – The following discussion provides some potentially useful heuristics on how state machines can be refined. These techniques are all based on practical experience. However, readers are reminded that this topic is still the subject of research, and that it is likely that other approaches may be defined either now or in the future.

Since state machines describe behaviors of generalizable elements, primarily classes, state machine refinement is used capture the relationships between the corresponding state machines. State machines use refinement in three different mappings, specified by the mapping attribute of the refinement meta-class. The mappings are refinement, substitution, and deletion.

To illustrate state machine refinement, consider the following example where one state machine attached to a class denoted 'Supplier,' is refined by another state machine attached to a class denoted as 'Client.'

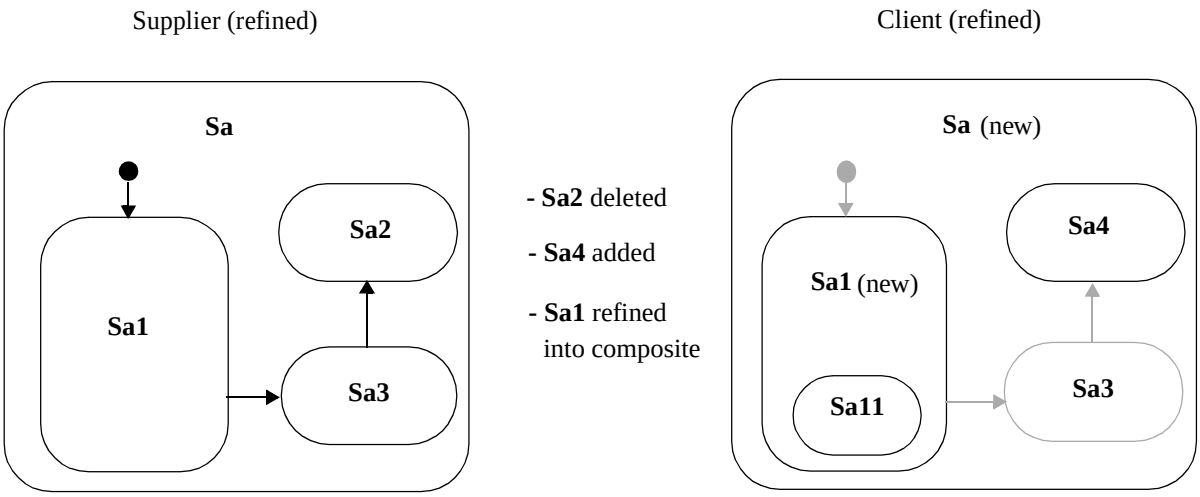


Figure 2-29 State Machine Refinement Example

In the example above, the client state (Sa(new)) in the subclass substitutes the simple substate (Sa1) by a composite substate (Sa1(new)). This new composite substate has a component substate (Sa11). Furthermore, the new version of Sa1 deletes the substate Sa2 and also adds a new substate Sa4. Substate Sa3 is inherited and is therefore common to both versions of Sa. For clarity, we have used a gray shading to identify components that have been inherited from the original. (This is for illustration purposes and is not intended as a notational recommendation.)

It is important to note that state machine refinement as defined here does not specify or favor any specific policy of state machine refinement. Instead, it simply provides a flexible mechanism that allows subtyping, (behavioral compatibility), inheritance (implementation reuse), or general refinement policies.

We provide a brief discussion of potentially useful policies that can be implemented with the state machine refinement mechanism.

Subtyping

The refinement policy for subtyping is based on the rationale that the subtype preserves the pre/post condition relationships of applying events/operations on the type, as

specified by the state machine. The pre/post conditions are realized by the states, and the relationships are realized by the transitions. Preserving pre/post conditions guarantee the substitutability principle.

States and transitions are only added, not deleted. Refinement is interpreted as follows:

- A refined state has the same outgoing transitions, but may add others, and a different set of incoming transitions. It may have a bigger set of substates, and it may change its concurrency property from false to true.
- A refined transition may go to a new target state which is a substate of the state specified in the base class. This comes to guarantee the post condition specified by the base class.
- A refined guard has the same guard condition, but may add disjunctions. This guarantees that pre-conditions are weakened rather than strengthened.
- A refined procedure contains the same actions (in the same sequence), but may have additional actions. The added actions should not hinder the invariant represented by the target state of the transition.

Strict Inheritance

The rationale behind this policy is to encourage reuse of implementation rather than preserving behavior. Since most implementation environment utilize strict inheritance (i.e. features can be replaced or added, but not deleted), the inheritance policy follows this line by disabling refinements which may lead to non-strict inheritance once the state machine is implemented.

States and transitions can be added. Refinement is interpreted as follows:

- A refined state has some of the same incoming transitions (i.e., drop some, add some) but a greater or bigger set of outgoing transitions. It may have more substates, and may change its concurrency attribute.
- A refined transition may go to a new target state but should have the same source.
- A refined guard may have a different guard condition.
- A refined procedure contains some of the same actions (in the same sequence), and may have additional actions.

General Refinement

In this most general case, states and transitions can be added and deleted (i.e., ‘null’ refinements). Refinement is interpreted without constraints (i.e., there are no formal requirements on the properties and relationships of the refined state machine element and the refining element):

- A refined state may have different outgoing and incoming transitions (i.e., drop all, add some).
- A refined transition may leave from a different source and go to a new target state.
- A refined guard has may have a different guard condition.

- A refined procedure need not contain the same actions (or it may change their sequence), and may have additional actions.

The refinement of the composite state in the example above is an illustration of general refinement.

It should be noted that if a type has multiple supertype relationships in the structural model, then the default state machine for the type consists of all the state machines of its supertypes as orthogonal state machine regions. This may be explicitly overridden through refinement if required.

2.12.5.4 Comparison to classical statecharts

The major difference between classical (Harel) statecharts and object state machines result from the external context of the state machine. Object state machines, such as ROOMcharts, primarily come to represent behavior of a type. Classical statechart specify behaviors of processes. The following list of differences result from the above rationale:

- Events carry parameters, rather than being primitive signals.
- Call events (operation triggers) are supported to model behaviors of types.
- Event conjunction is not supported, and the semantics is given in respect to a single event dispatch, to better match the type context as opposed to a general system context.
- Classical statecharts have an elaborated set of predefined actions, conditions and events which are not mandated by object state machines, such as entered(s), exited(s), true(condition), tr!(c) (make true), fs!(c).
- Operations are not broadcast but can be directed to an object-set.
- The notion of activities (processes) does not exist in object state machines. Therefore all predefined actions and events that deal with activities are not supported, as well as the relationships between states and activities.
- Transition compositions are constrained for practical reasons. In classical statecharts any composition of pseudostates, simple transitions, guards and labels is allowed.
- Object state machine support the notion of synchronous communication between state machines.
- Actions on transitions are executed in their given order.
- Classical statecharts do not support dynamic choice points.
- Classical statecharts are based on the zero-time assumption, meaning transitions take zero time to execute. The whole system execution is based on synchronous steps where each step produces new events that will be processed at the next step. In object-oriented state machines, these assumptions are relaxed and replaced with these of software execution model, based on threads of execution and that execution of actions may take time. antics.

2.13 Activity Graphs

2.13.1 Overview

The Activity Graphs package defines an extended view of the State Machine package. State machines and activity graphs are both essentially state transition systems, and share many metamodel elements. This section describes the concepts in the State Machine package that are specific to activity graphs. It should be noted that the activity graphs extension has few semantics of its own. It should be understood in the context of the State Machine package, including its dependencies on the Foundation package and the Common Behavior package.

An activity graph is a special case of a state machine that is used to model processes involving one or more classifiers. Its primary focus is on the sequence and conditions for the actions that are taken, rather than on which classifiers perform those actions. Most of the states in such a graph are action states that represent atomic actions (i.e., states that invoke actions and then wait for their completion). Transitions into action states are triggered by events, which can be

- the completion of a previous action state (completion events),
- the availability of an object in a certain state,
- the occurrence of a signal, or
- the satisfaction of some condition.

By defining a small set of additional subtypes to the basic state machine concepts, the well-formedness of activity graphs can be defined formally, and subsequently mapped to the dynamic semantics of state machines. In addition, the activity specific subtypes eliminate ambiguities that might otherwise arise in the interchange of activity graphs between tools.

2.13.2 Abstract Syntax

The abstract syntax for activity graphs is expressed in graphic notation in Figure 2-30 on page 2-171.

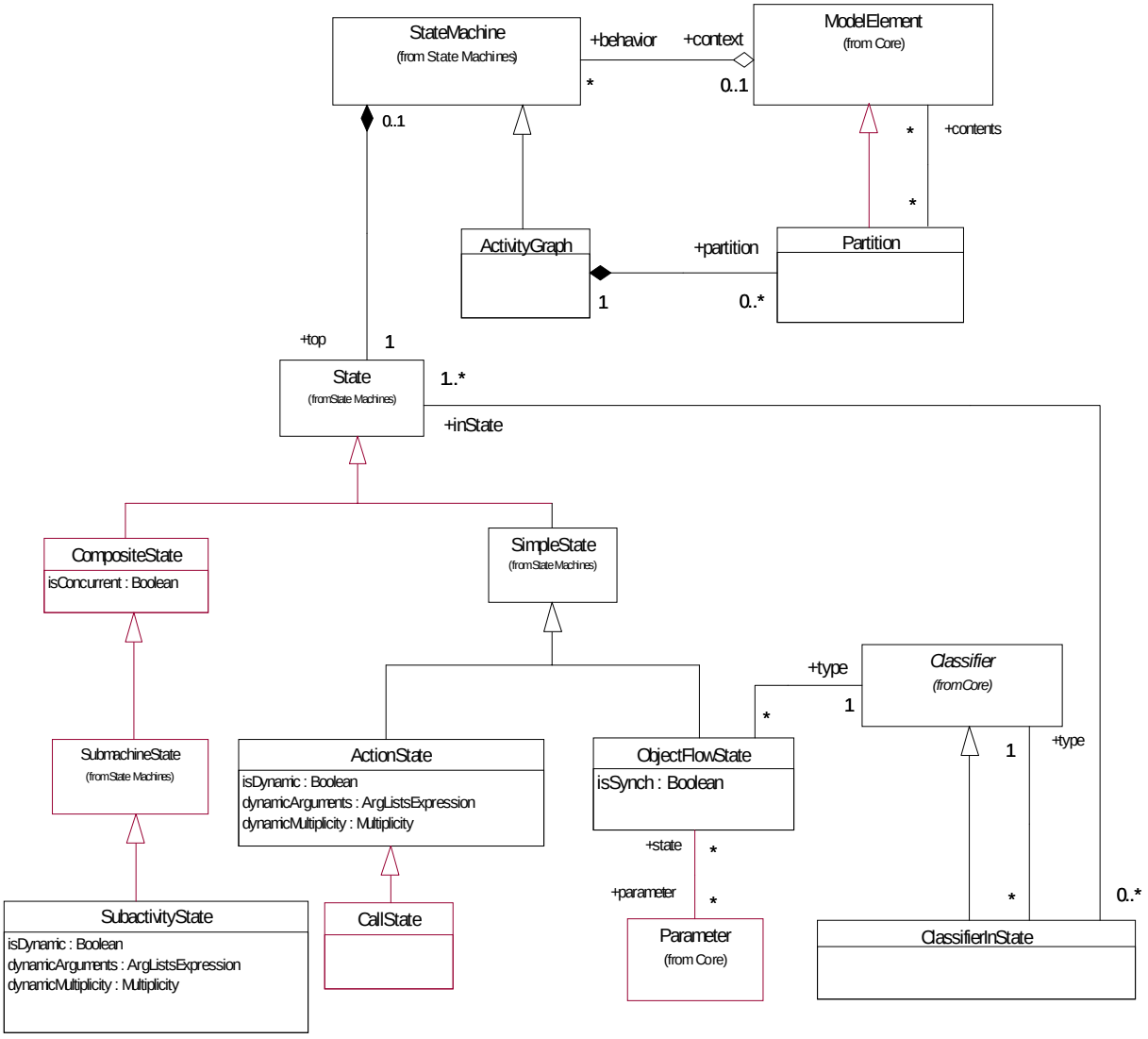


Figure 2-30 Activity Graphs

2.13.2.1 *ActionState*

An action state represents the execution of an atomic action, typically the invocation of an operation.

An action state is a simple state with an entry action whose only exit transition is triggered by the implicit event of completing the execution of the entry action. The state therefore corresponds to the execution of the entry action itself and the outgoing transition is activated as soon as the action has completed its execution.

An ActionState may perform more than one action as part of its entry action. An action state may not have an exit action, do activity, or internal transitions.

Attributes

dynamicArguments	An ArgListsExpression that determines at runtime the number of parallel executions of the actions of the state. The value must be a set of lists of objects, each list serving as arguments for one execution. This attribute is ignored if the isDynamic attribute is false.
dynamicMultiplicity	A Multiplicity limiting the number of parallel executions of the actions of state. This attribute is ignored if the isDynamic attribute is false.
isDynamic	A boolean value specifying whether the state's actions might be executed concurrently. It is used in conjunction with the dynamicArguments attribute.

Associations

entry	(Inherited from State) Specifies the invoked actions.

2.13.2.2 ActivityGraph

An activity graph is a special case of a state machine that defines a computational process in terms of the control-flow and object-flow among its constituent activities. It does not extend the semantics of state machines in a major way but it does define shorthand forms that are convenient for modeling control-flow and object-flow in organizational processes.

The primary purpose of activity graphs is to describe the states of an activity or process involving one or more classifiers. Activity graphs can be attached to packages, classifiers (including use cases) and behavioral features. As in any state machine, if an outgoing transition is not explicitly triggered by an event then it is implicitly triggered by the completion of the contained actions. A subactivity state represents a nested activity that has some duration and internally consists of a set of actions or more subactivities. That is, a subactivity state is a “hierarchical action” with an embedded activity subgraph that ultimately resolves to individual actions.

Junctions, forks, joins, and synchs may be included to model decisions and concurrent activity.

Activity graphs include the concept of Partitions to organize states according to various criteria, such as the real-world organization responsible for their performance.

Activity graphing can be applied to organizational modeling for business process engineering and workflow modeling. In this context, events often originate from inside the system, such as the finishing of a task, but also from outside the system, such as a customer call. Activity graphs can also be applied to system modeling to specify the dynamics of operations and system level processes when a full interaction model is not needed.

Associations

partition	A set of Partitions each of which contains some of the model elements of the model.

2.13.2.3 CallState

A call state is an action state that calls a single operation. It is useful in object flow modeling to reduce notational ambiguity over which action is taking input or providing output.

2.13.2.4 ClassifierInState

A classifier-in-state characterizes instances of a given classifier that are in a particular state or states. In an activity graph, it may be used to specify input and/or output to an action through an object flow state.

ClassifierInState is a child of Classifier and may be used in static structural models and collaborations (e.g., it can be used to show associations that are only relevant when objects of a class are in a given state).

Associations

type	Designates a classifier for the ClassifierInState to characterize the instances of.
inState	Designates a state that characterizes instances of the classifier of the ClassifierInState. The state must be a valid state of the corresponding classifier. This may have multiple states when referring to an object in orthogonal states.

2.13.2.5 ObjectFlowState

An object flow state defines an object flow between actions in an activity graph. An instance of a particular classifier, possibly in a particular state, is available when an object flow state is occupied.

The generation of an object by an action in an action state may be modeled by an object flow state that is triggered by the completion of the action state. The use of the object in a subsequent action state may be modeled by connecting the output transition of the object flow state as an input transition to the action state. Generally each action places the object in a different state that is modeled as a distinct object flow state.

Attributes

isSynch	A boolean value indicating whether an object flow state is used as a synch state.

Associations

type	Designates a classifier that specifies the classifier of the object. It may be a classifier-in-state to specify the state and classifier of the object.
parameter	Designates parameters which provide the object as output or take it as input.

Stereotypes

«signalflow»	Class	Signalflow is a stereotype of ObjectFlowState with a Signal as its type.

2.13.2.6 Partition

A partition is a mechanism for dividing the states of an activity graph into groups. Partitions often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the states of an activity graph. It should be noted that Partitions do not impact the dynamic semantics of the model but they help to allocate properties and actions for various purposes.

Associations

contents	Specifies the states that belong to the partition. They need not constitute a nested region.

2.13.2.7 SubactivityState

A subactivity state represents the execution of a non-atomic sequence of steps that has some duration (i.e., internally it consists of a set of actions and possibly waiting for events). That is, a subactivity state is a “hierarchical action,” where an associated subactivity graph is executed.

A subactivity state is a submachine state that executes a nested activity graph. When an input transition to the subactivity state is triggered, execution begins with the nested activity graph. The outgoing transitions of a subactivity state are enabled when the final state of the nested activity graph is reached (i.e., when it completes its execution), or when the trigger events occur on the transitions.

The semantics of a subactivity state are equivalent to the model obtained by statically substituting the contents of the nested graph as a composite state replacing the subactivity state.

Attributes

dynamicArguments	An ArgListsExpression that determines the number of parallel executions of the submachines of the state. The value must be a set of lists of objects, each list serving as arguments for one execution. This attribute is ignored if the isDynamic attribute is false.
dynamicMultiplicity	A Multiplicity limiting the number of parallel executions of the actions of state. This attribute is ignored if the isDynamic attribute is false.
isDynamic	A boolean value specifying whether the state's subactivity might be executed concurrently. It is used in conjunction with the dynamicArguments attribute.

Associations

submachine	(Inherited from SubmachineState) This designates an activity graph that is conceptually nested within the subactivity state. The subactivity state is conceptually equivalent to a composite state whose contents are the states of the nested activity graph. The nested activity graph must have an initial state and a final state.

2.13.2.8 Transition

Transition is inherited from state machines.

Tagged Values

usage	Association	Usage applies only to transitions leading into or out of an object flow state. It has a value of uses or modifies. A value of uses indicates that the action of the state at the other end of the transition from the object flow state uses but does not modify the object represented by the object flow state. A value of modifies indicates that the action of the state at the other end of the transition from the object flow state modifies and may use the object represented by the object flow state.

2.13.3 Well-formedness Rules

ActivityGraph

[1] An ActivityGraph specifies the dynamics of

(i) a Package, **or**

(ii) a Classifier (including UseCase), **or**

(iii) a BehavioralFeature.

```
(self.context.oclIsTypeOf(Package)    xor
 self.context.oclIsKindOf(Classifier) xor
 self.context.oclIsKindOf(BehavioralFeature))
```

2.13.3.1 ActionState

[1] An action state has a non-empty entry action.

self.entry->size > 0

[2] An action state does not have an internal transition, exit action, or a do activity.

self.internalTransition->size = 0 **and** self.exit->size = 0 **and** self.doActivity->size = 0

[3] Transitions originating from an action state have no trigger event.

self.outgoing->forAll(t | t.trigger->size = 0)

2.13.3.2 CallState

[1] The entry action of a call state is a single call action.

let a : Set = self.entry.action.allNestedActions in
a->size() = 1
and a->asSequence()->first().oclIsKindOf(CallOperationAction)

2.13.3.3 ClassifierInState

[1] Classifiers-in-state have no namespace contents.

self.allContents->size = 0

2.13.3.4 ObjectFlowState

[1] Parameters of an object flow state must have a type and direction compatible with classifier or classifier-in-state of the object flow state.

```
let ofstype : Classifier =
    (if self.type.IsKindOf(ClassifierInState)
        then self.type.type else self.type);
self.parameter->forAll(parameter |
    parameter.type = ofstype
    or (parameter.kind = #in
            and ofstype.allSupertypes->includes(type))
    or ((parameter.kind = #out or parameter.kind = #return)
            and type.allSupertypes->includes(ofstype))
    or (parameter.kind = #inout
            and (   ofstype.allSupertypes->includes(type)
            or type.allSupertypes->includes(ofstype))))
```

[2] Downstream states have entry actions that accept input conforming to the type of the classifier or classifier-in-state. The entry actions use the input parameters of the object flow state. Valid downstream states are calculated by traversing outgoing transitions transitively, skipping pseudo states, and entering and exiting subactivity states, looking for regular states. If the object flow state has no parameters, then the target of downstream actions must conform to the type of the classifier or classifier-in-state.

```
self.allNextLeafStates.size > 0 and
    self.allNextLeafStates->forAll(s | self.isInputAction(s.entry))
```

[3] Upstream states have entry actions that provide output or return values conforming to the type of the classifier or classifier-in-state. The entry actions use the output or return parameters of the object flow state. Valid upstream states are calculated by traversing incoming transitions transitively, skipping pseudo states, entering and exiting subactivity states, looking for regular states.

```
self.allPreviousLeafStates.size > 0 and
    self.allPreviousLeafStates->forAll(s |
                                        self.isOutputAction(s.entry))
```

Additional operations

[1] The operation `allNextLeafStates` results in the set of states immediately downstream of the object flow state that have the next actions that will be executed.

[2] The operation `allPreviousLeafStates` results in the set of states immediately upstream of the object flow state that have the next actions that were last executed.

[3] The operation `isInputAction` takes a procedure as input and results in a boolean telling whether it has an argument compatible with the object flow state.

[4] The operation `isOutputAction` takes a procedure as input and results in a boolean telling whether it has a result compatible with the object flow state.

2.13.3.5 PseudoState

[1] In activity graphs, transitions incoming to (and outgoing from) join and fork pseudostates have as sources (targets) any state vertex. That is, joins and forks are syntactically not restricted to be used in combination with composite states, as is the case in state machines.

```
self.stateMachine.oclIsTypeOf(ActivityGraph) implies
    ((self.kind = #join or self.kind = #fork) implies
        (self.incoming->forAll(t | t.source.oclIsKindOf(State) or
          source.oclIsTypeOf(PseudoState)) and
        (self.outgoing->forAll(t | t.source.oclIsKindOf(State) or
          source.oclIsTypeOf(PseudoState)))))
```

[2] All of the paths leaving a fork must eventually merge in a subsequent join in the model. Furthermore, multiple layers of forks and joins must be well nested, with the exception of forks and joins leading to or from synch state. Therefore the concurrency structure of an activity graph is in fact equally restrictive as that of an ordinary state machine, even though the composite states need not be explicit.

2.13.3.6 SubactivityState

[1] A subactivity state is a submachine state that is linked to an activity graph.

```
self.submachine.oclIsKindOf(ActivityGraph)
```

2.13.4 Detailed Semantics

2.13.4.1 ActivityGraph

The dynamic semantics of activity graphs can be expressed in terms of state machines. This means that the process structure of activities formally must be equivalent to orthogonal regions (in composite states). That is, transitions crossing between parallel paths (or threads) are not allowed, except for transitions used with synch states. As such, an activity specification that contains ‘unconstrained parallelism’ as is used in general activity graphs is considered ‘incomplete’ in terms of UML.

All events that are not relevant in a state must be deferred so they are consumed when they become relevant. This is facilitated by the general deferral mechanism of state machines.

2.13.4.2 ActionState

As soon as the incoming transition of an ActionState is triggered, its entry action starts executing. Once the entry action has finished executing, the action is considered completed. When the action is complete then the outgoing transition is enabled.

The isDynamic attribute of an action state determines whether multiple invocations of state might be executed concurrently, depending on runtime information. This means that the normal activities of an action state, namely its actions, may execute multiple times in parallel. If isDynamic is true, then the dynamicArguments attribute is evaluated at the time the state is entered. The size of the resulting set determines the number of parallel executions of the state. Each element of the set is a list, which is used as arguments for an execution. These arguments can be referred to within actions (e.g. by “object[i]” denoting the *i*th object in a list). If the isDynamic attribute is false, dynamicArguments is ignored. If the dynamicArguments expression evaluates to the empty set, then the state behaves as if it had no actions. It is an error if the dynamicArguments expression evaluates to a set with fewer or more elements than the number allowed by the dynamicMultiplicity attribute. The behavior is not defined in this case.

Dynamic states may be nested. In this case, you can't access the outer set of arguments in the inner nesting. If this should be necessary, arguments can be passed explicitly from the outer to the inner dynamic state.

2.13.4.3 ObjectFlowState

The activation of an object flow state signifies that an instance of the associated classifier is available, perhaps in a specified state (i.e., a state change has occurred as a result of a previous operation). This may enable a subsequent action state that requires the instance as input. As with all states in activity graphs, if the object flow state leads into a join pseudostate, then the object flow state remains activated until the other predecessors of the join have completed.

Unless there is an explicit ‘fork’ that creates orthogonal object states, only one of an object flow state’s outgoing transitions will fire as determined by the guards of the transitions. The invocation of the action state may result in a state change of the object, resulting in a new object flow state.

An object flow state may specify the parameter of an operation that provides the flowing object as output, and the parameter of an operation that takes the flowing object as input. The operations must be called in actions of states immediately preceding and succeeding the object flow state, respectively, although pseudostates, final states, synch states, and stub states may be interposed between the object flow state and the acting state. For example, an object flow state may transition to a subactivity state, which means at runtime the object is passed as input to the first state after the initial state of the subactivity graph. If no parameter is specified to take the flowing object as input, then it is used as an action target instead. Call actions are particularly suited to be used in conjunction with this technique because they invoke exactly one operation.

Object flow states may be used as synch states, indicated by the isSynch attribute being set to true. In this case, outgoing transitions can fire only if an object has arrived on the incoming transitions. Instead of a count, the state keeps a list of objects that arrive on the incoming transitions. These objects are pulled from the list as outgoing transitions

are fired. No outgoing transitions can fire if the list is empty. All objects in the list conform to the classifier and state specified by the object flow state. The list is not bounded as the count may be in synch states.

For applications requiring that actions or activities bring about an event as their result, use an object flow state with a signal as a classifier. This means the action or activity must return an instance of a signal. For multiple resulting events, transition the action or activity to a fork, and target the fork transitions at multiple object flow states.

2.13.4.4 SubactivityState

The isDynamic, dynamicArguments, and dynamicMultiplicity attributes of a subactivity state have a similar meaning to the same attributes of action states. They provide for executing the submachine of the subactivity state multiple times in parallel. See semantics of ActionState.

2.13.4.5 Transition

In activity graphs, transitions outgoing from forks may have guards. This means the region initiated by a fork transition might not start, and therefore not be required to complete at the corresponding join. Forks and joins must be well-nested in the model to use this feature (see rule #2 for PseudoState in Activity Graphs). The following mapping shows the state machine meaning for such an activity graph:

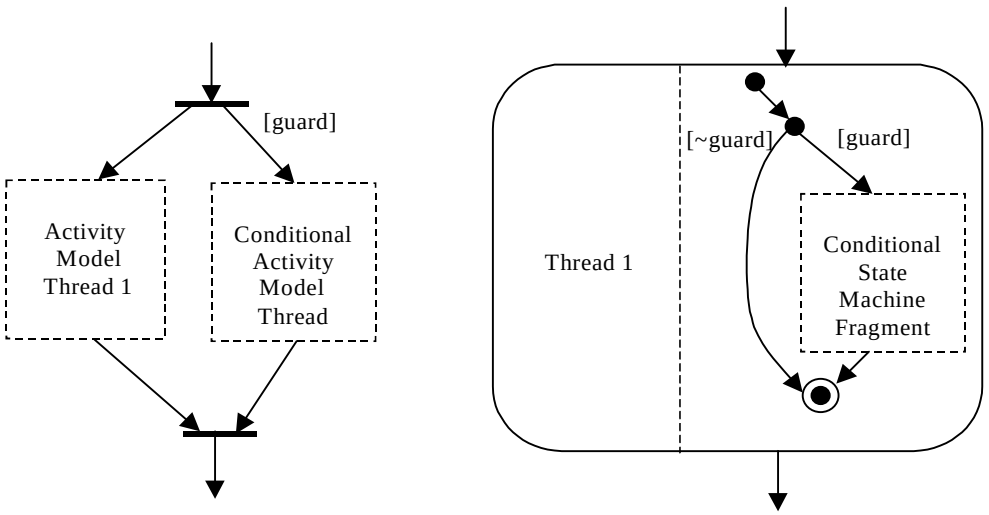


Figure 2-31 State Machine - Activity Graph

If a conditional region synchronizes with another region using a synch state, and the condition fails, then these synch states have their counts set to infinity to prevent other regions from deadlocking.

2.13.5 Notes

Object flow states in activity graphs are a specialization of the general dataflow aspect of process models. Object-flow activity graphs extend the semantics of standard dataflow relationships in three areas:

1. The operations in action states in activity graphs are operations of classifiers or types (e.g., ‘Trade’ or ‘OrderEntryClerk’). They are not hierarchical ‘functions’ operating on a dataflow.
2. The ‘contents’ of object flow states are typed. They are not unstructured data definitions as in data stores.
3. The state of the object flowing as input and output between operations may be defined explicitly. The event of the availability of an object in a specific state may form a trigger for the operation that requires the object as input. Object flow states are not necessarily stateless as are data stores.

2.14 Actions

See “Part 5 - Actions” on page 2-199.

Part 4 - General Mechanisms

This section defines the mechanisms of general applicability to models. This version of UML contains one general mechanisms package, Model Management. The Model Management package specifies how model elements are organized into models, packages, subsystems, and UML profiles.

2.15 Model Management

2.15.1 Overview

The Model Management package is dependent on the Foundation package. It defines Model, Package, and Subsystem, which all serve as grouping units for other ModelElements.

Models are used to capture different views of a physical system. Packages are used within a Model to group ModelElements. A Subsystem represents a behavioral unit in the physical system. UML Profiles are packages dedicated to group UML extensions.

In this section it is necessary to clearly distinguish between the *physical system* being modeled; that is, the subject of the model and the model element that represent the physical system in the model. For this reason, we consistently use the term *physical system* when we want to indicate the former, and the term (top-level) subsystem when we want to indicate the latter. An example of a physical system is a credit card service, which includes software, hardware, and wetware (people). The UML model for this physical system might consist of a top-level subsystem called CreditCardService, which is decomposed into subsystems for Authorization, Credit, and Billing. An

analogy with the construction of houses would be that the house would correspond to the physical system, while a blueprint would correspond to a model, and an element used in a blueprint would correspond to a model element.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Model Management package.

2.15.2 Abstract Syntax

The abstract syntax for the Model Management package is expressed in graphic notation in Figure 2-32.

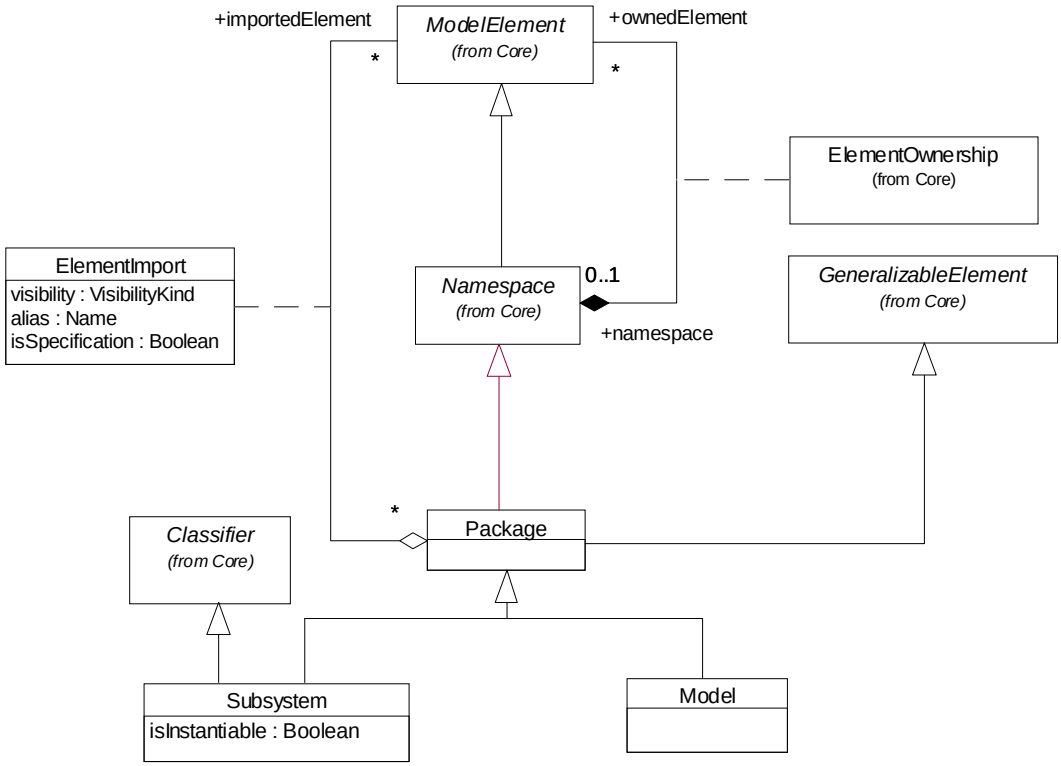


Figure 2-32 Model Management

2.15.2.1 Dependency (as extended)

Dependencies have specific extensions for modeling UML profiles.

Stereotypes

«modelLibrary»	Class	This dependency means that the supplier package is being used as a model library associated with a profile. The client is a package that is stereotyped as a profile and the supplier is a non-profile package that contains shared model elements, such as classes and data types.
«appliedProfile»	Class	This dependency is used to indicate which profiles are applicable to a package. Typically, the client is an ordinary package or a model (but could be any other kind of package), while the supplier is a profile package. This means that the profile applies transitively to the model elements contained in the client package, including the client package itself.

2.15.2.2 ElementImport

An element import defines the visibility and alias of a model element included in the namespace within a package, as a result of the package importing another package.

In the metamodel, an ElementImport reifies the relationship between a Package and an imported ModelElement. It allows redefinition of the name and the visibility for the imported ModelElement; that is, the ModelElement may be given another name (an alias) and/or a new visibility to be used within the importing Package. The default is no alias (the original name will be used) and private visibility relative to the importing Package.

Attributes

alias	The alias defines a local name of the imported ModelElement, to be used within the Package.
isSpecification	Specifies whether the ownedElement is part of the specification for the containing namespace (in cases where specification is distinguished from the realization). Otherwise the ownedElement is part of the realization. In cases in which the distinction is not made, the value is false by default.
visibility	An imported ModelElement is either public, protected, or private relative to the importing Package.

2.15.2.3 Model

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

In the metamodel, Model is a subclass of Package. It contains a containment hierarchy of ModelElements that together describe the physical system. A Model also contains a set of ModelElements that represents the environment of the system, typically Actors, together with their interrelationships, such as Dependencies, Generalizations, and Constraints.

Different Models can be defined for the same physical system, where each model represents a view of the physical system defined by its purpose and abstraction level (for example, an analysis model, a design model, an implementation model). Typically different models are complementary and defined from the perspectives (viewpoints) of different system stakeholders. For example, a use-case model may be defined from the viewpoint of a business analyst stakeholder. Each Model is a complete description of the physical system. When Models are nested, the container Model represents the comprehensive view of the physical system given by the different views defined by the contained Models.

Stereotypes

«systemModel»	Class	A systemModel is a stereotyped model that contains a collection of models of the same physical system. A systemModel also contains all relationships and constraints between model elements contained in different models.
«metamodel»	Class	A metamodel is a stereotyped model denoting that the model is an abstraction of another model; that is, it is a model of a model. Hence, if M2 is a model of the model M1, then M2 is a metamodel of M1. It follows then that classes in M1 are instances of metaclasses in M2. The stereotype can be recursively applied, as in the case of a 4-layer metamodel architecture.

2.15.2.4 Package

A package is a grouping of model elements.

In the metamodel, Package is a subclass of Namespace and GeneralizableElement. A Package contains ModelElements like Packages, Classifiers, and Associations. A Package may also contain Constraints and Dependencies between ModelElements of the Package.

Each ModelElement of a Package has a visibility relative to the Package stating if the ModelElement is available to ModelElements in other Packages with a Permission («access» or «import») or Generalization relationship to the Package. An «access» or «import» Permission from one Package to another allows public ModelElements in the target Package to be referenced by ModelElements in the source Package. They differ in that all public ModelElements in imported Packages are added to the Namespace within the importing Package, whereas the Namespace within an accessing Package is not affected at all. The ModelElements available in a Package are those in the contents of the Namespace within the Package, which consists of owned and imported ModelElements, together with public ModelElements in accessed Packages.

Associations

importedElement	The namespace defined by a package is extended by model elements in other, imported packages.

Stereotypes

«facade»	Class	A facade is a stereotyped package that contains references to model elements owned by another package. It is used to provide a ‘public view’ of some of the contents of a package. A facade does not contain any model elements of its own.
«framework»	Class	A framework is a stereotyped package that contains model elements that specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns, or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks.
«modelLibrary»	Class	A model library is a stereotyped package that contains model elements that are intended to be reused by other packages. A model library differs from a profile in that a model library does not extend the metamodel using stereotypes and tagged definitions. A model library is analogous to a class library in some programming languages.
«profile»	Class	A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose using extension mechanisms, such as stereotypes, tagged definitions, and constraints. A profile may also specify model libraries on which it depends and the metamodel subset that it extends. (The latter is specified via an *applicableSubset* tag definition.)
«stub»	Class	A stub is a stereotyped package that represents only the public parts of another package.
«topLevel»	Class	TopLevel is a stereotyped package that denotes the highest level package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces “see” outwards. A topLevel subsystem is the top of a subsystem containment hierarchy; that is, it is the model element that represents the boundary of the entire physical system being modeled.

Tag Definitions

{applicableSubset}	This tag definition, which only applies to profile packages, lists the metaelements that are used by the associated profile. The value associated with this tag definition is a set of strings, where each string represents the name of an applicable metaelement. Note that the use of applicable subset does not necessarily exclude the use of any metaelements, but clearly identifies which ones are referenced from the associated profile. Further note that the tag definition applies only to the immediately associated profile. If a profile combines several other profiles using import or generalizations, the applicable subset only applies to the immediately associated profile. The absence of an applicable subset tag definition means that the whole UML metamodel is applicable.

2.15.2.5 Subsystem

A subsystem is a grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem are partitioned into specification and realization elements, where the former, together with the operations of the subsystem, are realized by the latter.

In the metamodel, Subsystem is a subclass of both Package and Classifier. As such it may have a set of Features, which are constrained to be Operations and Receptions, and Associations.

The contents of a Subsystem are divided into two subsets: specification elements and realization elements. The former subset provides, together with the Operations of the Subsystem, a specification of the behavior contained in the Subsystem, while the ModelElements in the latter subset jointly provide a realization of the specification. Any kind of ModelElement can be a specification element or a realization element. The relationships between the specification elements and the realization elements can be defined in different ways (for example, with Collaborations or «realize» dependencies).

Attributes

isInstantiable	States whether a Subsystem is instantiable or not. If false, the Subsystem represents a unique part of the physical system; otherwise, there may be several system parts with the same definition.

2.15.3 Well-formedness Rules

The following well-formedness rules apply to the Model Management package.

2.15.3.1 ElementImport

No extra well-formedness rules.

2.15.3.2 Model

No extra well-formedness rules.

2.15.3.3 Package

[1] No imported element (excluding Association) may have the same name or alias as any element owned by the Package or one of its supertypes.

```
self.allImportedElements->reject( re |
    re.oclIsKindOf(Association) )->forAll( re |
        (re.elementImport.alias <> '' implies
            not (self.allContents - self.allImportedElements)->
                reject( ve |
                    ve.oclIsKindOf (Association) )->exists ( ve |
                        ve.name = re.elementImport.alias))
        and
        (re.elementImport.alias = '' implies
            not (self.allContents - self.allImportedElements)->
                reject ( ve |
                    ve.oclIsKindOf (Association) )->exists ( ve |
                        ve.name = re.name) ) )
```

[2] Imported elements (excluding Association) may not have the same name or alias.

```
self.allImportedElements->reject( re |
    not re.oclIsKindOf (Association) )->forAll( r1, r2 |
        (r1.elementImport.alias <> '' and
            r2.elementImport.alias <> '' and
            r1.elementImport.alias = r2.elementImport.alias
            implies r1 = r2)
        and
        (r1.elementImport.alias = '' and
            r2.elementImport.alias = '' and
            r1.name = r2.name implies r1 = r2)
        and
        (r1.elementImport.alias <> '' and
            r2.elementImport.alias = '' implies
                r1.elementImport.alias <> r2.name))
```

[3] No imported element (Association) may have the same name or alias combined with the same set of associated Classifiers as any Association owned by the Package or one of its supertypes.

```
self.allImportedElements->select( re |
    re.oclIsKindOf(Association) )->forAll( re |
        (re.elementImport.alias <> '' implies
            not (self.allContents - self.allImportedElements)->
            select( ve |
                ve.oclIsKindOf(Association) )->exists(
                ve : Association |
                    ve.name = re.elementImport.alias
                    and
                    ve.connection->size = re.connection->size and
                    Sequence {1..re.connection->size}->forAll( i |
                        re.connection->at(i).participant =
                        ve.connection->at(i).participant ) ) )
        and
        (re.elementImport.alias = '' implies
            not (self.allContents - self.allImportedElements)->
            select( ve |
                not ve.oclIsKindOf(Association) )->exists( ve :
                Association |
                    ve.name = re.name
                    and
                    ve.connection->size = re.connection->size and
                    Sequence {1..re.connection->size}->forAll( i |
                        re.connection->at(i).participant =
                        ve.connection->at(i).participant ) ) ) )
```

[4] Imported elements (Association) may not have the same name or alias combined with the same set of associated Classifiers.

```
self.allImportedElements->select ( re |
    re.oclIsKindOf (Association) )->forAll ( r1, r2 : Association |
        (r1.connection->size = r2.connection->size and
        Sequence {1..r1.connection->size}->forAll ( i |
            r1.connection->at (i).participant =
                r2.connection->at (i).participant  and
            r1.elementImport.alias <> '' and
            r2.elementImport.alias <> '' and
            r1.elementImport.alias = r2.elementImport.alias
            implies r1 = r2))
        and
        (r1.connection->size = r2.connection->size and
          Sequence {1..r1.connection->size}->forAll ( i |
            r1.connection->at (i).participant =
                r2.connection->at (i).participant and
            r1.elementImport.alias = '' and
            r2.elementImport.alias = '' and
            r1.name = r2.name
            implies r1 = r2))
        and
        (r1.connection->size = r2.connection->size and
        Sequence {1..r1.connection->size}->forAll ( i |
            r1.connection->at (i).participant =
                r2.connection->at (i).participant and
            r1.elementImport.alias <> '' and
            r2.elementImport.alias = ''
            implies r1.elementImport.alias <> r2.name)))
```

[5] A Package may only own or reference Packages, Classifiers, Associations, Generalizations, Dependencies, Comments, Constraints, Collaborations, StateMachines, Stereotypes, and TaggedValues.

```
self.contents->forAll ( c |
c.oclIsKindOf(Package) or
c.oclIsKindOf(Classifier) or
c.oclIsKindOf(Association or
c.oclIsKindOf(Generalization) or
c.oclIsKindOf(Dependency) or
c.oclIsKindOf(Comment) or
c.oclIsKindOf(Constraint) or
c.oclIsKindOf(Collaboration or
c.oclIsKindOf(StateMachine) or
c.oclIsKindOf(TaggedValue) or
c.oclIsKindOf(Stereotype))
```

Additional Operations

[1] The operation contents results in a Set containing the ModelElements owned by or imported by the Package.

```
contents : Set(ModelElement)
contents = self.ownedElement->union(self.importedElement)
```

[2] The operation allImportedElements results in a Set containing the ModelElements imported by the Package or one of its parents.

```
allImportedElements : Set(ModelElement)
allImportedElements = self.importedElement->union(
self.parent.oclAsType(Package).allImportedElements->select( re |
    re.elementImport.visibility = #public or
    re.elementImport.visibility = #protected))
```

[3] The operation allContents results in a Set containing the ModelElements owned by or imported by the Package or one of its ancestors.

```
allContents : Set(ModelElement);
allContents = self.contents->union(
    self.parent.allContents->select(e |
        e.elementOwnership.visibility = #public or
        e.elementOwnership.visibility = #protected))
```

2.15.3.4 Profile

[1] The base classes of all stereotypes in a profile must be part of the applicable subset of this profile.

```
self.applicableSubset->
    includesAll(self.stereotypes->collect(baseClass))
```

[2] A profile package can only contain tag definitions, stereotypes, constraints and data types.

```
self.contents->forAll(e |
    e.oclIsKindOf(Stereotype) or
    e.oclIsKindOf(Constraint) or
    e.oclIsKindOf(TagDefinition) or
    e.oclIsKindOf (DataType))
```

2.15.3.5 Subsystem

[1] For each Operation in an Interface offered by a Subsystem, the Subsystem itself or at least one contained specification element must have a matching Operation.

```
self.specification.allOperations->forAll(interOp |
    self.allOperations->union
        (self.allSpecificationElements->select(specEl|
            specEl.oclIsKindOf(Classifier))->forAll(c|
                c.allOperations))->exists
                    ( op | op.hasSameSignature(interOp) ) )
```

[2] For each Reception in an Interface offered by a Subsystem, the Subsystem itself or at least one contained specification element must have a matching Reception.

```
let allReceptions : set(Reception) = self.allFeatures->select(f |
    f.oclIsKindOf(Reception)) in
self.specification.allReceptions->forAll(interRec |
    self.allReceptions->union
        (self.allSpecificationElements->select(specEl|
            specEl.oclIsKindOf(Classifier))->forAll(c|
                c.allReceptions))->exists
                    ( rec | rec.hasSameSignature(interRec) ) )
```

[3] The Features of a Subsystem may only be Operations or Receptions.

```
self.feature->forAll(f |        f.oclIsKindOf(Operation) or
                                f.oclIsKindOf(Reception))
```

[4] A Subsystem may only own or reference Packages, Classes, DataTypes, Interfaces, UseCases, Actors, Subsystems, Signals, Associations, Generalizations, Dependencies, Constraints, Collaborations, StateMachines, and Stereotypes.

```
self.contents->forAll ( c |
        c.oclIsKindOf(Package) or
        c.oclIsKindOf(Class) or
        c.oclIsKindOf(DataType) or
        c.oclIsKindOf(Interface) or
        c.oclIsKindOf(UseCase) or
        c.oclIsKindOf(Actor) or
        c.oclIsKindOf(Subsystem) or
        c.oclIsKindOf(Signal) or
        c.oclIsKindOf(Association) or
        c.oclIsKindOf(Generalization) or
        c.oclIsKindOf(Dependency) or
        c.oclIsKindOf(Constraint) or
        c.oclIsKindOf(Collaboration) or
        c.oclIsKindOf(StateMachine) or
        c.oclIsKindOf(Stereotype) )
```

Additional Operations

[1] The operation allSpecificationElements results in a Set containing the Model Elements specifying the behavior of the Subsystem

```
allSpecificationElements : Set(ModelElement)
allSpecificationElements = self.allContents->select(c | c.elementOwnership.isSpecification)
```

[2] The operation contents results in a Set containing the ModelElements owned by or imported by the Subsystem.

```
contents : Set(ModelElement)
contents = self.ownedElement->union(self.importedElement)
```

2.15.4 Semantics

2.15.4.1 Package

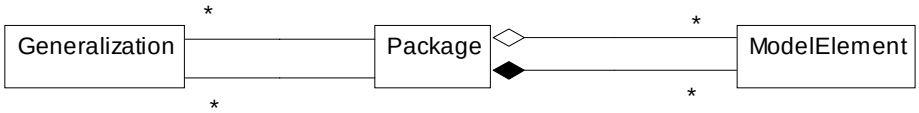


Figure 2-33 Package illustration - shows Package and its environment in the metamodel by flattening the inheritance hierarchy.

The purpose of the *package* construct is to provide a general grouping mechanism. A package cannot be instantiated, thus it has no runtime semantics. In fact, its only semantics is to define a namespace for its contents. The package construct can be used for organizing elements for any purpose; the criteria to use for grouping elements together into one package are not defined within UML.

A package owns a set of model elements, with the implication that if the package is removed from the model, so are the elements owned by the package. Elements with names, such as classifiers, that are owned by the same package must have unique names within the package, although elements in different packages may have the same name.

There may be relationships between elements contained in the same package, and between an element in one package and an element in a surrounding package at any level. In other words, elements “see” all the way out through nested levels of packages. (Note that a package with the stereotype «topLevel» defines the outer limit of this outward visibility.) Elements in peer packages, however, are encapsulated and are not *a priori* visible to each other. The same goes for elements in contained packages; that is, packages do not see “inwards.” There are two ways of making elements in other packages available: by importing/accessing these other packages, and by defining generalizations to them.

An *import* dependency (a Permission dependency with the stereotype «import») from one package to another means that the first package imports all the elements with sufficient visibility in the second package. Imported elements are not owned by the package; however, they may be used in associations, generalizations, attribute types, and other relationships owned by the package. A package defines the *visibility* of its contained elements to be private, protected, or public. Private elements are not available at all outside the containing package. Protected elements are available only to packages with generalizations to the package owning the elements, and public elements are available also to importing and accessing packages. Note that the visibility mechanism does not restrict the availability of an element to peer elements in the same package.

When an element is imported by a package it extends the namespace of that package. It is possible to give an imported element an alias to avoid name conflicts with the names of the other elements in the namespace, including other imported elements. The alias will then be the name of that element in the namespace; the element will not appear under both the alias and its original name. An imported element is by default

private to the importing package. It may, however, be given a more permissive visibility relative to the importing package; that is, the local visibility may be defined as protected or public.

A package with an import dependency to another package imports all the public contents of the namespace defined by the supplier package, including elements of packages imported by the supplier package that are given public visibility in the supplier.

The *access* dependency (a Permission dependency with the stereotype «access») is similar to the import dependency in that it makes elements in the supplier package available to the client package. However, in this case no elements in the supplier package are included in the namespace of the client. They are simply referred to by their full pathname when referenced in the accessing package. Clearly, they are not visible to packages in turn accessing or importing this package.

A package can have *generalizations* to other packages. This means that the public and protected elements owned or imported by a package are also available to its children, and can be used in the same way as any element owned or imported by the children themselves. Elements made available to another package by the use of a generalization are referred to by the same name in the child as they are in the parent. Moreover, they have the same visibility in the child as they have in the parent package. Relationships between the ancestor package and other model elements are also inherited by the child package.

A package can be used to define a *framework*, specifying a reusable architecture for all or part of a system. Frameworks may include reusable classes, patterns or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks.

2.15.4.2 Profile

A profile stereotype of Package contains one or more related extensions of standard UML semantics (refer to Section 2.6, “Extension Mechanisms,” on page 2-73). These are normally intended to customize UML for a particular domain or purpose. Profiles can contain stereotypes, tag definitions, and constraints. They can also contain data types that are used by tag definitions for informally declaring the types of the values that can be associated with tag definitions.

In addition, a profile package can specify a related model library and identify a subset of the UML metamodel that is applicable for the profile. In principle, profiles merely refine the standard semantics of UML by adding further constraints and interpretations that capture domain-specific semantics and modeling patterns. They do not add any new fundamental concepts.

Relationships between profiles

A profile package can have the usual relationships with other packages such as generalization, import, and access. These have the usual semantics. They are useful to profile designers who may want to import elements from one profile into another, or to combine two or more profiles. However, care should be taken to combine these in a

consistent way. For example, extensions from different profiles may be incompatible and their respective constraints may contradict each other. In this revision of UML, no formal mechanisms are defined to verify that a combination of two or more profiles is mutually consistent.

Profile generalization

Generalization of profiles is a relationship between a profile and a more general profile. The more specific profile must be fully consistent with the more general profile; that is, it has all the same tag definitions, stereotypes, and constraints, and may add further refinements, which must not contradict its parent. Note that the subset of UML defined as applicable by a profile is *not* inherited by specializing profiles, whereas relationships to model libraries are.

Access and import dependencies between profiles

Profiles can have access and import dependencies with the usual semantics. This allows elements in one profile to access or use elements in the related profiles. An applied profiles dependency will allow a client package to use all stereotypes and tag definitions accessible by the supplier package. As in all other types of packages, a profile can own other profiles with standard semantics of ownership and accessibility.

Applying a profile to a package

A UML model can be based on a number of different UML profiles. The applicable profiles are identified by specially stereotyped «appliedProfile» dependencies from the UML model package to the appropriate profile packages. This declaration enables the UML model to access the stereotypes and tag definitions of these profiles.

2.15.4.3 Subsystem

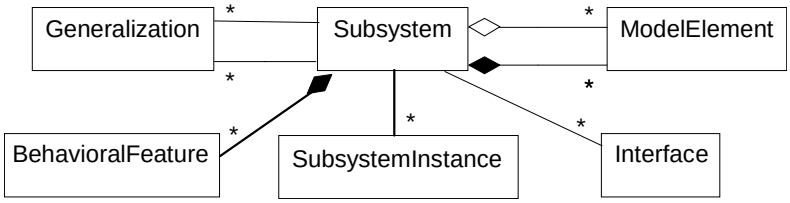


Figure 2-34 Subsystem illustration - shows Subsystem and its environment in the metamodel by flattening the inheritance hierarchy.

The purpose of the *subsystem* construct is to provide a grouping mechanism for specifying a behavioral unit of a physical system. Apart from defining a namespace for its contents, a subsystem serves as a specification unit for the behavior of its contained model elements.

The contents of a subsystem are defined in the same way as for a package, thus it consists of owned elements and imported elements, with unique names or aliases within the subsystem. The contents of a subsystem are divided into two subsets: 1) *specification elements* and 2) *realization elements*. The specification elements, together with the

operations and receptions of the subsystem, are used for giving an abstract specification of the behavior offered by the realization elements. The collection of realization elements model the interior of the behavioral unit of the physical system. Consequently, subsystems contained in the realization part represent subordinate subsystems; that is, subsystems at the level below in the containment hierarchy, hence owned by the current subsystem.

The *specification* of a subsystem thus consists of the specification elements together with the subsystem's features (operations and receptions). It specifies the behavior performed jointly by instances of classifiers in the realization subset, without revealing anything about the contents of this subset. The specification is typically made in terms of model elements such as use cases and/or operations, although other kinds of model elements like classes, interfaces, constraints, relationships between model elements, state machines may also be used. Use cases are used to specify complete sequences performed by the subsystem; that is, by instances of its contained classifiers interacting with its surroundings. Operations are suitable to represent simpler subsystem services that are used independently of each other; that is, not in any particular order.

A subsystem has no behavior of its own. All behavior defined in the specification of the subsystem is jointly offered by the elements in the realization subset of the contents. In general, since subsystems are classifiers, they can appear anywhere a classifier is expected. It follows that, since the subsystem itself has no behavior of its own, the requirements posed on the subsystem in the context where it occurs are fulfilled by the realization of the subsystem.

The correspondence between the specification and the realization of a subsystem can be specified in several ways, including collaborations and «realize» dependencies. A collaboration specifies how instances of the realization elements cooperate to jointly perform the behavior specified by a use case, an operation, etc. in the subsystem specification; that is, how the higher level of abstraction is transformed into the lower level of abstraction. A stimulus received by an instance of a use case (higher level of abstraction) corresponds to an instance conforming to one of the classifier roles in the collaboration receiving that stimulus (lower level of abstraction). This instance communicates with other instances conforming to other classifier roles in the collaboration, and together they perform the behavior specified by the use case. All stimuli that can be received and sent by instances of the use cases are also received and sent by the conforming instances, although at a lower level of abstraction. Similarly, application of an operation of the subsystem actually means that a stimulus is sent to a contained instance that performs a method.

There are two ways of communicating with a subsystem, either by sending stimuli to the subsystem itself to be re-directed to the proper recipient inside the subsystem, or by sending stimuli directly to the recipient inside the subsystem. In the first case, an association is defined with the subsystem itself to enable stimuli sending. (In the abstract syntax, this is handled by a single subsystem instance being connected by links corresponding to this association, receiving stimuli sent to the subsystem, and re-directing them to instances within the subsystem instance. Hence the subsystem instance is the "runtime representative" of the subsystem. Note that this subsystem

instance still does not perform any of the behavior specified in the subsystem specification.) How stimuli sent to the subsystem are re-directed to internal instances is not defined but left as a semantic variation point.

Communicating with a subsystem by sending stimuli directly to instances within the subsystem requires that the classifiers of these instances are available within the sender's namespace so that they can be connected by associations. This can be achieved by import or access permissions. *Importing* and *accessing* subsystems is done in the same way as with packages, using the *visibility* property to define whether elements are public, protected, or private to the subsystem. Both the specification part and the realization part of a subsystem may include imported elements.

A subsystem can have *generalizations* to other subsystems. This means that the public and protected elements in the contents of a subsystem as well as operations and receptions are also available to its heirs. Furthermore, relationships between an ancestor subsystem and other model elements are inherited by specializing subsystems. In a concrete (non-abstract) subsystem all elements in the specification, including elements from ancestors, are completely realized by cooperating realization elements, as specified with, for example, a set of collaborations. This may not be true for abstract subsystems.

A subsystem may offer a set of *interfaces*. This implies that for each operation defined in an interface, the subsystem offering the interface must have a matching operation, either as a feature of the subsystem itself or of a specification element. The relationship between interface and subsystem is not necessarily one-to-one. Interfaces of a subsystem are usually contained in the same namespace as the subsystem itself, but may also be contained in the specification of the subsystem. In the latter case, elements using these interfaces must have an import or access relationship with the subsystem to gain access to the interfaces.

In cases when the physical system has several parts with the same definition, the subsystem is specified to be *instantiable*. The parts are then instances of this subsystem. Note, however, that all behavior specified for the subsystem is still performed by instances contained in the subsystem instances, not by the subsystem instances themselves.

2.15.4.4 Model

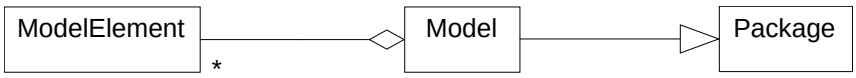


Figure 2-35 Model illustration - shows Model and its environment in the metamodel by flattening the inheritance hierarchy.

A *model* is a description of a physical system with a certain purpose, such as to describe logical or behavioral aspects of the physical system to a certain category of readers. Examples of different kinds of models are 'use case,' 'analysis,' 'design,' and 'implementation,' or 'computational,' 'engineering,' and 'organizational' each representing one view of a physical system.

Thus, a model is an abstraction of a physical system. It specifies the physical system from a certain vantage point (or viewpoint); that is, for a certain category of stakeholders (for example, designers, users, or orderers of the system), and at a certain level of abstraction, both given by the purpose of the model. A model is complete in the sense that it covers the whole physical system, although only those aspects relevant to its purpose; that is, within the given level of abstraction and vantage point, are represented in the model. Furthermore, it describes the physical system only once; that is, there is no overlapping; no part of the physical system is captured more than once in a model.

A model consists of a containment hierarchy where the top-most package or subsystem represents the boundary of the physical system. This package/subsystem may be given the stereotype «topLevel» to emphasize its role within the model. It is possible to have more than one containment hierarchy within a model; that is, the model contains a set of top-most packages/subsystems each being the root of a containment hierarchy. In this case there is no single package/subsystem that represents the physical system boundary.

The model may also contain model elements describing relevant parts of the system's environment. The environment is typically modeled by actors and their interfaces. As these are external to the physical system, they reside outside the package/subsystem hierarchy. They may be collected in a separate package, or owned directly by the model. These model elements and the model elements representing the physical system may be associated with each other.

A model may be a *specialization* of another model via a generalization relationship. This implies that all public and protected elements in the ancestor are also available in the specialized model under the same name and interrelated as in the ancestor.

A model may *import* or *access* another model. The semantics is the same as for packages. However, some of the actors of the supplier model may be internal to the client. This is the case, for example, when the imported model represents a lower layer of the physical system than the client model represents. Then some of the actors of the lower layer model represent the upper layer. The conformance requirement is that there must be classifiers in the client whose instances may play the roles of such actors.

The contents of a model is the transitive closure of its owned model elements, like packages, classifiers, and relationships, together with inherited and imported elements.

There may be relationships between model elements in different models, such as refinement and trace. A *trace*; that is, an abstraction dependency with the stereotype «trace» indicates that the connected (sets of) model elements represent the same concept. Trace is used for tracing requirements between models, or tracing the impact on other models of a change to a model element in one model. Thus traces are usually non-directional dependencies. Relationships between model elements in different models have no impact on the model elements' meaning in their containing models because of the self-containment of models. Note, though, that even if inter-model relationships do not express any semantics in relation to the models, they may have semantics in relation to the reader or in deriving model elements as part of the overall development process.

Models may be nested (for example, several models of the same physical system may be collected in a model with the stereotype «systemModel»). The models contained in the «systemModel» all describe the physical system from different viewpoints, the viewpoints not necessarily disjoint. The «systemModel» also contains all inter-model relationships. A «systemModel» constitutes a comprehensive specification of the physical system.

A large physical system may be composed by a set of subordinate physical systems together making up the large physical system. In this case each subordinate physical system is described by its own set of models collected in a separate «systemModel». This is an alternative to having each part of the physical system defined as a subsystem.

2.15.5 Notes

In UML, there are three different ways to model a group of elements contained in another element; by using a package, a subsystem, or a class. Some pragmatics on their use include:

- Packages are used when nothing but a plain grouping of elements is required.
- Subsystems provide grouping suitable for top-down development, since the requirements on the behavior of their contents can be expressed before the realization of this behavior is defined. Furthermore, from a bottom-up perspective, the specification of a subsystem may also be seen as a provider of “high level APIs” of the subsystem.
- Classes are used when the container itself should have instances, so that it is possible to define composite objects.

As Subsystem and Model, both are Packages. In the metamodel, all three constructs can be combined arbitrarily to organize a containment hierarchy. For example, a Subsystem may be defined using a set of Models, in which case these Models are contained in the Subsystem. Another example is a set of components defined by Subsystems, collected in a Package defining a reuse library.

It is a tool issue to decide how many of the imported elements must be explicitly referenced by the importing package; that is, how many ElementImport links to actually implement. For example, if all elements have the default visibility (private) and their original names in the importing package, the information can be retrieved directly from the imported package.

If a tool does not support the separation of specification and realization elements for Subsystem, then the value of the isSpecification attribute for ElementOwnership should be false by default. See the Core package, where ElementOwnership is defined, for details.

The issue of how to represent the runtime presence of a Subsystem has been solved by introducing SubsystemInstance, even for a non-instantiable Subsystem. An alternative, less intuitive, solution would be to have the metaclass Subsystem inherit the metaclass Instance, thus getting the desired characteristics.

Because this is a logical model of the UML, distribution or sharing of models between tools is not described. It is expected that tools will manage presentation elements, in particular diagrams, that are attached to model elements.

Part 5 - Actions

This section defines the syntax and semantics of executable actions and procedures, including their run-time semantics. This part contains one package, Actions. The Actions package defines the various kinds of actions that may compose a procedure.

2.16 Action Package

The action package structure is shown in Figure 2-36.

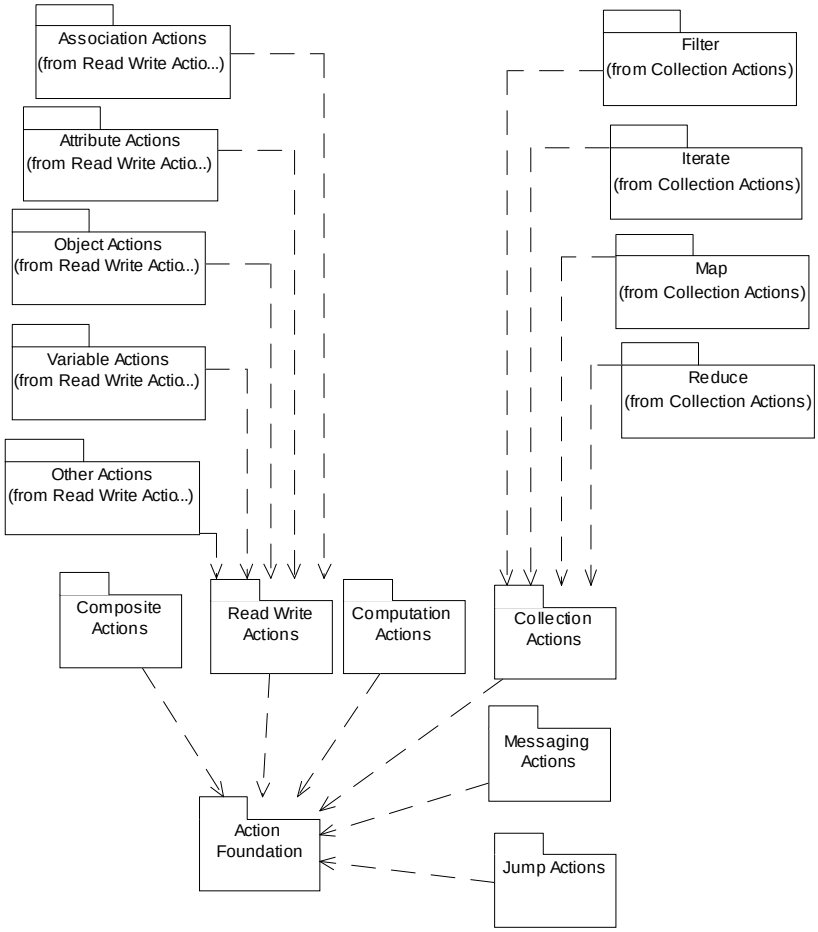


Figure 2-36 Action package

See Section 2.17.3, “The Actions,” on page 2-204 for the description of packages.

2.17 Actions Overview

The section provides an overview of the actions that will be used by modelers.

2.17.1 Action Metamodel

The classes defined in this package play the same role as the classes in the remainder of the UML metamodel. Just as the classes Class and Attribute provide a definition for the meaning of these concepts and the relationships between them, so a class CreateObjectAction in the action metamodel provides a precise definition for the concept of creating an object. Figure 2-37 shows a portion of the action metamodel that deals with creating, destroying and reclassifying objects. It shows several subclasses of the general class Action that are related to other classes in the UML metamodel to form a whole that a modeler can use to build complete, executable specifications.

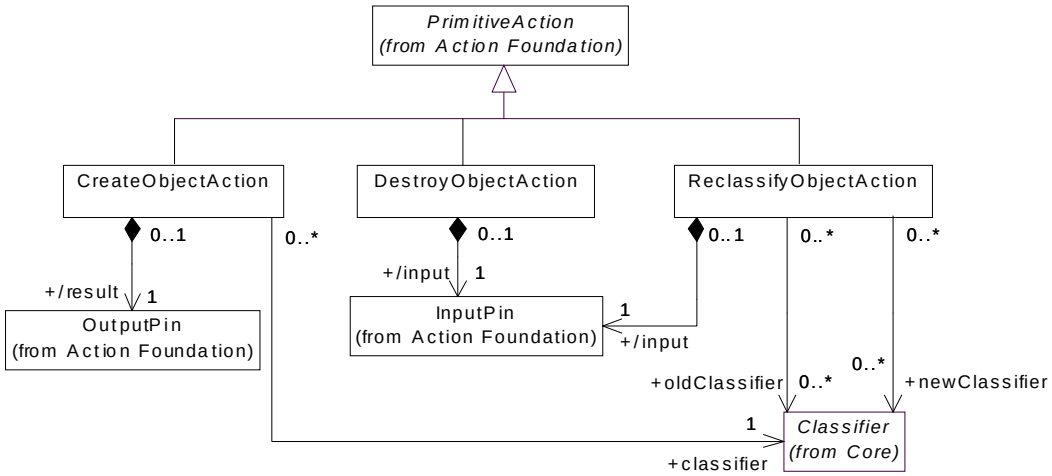


Figure 2-37 A fragment of the metamodel for actions

One role of the action metamodel specifically is to define all the actions that a modeler can use. Figure 2-37 shows the actions CreateObjectAction, DestroyObjectAction, and ReclassifyObjectAction, each of which is defined in detail in the action package. The metamodel further asserts that a CreateObjectAction requires the specification of a classifier, the one to be created, and that the action also produces a single result, OutputPin. This class captures the concept of the result of the CreateObjectAction, which can then be manipulated by other actions. The action metamodel then, together with its supporting class descriptions and well-formedness rules, declares what actions exist and how they relate to other concepts required to support actions.

A user model uses instances of classes from the metamodel. The name “customer” describes an instance of Class. A user model also contains anonymous instances of actions. A modeler can create an instance of a metamodel class, such as “customer” as

an instance of “Class,” using an interactive graphical tool or other means. Similarly, a complier can create instances of actions from a user model expressed in a surface programming language. Figure 2-37 shows an instance diagram for a statement such as “Customer := new Customer” in a developer model. Execution of the statement creates an instance of class Customer and binds the new instance to a variable called “customer.” To designate the result of the action, the developer model attaches an instance of OutputPin to an instance of the CreateObjectAction. An instance of DataFlow connects the users of a value (InputPins) to the producer of a value (OutputPin) within the user model.

In summary, the metaclasses in the Action Package play the same role and have the same properties as classes in other packages comprising UML.

2.17.2 Design Principles and Rationale

The section outlines the design principles that animate the specification, thus providing the design rationale.

2.17.2.1 Interface to Other UML Packages

This package defines the interface to other UML packages in terms of a *procedure*. A procedure is a group of actions caused to execute as a unit. Examples include the body of a method of a class, a group of transition actions, and so on. This package relies on the rest of UML to define the semantics for state machines and other uses of procedures, and therefore to define when a procedure executes. The action semantics defines the behavior of the actions in the procedure and exactly what data is accessible to the procedure once triggered.

2.17.2.2 Undefined Semantics

When this package leaves semantics “undefined,” this means that it leaves the semantics to be defined by other packages in UML, or if UML does not define it, to the implementation. There are several cases where semantics may be left undefined by this package:

- There is no general agreed-upon meaning.
- The meaning is ambiguous in the part of UML that should define it.
- The action or execution foundation is not able to support the desired semantics yet.

An example of the first case is that no semantics is defined for using a create-object action to instantiate an abstract class. An example of second is using a write-attribute action on an ordered attribute without specifying an insertion point. An example of the third is a target scope of “classifier” for attributes and association ends. These cases of undefined semantics are described by well-formedness rules applying to the user model, and the execution rules applying to runtime execution.

The fact that no semantics is given for these situations does not mean that users cannot define their own. Even though this package sometimes uses the value-laden term “ill formed” for these models, it is a purely technical term and should not be taken to mean

this specification excludes semantics from ever being defined for these cases. For example, some users might want to instantiate abstract classes for purposes of testing the root classes of their model. Or some may want the lack of insertion point in write-attribute action to mean that the value is inserted at the end. Finally, some users may even want to extend the action semantic foundation to support the “classifier” target scope. These users could agree on semantics among themselves that covers their needs and does not conflict with this specification. That is perfectly consistent with the action semantics and UML in general.

2.17.2.3 Specification and Software Structure

At the very simplest, actions need access to data, they need to transform and test data, and actions may require sequencing. A simple, sequential model could be adequate relative to a sequential computing environment, but it is not adequate today because it is unreasonably costly to map to today’s distributed computing environments. Consequently, the specification language has to include concepts of distributed, concurrent execution. This specification therefore allows for several (logical) threads of control executing at once and synchronization mechanisms to ensure that procedures execute in a specified order. Semantics based on concurrent execution can then be mapped easily into a distributed implementation.

On the other hand, the fact that a specification language allows for concurrently executing objects does not necessarily imply a distributed software structure. Some implementations may group together objects into a single task and execute sequentially—so long as the behavior of the implementation is the same as the specification.

The implication is that the action semantics do not require the specification of software components, such as tasking structures, or of different forms of transfer of control, such as remote procedure calls vs. messages. Indeed the specification language need not actually specify class structure: the data and behavior of a “class” in the specification may not be rendered as a class in the implementation at all.

In short, the modeler can define concurrent, distributed abstract behavior using actions, but not software structure.

2.17.2.4 Mappings

There are potentially many ways of implementing the same specification, and any implementation that preserves the information content and behavior of the specification is acceptable. Because the implementation can have a different structure from that of the specification, there is a *mapping* between the specification and its implementation. A one-to-one mapping would implement each class as a class and each state machine directly. But the mapping need not be one-to-one: an implementation may not use classes, or it might choose a different set of classes from the original specification.

The mapping may be carried out by hand by overlaying physical models of computers and tasks for implementation purposes, or the mapping could be carried out automatically. This specification neither provides the overlays, nor does it provide for code generation explicitly, but the specification makes both approaches possible.

2.17.2.5 Primitives

The action semantics defines simple, primitive constructs. These primitive actions are defined in such a way as to enable the maximum range of mappings. Specifically, we define the primitive actions so that they either carry out a computation or access object memory, and never both. This approach enables clean mappings to a physical model, even those with data organizations different from that suggested by the specification. In addition, any re-organization of the data structure will leave the specification of the computation unaffected.

A particular action language could implement each semantic construct one-to-one, or it could define higher-level, composite constructs to offer the modeler both power and convenience. These higher-level constructs do not need to be defined as a part of the semantics, because they already exist as composites of primitives.

Primitive actions may be grouped into composite actions, including control actions such as conditionals, loops, etc. In addition, a surface language may map higher-level constructs to lower-level action model constructs. For example, in a composition association where the deletion of an instance implies the deletion of all its components, the specification defines the delete action to remove only the single instance, and the specification requires further deletions for each of the component instances. A surface language could choose to define a delete-composition operation as a single unit as a shorthand for several deletions that cascade across other associations.

This specification, then, expresses the fundamental semantics in terms of primitive behavioral concepts that are conceptually simple to implement. The semantics do not define any “magic” that takes place behind the scenes. Modelers can work in terms of higher-level constructs as provided by their chosen surface language or notation.

2.17.2.6 Execution Engines

The semantic primitives are defined to enable the construction of multiple *execution engines*, each of which may have different performance characteristics. A model compiler builder can optimize the structure of the software to meet specific performance requirements, so long as the semantic behavior of the specification and the implementation remain the same. For example, one engine might be fully sequential within a single task, while another may separate the classes into different processors based on potential overlapping of processing, and yet others may separate the classes in a client-server, or even a three-tier model.

The modeler can provide “hints” to the execution engine when the modeler has special knowledge of the domain solution that could be of value in optimizing the execution engine. For example, instances could—*by design*—be partitioned to match the distribution selected, so tests based on this partitioning can be optimized on each processor. The execution engines are not required to check or enforce such hints. An execution engine can either assume that the modeler is correct, or just ignore it. An execution engine is *not* required to verify that the modeler’s assertion is true.

2.17.3 The Actions

This section provides an overview of the various actions that can be included in developer models. Related actions are organized into sections that correspond to subsequent chapters.

2.17.3.1 Foundation

Traditional programming languages overspecify sequence because actions, even within the same procedure, can potentially execute concurrently on different machines. For example, some execution engines might rely on a file server, and execute all data accesses on that separate machine, but over-specifying sequence inhibits such re-organization of the actions. The issue here is not concurrency of actions *per se*, but rather the requirement to be able to re-organize the actions for an efficient implementation.

This specification therefore treats all actions as executing concurrently unless explicitly sequenced by a flow of data or control. In addition, each action is defined so that it is free of any context that describes how it is used, so that data access and other computations are two separate primitive actions connected together when required, and each action is unaware of the source or destination of the data.

To meet these goals, this specification defines the concept of a *data flow.* A data flow carries data between two actions and in so doing effectively sequences the actions. Hence, we may execute a read action to access some data, flow that data to a computation action that computes the square, and then flow that result to a write action. The three actions *have to* execute in this order because each one cannot execute until the previous one produces its data.

The technical reasons for this choice are first that data flow makes this form of sequencing explicit. Second, because data flows are produced only once (as a result of a specific execution of an action), we may make assertions about the values on the data flow for verification and translation purposes.

A data flow connects to other components via *pins*. Each data flow has a source that is the output pin of some other element, typically an action, and each data flow has a destination that is an input pin to some other element.

This specification also provides the ability to read and write variables local to the procedure for upward compatibility with existing languages.

Finally, the specification provides for *control flows* between actions for explicit sequencing that does not rely on data flow.

2.17.3.2 Composite Actions

This specification provides for conditionals and iteration. In both cases, there is a need to group actions together so they may be executed (or not) as unit. Such groupings may be nested, and may accept and receive control flows. Data flows are routed transparently to the actions involved. The presence of concurrency also affects how these traditional structures operate.

A conditional action comprises a group of clauses that execute concurrently, subject to optional execution ordering constraints among them. Each clause has two parts: a test and a body. A test produces a Boolean value. If the value is true, the associated body may be executed. If the value is false, the body is not executed and clauses dependent on the given clause may execute their tests. If there are no constraints among a set of clauses, they may be tested concurrently. Potentially more than one test may produce a true value during execution, but only one body is executed. If more than one clause yields a true test, the body of exactly one of them is nondeterministically chosen for execution.

The modeler may, of course, constrain the sequencing of the tests so that when one test evaluates to true, no further tests are evaluated, and the structure then behaves like a conventional if-then-else. The modeler may also assert that only one test is ever true *by design of the tests* (that is, they are *mutually exclusive*), so once a test has evaluated to true, no further tests need to be evaluated (even in the absence of explicit sequencing). An execution engine is not required to verify explicitly a modeler's assertion of mutual exclusion.

Some kinds of iteration require sequential execution, for example, a regression that takes the outputs of one cycle and feeds it as inputs to the next. This kind of iteration is managed by the loop action. It comprises a single clause that, in turn, comprises a test and a body. The body is executed repetitively while the test is true. The body of a loop accepts a set of input values and produces a set of output values. The output values of one iteration of the body become the input values for the next iteration of the loop. These values are known as the loop variables. (Iterations that scan the elements of a collection are handled by collection actions. These include both concurrent scans and sequential scans. See below.)

2.17.3.3 Read and Write Actions

Objects, attributes, links, and variables have values that are available to actions. Objects have classifiers and objects can be created and destroyed; attributes and variables have values; links may be created and destroyed, and they have link ends, and qualifier values; all of which are available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Read and write actions share the structures for identifying the attributes, links, and variables that they are accessing.

Read actions do not modify the values they access, while write actions, as a general policy, have the most minimal effect possible. For example, creating an object does not execute constructors. Languages requiring additional semantics can define higher-level actions on the primitive ones given here. Unlike read actions, write actions may leave semantics unspecified in some cases. No semantics is usually given when an action violates those aspects of static UML modeling that constrain runtime behavior. For example, no semantics is given for creating an instance of an abstract class. The only exception is minimum multiplicity, which is given a semantics equivalent to the lower multiplicity being zero. Modelers of language bindings can assign their own semantics for undefined cases.

2.17.3.4 Computation Actions

Computation actions transform a set of input values to produce a set of output values. These actions work directly on values and produce values; they embody mathematical functions. They do not interact with object memory: they do not read or write attribute or link values. They do not interact with other objects or other executions, so their control is entirely self-contained. These actions supply the primitive functions out of which computations are constructed.

This specification allows for the incorporation of primitive functions, such as mathematical functions or string manipulations, that may be gathered together to form a profile for specific uses, but it does not define a set of primitive functions as a part of the specification. This allows the actions to be extensible.

2.17.3.5 Collection Actions

Collection actions permit the application of an action to a set of data elements, possibly in parallel. They avoid the need for explicit indexing and extracting of elements from collections, avoiding the overspecification of control that would otherwise be necessary.

Each collection action contains a subaction, an embedded action that is executed once for each element in the input collection. There are four kinds of collection action. The map action applies a subaction in parallel to each of the elements of a collection of data, resulting in an output that is a collection of the same size and shape. The filter action selects a subset of the elements in a collection into a new collection of the same shape, based on a boolean result of the subaction applied to each element. The iterate action applies a subaction repeatedly to each of the elements in a collection, accumulating the effects in loop variables. The reduce action repeatedly applies a binary subaction to pairs of adjacent elements in a collection, implicitly replacing a pair of elements by the result, until the final result is a single element of the type in the collection. The binary subaction must be associative, therefore it may be applied in parallel to many pairs of elements, because the exact order of application will not affect the final result. For example, summation or matrix multiplication are reduction operators.

There is also a separate loop action that carries out actions repetitively subject to an arbitrary test. The iteration action can be regarded as a specialization of the loop in which a separate element of the collection is selected for input to each iteration and the while-test fails when all elements have been processed.

2.17.3.6 Messaging Actions

These actions exchange messages among objects. An initial message from one object to another is called a *request*. The sender of a request may simply continue execution immediately without concern for the behavior invoked by the request (an asynchronous request, or *send*), or it may choose to suspend execution until the activity invoked by the request reaches a well-defined point and sends a reply message back to the requestor, with optional return values (a synchronous request, or *call*). If the request is synchronous, the behavior of the receiver must have a well-defined reply point; if the

request is asynchronous, a reply is optional and will be ignored. The receiver may handle a request in various ways based on its organization, including procedure execution and triggering a state machine. The requestor need not be aware of how the request will be handled. The messaging model covers a wide range of ways to match behavior to requests, including state machine triggers, fixed procedures, class-based method lookup, method combination (such as before-after methods), object-based delegation (as in *self*), and so on. In all cases, the effect is processed by a distinct context from the context of the requestor and the messaging information is transmitted among requestor and target by value. This messaging model fully supports distributed processing without special mechanisms. This model unifies operations and signals into a single concept.

2.17.3.7 Jump Actions

All flow of control for a procedure *could* use Dijkstra-style, fully nested flow-of-control constructs, but this style can be awkward and obscure when dealing with unusual or secondary conditions that do not follow the main line. Programming languages include constructs such as *break*, *continue*, and *exceptions* for dealing with these situations. When a non-mainline situation occurs, the normal flow of control is abandoned and a different flow of control, specified in the program, is taken. The UML jump construct unifies these nonlinear flow-of-control mechanisms while providing the functionality found in most modern programming languages.

2.18 Action Conventions

This describes the structure and conventions applied in the chapters that describe actions.

2.18.1 Chapter Structure

Each chapter begins with a brief introduction that outlines the content of the chapter, the theme behind it, and any high-level design decisions or criteria that guide the chapter.

The following several sections in each chapter introduce the key abstractions for readers. The material is organized for understanding, not for detailed reference. The purpose is to give the reader an intuitive feel for the concepts, and, particularly, to show how they work together and the reason that they are needed. These sections strive for clarity at the expense of detail, and they are not normative. Examples, diagrams, and partial models help present concepts. The normative specifications appear in the precise descriptions of classes in subsequent chapters.

Metamodel diagrams of the actions are included in these conceptual sections. These are normative and are the basis of the interchange format.

The final sections of each chapter (with titles "... Classes") define the actions in alphabetical order. Each class section covers the semantics of the class, including its attributes and associations, inputs and outputs, static well-formedness rules, and additional OCL operations. These sections are normative. The format used is the subject of the next section of this chapter.

2.18.2 Description of a Class

The formal class descriptions begin with the technical meaning of the class. If it is an action, it describes what the action does. It calls out all semantic features, unless detailed aspects are given in the semantic section.

The following subsections are lists with specific formats for their contents.

- Attributes
- Associations
- Inputs
- Outputs
- Well-formedness rules
- Additional operations

A subsection is omitted if it has no elements, except for Inputs and Outputs, which are included for all concrete action classes and omitted for abstract action classes.

2.18.2.1 Attributes

This subsection lists all attributes of the class, including their types and multiplicity.

For enumerations that are used as the type of only one attribute, a list of enumeration literals may follow, one to a line.

- name: type [multiplicity] A description of the attribute. For enumerations:

  foo—description of the literal

  bar—description of the literal

Inherited attributes are omitted.

2.18.2.2 Associations

This subsection lists all associations that have end names opposite from the class being described. If an association has no opposite end name, it is omitted. All associations are described in at least one direction. For example:

- endName: TargetClassName [multiplicity]
  Description of the association in the target direction

Inherited associations ends are omitted unless they are derived. Associations are sometimes derived from more general ones to limit the kinds of classes that can participate. The parent and association end from which an end is derived are given at the beginning of the description:

- derivedEndName : TargetClassName [multiplicity]
  First, in parentheses, give the name of the parent class and association end using the notation Class:endName. Then describe the association.

The following is an example from CreateLinkObjectAction:

- result [1..1] : OutputPin [1..1]
  (Derived from Action:outputPin) Gives the output pin on which the result is put.

2.18.2.3 Inputs

In describing an action, it is necessary to refer to the values that are delivered to the action on its input pins and sent by the action on its output pins. The action metamodel in Section 2.19, “Action Foundation,” on page 2-211, defines the association of an abstract action with a set of input pins and a set of output pins. Each kind of action requires a specific set of inputs and produces a specific set of outputs. For example, a read-attribute action has a single pin to which the object to be read is supplied and a single output pin that holds the value that is read. For clarity, the action metamodel contains derived associations that delineate the specific set of input and output pins required for an action. These are listed in the Associations section.

For additional clarity, the input and output pins are also listed in separate sections that give information normally spread out in the metamodel and well-formedness rules. This section lists the pins by the name of the corresponding derived association end. For each derived pin association, the multiplicity is given resulting from combining the multiplicity of the association at the pin end and the multiplicity of the pin itself. The type of the pin itself is not always constant, so is not given. The description gives any static constraints on the typing of the input pins that are in the well-formedness rules.

This is an example from AddAttributeValueAction:

- value : RuntimeInstance [1..1]
  (Inherited from WriteAttributeAction) Value of attribute to add. Its type is the same as the type of the attribute.

Inherited pins are included and the parent class from which they are inherited is given at the beginning of the description in parentheses.

- inputName : type [multiplicity]
  (Inherited from Foo) Description of inherited pin.

Here’s the more general format:

- inputName [multiplicity] : type
  First, in parentheses, give the name of the parent class from which the pin is inherited, if any. Then describe the pin, including any constraints on the type given in the well-formedness rules.

Sometimes getting to a pin from an action requires navigating more than one association. In this case, the entire navigation path is given using the OCL dot operator. The multiplicity of the association at the pin end in this case should actually be a combination of all the multiplicities navigating from the action to the pin, and reflect well-formedness constraints on those multiplicities that do not depend on the user model.

Here's an example from CreateLinkAction:

- endData.qualifier.value : RuntimeInstance [0..*]
  (Inherited from LinkAction) Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier attribute. See LinkEndData.

In the above example, the qualifier value is reached from a CreateLinkAction by navigating through the endData association, then the qualifier association, then the value association.

The order in which the input pins are listed specifies the order in which pins must be linked to the action. This represents well formed rules on the action model. For example, suppose an action has two input pins with derived association end names "object" and "value." If the pins are listed in that order, then the corresponding well-formedness rules on the action model are:

[1] self.object = self.inputPin.at(1)

[2] self.value = self.inputPin.at(2)

The well-formedness formally specify derivation of the pin associations.

This subsection is included for concrete action classes even if there are no input pins, with the text "None." It is not included for abstract action classes.

2.18.2.4 Outputs

This subsection lists the output pins using the same format as input pins. The well-formedness rules represented by the listing order are also the same, except they apply to output pins instead of input pins.

2.18.2.5 Well-formedness Rules

This subsection lists static constraints on user models. The constraints are expressed first in English, then in OCL. The constraints do not cover aspects already expressed in the model, such as association multiplicity. Well-formedness rules defined for a class are inherited by all its subclasses.

2.18.2.6 Additional Operations

This section defines additional OCL operations that apply to the class. These also apply to all the descendants of the class.

2.18.2.7 Semantics

This section contains more detailed specification of semantics, if needed. It is primarily for actions that go through intermediate states of execution.

2.19 Action Foundation

This section describes the structure of procedures and actions in the UML metamodel. The foundational concepts discussed in this chapter apply to all kinds of actions. The details of specific kinds of actions are described in subsequent chapters.

2.19.1 Action Specification

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. In addition, some actions modify the state of the system in which the action executes. The inputs to an action may be obtained from the results of other actions, and the outputs of the action may be provided as inputs to other actions, some of which have the sole purpose of reading or writing object memory.

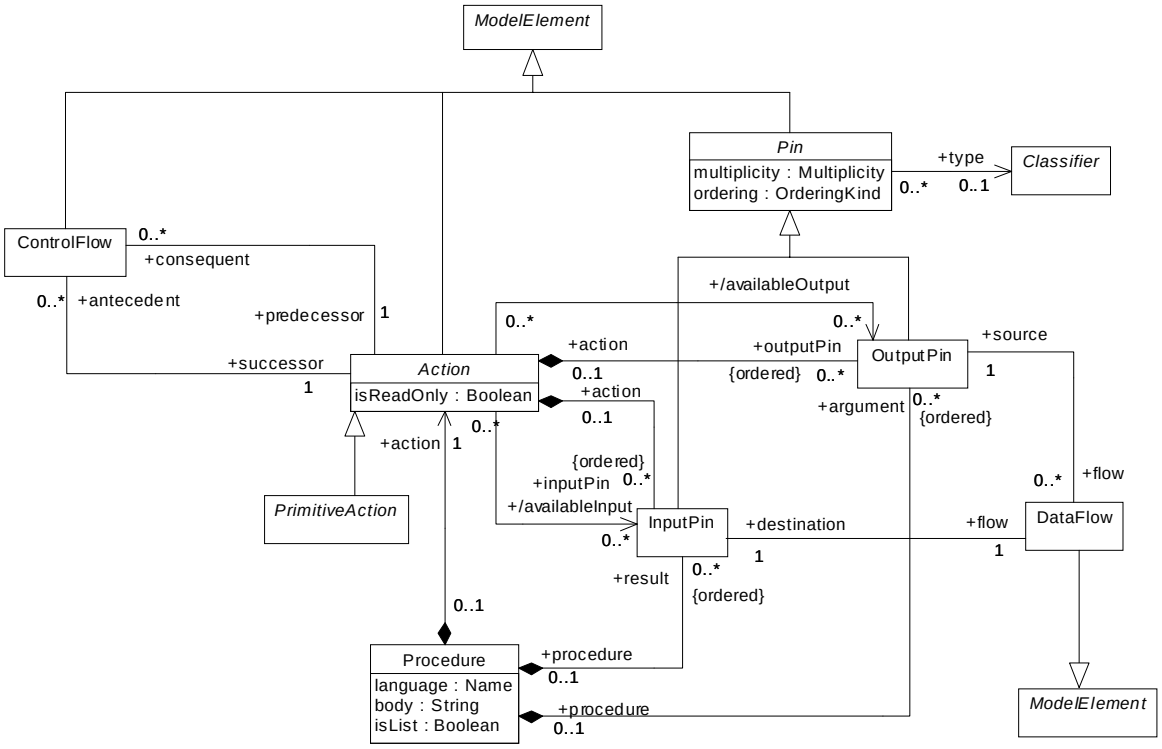


Figure 2-38 Action foundation model

2.19.1.1 Pins

An action takes some input values, possibly accesses the state of the containing system, performs some processing, possibly modifies the state of the system, and produces some set of output values. The required inputs and outputs of an action are specified as *pins* of the action.

A pin specifies the type and multiplicity of values that may be held by the pin. A pin may hold multiple values, subject to the specified multiplicity. Each of the values held by a pin must conform to the type specified for the pin. This type conformance is determined as follows.

- If the specified type is a primitive type, then the pin can only hold primitive values of the given type.
- If the specified type is an enumeration type, then the pin can only hold enumeration values of the given type.
- If the specified type is a data type other than a primitive or enumeration type, then the pin can hold values of the given type or any specialization of the given type.
- If the specified type is a classifier other than a data type, then the pin can hold object identities that have the given type or a specialization of the given type.

Input pins are the connection points for delivering the input values to actions. *Output pins* are the connection points for obtaining the output values from actions. The types and multiplicities of the input and output pins of an action are constrained by the *form* of the action. For example, an action that reads a certain attribute has an output pin with the type and multiplicity of that attribute, while an action that writes to an attribute has an input pin with the type and multiplicity of that attribute. An action that calls an operation has (possibly empty) sets of input and output pins with types and multiplicities corresponding to the input and output parameters of the operation.

The entire context for an action is represented in its pins, including the “current object” that is also passed to an action on a pin. There is no “back door” by which an action can access objects implicitly.

Note – The types of the input and output pins of an action form the input and output “signature” of the action. The signature is modeled by derivation from the types of the input and output pins; it is not modeled explicitly. Some actions impose constraints on the types of some of their pins.

2.19.1.2 Data Flow

A *data flow* from an output pin to an input pin indicates that an output value of one action is used as an input value of another action. Data flows link chains of actions without having to store values temporarily.

A single output pin can be connected to zero or more input pins, but each input pin can have at most one connection. Thus, input pins are “single assignment” data holders—once they receive a value, this value does not change. In other words, normal dataflow connections allow “fan out,” but not “fan in.”

The input pin that is the destination of a data flow must *conform* to the output pin that is the source of the flow: the type of the output pin is the same as or a descendant of the type of the input pin, and all cardinalities allowed by the multiplicity of the output pin are allowed by the multiplicity of the input pin. For example, if the type of the output pin is money, and the type of the input pin is real, and money is a descendant of real, then the types conform.

2.19.1.3 Control Flow

A *control flow* indicates an ordering constraint between a predecessor action and a successor action without explicit data flow. A predecessor action must complete execution before its successor action can execute. Such control constraints are often required for actions that affect object memory, such as when a value written to an attribute by one action is to be read by a subsequent action. Control flow specifies the order of actions that require such constraints.

Other actions create or destroy objects, communicate among objects, or have external effects outside the system. Control flow is used to order these actions as well. Control flow represents an implicit potential communications path among actions—through object memory, by signal transmission, or outside of the system.

Traditional programming languages have implicit control flows between each statement, but they overspecify execution order. The model defined here permits control flows, data flows, and concurrent actions as needed.

A data flow implies a control dependency in the sense that an executing action cannot consume an input value during execution until it has been produced by the source action. Control sequencing is implicit between actions connected by data flows; it is unnecessary to include explicit control flows between such actions.

The network of actions connected by data and control flows forms an acyclic directed graph because an action cannot be both a predecessor and successor of another at the same time.

2.19.1.4 Primitive Actions

A *primitive action* is one that cannot be decomposed into other actions. Primitive actions include purely mathematical functions, such as arithmetic and string functions; actions that work on object memory, such as read actions and write actions; and actions related to object interaction, such as messaging actions. Each kind of primitive action has a form that specifically defines sets of input and output pins. The details of the various kinds of primitive actions are discussed in later chapters in this part.

2.19.1.5 Procedures

Within a user model, actions may be nested at various levels. The highest-level such grouping is the *procedure*. A procedure is a set of actions that may be attached as a unit to other parts of the user model, for example, as the body of a method. Conceptually a procedure takes a single request object as argument and produces a single reply object as result. Both the type and the attributes of the request object and the reply object may convey information. As a convenience, if the isList flag is true, the procedure may be written with multiple arguments and results; the attributes of the request object are automatically unmarshalled into a list of arguments, and a list of results are automatically marshalled into a reply object; the type of the request and reply objects are implicitly generated for each such procedure. All uses of procedures may have an *argument* (or arguments, if the *isList* flag is true), corresponding to the input parameters. Procedures executed as the result of synchronous calls may have a result (or results, if the *isList* flag is true); procedures executed as a result of asynchronous invocations do not have results (or rather, if they do have result pins, the results are ignored). See Section 2.24, “Messaging Actions,” on page 2-311.

The arguments supply values to the action by *output* pins within the procedure, and each one may be connected to zero or more inputs of the action. Similarly, the results are represented as *input* pins within the procedure, and each one must be connected to exactly one output of the action. The data flow connections then follow the normal connection rules for input and output pins. The types of the actual arguments to a procedure can be descendants of the declared types. The types of the actual results of a procedure can be descendants of the declared types.

As with any action, the pins on a procedure action must match the types and multiplicities of the corresponding parameters supplied to the method, or the supplemental data items on the triggering event.

Procedures can have a textual specification entered in the body attribute, with the kind of specification given in the language attribute.

2.19.2 Action Execution Model

An *action execution* corresponds to an individual execution of a particular action. Similarly, a *procedure execution* is an individual run-time execution of a procedure. Each action in a procedure may execute zero, one, or more times for each procedure execution, depending on the use of composite and collection actions.

Execution is not instantaneous, but takes place over a period of time. Procedures and other groupings of actions must be executed in a number of steps, including control of the execution of nested actions. Thus, procedure and action executions, in general, require the maintenance of state information over time.

2.19.2.1 Pin Values

Both procedures and actions have pins that get values during procedure and action execution. A *pin value* represents the value of a single pin at a specific point in the execution of a procedure or action. There may actually be a collection of instance values associated with a pin value, consistent with the multiplicity of the corresponding pin.

2.19.2.2 Action Execution

The action semantics place no restriction on the relative execution order of two or more actions, unless they are explicitly constrained by data flow or control flow relationships. Hence every action within a procedure may execute at once, though a specific execution engine may actually perform the executions sequentially or in parallel.

The execution of an action proceeds through a life cycle whose stages are as follows. These stages can be understood in terms of the constraints they place on the action execution at various points in its history.

- *Waiting* - An action execution may be created at any time after the procedure execution for its containing procedure is created. On creation, an action execution has the status *waiting* and no pin values.
- *Ready* - An action execution with status *waiting* becomes *ready* on the completion of the execution of all *prerequisite* actions (that is all actions that are the sources of data flows or predecessors of control flows into the action becoming ready). This synchronization captures the ordering in time between the completion of prerequisite action executions and the readying of their target. The values of the input pins of the target action execution are determined by the values of the output pins from the prerequisite action executions for actions that are the sources of data flows. (Enclosing composite actions may also place constraints on the execution of nested actions—for instance, conditional execution.)
- *Executing* - Once it is ready, an action execution eventually begins *executing*. An action need not begin executing immediately upon being ready and the action semantics do not determine the specific time delay (if any) between becoming ready and actually executing. For a primitive action, there will be only one executing step in the history of the execution. However, a composite action may require several steps to complete execution.
- *Complete* - When it has finished executing, the action execution becomes *complete*. The action execution then has pin values for all output pins of the action as well as all input pins. The specific semantics of each kind of action determine how these output pin values are computed. After the output values from a completed execution have been copied, there is no longer any way for another execution to access the completed execution. Because a completed execution is inaccessible, the action semantics does not supply any clean up or garbage collection rules, as they cannot affect the outcome of the computation.

This life cycle may be depicted informally using a statechart diagram, as shown in Figure 2-39 on page 2-217.

2.19.2.3 Procedure Execution

A procedure execution also has four statuses.

- *Ready* - A procedure execution begins with status *ready.* Input parameters are passed to the procedure as values of its argument pins. These are captured as pin values that must be associated with the procedure execution on its creation. (Procedure executions are generally created as the direct or indirect result of messaging actions.)
- *Executing* - Once it is ready, a procedure execution eventually beings *executing*. A procedure need not begin executing immediately upon being ready and the action semantics do not determine the specific time delay (if any) between becoming ready and actually executing. When a procedure beings executing, it causes the start of the execution of the nested action.
- *Returning* - When its nested action has completed execution, it transitions to the status *returning*. At this point the procedure execution has pin values for all the result pins of the procedure. In the specification of the procedure, the result pins are connected by data flows to the output pins of actions within the procedure. This specification and the procedural context of completed action executions within the procedure execution then determine the values for the result pins. If the procedure execution was triggered by a messaging action, then it may package its results for transmission back to the invoker.
- *Complete* - Once the procedure execution has completed returning, it becomes *complete*.

This life cycle may be depicted informally using a statechart diagram, as shown in Figure 2-39.

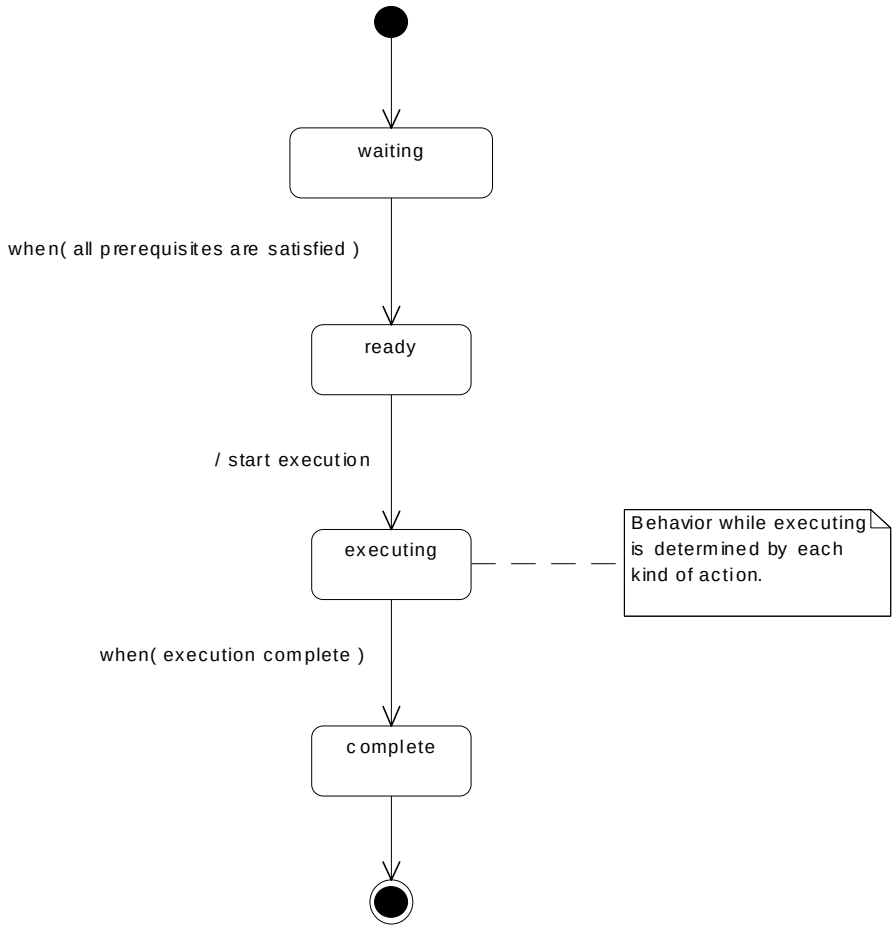


Figure 2-39 Life cycle for action execution

A procedure execution acts as a *context* for all action executions nested directly or indirectly within it. This contextual information includes the following.

- *Host* - A procedure may execute as the result of the invocation of a method or an event occurrence triggering a transition. The host of the execution is the instance that owns the invoked method or the state machine for the transition. The host remains frozen throughout the execution. (There is no host instance for procedures associated with a method with classifier scope or for procedures acting as expressions.)
- *Action executions* - A procedure is the root of a tree of current action executions. Each of these action executions can reference its parent action execution. However, it is generally not possible to determine statically which actions within the procedure will actually execute, or even at all. The current set of action executions

nested within a procedure execution will generally change over time. Nevertheless, this set provides the ability to map from the specification of an action within a procedure to the specific, current execution of that action within the context of a given procedure execution (if, of course, that action is currently executing at all).

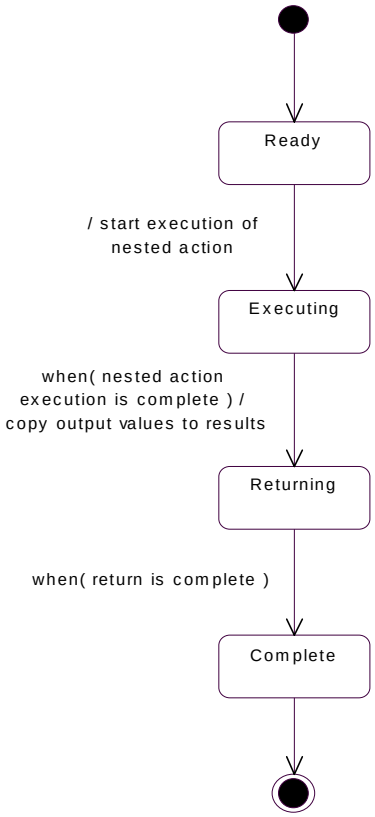


Figure 2-40 The life cycle for procedure execution

Note – A UML constraint contains an expression that can be modeled using a procedure with a Boolean result. However, when such procedures should be executed and what should be done when they fail is not a simple question. Some apply at all times, but most can be violated during the actions of a transition. Some may be valid after certain actions. Currently, the action semantics specification does not attempt to verify or enforce user-defined constraints that are made invalid by actions. In effect, such constraints are considered to be design statements and the system implementer is obliged to ensure that they are not violated.

2.19.3 Action Foundation Classes

2.19.3.1 Action

An action is the fundamental unit of behavior specification. An action takes a set of input values and uses them to produce a set of output values, though either or both sets may be empty. If both inputs and outputs are missing, the action must have some kind of fixed, nonparameterized effect on the system state, or be performing some effect external to the system. Actions may access or modify accessible, mutable objects. A reference to an object to read or write is an input of the action. Composite actions may include data-transformation actions as well as object-access actions.

An action may have a set of incoming data flows as well as a set of explicit control flow dependencies on predecessor actions. An action will not begin execution until all of its input values (if any) have been produced by preceding actions *and* all predecessor actions have completed. The completion of the execution of an action may enable the execution of a set of successor actions and actions that take their inputs from the outputs of the action. Actions come in various kinds, each of which has its own formation rules. An action must be one of those kinds.

Attributes

- isReadOnly : Boolean
  If true, then this action must not make any changes to variables outside the action or to object memory. (This is an assertion, not an executable property. It may be used by an execution engine to optimize model execution. If the assertion is violated by the action, then the model is ill formed.)

Associations

- antecedent : ControlFlow [0..*]
  The set of control flows that must be enabled before this action can execute.
- availableInput : InputPin [0..*]
  (Derived from Action::inputPin) The set of all input pins available to be the destinations of data flows entering the action. Such pins must not have data flows from pins within the action. The derivation rule is defined for each kind of action.
- availableOutput : OutputPin [0..*]
  (Derived from Action::outputPin) The set of all output pins available to be the sources of data flows leaving the action. These pins may also be connected by data flows to input pins within the action. The derivation rule is defined for each kind of action.
- consequent : ControlFlow [0..*]
  The set of control flows that are enabled when this action finishes executing.
- inputPin : InputPin [0..*]
  The ordered set of input pins owned by the action, which act as connection points for providing values consumed by the action.

- outputPin : OutputPin [0..*]
  The ordered set of output pins owned by the action, which act as connection points for obtaining values generated by the action.

Well-formedness Rules

[1] There must be no cycles in the graph of actions and flows, where a cycle is defined as a path that begins and ends at the same action and a path is constructed from directed edges between actions, with control flows traversed from predecessor action to successor action and data flows traversed from the action of the source pin to the action of the destination pin. A control flow from or to a group action is treated as being a set of control flows from or to each action within the group action.

```
not self.allSuccessors()->includes(self)
```

Note – This is a necessary but not sufficient condition to prevent ill formed control cycles. There are additional conditions related to conditional and loop actions that are handled as well-formedness rules for those kinds of actions.

Additional Operations

[1] This operation returns the set of all immediately nested actions of this action. The actual set returned is defined in concrete descendants of Action.

```
nestedActions() : Set(Action)
```

[2] This operation returns all nested actions of an action, nested to any depth.

```
allNestedActions() : Set(Action)
allNestedActions() = self.nestedActions()->union(self.nestedActions().allNestedActions())
```

[3] This operation returns the set of immediate subactions of this action, whose output pins may be directly connected to the input pins of actions outside this action. The actual set returned is defined in concrete descendants of Action. (This is only intended to include subactions that are “visible” through a “porous” boundary, which currently includes only the subactions of GroupAction).

```
subactions() : Set(Action)
```

[4] This operation returns all subactions of an action, nested to any depth.

```
allSubactions() : Set(Action)
allSubactions() = self.subactions()->union(self.subactions().allSubactions())
```

[5] This operation returns the set of all input and output pins of an action.

```
allPins() : Set(Pin)
allPins() = self.inputPin->union(self.outputPin)->asSet()
```

[6] This operation returns true if the action is a subaction, at any depth, of another given action.

```
isSubaction(otherAction: Action):Boolean
isSubaction(otherAction) = otherAction.allSubactions()->includes(self)
```

[7] This operation returns the set of actions whose execution must complete before this action can execute: the source actions in data flows and predecessor actions in control flows.

```
prerequisites() : Set(Action)
prerequisites() = self.inputPin.flow.source.action->union(self.antecedent.predecessor)
```

[8] This operation returns all the actions which are destinations of data flows or successors of control flows leaving an action. A control flow from or to a group action is treated as being a set of control flows from or to each action within the group action.

```
successors() : Set(Action)
successors() = self.outputPin.flow.destination.action
    ->union((self.consequent.successor->union(self.group.consequent.successor))
        ->collect(a : Action | a.subactions()->including(a)))->asSet()
```

[9] This operation returns the transitive closure of all data-flow and control-flow successors (defined as above) of an action.

```
allSuccessors() : Set(Action)
allSuccessors() = self.successors()->union(self.successors().allSuccessors()->asSet()
```

Semantics

An *action execution* represents the general run-time behavior of executing an action within a specific context. Action is an abstract class; therefore all action executions may be executions of specific kinds of actions.

1. An action execution is created with status *waiting* with no pin values initialized. The action has the same procedural context as the action execution that created it. For the top-level action execution in a procedure, a procedural context is created with empty slots for all of the variable values or data flow values that may be created during an execution of a procedure. All action executions within a procedural execution share the same procedural context, that is, they share access to the same set of variable and data flow values.

2. An action execution that is waiting becomes *ready* when all its data flow and control flow prerequisites have satisfied, at which point it obtains input pin values from each of its data-flow sources. After all the values have been obtained, the action execution begins executing.

3. An action continues executing until it has completed. Details of execution are given under the description of each particular kind of action.

4. When completed, an action execution produces values on all its output pins and terminates execution. The action execution satisfies control flow or data flow prerequisites for any other potential action executions that depend on it or one of its data values.

2.19.3.2 ControlFlow

A control flow is a sequencing dependency between two actions. The successor action of the flow may not execute until the predecessor action has completed execution.

Associations

- predecessor : Action [1..1]
  The action that must finish executing before the successor can execute.
- successor : Action [1..1]
  The action that cannot execute until the predecessor completes execution.

Semantics

An action execution has a control flow prerequisite on each action execution in the same procedural context for which the respective actions have a successor-predecessor relationship. The prerequisite is satisfied when the predecessor action execution has completed execution.

2.19.3.3 DataFlow

A data flow carries values from a source output pin to a destination input pin. When a value is generated on the source pin, it is copied to the destination pin. The source pin must therefore conform in type and multiplicity to the destination pin.

Associations

- destination : InputPin [1..1]
  The input pin that receives the data carried by the flow.
- source : OutputPin [1..1]
  The output pin that provides the data carried by the flow.

Well-formedness Rules

Note – Since UML does not provide any standard classifier that is the ancestor of all other classifiers, untyped pins can be used for the purpose of accepting input of “any” type.

.

[1] The type of the source pin must be the same as or a descendant of the type of the destination pin. An untyped pin has a type that is an ancestor of any classifier.

```
self.destination.type->isEmpty()
    or (self.source.type->notEmpty()
        and (self.source.type = self.destination.type
            or self.destination.type.allParents()->includes(self.source.type)))
```

Note – The operation “allParents” is defined for GeneralizableElement.

[1] All cardinalities allowed by the multiplicity of the source pin must be allowed by the multiplicity of the destination pin.

```
self.source.multiplicity.compatibleWith(self.destination.multiplicity)
```

Semantics

An action execution *e* corresponding to action *a* has a data flow prerequisite on each output pin *p* for which the output pin *p* is connected to any input pin of *a*. The prerequisite is satisfied when the output pin *p* holds a value within the same procedural context as the action execution *e*.

2.19.3.4 InputPin

An input pin holds input values to be consumed by an action. An input pin may be the destination for exactly one data flow. The input pin receives its values from the source output pin of the data flow.

Associations

- action : Action [0..1]
  The action that owns the pin as an input. This value only applies to Actions.
- flow : DataFlow [1..1]
  The data flow for which this input pin is the destination.
- procedure : Procedure [0..1]
  The procedure that owns the pin as a result. This value only applies to Procedures.

Well-formedness Rules

[1] An input pin must be owned by either an action or a procedure but not both.

```
self.action->size() + self.procedure->size() = 1
```

Semantics

An input pin holds a potential value within each procedural context holding an execution of the action to which the pin is an input. When the procedural context is created, all input values are initially empty. Input values are initialized just before the execution of their action.

2.19.3.5 OutputPin

An output pin holds output values generated by an action. A single output pin may have several data-flow connections to several input pins. In this case, the output pin provides a copy of its value to each of the associated input pins.

Associations

- action : Action [0..1]
  The action that owns the pin as an output. This value only applies to Actions.
- flow : DataFlow [1]
  The data flow for which this output pin is the source.
- loop : LoopAction [0..1]
  The loop action that owns the pin as a loop variable (see the discussion under Composite Actions). This value only applies if the pin is a loop variable.
- procedure : Procedure [0..1]
  The procedure that owns the pin as an argument. This value only applies to Procedures.

Well-formedness rules

[1] An output pin must be owned by exactly one of an action (as an input), a loop action (as a loop variable), or a procedure (as a result).

```
self.action->size() + self.loop->size() + self.procedure->size() = 1
```

Semantics

An output pin holds a potential value within each procedural context holding an execution of the action to which the pin is an output. When the procedural context is created, all output values are initially empty, except those output values initialized as parameters of the overall procedure. Output values are initialized at the completion of execution of their action.

2.19.3.6 Pin

A pin is a connection point for delivering input values to or obtaining output values from an action. Any values passing through the pin must conform to the type of the pin and have cardinalities allowed by the multiplicity of the pin. (A pin without a type specification can hold any value.) Pin is completely specialized into input and output pins.

Attributes

- multiplicity : Multiplicity [1..1]
  A specification of the number of values a pin may hold at any one time.
- ordering : OrderingKind [1..1]
  Indicates whether the set of values held by this pin is to be considered ordered or not.

Associations

- type : Classifier [0..1]
  A classifier specifying the allowed classifiers of values passing through the pin. The actual classifier of a value must conform to the type specification of the pin.

Semantics

See InputPin and OutputPin.

2.19.3.7 PrimitiveAction

A primitive action is one that does not contain any nested actions, so all available inputs and outputs of the action are pins directly owned by the action.

Well-formedness Rules

[1] The available inputs of a primitive action are the input pins of the action.

```
self.availableInput = self.inputPin->asSet()
```

[2] The available outputs of a primitive action are the output pins of the action.

```
self.availableOutput = self.outputPin->asSet()
```

Note – PrimitiveAction is really only defined as a convenience to provide a common definition of these well-formedness rules for all kinds of primitive actions.

Additional Operations

[1] A primitive action has no subactions. (The subactions operation is defined for each kind of action.)

```
subactions() : Set(Action)
subactions() = Set{}
```

Semantics

See Action for the general rules of starting and finishing execution. See the particular kind of action for the rules of executing the action and producing output values.

2.19.3.8 Procedure

A procedure is a set of actions that may be attached as a unit to other parts of the user model. The behavior of the procedure is specified using an action, which is usually composite. When it is activated, a procedure execution receives a request object from the caller; during its execution it produces a reply object that is returned to the caller as the result. The procedure has a single input pin for the request object and a single result pin for the reply object. If the *isList* flag is set in the action, the procedure may have multiple argument and result pins; the request object is broken apart into an ordered list of arguments, one per attribute of the request object; and the reply object is composed from an ordered list of results, one per attribute of the reply object. This option presents the inputs and outputs to the procedure in the traditional fashion as a list of explicit parameters, although they are composed into a request object and a reply object for use by the invocation action.

Pins of procedures have two directions: in and out, while method and operation parameters have direction in, out, inout, and return. Since parameters and pins are ordered on methods and procedures respectively, the parameters of methods can be matched to pins on procedures unambiguously, assuming the last output pin is matched to the return parameter. Parameters of kind in and inout match input pins, while parameters of kind out, inout, and return match output pins.

Attributes

- body : String
  A textual representation of the procedure in the named surface language. (Presumably the procedure is the result of parsing this representation, though that correspondence cannot be guaranteed by the rules of the metamodel.)
- language : Name
  The name of the language in which the textual procedure body is represented. (This language name should follow the conventions for language names in UML, as described for the language attribute of Expression.)
- isList: Boolean
  If true, the request is presented to the procedure as a list of zero or more separate argument pins and the result is delivered as a list of zero or more separate result pins. If false, there is a single argument pin that receives the request object and a single result pin that receives the reply object.

Associations

- action : Action [1..1]
  The action that provides the behavioral specification of the procedure.
- argument : OutputPin [0..*]
  if *isList* is true: The ordered set of output pins representing procedure arguments.
  if *isList* is false: One output pin representing the request object. Any extra pins are ignored.
- result : InputPin [0..*]
  if *isList* is false: The ordered set of input pins representing procedure results.
  if *isList* is true: One input pin representing the reply object. Any extra pins are ignored.

Well-formedness Rules

[1] All available inputs of the action of a procedure must be the destinations of flows, the sources of which are arguments of the procedure.

```
self.argument->includesAll(self.action.availableInput.flow.source)
```

[2] All results of a procedure must be the destination of flows, the sources of which are available outputs of the action of the procedure.

self.action.availableOutput->includesAll(self.result.flow.source)

[3] If the arguments/results are presented as request/reply objects, there must be exactly one argument pin and exactly one result pin.

self.isList implies (self.argument->size = 1 and self.result->size = 1

Semantics

A procedure execution is a single execution of a procedure as a result of its invocation by a call, the firing of a state machine transition, or some other means that might be defined in the future. A procedure in a user model may correspond to many procedure executions over time or at the same time. Each is an independent execution of the procedure.

A procedure execution maintains the context for all action executions within the procedure execution. It ties together all of the action executions and values corresponding to a single execution of a procedure. It represents a logical memory space with slots for variable values, data flow values, and action execution states. A procedure execution could be implemented in many different ways; the reader should not assume that it must be implemented directly as described.

1. When a procedure is invoked, a procedure execution is created. For each argument pin of the procedure, an output pin value in the procedure execution is initialized with a value copied from the corresponding argument value in the invocation. All other input and output pins and variables in the procedure are initialized to empty values. For each action in the procedure, the procedure execution contains an action execution in the *waiting* state.

2. After a procedure execution has been initialized, the execution of its nested action is placed in the *ready* state and begins execution. The effects of this execution eventually work their way through the entire procedure. When all of the subordinate actions have completed, the execution of the top-level nested action will be complete.

3. When the execution of the top-level nested action is complete, the procedure execution must wrap up:

- If the procedure invocation was asynchronous and not repliable, the execution is complete.
- If the procedure invocation was asynchronous and repliable, the (possibly empty set of) output values corresponding to result output pins of the procedure are formed into a result object that is transmitted to the object whose action invoked the procedure.
- If the procedure invocation was synchronous, the (possibly empty set of) output values corresponding to result output pins of the procedure are formed into a result object that is returned to the action execution that invoked the procedure.

2.20 Composite Actions

Composite actions are recursive structures that permit complex actions to be composed from simpler actions. A composite action can contain other composites as well as primitive actions. The leaves of the tree of action decomposition are primitive actions. The composite actions discussed in this section also provide for control structures beyond explicit data flows and control flows.

2.20.1 Composite Action Specification

This section gives a schematic description of the structure and behavior of composite actions. Figure 2-41 shows the model for these actions. Section 2.20.3, “Composite Action Classes provides the formal definitions of all the classes shown in Figure 2-41.

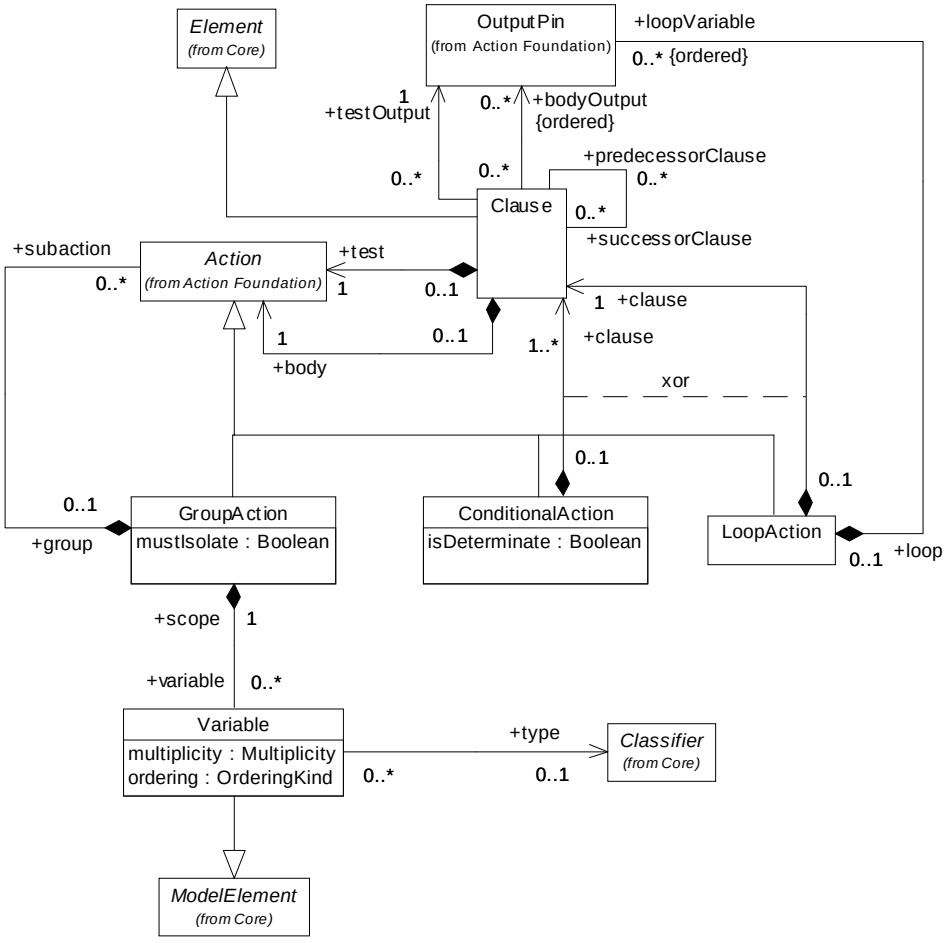


Figure 2-41 Composite actions metamodel

There are three kinds of composite action, each of which is treated in turn in the following subsections. Group actions group actions into larger units for use in procedures, conditionals, and loops. Conditional actions provide for contingent execution of subactions depending on a run-time test. All branches must have the same outputs. Loop actions permit the repeated execution of a subaction depending on a repeated run-time test, with outputs of one iteration used as inputs for the next iteration.

2.20.1.1 Available Inputs and Outputs

A composite action may have input and output pins of its own that allow data flows to be connected directly to the composite action as a whole. However, composite actions may also allow some data flow connections to "cross the boundary" of the composite action and connect directly to pins on subactions within the composite. This allows the subactions so connected to access values computed in the context of the composite action, similar to the way in which scoped languages in an inner scope to access variables defined in an enclosing scope.

The union of the set of input pins owned by a composite action and the set of input pins of subactions that may be the destination of data flows with sources outside the composite is called the set of *available inputs* of the composite action. Similarly, the union of the set of output pins owned by a composite action and the set of output pins of subactions that may be the source of data flows with destinations outside the composite is called the set of *available outputs* of the composite. The specification of each kind of composite action defines what the available inputs and outputs are for that kind of action.

The concept of available inputs and outputs is important because when a composite action is viewed from the outside as a black box, it is the complete sets of available inputs and outputs that provide the allowable connection points to the action for data flows. Thus, the available inputs and outputs for a composite are generally defined in terms of the available inputs of its subactions, without any need to "look inside" the subactions if they are themselves composites. By definition, the available inputs and outputs of a primitive action are simply the input and output pins owned by that action.

2.20.1.2 Group Action

The simplest form of composite action is the *group action*. A group action composes a set of subordinate actions into a higher-level unit. The actions may form a sequence, a concurrent set, or some combination of both.

A group action does not own any pins of its own. Data flows are not connected to a group action, instead they may "cross the boundary" of a group action to connect to input and output pins of actions within the group, making the boundary of the group action "porous." A group action does not encapsulate its contents, rather this approach to grouping focuses on convenience in organizing a computation, not encapsulation or decomposition.

Since an input pin may be the destination of only one data flow, the available inputs of a group action are just the available inputs of its subactions that are not connected to other actions within the same group. Output pins, however, may be the source for multiple data flows, so the available outputs of a group action include *all* the available outputs of all the subactions.

Group actions as a whole can be predecessors and successors in control flows, so they provide the ability to synchronize the execution of groups of actions. A control flow whose destination is a group action requires that the predecessor must complete before *any* action within the group action may begin. A control flow whose source is a group action requires that *all* actions within the group action must complete before the successor may begin.

Control flows may also cross the boundary of a group action. These control flows are in addition to any control flows to or from the group action itself. A subordinate action may execute if it has all of its data inputs and all of its control flow inputs and if the group action itself has all of its control flow inputs. Similarly, an external successor of a subordinate action must wait until the subordinate action is complete, but it need not wait for the entire group action to complete (unless it is also a successor of the group action as a whole).

2.20.1.3 Conditional Action

A *conditional action* provides the conditional execution of contained actions depending on the result of test actions.

A conditional action consists of some number of *clauses*, each of which has an embedded test action and body action. Each clause designates an output pin of its test action as the test output. If it evaluates to true, the body is executed. Each clause designates a bank of output pins from its body that have conforming types and multiplicities to the output pins of the conditional action. In this way, the conditional action will produce the same types of output values, regardless of which clause body executes. If the body of a specific clause executes, then the values of the bank of output pins designated by the clause become the values of the output pins of the overall conditional action.

Conditional actions provide the only points of “fan in” of data flow. This fan in within a conditional action does not violate the single assignment principle, since only one of the body actions can execute during any execution of the conditional action, so that each conditional output pin will receive only one value. Also, since exactly one body action must execute, each conditional output pin will always receive a value.

A conditional action has no explicit input pins of its own. However, the inputs for all test actions must come from outside the conditional action. The inputs for a body action may come from either outside the conditional action, or from the outputs of the test action in the same clause (there are no data flows allowed between test and body actions in different clauses). The different test and body actions may or may not use the same input values. Thus, the available inputs of a conditional action include all available inputs of all test actions and the available inputs of body actions that are not already connected to the outputs of test actions.

The output pins of a body action may not be directly connected to input pins outside the body action. The output pins of test actions may not be connected to input pins outside the clause. There are no explicit data flows from the output pins of the clauses to the output pins of the overall conditional action (since data flows can only connect output to input pins). The connection is implicit in the structure of the conditional action itself. Thus the only available outputs of a conditional action are the output pins explicitly owned by the conditional.

The clauses of a conditional action may have noncyclic predecessor-successor relationships among them. Clauses with no predecessor-successor relationships may execute their test actions concurrently. If more than one of these is true, only one body action will execute, but its selection is indeterminate. If the tests are not guaranteed to be exhaustive, the user may provide a default action with a test of "true" as a successor of all other tests. A conditional is not required to have an explicit "else" clause, but the modeler must ensure that at least one test action is true or the model is incorrect.

Note – One exception is allowed to the "exactly one body action must execute" rule. In the case that the conditional action has *no* output pins, then it is allowable for *no* body actions to execute (i.e., for *all* test actions to fail). In this case, any effect of the conditional action is by affecting object memory or the external world. This is equivalent to providing an "else" clause with an empty body.

In cases when the tests in a conditional will be both exhaustive and mutually exclusive by design, the conditional action can be explicitly tagged as being "determinate." In this case, *exactly one* concurrent test action must evaluate to true. Determinism is an assertion by the designer—if it is not correct, the model is ill formed. It can be achieved either by complete, explicit sequencing of the test actions or through the designer's knowledge that tests that are not sequenced are mutually exclusive. The latter case does not imply a need for the run-time system to test that the other tests do indeed evaluate to false—the developer has the responsibility to guarantee mutual exclusion.

2.20.1.4 Loop Action

The final kind of composite action is the *loop action*. The loop action provides for repeated execution of a contained action so long as a test action results in an appropriate value.

A loop action contains a single clause with a test action and a body action. The body action is executed repeatedly as long as the test action yields "true." The test and the body actions have access to the values of a set of "loop variables," which are represented as an ordered set of output pins owned by the loop action.

The loop variables may be connected to input pins of the test and body actions, thus providing the "current values" of the loop variables during a loop iteration. The clause designates a set of output pins within its body subaction. The types of these pins must conform to the types and multiplicities of corresponding loop variables. At the completion of execution of the body action, the values of these pins become the values of the loop variable pins for the next iteration of the loop.

The loop action also has a list of input pins and a list of output pins. Both lists must conform to the loop variable pins and the body output pins. Before the first execution of the loop clause, the values of the loop inputs become the values of the loop variables. During each iteration, the test action of the clause is executed. If its designated test output pin is false, then the values of the loop variable pins become the values of the output pins of the loop action, and the execution of the loop is complete. If the test value is true, the body action of the clause is executed. When it is complete, the body output values become the new values of the loop variables.

The loop variables are not explicitly “reassigned” in the body action. Instead, the input and output pins for the body action hold the “old values” and the “new values” of the loop variables, respectively, during a single iteration. This preserves the single-assignment principle. There are no explicit dataflow connections from the loop input pins to the loop variable pins, from the body output pins to the loop variable pins, or from the loop variable pins to the loop output pins. The dataflow is implicit, based on the structure of the loop action itself.

During execution of a loop action, the test action and the body action have access to output pins outside the loop action. For any one execution of the loop, the value on one of these pins will be fixed during all the iterations of the loop. Thus, the available inputs of a loop action include the input pins explicitly owned by the loop action, the available inputs of the test action that are not connected to loop variables and the available inputs of the body action that are not connected to loop variables or output pins of the test action.

The output pins of the body action may not be directly connected to input pins outside the body action. The output pins of test actions may not be connected to input pins outside the single clause of the loop action. There are no explicit data flows from the output pins of the clause to the loop variables (since data flows can only connect output to input pins). The connection is implicit in the structure of the loop action itself. Thus the only available outputs of a loop action are the output pins explicitly owned by the loop.

2.20.1.5 Local Variables

In addition to data flows, it is also possible to pass data between actions using *local variables*. A local variable is a slot for values shared by the actions within a group but not accessible outside it. The output of one action may be written to a variable and used as the input to a subsequent action, providing an indirect communication path.

The inclusion of variables supports both traditional imperative programming and data-flow programming, as well as mixtures of the two styles. In an imperative style, the result of an action is placed in an object attribute or a local variable, and a subsequent action retrieves it. Because there is no explicit relationship between the actions, they must be sequenced by a control flow. And because there may be many reads and writes to a particular attribute or variable, there is a danger of conflict that must be considered and prevented by the specifier.

In a purely imperative style in which each action is considered to operate on local variables, the UML action model uses data flows to connect read-variable or write-variable actions to the action actually performing the computation on these variables. The data flows in this case can be considered purely formal, and the different actions are otherwise disconnected and communicate by values in variables.

In such an imperative style, local variables can also be used instead of data-flow outputs from conditional actions or loop variables in loop actions. For instance, the body of each clause of a conditional would update the variables that it changes and leave the others unchanged. Since there are no data-flow results from any clause, or from the conditional as a whole, all clauses automatically meet the conditional of having outputs consistent with the conditional outputs.

2.20.1.6 Isolation

Because of the concurrent nature of the execution of actions within and across procedures, it can be difficult to guarantee the consistent access and modification of object memory.

For example, suppose the temperature and pressure attributes of a tank object are being periodically updated by consistent sensor readings. Another procedure may involve reading these attributes in order to compute some current properties of the contents of the tank. But reading both attributes requires two separate read actions. It is possible that a concurrent update of the tank attributes could be interleaved with the two reads, resulting in the temperature and pressure values read not being consistent.

As another example, consider reading a list of account balances, adding a fee amount to each one and then updating each balance. Any concurrent change to a balance before the fee update is complete could result in an inconsistent state. Further, concurrent accesses to the list could result in inconsistencies, with some balances updated but not others.

In order to avoid these problems, it is necessary to *isolate* the effects of a group of actions from the effects of actions outside the group. This is indicated by setting the mustIsolate attribute to “true” on a group action. If a group action is isolated, then any object used by an action within the group cannot be accessed by any action outside the group until the group action as a whole completes. Any concurrent actions that would result in accessing such objects are required to have their execution deferred until the completion of the group action.

In the first example above, if the read actions on the temperature and pressure attributes are wrapped in a group action with mustIsolate set to “true,” then the temperature and pressure values read are assured to be consistent, since no changes can intervene between the two reads. Similarly, if an isolated group is used for the second action, then the update is assured to be consistent, since no action outside the group can change the list until the update is complete.

Note – The term “isolation” is used here in the sense used in traditional transaction terminology. An execution engine may achieve any required isolation using locking mechanisms, or it may simply sequentialize execution to avoid concurrency conflicts. Isolation is different than the property of “atomicity,” which is the guarantee that a group of actions either all complete successfully or have no effect at all. Atomicity generally requires a rollback mechanism to prevent committing partial results. This is beyond the scope of what can be guaranteed by the basic action semantics.

2.20.2 Composite Action Execution

2.20.2.1 Clause Execution

Figure 2-42 shows the life cycle for the execution of a clause. Unlike an action, a clause does not have prerequisites, but it may have other predecessor clauses, which must be clauses in the same conditional action (a loop action has only one clause, which therefore may not have any predecessors). If a clause has one or more predecessors, then it must wait for these to complete. However, a clause only moves out of this *waiting* status if all of its predecessor clauses have completed with “false” outputs.

If all the predecessor clauses of a clause complete with “false” outputs, then the clause’s test action is executed (assuming all data-flow prerequisites of the test action are satisfied). Once the test action has completed, then the clause either passes or fails the test, depending on the result of the test-action execution. The body of a clause is not automatically executed if the clause passes, since multiple clauses may pass in a conditional, but only one body is executed.

The execution of clauses is separately modeled to capture the above special behavior associated with clause predecessors. Since only a clause in a conditional action may have other clauses as predecessors, clause execution need only be explicitly modeled for conditional actions. For loop actions, clauses simply provide a convenient syntactic grouping of the test and body actions of the loop and have no semantic significance.

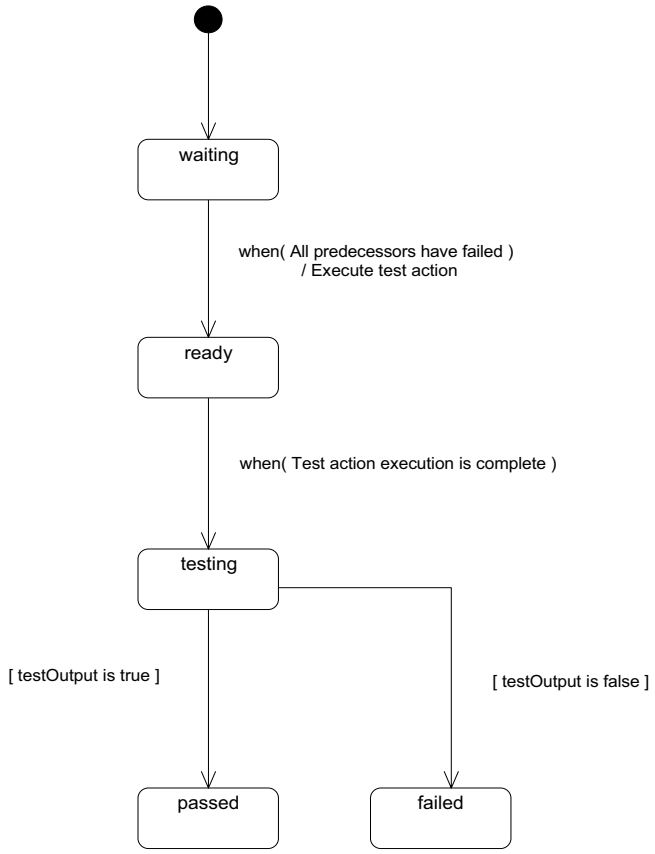


Figure 2-42 Life cycle for clause execution

2.20.2.2 Conditional Action Execution

Figure 2-43 on page 2-237 shows the life cycle for the execution of a conditional action. As with a group action, a conditional action may only have control-flow prerequisites. Once these are satisfied, the clauses of the conditional action may begin executing. Once the clauses have completed execution, then the body of exactly one clause with status *passed* is executed, and the outputs of this body action become the outputs of the conditional action. (The one exception is in the case of a conditional action that has no outputs, in which case it is allowable for no clauses to pass.)

We need to interpret “clause execution complete” carefully, as the tests of some clauses may never be executed, because one of its predecessors was successful or one or more of its predecessors never executed tests themselves. Thus, clause execution may be considered complete when every clause satisfies one of the following:

- The clause has the status *passed*.
- The clause has the status *failed*.

- The clause has the status *waiting* and at least one predecessor with the status *passed* or *waiting*. (A clause with no predecessor clauses cannot meet this condition and therefore must execute in the “first wave.”)

Since the execution of the clause tests must complete before a body is executed, there will be multiple steps in the history of a conditional execution in which it has the status *executing*. These correspond to the similarly named substates of the state *executing* shown in Figure 2-43.

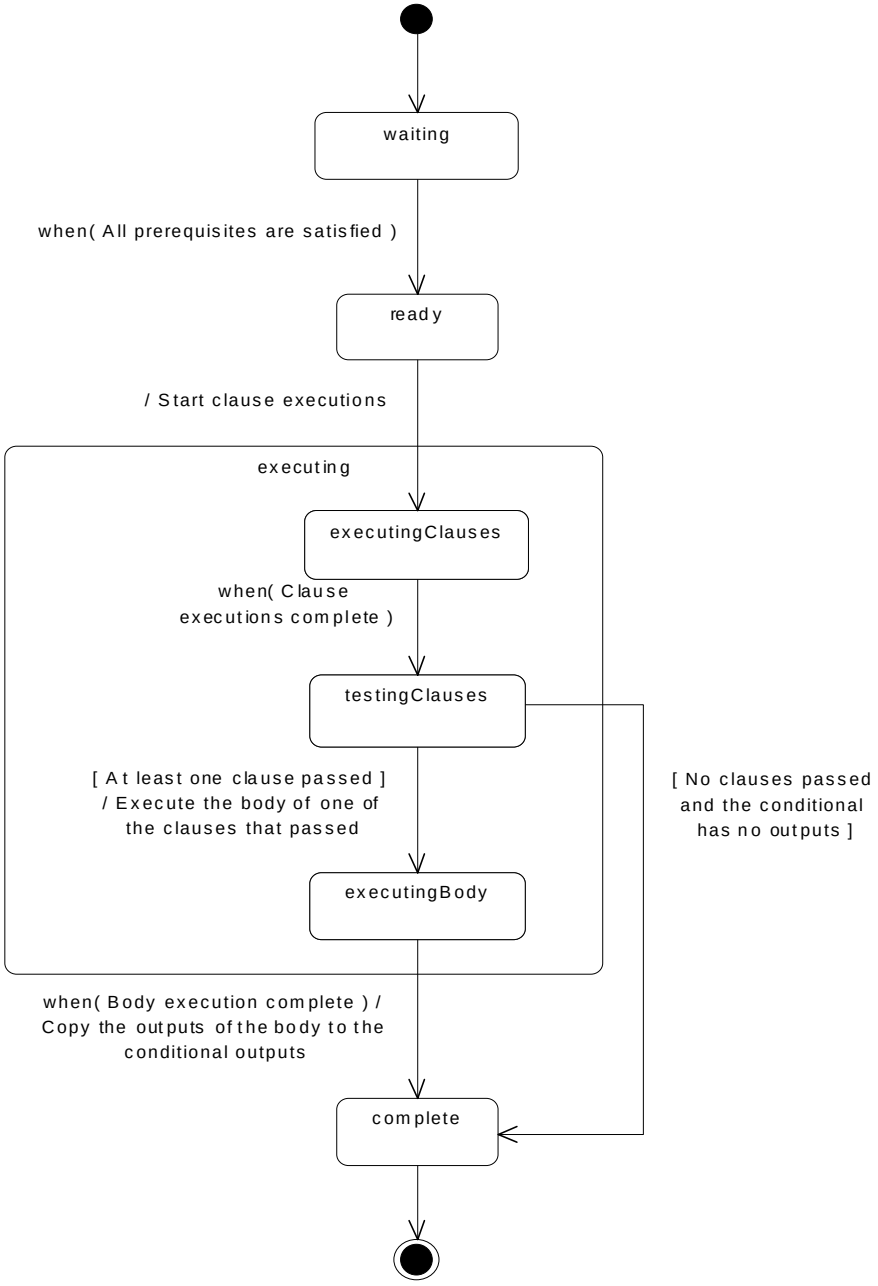


Figure 2-43 Life cycle for conditional-action execution

2.20.2.3 Loop Action Execution

Figure 2-44 on page 2-239 shows the life cycle for the execution of a loop action. A loop action may have both control-flow and data-flow prerequisites, since a loop action may directly own input pins. Once these prerequisites have been satisfied, the values of the loop-action input pins are copied to the loop variables as their initial values and the loop test is executed. If the test output is “true:,” then the loop body is executed, its outputs are copied to the loop variables and the test is executed again. This continues until the loop test fails, in which case the loop variables are copied to the loop outputs and the loop is complete

The iterative nature of a loop execution potentially results in a whole sequence of steps in its execution history in which it has the status *executing*. A loop execution with this status will cycle between the substatuses *executingTest*, *testing*, and *executingBody* (corresponding to the states in Figure 2-44) until the test fails, at which point it will move to status *complete*.

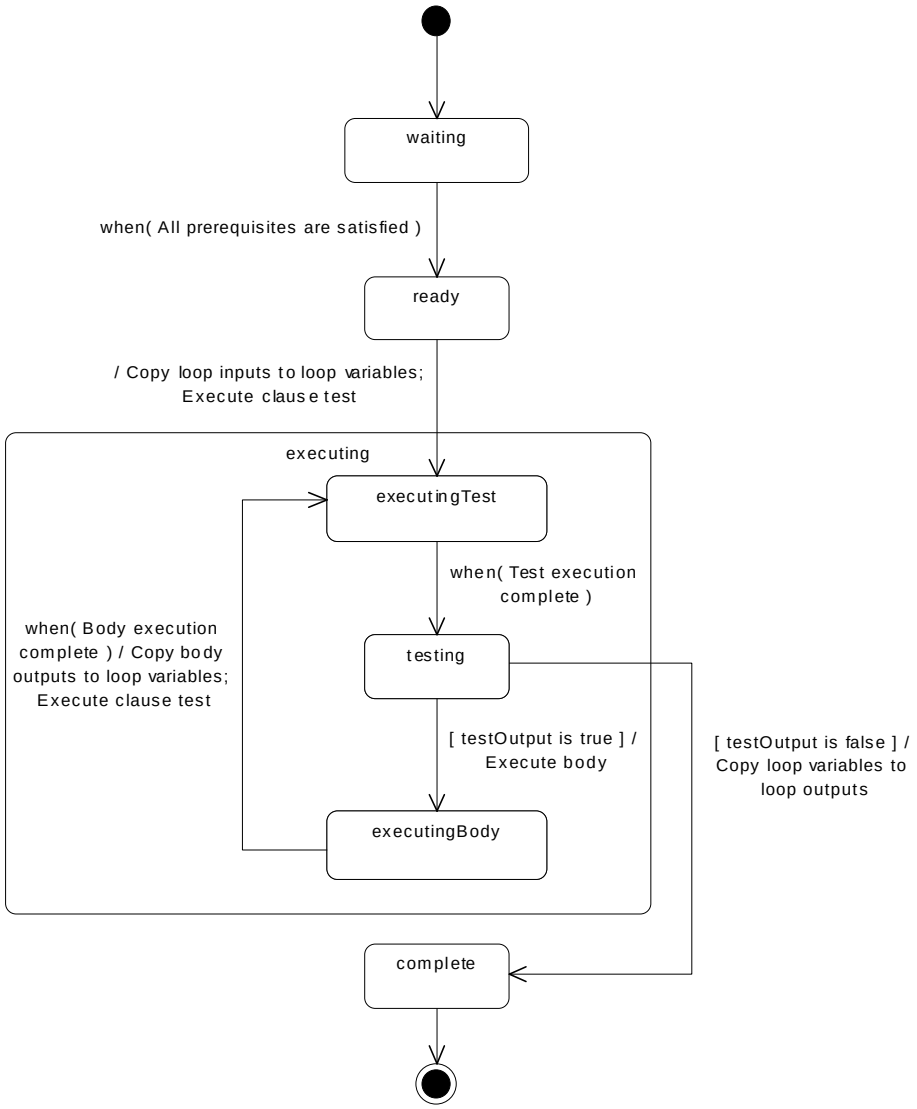


Figure 2-44 The life cycle for loop-action execution

2.20.3 Composite Action Classes

2.20.3.1 Clause

A clause is a part of a conditional or loop action. A clause contains a test action and a body action. Both are arbitrary actions (usually group actions) subject to connectivity constraints described under the various composite actions. The execution of the body action is contingent on the corresponding test action producing a “true” value. The

clause specifies one output pin of the test action, which must have type Boolean; the body action is only executed if this output produces the value “true” after execution of the test action. (Additionally, for a conditional, only one clause body is executed even if more than one of their tests is true.) The outputs of a clause are an ordered subset of the outputs of the body action. No other output of the body action may be connected outside the clause.

Clauses within a conditional action may be linked by a set of (noncyclic) predecessor/successor relationships. The test action of a clause may not execute unless all the predecessors of the clause not only have completed execution, but have also “failed.” These relationships thus act, in effect, as specialized kinds of control flows. Neither the test or body actions of a clause can participate in any normal, explicit control flows.

During execution of an enclosing loop action, a test action may not execute for the first time unless all predecessors of the loop action have executed. It may not execute for a subsequent time unless the previous execution of the body action is complete, that is, there is an implicit control flow between the execution of a body action and the next iteration of the test action.

In a conditional or a loop action, a body action may not execute until its test action completes and yields “true.”

Associations

- test : Action [1..1]
  The action whose Boolean result (designated by testOutput) must be true for execution of the body action to proceed.
- testOutput : OutputPin [1..1]
  The output pin of the test action whose value is the test result. If this value evaluates to “true,” the body action may begin execution.
- body : Action [1..1]
  The action whose execution is contingent on the result of the test action being true.
- bodyOutput : OutputPin [0..*]
  The ordered set of outputs of the body that are considered to be results of the clause.
- predecessorClause : Clause [0..*]
  The set of clauses that must fail before this clause can execute its test action.
- successorClause : Clause [0..*]
  The set of clauses that cannot execute their test actions unless this clause fails.

Well-formedness Rules

[1] The available outputs of the test action of a clause may not be connected to destinations outside the clause.

```
self.test.subactions().availableInput->union(self.body.availableInput)
    ->includesAll(self.test.availableOutput.flow.destination)
```

[2] The available outputs of the body action of a clause may not be connected to destinations outside the body action.

```
self.body.subactions().availableInput->includesAll(self.body.availableOutput.flow.destination)
```

[3] The testOutput of a clause must be an available output of the test action.

```
self.test.availableOutput->includes(self.testOutput)
```

[4] The testOutput pin must conform to type Boolean and multiplicity 1..1.

```
self.testOutput.type = booleanType and
self.testOutput.multiplicity.range->size = 1 and
self.testOutput.multiplicity.range->forAll(r : MultiplicityRange | r.lower = 1 and r.upper = 1)
```

Note – The term “booleanType” is used here to indicate the Boolean enumeration type (instance of Enumeration).

[1] None of the actions within the test action of a clause (if any) may have control-flow connections with actions outside the test action.

```
self.test.allSubactions()->forAll(action : Action |
action.antecedent.predecessor->union(action.consequent.successor)->forAll(a : Action |
a.isSubaction(self.test))
```

[2] None of the actions within the body action of a clause (if any) may have control-flow connections with actions outside the body action.

```
self.body->allSubactions()->forAll(action : Action |
action.antecedent.predecessor->union(action.consequent.successor)->forAll(a:Action |
a.isSubaction(self.body))
```

[3] The test action of a clause may not participate in control flows.

```
self.test.antecedent->isEmpty() and self.test.consequent->isEmpty()
```

[4] The body action of a clause may not participate in control flows.

```
self.body.antecedent->isEmpty() and self.body.consequent->isEmpty()
```

[5] The body outputs of a clause must be available outputs of the body of the clause.

```
self.body.availableOutput->includesAll(c.bodyOutput)
```

[6] There cannot be any cycles in the predecessor/successor relationships among clauses.

```
self.allClauseSuccessors()->excludes(self)
```

Additional Operations

[1] This operation returns the transitive closure of all successors of this clause.

```
allClauseSuccessors() : Set(Clause)
allClauseSuccessors() = self.successorClause-
>union(self.successorClause.allClauseSuccessors()->asSet())
```

[2] This operation returns the available inputs of the test and body actions of a clause that are available outside the clause.

```
clauseInputs() : Set(InputPin)
clauseInputs() = self.body.availableInput->reject(i : InputPin | self.test.availableOutput-
>includes(i.flow.source))
    ->union(self.test.availableInput)
```

Semantics

A clause is executed as part of a conditional action or a loop action. See those actions for a description of how the clause is used in each case.

2.20.3.2 ConditionalAction

A conditional action consists of a set of one or more clauses, exactly one of whose bodies is executed during any execution of the conditional action. If more than one clause has a test that yields "true," exactly one of the corresponding body actions is selected for execution, but it is unspecified which one. (If the conditional action is declared to be determinate, this is an assertion that exactly one concurrent clause test will yield true.) The clause must have an ordered list of output pins that conform to the ordered list of output pins of the conditional. This list must be drawn from accessible outputs of the body action. The only outputs of the conditional action accessible outside it are the output pins directly owned by the conditional action.

Attributes

- isDeterminate : Boolean
  If true, then whenever the conditional action is executed, the execution of exactly one test action must result in "true." (This is an assertion, not an executable property. If the assertion is violated by the action, the model is ill formed.)

Associations

- clause : Clause[1..*]
  The set of clauses contained in the conditional action.

Inputs

none.

Note – There are no explicit input pins. Embedded actions may have input pins that are available inputs of the conditional action.

Outputs

- testOutput: Boolean [1..*]
  [These output pins are referenced by the Clause in the ConditionalActon but are not owned by it. They are owned by test actions within the clause.] A value used to control execution of the conditional body. If the value of a testOutput pin is true at the completion of execution of the test action of a clause, then the body action of the clause is a candidate for execution. Regardless of the number of clauses that produce true values, only one body action will be executed. The manner of selecting among multiple candidates is indeterminate. A test action may begin execution only if all of its predecessorClauses in the conditional action have completed execution and the value of all of their testOutput pins is false. If no clause produces a true test value during an execution of the conditional action, the model is ill formed.
- bodyOutput: T [1..*], where T are user classes
  [These output pins are referenced by the Clause in the ConditionalActon but are not owned by it. They are owned by test actions within the clause.] The output values produced by the conditional body. The list of output pins for each clause must be equal in number and respective types to the list of output pins for the conditional action. At the completion of execution of the body action of the one clause selected for execution, the values on the output pins designated by that clause are copied onto the output pins of the conditional action.
- output: T [0..*], where T are the same user classes as in *bodyOutput*
  [These output pins are owned by the ConditionalAction.] A list of zero or more values that are the only available outputs of the conditional action. At the completion of execution of the conditional action, each output pin has a value equal to the corresponding output pin of the clause that executed.

Note – There are no available outputs of the conditional action except for the explicit output pins of the action itself.

Well-formedness Rules

[1] Each clause of a conditional action must have a number of outputs equal to the number of output pins of the conditional action. Each output of a clause must conform in type and multiplicity to the corresponding output of the conditional.

```
self.clause->forAll(c : Clause |
    c.output->size() = self.outputPin->size()
    and Sequence{1..c.output->size()}->forAll(i : Integer |
        let cOutput : OutputPin = c.output->at(i) in
        let selfOutput : OutputPin = self.outputPin->at(i) in
            (cOutput.type = selfOutput.type
                or cOutput.type.allParents()->includes(selfOutput.type))
                    and cOutput.multiplicity.compatibleWith(selfOutput.multiplicity))
```

[2] The predecessors and successors of a clause in a conditional action must be clauses in the same conditional action.

```
self.clause->includesAll(self.clause.predecessor->union(self.clause.successor))
```

[3] A conditional action owns no input pins.

```
self.inputPin->isEmpty()
```

[4] The available inputs of a conditional action are the union of the available inputs from all the clauses of the conditional action.

```
self.availableInput = self.clause.clauseInputs()
```

[5] The available outputs of a conditional action are the output pins of the conditional action.

```
self.availableOutput = self.outputPin->asSet()
```

[6] There may be no path from a conditional action to any action within the conditional action (where a path is defined as in rule [1] for Action and includes both data flows and control flows).

```
self.allSuccessors()->excludesAll(self.clause.test.allSubactions()-
>union(self.clause.body.allSubactions()))
```

Additional Operations

[1] The nested actions of a conditional action are the test and body actions of its clauses.

```
nestedActions() : Set(Action)
nestedActions() = self.clause.test->union(self.clause.body)->asSet()
```

[2] A conditional action has no subactions.

```
subactions() : Set(Action)
subactions() = Set{}
```

A conditional execution represents the execution of a conditional action.

Semantics

A conditional execution represents the execution of a conditional action.

1. When the control and data flow prerequisites of a conditional action are satisfied, it enables its clauses.
2. Any clause lacking a predecessor clause may begin execution of its test subaction immediately. The test subactions of multiple clauses may execute concurrently. The test subaction of a clause with predecessors may execute if the execution of the test actions of all of its predecessor clauses completed and yielded false values for all test actions.
3. If the test action of any clause yields a true value, the body action of that clause *may* be executed. If multiple test actions yield true values, nevertheless only one body action will be executed, but the choice of which one to execute is not specified.

4. When a body action completes execution, the values on the pins designated by the *bodyOutput* association from the clause containing the body action are copied to the output pins of the conditional action. Note that the list of bodyOutput pins and the list of output pins of the conditional action must match (well-formedness rule on the action).

5. If all clauses have been tested and no test value has been true, and the conditional action has no output pins, then the conditional execution completes execution and the control flow prerequisite is satisfied on any successor actions.If no test value has been true and the conditional action has one or more output pins, then the conditional action is ill formed and the behavior of the condition action is undefined. This represents an error in the model.

2.20.3.3 GroupAction

A group action represents a simple composition of a set of subactions. A group action does not own any pins of its own, but data-flow connections may (generally) be made from actions outside the group to pins owned by actions within the group action. The group action as a whole may participate in control flows and actions within the group action may also participate in control flows with actions outside the group action. There is an implicit control flow from the group action to each action within it; this is important only if there is a control flow to the group action. Similarly, there is an implicit control flow from each action in the group action to the group action; this is important only if there is a control flow from the group action. Finally, a set of local variables can be declared in association with a group action. The group action serves as the scope for use of the variables.

Attributes

- mustIsolate : Boolean
  If true, then the actions in the group execute in isolation.

Associations

- subaction : Action [0..*]
  The set of actions contained in the group.
- variable : Variable [0..*]
  The set of variables declared with the group as their scope.

Inputs

none

Note – A group action has available inputs, but no explicit input pins. Its embedded actions may have input pins.

Outputs

none

Note – A group action has available outputs, but no explicit output pins. Its embedded actions may have output pins.

Well-formedness Rules

[1] A group action does not own any pins.

```
self.outputPin->isEmpty() and self.inputPin->isEmpty()
```

[2] The set of available inputs of a group action is the union of the available inputs of all the subactions of the group action not connected within the group action to an available output of a subaction of the group action.

```
self.availableInput = self.subaction.availableInput->asSet()->reject(i : InputPin |
self.subaction.availableOutput.includes(i.flow.source))
```

[3] The set of available outputs of a group action is the union of the available outputs of all the subactions of the group action.

```
self.availableOutput = self.subaction.availableOutput->asSet()
```

Additional Operations

[1] The nested actions of a group action are its subactions.

```
nestedActions() : Set(Action)
nestedActions() = self.subaction
```

[2] A group action has explicit subactions.

```
subactions() : Set(Action)
subactions() = self.subaction
```

Semantics

Figure 2-45 on page 2-247 shows the life cycle for the execution of a group action. As with any action, a group execution becomes ready when the execution of all its prerequisite actions have completed. Since a group action does not directly own any inputs itself, it can only have control-flow prerequisites. The subactions within a group action may not start executing until the group execution as a whole is ready. A subaction may have its own prerequisite data and control flows that must be satisfied before the subaction may execute. In this sense, the group action is an additional prerequisite for all its subactions.

The group execution maintains its *executing* status until all its subactions have completed. As individual subaction executions within the group execution complete, they trigger any data or control flows attached directly to them, independently of the completion of the group action as a whole. However, control flows with the group action itself as their source will not be triggered until the group execution reaches the *complete* status.

A group action may also act as the scope for a set of local variables. The execution of the group action must therefore also maintain the state of those variables. A variable value associates a sequence of values with the variable, consistent with the multiplicity of the variable. As with pin values, the association of a variable with the instance values is via intermediate variable elements, which allow for the possibility of the variable containing duplicate values, and the ordering of these elements is only significant if the variable is marked as ordered.

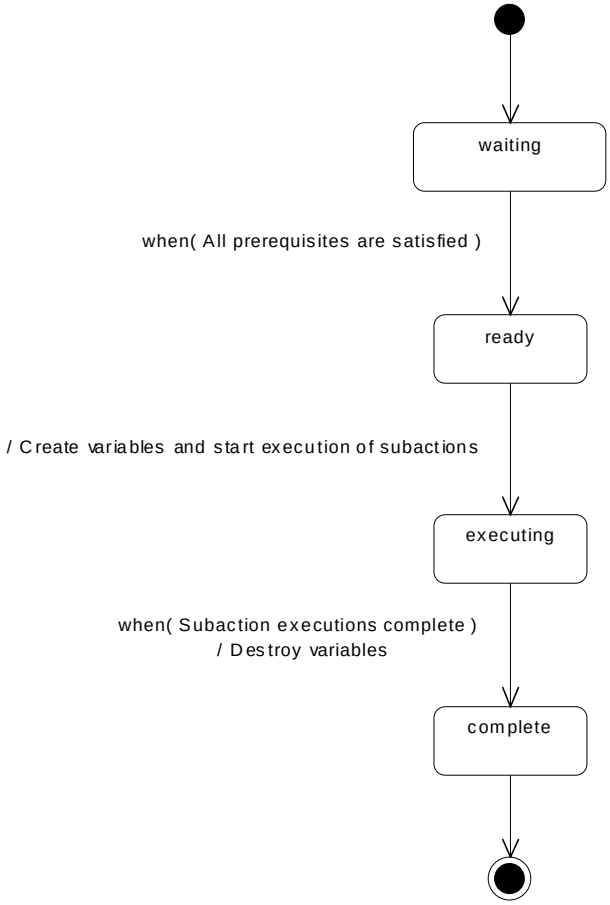


Figure 2-45 Life cycle for group-action execution

1. A group action begins execution when all of its control flow prerequisites are satisfied. (A group action may not have data flow prerequisites.) When a group action begins execution, any variables declared in its scope are created with undefined values and all of its subactions are enabled. Any subactions without control or data flow prerequisites may begin execution immediately. Subactions with control or data flow prerequisites must wait until the prerequisites are satisfied.

2. When all of the subactions have completed execution, the execution of the group action is complete. The values of any variables declared in its scope are destroyed. Any control flow prerequisites in which the group action is a predecessor are satisfied. (A group action may not have data flow outputs.)

Note – The execution model does not currently formalize the semantics of group actions with mustIsolate = true.

2.20.3.4 LoopAction

A loop action contains a single clause whose test action and body action are executed repeatedly as long as the test action yields “true.” A list of output pins acts as “loop variables” for the loop action. The loop variables are not directly available outside the loop. Input pins of the overall loop action provide initial values that are copied into these loop variables before the first iteration of the loop. As output pins, the loop variables may be connected to available inputs of the test and body actions using normal data flows, to provide the “old values” of the loop variables during an iteration. The outputs of the loop clause provide “new values” that are copied to the loop variables at the completion of an iteration. The test is executed after the initial values are copied to the loop variables, and after each execution of the body action. The body action is executed only if the test yields *true*. When the loop terminates, the final values of the loop variables are copied to the regular output pins of the loop action, which are the only available outputs for the loop action as a whole.

Associations

- clause : Clause [1..1]
  The clause that contains the test and body actions for the loop.
- loopVariable : OutputPin [0..*]
  The set of loop-action output pins that act as loop variables for the loop. These are owned directly by the loop action, but they are not available outside the loop.

Inputs

- loopVariableInput: T [0..*], where T are user classes
  [These input pins are owned by the LoopAction itself.] An ordered list of input pins holding values. Each pin can have a different type. The number and irrespective types of the pins must match the loop variable pins. These pins represent the initial values for the loop variables. During the first execution of the subaction, the loop variable pins hold copies of the values on the corresponding loop variable input pins.

Outputs

- result: T [0..*], where T are the classes as in loopVariableInput
  [These output pins are owned by the loop action itself.] An ordered list of output pins holding collections of output values. The number and type of each pin must match the corresponding loop variable input pin and loop variable pin. When the execution of the loop action is complete (because the test condition is false), the

output pins have values equal to the values on the corresponding bodyOutput pins on the final execution of the body action. If the testOutput is false on first execution of the clause (which consequently does not execute the body action), the result pins have values equal to the values on the loopVariableInput pins.

- loopVariable: T [0..*], where T are the same classes as in loopVariableInput
[These output pins are owned directly by the LoopAction. These are internal output pins, visible only within the LoopAction itself.] An ordered list of output pins holding collections of output values. The number and type of each pin must match the corresponding loop variable input pin and result pin. During the first execution of the body action, the loopVariable pins have values equal to the values of the corresponding loopVariableInput pins. On each subsequent execution of the body action, each loopVariable pin has a value equal to the value of the corresponding body action output pin at the completion of the previous execution of the body action. The values of the loopVariable pins are available inputs of actions nested within the clause, including its test action and body action.

- bodyOutput: T [0..*], where T are the same classes as in loopVariableInput
[These output pins are owned by actions nested within the body action of the clause of the LoopAction. They are visible only within the LoopAction itself.] An ordered list of output pins holding collections of output values. The number and type of each pin must match the corresponding loop variable input pin and result pin. At the completion of each execution of the body action, each pin holds a value computed within the body action from its available inputs, including loop variable values. The value of each bodyOutput pin determines the value of the corresponding loopVariable pin on the subsequent execution of the body action and the corresponding result pin if the body action does not execute subsequently. The bodyOutput values are not otherwise available outside the body action itself.

- testOutput : Boolean
[These output pins are owned by an action nested within the test action of the loop action. They are visible only within the LoopAction.] An output pin holding a Boolean value computed within the test action of the clause of the loop action. During each iteration of the loop, the test action executes and the value on the testOutput pin determines whether the execution of the loop action is complete. If the value is false, execution is complete. If the value is true, the body action is executed again.

Well-formedness Rules

[1] The clause of a loop action must have the same number of outputs as the number of loop variables of the loop and each clause output in the ordered list must conform to the corresponding loop variable in type and multiplicity.

```
self.clause.output->size() = self.loopVariable->size()
    and Sequence{1..clause.output->size()}->forAll(i : Integer |
        let clauseOutput : OutputPin = clause.output->at(i) in
            let v : OutputPin = self.loopVariable->at(i) in
                (clauseOutput.type = v.type or clauseOutput.type.allParents()-
>includes(v.type))
                    and clauseOutput.multiplicity.compatibleWith(v.multiplicity))
```

[2] A loop action must have the same number of inputs pins as loop variables and each input pin must conform to the corresponding loop variable in type and multiplicity.

```
self.inputPin->size() = self.loopVariable->size()
    and Sequence{1..self.inputPin->size()}->forAll(i : Integer |
        let p : InputPin = self.inputPin->at(i) in
        let v : OutputPin = self.loopVariable->at(i) in
            (p.type = v.type or p.type.allParents()->includes(v.type)
                and p.multiplicity.compatibleWith(v.multipicity))
```

[3] A loop action must have the same number of output pins as the number of loop variables and each loop variable in the ordered list must conform to the corresponding output pin (in order) in type and multiplicity.

```
self.outputPin->size() =self. loopVariable->size()
    and Sequence{1..self.outputPin->size()}->forAll(i : Integer |
        let p : OutputPin = outputPin->at(i) in
            let v : OutputPin = loopVariable->at(i) in
                (v.type = p.type or v.type.allParents()->includes(p.type))
                    and v.multiplicity.compatibleWith(p.multipicity))
```

[4] The clause of a loop action may not have predecessors or successors.

```
self.clause.predecessor->isEmpty() and self.clause.successor->isEmpty()
```

[5] The set of available inputs of a loop action is the union of the loop-action input pins and the available inputs of the clauses of the loop action that are not connected to loop variables.

```
self.availableInput = self.inputPin->union(self.clause.clauseInputs()
    ->reject(i : InputPin | self.loopVariable->includes(i.flow.source)))
```

[6] The available outputs of a loop action are the output pins of the loop action.

```
self.availableOutput = self.outputPin
```

[7] There may be no path from a loop action to any action within the loop action (where a path is defined as in Rule [1] under Action).

```
self.allSuccessors()->excludesAll(self.clause.test.allSubactions()-
>union(self.clause.body.allSubactions()))
```

Additional operations

[1] The nested actions of a loop action are its test and body actions.

```
nestedActions() : Set(Action)
nestedActions() = Set{self.clause.test, self.clause.body}
```

[2] A loop action has no subactions.

```
subactions() : Set(Action)
subactions() = Set{}
```

Semantics

1. When all control flow and data flow prerequisites of a loop action are satisfied, the execution of the loop begins. All of the values on input pins of the loop execution are copied into a set of loop variable values owned by the loop execution. The execution of the clause owned by the loop action begins.
2. When the execution of a clause begins, its test subaction is executed.
3. When the test action has completed execution, if the test action yields a false value for its *testOutput* pin, the loop execution completes. The values of the loop variables are copied to the loop output pins.
4. When the test action has completed execution, if the test action yields a true value, the body action of the clause is executed. Before execution begins, any control flow conditions, data flow values, or variables in the scope of the action from any previous iterations of the body action are cleared.
5. When the body action has completed execution, the values on the output pins of the body action are copied to the values of the loop variables. Then the test action is executed again.

2.20.3.5 Variable

A variable is the specification of a data slot that represents a local variable shared by the actions within a group. There are actions to write and read variables. These actions are treated as side effect actions, similar to the actions to write and read object attributes and associations. There are no automatic sequencing constraints among actions that access the same variable. Such actions must be explicitly sequenced by control flows (unless their sequencing follows from other constraints anyway).

Any values contained by a variable must conform to the type of the variable and have cardinalities allowed by the multiplicity of the variable. A variable without a type specification can hold any value.

Attributes

- multiplicity : Multiplicity
  A specification of the number of values a variable execution may hold at any one time.

- ordering : OrderingKind
  Indicates whether the set of values held by this variable is to be considered ordered or not.

Associations

- scope : GroupAction [1..1]
  The group action that owns the variable.
- type : Classifier [0..1]
  A classifier specifying the allowed classifiers of values held by the variable. The actual classifier of a value must conform to the type specification of the variable.

Additional Operations

[1] This operation checks whether the given action is within the scope of this variable.

```
isAccessibleBy(a : Action) : Boolean
isAccessibleBy(a) = self.scope.allNestedActions()->includes(a)
```

2.21 Read and Write Actions

Objects, attributes, links, and variables have values that are available to actions. Objects have classifiers and they can be created and destroyed; attributes and variables have values; links can be created and destroyed, have object ends and qualifier values; all of which are available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Read and write actions share the structures for identifying the attributes, links, and variables they access. Objects, attributes, links, and variables each have their own section in this chapter.

Following the philosophy of providing simple actions from which languages can compose powerful constructs, read actions do not modify the values they access, while write actions have the minimum possible effect. For example, creating an object does not execute constructors. Languages requiring higher-level semantics can define higher-level actions from the primitive ones given here.

When an action violates those aspects of static UML modeling that constrain runtime behavior, the semantics is left undefined. For example, an attempt to create an instance of an abstract class is undefined: some languages may make this action illegal, others may create a partial instance for testing purposes. The semantics are also left undefined in situations that require classes as values at runtime. The only exception is minimum multiplicity, which is defined to be equivalent to the lower multiplicity being zero. Runtime situations violating minimum multiplicity do not conform to their model, but are necessary to allow intermediate stages of initializing runtime objects. The modeler must determine when minimum multiplicity should be enforced.

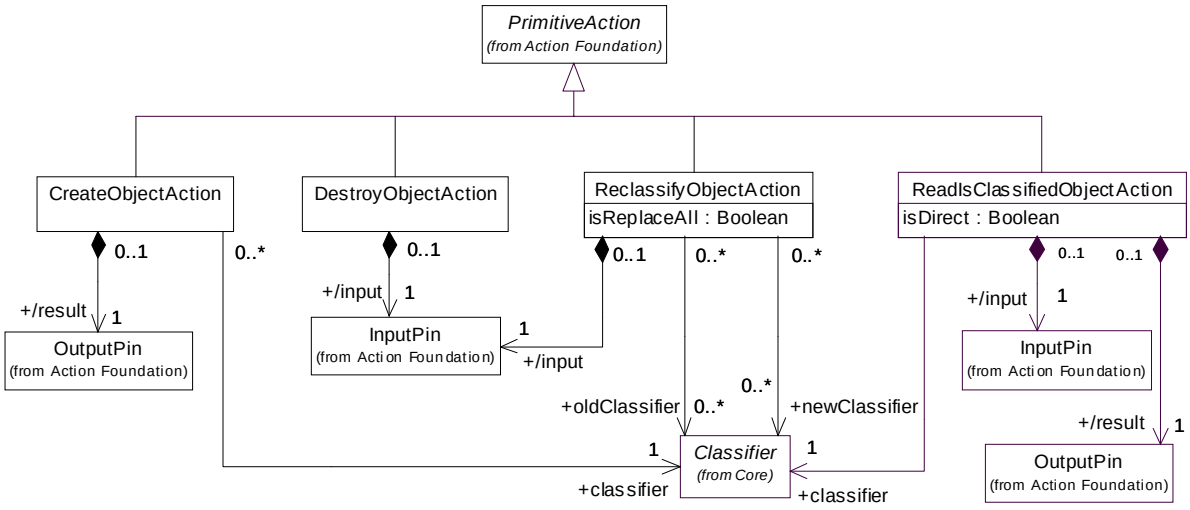


Figure 2-46 Object Action metamodel

2.21.1 Object Actions

The only properties an object has directly are its classes and whether it exists or not. All the other aspects of an object are handled through other elements, such as attributes and associations. This section covers the direct properties of objects.
CreateObjectAction creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime. *DestroyObjectAction* destroys the object on its input pin at runtime. When the object is also a link object, the link is also destroyed with the same semantics as a *DestroyLinkAction*. *ReclassifyObjectAction* adds and removes statically specified classifiers to and from the object given on its input pin at runtime. It supports adding and removing multiple classifiers at a time. No constructors or destructors are executed by the object actions, nor is there any effect on the state machines of the object. It has the option of removing all existing classifiers of the object before new ones are added. *ReadIsClassifiedObject* determines whether an object is classified by a classifier that is specified at modeling time, either as a direct instance or indirect descendant of the classifier. The actions on objects in general are applicable to link objects. See descriptions of classes for more information on their semantics.

2.21.2 Attribute Actions

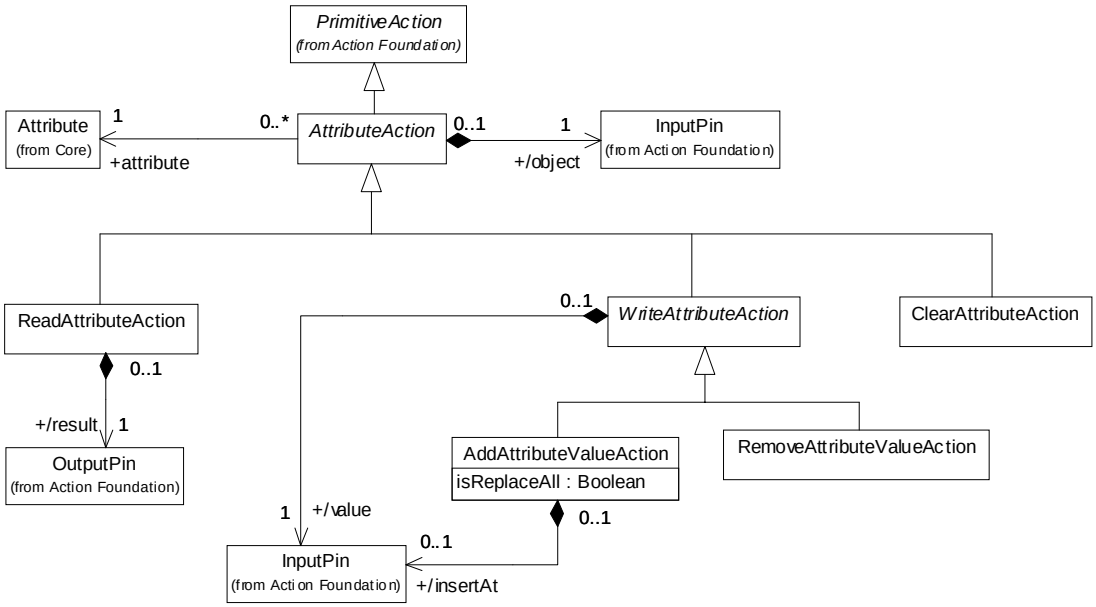


Figure 2-47 Attribute action metamodel

Attributes have values that can be read or written, as modeled in Figure 2-47 on page 2-254. The abstract metaclass *AttributeAction* statically specifies the attribute being accessed. The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified attribute, and the value's multiplicity is 1..1.

When an attribute is read with *ReadAttributeAction*, the values of the attribute of the input object are placed on the output pin of the action. The type and ordering of the output pin are the same as the specified attribute. The multiplicity of the attribute must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the attribute only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the attribute.

Adding a value with *AddAttributeValueAction* has the option of removing existing values of the attribute beforehand, even if the value already exists. Attributes can also have all values removed with no new values added, using *ClearAttributeAction*.

The semantics of adding a value that violates the maximum multiplicity of the attribute is undefined. Removing a value succeeds even when that violates the minimum multiplicity—the same as if the minimum were zero. The same applies to clearing the attribute. The modeler must determine when minimum multiplicity of attributes should be enforced.

Values of an attribute may be ordered or unordered, even if the multiplicity maximum is 1. The insertion point for adding new values to ordered attributes is specified at runtime by an additional input pin. The insertion point is a positive integer giving the position to insert the value, or the special value *unlimited*, to insert at the end. Reinserting an existing value at a new position moves the value to that position. (This works because attribute values are sets.) The insertion point is required for ordered attributes and omitted for unordered attributes.

The semantics of actions that read and write attributes with classifier ownerScope or targetScope is undefined.

The attributes of an object may change over time due to dynamic classification. However, the attribute specified in an attribute action is inherited from a single classifier, and it is assumed that the object passed to an attribute action is classified by that classifier directly or indirectly. The attribute is referred to as a user model element, so it is uniquely identified, even if there are other attributes of the same name on other classifiers.

2.21.3 Association Actions

In the following discussion, “associations” does not include those modeled with association classes, unless specifically indicated. Similarly, a “link” is not a link object unless specifically indicated. The actions on objects in general are applicable to link objects. The term “link end object” or “end object” refers to the object participating in a link at a particular end. The semantics of actions that read and write associations that have any end with classifier targetScope is undefined.

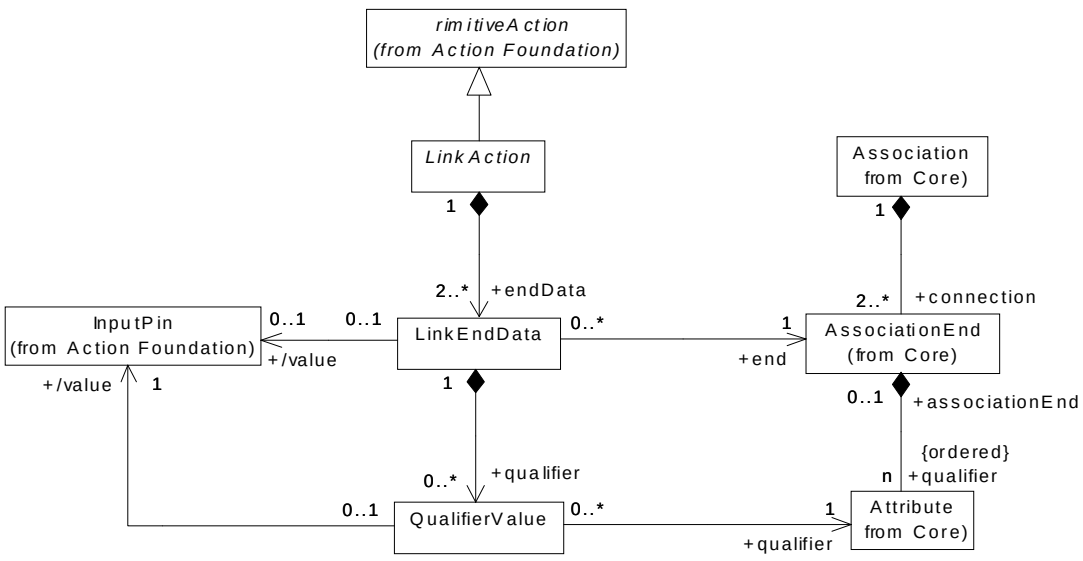


Figure 2-48 Link identification metamodel

2.21.3.1 Identifying a Link

A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, as required. This requires more than one piece of data, namely, the static end in the user model, the object on the end, and the qualifier values for that end. These pieces are brought together around *LinkEndData*. Each association end is identified separately with an instance of the *LinkEndData* class.

For write actions, all association ends must have a corresponding input pin so that all end objects are specified when creating or deleting a link. An input pin identifies the end object by being given a value at runtime. It has the type of the association end and multiplicity of 1..1, since a link always has exactly one object at its ends.

Read actions omit exactly one input pin for an object end. The model, shown in Figure 2-48, therefore abstracts the association to an input to be optional.

When a qualifier must be specified, the input pin for the qualifier attribute has a type given by that qualifier, and multiplicity of 1..1.

2.21.3.2 Navigating Across an Association

Navigation of a binary association requires the specification of the source end of the link. The target end of the link is not specified. When qualifiers are present, one navigates to a specific end by giving objects for the source end of the association and qualifier values for all the ends. These inputs identify a subset of all the existing links of the association that match the end objects and qualifier values. The result is the collection of objects for the end being navigated towards, one object from each identified link.

Figure 2-49 shows the model for a *ReadLinkAction*, generalized for n-ary associations. One of the link-end data must have an unspecified object (the “open” end). The result of the action is a collection of objects on the open end of links of the association, such that the links have the given objects and qualifier values for the other ends and the given qualifier values for the open end. This result is placed on the output pin of the action, which has a type and ordering given by the open end. The multiplicity of the open end must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the open end only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the open end. The semantics are defined only when the open end is navigable, and visible to the host object of the action.

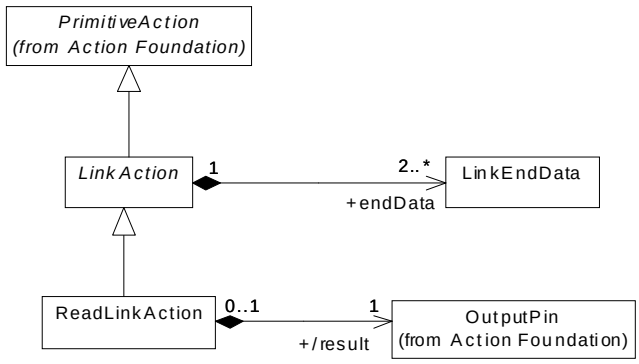


Figure 2-49 Read-link action metamodel

2.21.3.3 Reading Link Objects

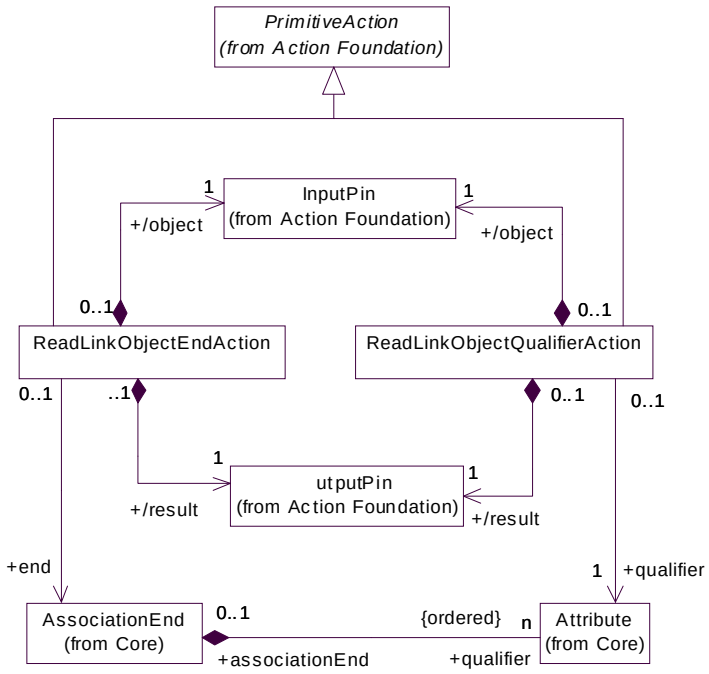


Figure 2-50 Read link object action metamodel

Link objects are instances of association classes. They have their own identity because they are objects, so it is possible to read the end objects and qualifier values of a link object in a more direct fashion than for ordinary links. Figure 2-50 shows the model for link object reading actions.

ReadLinkObjectEndAction reads a link object to retrieve an end object. The association end to retrieve the object from is given statically, and the link object to read is provided on the input pin at run time. *ReadLinkObjectQualifierAction* determines the association end through a qualifier attribute, which UML restricts to being on exactly one association end, and returns the value of the qualifier attribute. For both actions, the input and output pins have multiplicity 1..1. Qualifier attributes must have exactly one value. The type of the output pin is the type of the specified association end for *ReadLinkObjectEndAction*, and the type of the qualifier attribute for *ReadLinkObjectQualifierAction*.

2.21.3.4 Writing Links

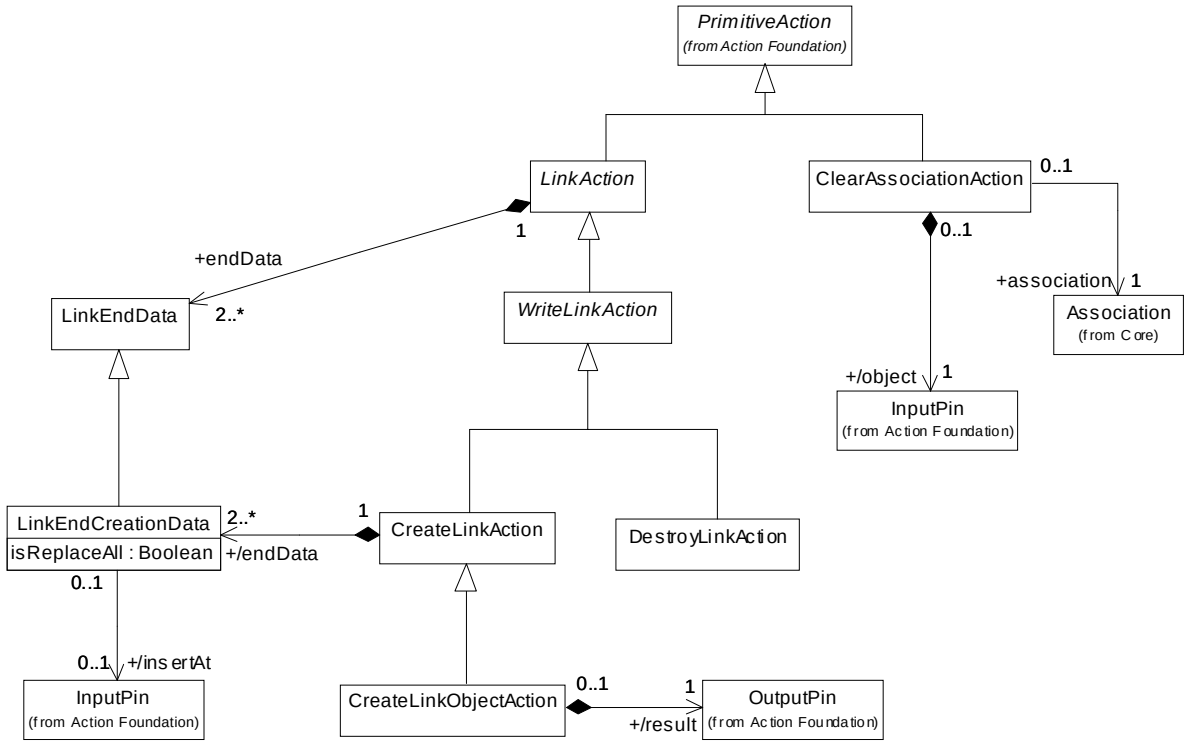


Figure 2-51 Write link action metamodel

Figure 2-51 shows the classes for creating and destroying links. Both inherit the elements for identifying associations from *LinkAction* (see Subsection ”Identifying a Link”). Both inherit well-formedness rules from *WriteLinkAction*.

CreateLinkAction can be used to create links and link objects. In neither case is the created link object returned. This has the happy effect of requiring no change of the action if the association is changed to an association class or vice versa. *CreateLinkAction* uses a specialization of *LinkEndData* called *LinkEndCreationData*, to support ordered associations and for replacing all links at an end. The insertion point is specified at runtime by an additional input pin, which is required for ordered association ends and omitted for unordered ends. The insertion point is a positive integer giving the position to insert the link, or the special value *unlimited*, to insert at the end. Reinserting an existing end at a new position moves the end to that position.

CreateLinkAction also supports the destruction of existing links of the association that connect any of the objects of the new link. When the link is created, this option is available on an end-by-end basis, and causes all links of the association emanating from the specified ends to be destroyed before the new link is created.

CreateLinkObjectAction is exclusively for creating links of association classes. It returns the created link object. *DestroyLinkAction* deletes links, including link objects. *ClearAssociationAction* destroys all links of an association in which an object given at runtime participates.

Creating a link that violates the maximum multiplicities of the association has undefined semantics. The semantics of destroying a link that violates the minimum multiplicities of the association is that same as if the minimum were zero, that is, the link is destroyed. The modeler must determine when minimum multiplicity of associations should be enforced.

2.21.4 Variable Actions

Variables have values that can be read or written, as modeled in Figure 2-52 on page 2-260. The abstract metaclass *VariableAction* statically specifies the variable being accessed. Variable actions can only access variables within the procedure of which the action is a part.

When a variable is read with *ReadVariableAction*, the values of the variable are placed on the output pin of the action. The type and ordering of the output pin are the same as the specified variable. The multiplicity of the variable must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the variable only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the variable.

Adding a value with *AddVariableValueAction* has the option of removing existing values of the variable beforehand, even if the value already exists. Variables can also have all values removed with no new value added, using *ClearVariableAction*.

The semantics of adding a value that violates the maximum multiplicity of the variable is undefined. Removing a value succeeds even when that violates the minimum multiplicity—the same as if the minimum were zero. The same applies to clearing the variable.

Values of a variable may be ordered or unordered, even if the multiplicity maximum is 1. The insertion point for adding new values to ordered variables is specified at runtime by an additional input pin. The insertion point is a positive integer giving the position to insert the value, or the special value *unlimited*, to insert at the end. Reinserting an existing value at a new position moves the value to that position. (This works because variable values are sets.) The insertion point is required for ordered variables and omitted for unordered variables.

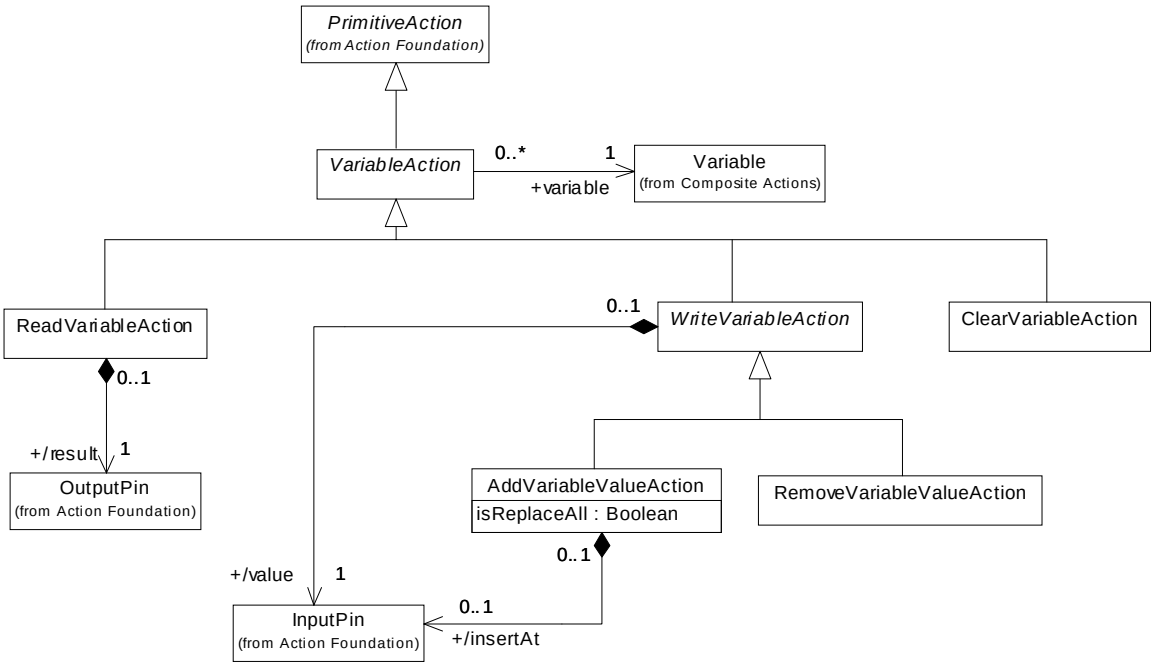


Figure 2-52 Variable action metamodel

2.21.5 Other Actions

Additional actions support navigation to the object hosting the action, reading of the extent of a classifier, and starting the state machines of an object. Figure 2-53 on page 2-261 shows the model for these actions. Every action is ultimately a part of some procedure, which in turn is attached in some way to the specification of a classifier—for example as the body of a method or as part of a state machine. When the procedure executes, it does so in the context of some specific host instance of that classifier. *ReadSelfAction* produces this host instance on its output pin. The type of the

output pin is the classifier to which the procedure is statically associated. *ReadExtentAction* reads the current extent of a given classifier. *StartObjectStateMachineAction* puts the state machines of an object in their top state, if they have not been there already. This can only be used once per object. *CallProcedureAction* starts a procedure passing inputs, and waiting for outputs if it is synchronous.

Note – The extent of a classifier is the set of all instances of a classifier that exist at any one time. It is not generally practical to require that reading the extent produce all the instances of the classifier that exist in the entire universe. Rather, any real execution engine will manage only a limited subset of the theoretical extent of any classifier and may actually manage multiple distributed extents for any one classifier. It is not formally specified in general by the execution semantics which managed extent is actually read by a read-extent action.

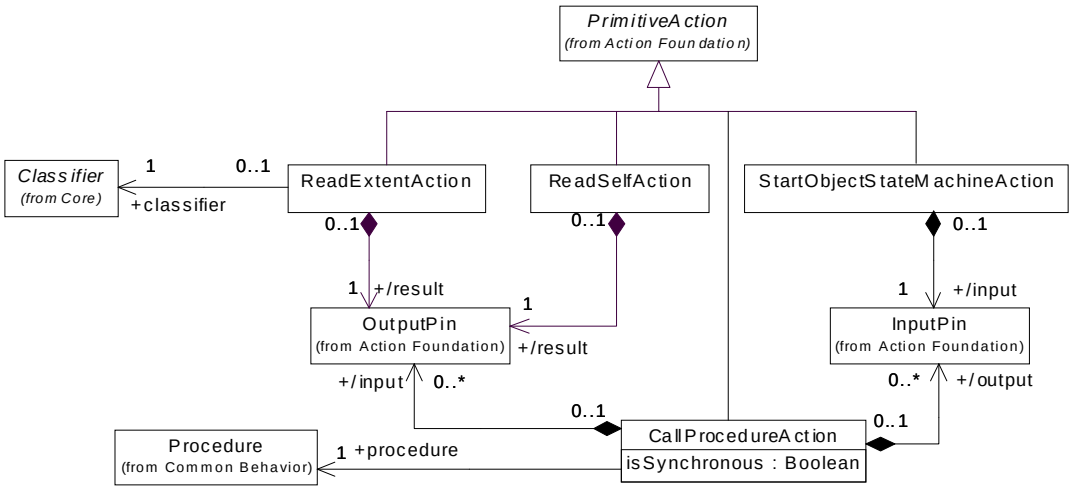


Figure 2-53 Other action metamodel

2.21.6 Additional OCL Operations for Read and Write Actions

Some additional OCL operations are defined for this chapter.

Action

[1] *procedure* operates on *Action*. It returns the procedure containing the action.

```
procedure() : Procedure;
procedure = if self.Procedure->size() > 0 then self.Procedure else self.group.procedure()
endif
```

Procedure

[1] *hostClassifier* operates on *Procedure*. It returns the classifier hosting the procedure. This is the classifier on which the procedure is defined as a method, action in a state machine, sender of a message in a collaboration, or sender of a stimulus in a CollaborationInstance.

```
hostClassifier() : Classifier;
hostClassifier = if self.Method->size() > 0
                    then self.Method.owner
                    else if self.State->size() > 0
                         then self.oclAsType(StateVertex).hostClassifier()
                         else if self.Transition->size() > 0
                              then self.Transition.source.hostClassifier()
                              else if self.Message->size()>0
                                   then self.Message.sender.base
                                   else if self.Stimulus->size>0
                                        then self.Stimulus.sender.classifier
                                        endif
                                   endif
                              endif
                         endif
```

[2] *hostElement* operates on *Procedure*. It returns the "innermost" element in the user model that is hosting the procedure. This will be either a *Method*, *State*, *Transition*, *Message*, or *Stimulus*.

```
hostElement() : ModelElement;
hostElement = if self.Method->size() > 0
                 then self.Method
                 else if self.State->size() > 0
                      then self.State
                      else if self.Transition->size() > 0
                           then self.Transition
                           else if self.Message->size()>0
                                then self.Message
                                else if self.Stimulus->size>0
                                     then self.Stimulus
                                     endif
                                endif
                           endif
                      endif
                 endif
```

StateVertex

[1] *hostClassifier* operates on *StateVertex*. It returns the classifier hosting the state machine of the vertex.

```
hostClassifier() : Classifier;
hostClassifier = if self.top->size() > 0
                 then if self.top.context.oclIsType(Classifier)
                      then self.top.context
                      endif
                 else self.container.hostClassifier()
                 endif
```

2.21.7 Read and Write Action Classes

2.21.7.1 AddAttributeValueAction

This action adds values to attributes. Attributes are potentially multi-valued. It also supports the removal of existing values of the attribute before the new value is added. If the new value already exists, then it is not removed under this option. Otherwise, adding an existing value has no effect.

The semantics is undefined for adding a value that violates the upper multiplicity of the attribute. The semantics is undefined for adding a new value for an attribute with changeability *frozen* after initialization of the owning object.

Adding values to ordered attributes requires an insertion point for a new value using the *insertAt* input pin. The pin is of type *UnlimitedInteger*. A positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values, with the integer one meaning the new value will be first in the sequence. A value of *unlimited* for *insertAt* means to insert the new value at the end of the sequence. The semantics is undefined for a value of zero or an integer greater than the number of existing values. The *insertAt* input pin does not exist for unordered attributes. Reinserting an existing value at a new position moves the value to that position.

Attribute

- isReplaceAll : Boolean [1..1]
  Specifies whether existing values of the attribute of the object should be removed before adding the new value.

Associations

- insertAt : InputPin [0..1]
  (Derived from Action:inputPin) Gives the position at which to insert a new value or move an existing value in ordered attributes. This pin is omitted for unordered attributes.

Inputs

- value : T [1..1], where T is self.attribute.type
  (Inherited from WriteAttributeAction) Value of attribute to add. Its type is the same as the type of the attribute.
- insertAt : UnlimitedInteger [0..1]
  Position at which to insert a new value or move an existing value in ordered attributes. This pin is omitted for unordered attributes.
- object : U [1..1], where U is self.attribute.owner
  (Inherited from AttributeAction) Object to which to add a value. Its type is the same as the type of the owner of the attribute being written.

Outputs

None.

Well-formedness rules

[1] Actions adding a value to ordered attributes must have a single input pin for the insertion point with type *UnlimitedInteger* and multiplicity of 1..1, otherwise the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.attribute.ordering = #unordered
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin = insertAts->asSequence()->first() in
            insertAtPins->size() = 1
            and insertAtPin.type = UnlimitedInteger
            and insertAtPin.multiplicity.is(1,1))
    endif
```

2.21.7.2 AddVariableValueAction

This action adds values to variables. Variables are potentially multi-valued. It also supports the removal of existing values of the attribute before the new value is added. If the new value already exists, then it is not removed under this option. Otherwise, adding an existing value has no effect.

The semantics is undefined for adding a value that violates the upper multiplicity of the variable.

Adding values to ordered variables requires an insertion point for a new value using the *insertAt* input pin. The pin is of type *UnlimitedInteger*. A positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values, with the integer one meaning the new value will be first in the sequence. A value of *unlimited* for *insertAt* means to insert the new value at the end of the sequence. The semantics is undefined for a value of zero or an integer greater than the number of existing values. The *insertAt* input pin does not exist for unordered variables. Reinserting an existing value at a new position moves the value to that position.

Attributes

- isReplaceAll : Boolean [1..1]
  Specifies whether existing values of the variable should be removed before adding the new value.

Associations

- insertAt : InputPin [0..1]
  (Derived from Action:inputPin) The input pin giving the position at which to insert values into an ordered variable. This pin is omitted for unordered variables.

Inputs

- value : T [1..1] , where T is self.variable.type
  (Inherited from WriteVariableAction) Value to add. Its type is the same as the type of the variable.
- insertAt : UnlimitedInteger [0..1]
  The position at which to insert values into an ordered variable. This pin is omitted for unordered variables.

Outputs

None.

Well-formedness rules

[1] Actions adding values to ordered variables must have a single input pin for the insertion point with type *UnlimitedInteger* and multiplicity of 1..1, otherwise the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.variable.ordering = #unordered
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin = insertAts->asSequence()->first() in
            insertAtPins->size() = 1
            and insertAtPin.type = UnlimitedInteger
            and insertAtPin.multiplicity.is(1,1))
    endif
```

2.21.7.3 AttributeAction (abstract)

An attribute action operates on a statically specified attribute of some classifier. The action requires an object on which to act, provided at runtime through an input pin. The semantics is undefined for accessing an attribute that violates its visibility. The semantics is undefined for attributes with ownerScope or targetScope equal to classifier.

Attributes

- isSynchronous: Boolean [1..1]
  Specifies whether the execution of the action waits for the started procedure to finish before continuing.

Associations

- attribute : Attribute [1..1]
  Attribute to be read.
- object : InputPin [1..1]
  (Derived from Action:inputPin) Gives the input pin from which the object whose attribute is to be read or written is obtained. Must be of the same type as the attribute.

Well-formedness rules

[1] The attribute must have an *ownerScope* of *instance*.

```
self.attribute.ownerScope = #instance
```

[2] The attribute must have a *targetScope* of *instance*.

```
self.attribute.targetScope = #instance.
```

[3] The type of the input pin is the same as the type of the object owning the attribute.

```
self.object.type = self.attribute.owner
```

[4] If the action has an input pin, then its multiplicity must be 1..1.

```
self.object→forall(multiplicity.is(1,1))
```

[5] Visibility of attribute must allow access to the object performing the action.

```
let host : Classifier = self.procedure().hostClassifier() in
    self.attribute.visibility = #public
    or host = self.attribute.owner.type
    or (self.attribute.visibility = #protected and
host.allSupertypes→includes(self.owner.type)))
```

2.21.7.4 CallProcedureAction

This action starts a statically-specified procedure, passing inputs, and waiting for outputs if it is synchronous.

Associations

- calledProcedure: Procedure [1..1]
  Procedure to be started.
- input: InputPin [0..*]
  (Derived from Action:inputPin) Gives the input pin from which is obtained the inputs for starting the procedure.

- output: OutputPin [0..*]
  (Derived from Action:outputPin) Gives the output pin from which is obtained the outputs of a synchronously started procedure.

Inputs

- input: T [0..*], where T matches the order and types of the procedure inputs.

Outputs

- output: T [0..*], where T matches the order and types of the procedure outputs.

Well-formedness rules

[1] Asynchronous calls can have no output pins.

```
self.isSynchronous = #false implies self.output->size() = 0
```

[2] The number, type, and order of the input and output pins must be the same as the number, type, and order of the procedure inputs and outputs.

```
self.input->size( ) = self.calledProcedure.argument->size( )
and Sequence {1..self.input->size( )}
        -> forAll (i:Integer |
            let inputi = self.input->at(i) in
            let argi = self.calledProcedure.argument->at(i) in
                inputi.type = argi.type)
and self.output->size( ) = self.calledProcedure.result->size( )
and Sequence {1..self.calledProcedure.result->size( )}
    -   > forAll (i:Integer |
            let outputi = self.output->at(i) in
            let resulti = self.calledProcedure.result>at(i) in
                outputi.type = resulti.type)
```

2.21.7.5 ClearAssociationAction

This action destroys all the links of an association in which a particular object participates. This action has a statically-specified association end. It has an input pin for a runtime object that must be of the same type as at least one of the association ends of the association. All links of the association in which the object participates are destroyed even when that violates the minimum multiplicity of any of the association ends. If the association is a class, then link object identities are destroyed.

Associations

- association : Association [1..1]
  Association to be cleared.
- object : InputPin [1..1]
  (Derived from Action:inputPin) Gives the input pin from which is obtained the object whose participation in the association is to be cleared.

Inputs

- object : T [1..1], where T is the type of at least one of self.association.end.participant
  The object on which to clear the association. It must be of the same type as at least one of the association ends of the association.

Outputs

None.

Well-formedness rules

[1] The type of the input pin must be the same as the type of at least one of the association ends of the association.

```
self.association->exists(connection.participant = self.object.type)
```

[2] The multiplicity of the input pin is 1..1.

```
self.object.multiplicity.is(1,1)
```

2.21.7.6 ClearAttributeAction

This action removes all values of an attribute. All values are removed even when that violates the minimum multiplicity of the attribute. The semantics is undefined if the attribute changeability is *addonly*, or the attribute changeability is *frozen* after initialization of the object owning the attribute, unless the attribute has no values.

Inputs

- object : T [1..1], where T is self.attribute.owner
  (Inherited from AttributeAction) The object on which to clear the attribute. It must be of the same type as the owner of the attribute.

Outputs

None.

2.21.7.7 ClearVariableAction

This action removes all values of a variable. All values are removed even when that violates the minimum multiplicity of the variable.

Inputs

None

Outputs

None.

2.21.7.8 CreateLinkAction

This action creates a link or link object for an association or association class. This action has no output pin, because links are not necessarily values that can be passed to and from actions. When the action creates a link object, the object could be returned on output pin, but it is not for consistency with links. This allows actions to remain unchanged when an association is changed to an association class or vice versa. The semantics of *CreateLinkObjectAction* applies to creating link objects with *CreateLinkAction*.

This action also supports the destruction of existing links of the association that connect any of the objects of the new link. This option is available on an end-by-end basis, and causes all links of the association emanating from the specified ends to be destroyed before the new link is created. If the link already exists, then it is not destroyed under this option. Otherwise, recreating an existing link has no effect.

The semantics is undefined for creating a link for an association class that is abstract. The semantics is undefined for creating a link that violates the upper multiplicity of one of its association ends. A new link violates the upper multiplicity of an end if the cardinality of that end after the link is created would be greater than the upper multiplicity of that end. The cardinality of an end is equal to the number of links with objects participating in the other ends that are the same as those participating in those other ends in the new link, and with qualifier values on all ends the same as the new link, if any.

The semantics is undefined for creating a link that has an association end with changeability *frozen* after initialization of the other end objects, unless the link being created already exists. Objects participating in the association across from an unfrozen end can have links created as long as the objects across from the frozen ends are still being initialized. This means that objects participating in links with two or more frozen ends cannot have links created unless all the linked objects are being initialized.

Creating ordered association ends requires an insertion point for a new link using the insertAt input pin of LinkEndCreationData. The pin is of type *UnlimitedInteger* with multiplicity of 1..1. A pin value that is a positive integer less than or equal to the current number of links means to insert the new link at that position in the sequence of existing links, with the integer one meaning the new link will be first in the sequence. A value of *unlimited* for *insertAt* means to insert the new link at the end of the sequence. The semantics is undefined for value of zero or an integer greater than the number of existing links. The *insertAt* input pin does not exist for unordered association ends. Reinserting an existing end at a new position moves the end to that position.

Associations

- endData : LinkEndCreationData [2..*]
  (Derived from LinkAction:endData) Specifies ends of association and inputs.

Inputs

- endData.value : T [2..*], where T are self.endData.end.participant
  (Inherited from LinkAction) Gives the object at the association end. It is of the same type as the end.
- endData.qualifier.value : U [0..*], where U are self.endData.end.qualifier.type
  (Inherited from LinkAction) Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier attribute. See LinkEndData.
- endData.insertAt : UnlimitedInteger [0..1]
  Gives insertion point for ordered association ends. This pin is omitted for unordered ends.

Outputs

None.

Well-formedness rules

[1] The association cannot be an abstract class.

```
not (if self.association().oclIsKindOf(Classifier) then (true = self.association().isAbstract)
else false endif)
```

[2] The end data must be LinkEndCreationData.

```
self.endData->forall(oclIsKindOf(LinkEndCreationData))
```

2.21.7.9 CreateLinkObjectAction

This action creates a link object. It inherits the semantics of *CreateLinkAction*, except that it operates on association classes to create a link object. The additional semantics over *CreateLinkAction* is that the new or found link object is put on the output pin. If the link already exists, then the found link object is put on the output pin. The semantics of *CreateObjectAction* applies to creating link objects with *CreateLinkObjectAction.*

Associations

- result [1..1] : OutputPin [1..1]
  (Derived from Action:outputPin) Gives the output pin on which the result is put.

Inputs

- endData.value : T [2..*], where T are self.endData.end.participant
  (Inherited from CreateLinkAction) Gives object at association end. It is of the same type as the end.
- endData.qualifier.value : U [0..*], where U are self.endData.end.qualifier.type
  (Inherited from CreateLinkAction) Gives qualifier values of an association end. They are the same type as the qualifier attribute. See LinkEndData.

- endData.insertAt : UnlimitedInteger [0..1]
  (Inherited from CreateLinkAction) Gives insertion point for ordered association ends. This pin is omitted for unordered ends.

Outputs

- result : V [1..1], where V is self.endData.end.association
  The link object created of the same type as the association of the action.

Well-formedness rules

[1] The association must be an association class.

```
self.association().oclIsKindOf(Classifier)
```

[2] The type of the result pin must be the same as the association of the action.

```
self.result.type = self.association()
```

[3] The multiplicity of the output pin is 1..1.

```
self.result.multiplicity.is(1,1)
```

2.21.7.10 CreateObjectAction

This action instantiates a concrete classifier. The new object is created, and the classifier of the object is set to the given classifier. The new object is returned as the value of the action. The action has no other effect. In particular, no constructors are executed, no initial expressions are evaluated, and no state machines transitions are triggered. The new object has no attributes values and participates in no links.

The semantics is undefined for creating objects from abstract classifiers or from association classes.

Associations

- classifier : Classifier [1..1]
  Classifier to be instantiated.
- result : OutputPin [1..1]
  (Derived from Action:outputPin) Gives the output pin on which the result is put.

Inputs

None.

Outputs

- result : T [1..1], where T is self.class
  The created object. The type of the runtime object is the classifier specified for the action.

Well-formedness rules

[1] The classifier cannot be abstract.

not (self.classifier.isAbstract = true)

[2] The classifier cannot be an association class.

not self.classifier.oclIsKindOf(AssociationClass)

[3] The classifier of the result pin must be the same as the classifier of the action.

self.result.type = self.classifier

[4] The multiplicity of the output pin is 1..1.

self.result.multiplicity.is(1,1)

2.21.7.11 DestroyLinkAction

This action destroys a link or a link object. Link objects can also be destroyed with *DestroyObjectAction*.

The link is specified in the same way as link creation, even for link objects. This allows actions to remain unchanged when their associations are transformed from ordinary ones to association classes and vice versa.

Destroying a link that does not exist has no effect. The semantics of *DestroyObjectAction* applies to destroying a link that has a link object with *DestroyLinkAction*.

The semantics is undefined for destroying a link that has an association end with changeability *addonly*, or *frozen* after initialization of the other end objects, unless the link being destroyed does not exist. Objects participating in the association across from an unfrozen end can have links destroyed as long as the objects across from the frozen ends are still being initialized. This means that objects participating in links with 2 or more frozen ends cannot have links destroyed unless all the linked objects are being initialized.

Inputs

- endData.value : T [2..*], where T are self.endData.end.participant
  (Inherited from LinkAction) Gives the object at the association end. It is of the same type as the end.
- endData.qualifier.value : U [0..*], where U are self.endData.end.qualifier.type
  (Inherited from LinkAction) Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier attribute. They are the same type as the qualifier attribute. See *LinkEndData*.

Outputs

None.

2.21.7.12 DestroyObjectAction

This action destroys an object. The object may be a link object, in which case the semantics of *DestroyLinkAction* also applies.

The classifiers of the object are removed as its classifiers, and the object is destroyed. The action has no other effect. In particular, no destructors are executed, no state machines transitions are triggered, and references to the objects are unchanged.

Destroying an object that is already destroyed has no effect.

Associations

- input : InputPin [1..1]
  (Derived from Action:inputPin) The input pin providing the object to be destroyed.

Inputs

- input : T [1..1], where T is any class.
  The object to be destroyed. There is no restriction on its type, other than it must be a class.

Outputs

None.

Well-formedness rules

[1] The multiplicity of the input pin is 1..1.

```
self.input.multiplicity.is(1,1)
```

[2] The input pin has no type.

```
self.input.type->size() = 0
```

2.21.7.13 LinkAction (abstract)

A link action creates, destroys, or reads links. A link is identified by its end objects and qualifier values, if any. The semantics is undefined for links of associations that have targetScope equal to classifier on any end.

Associations

- endData : LinkEndData [2..*]
  Data identifying link ends.

Well-formedness rules

[1] The association ends of the link end data must all be from the same association and include all and only the association ends of that association.

```
self.endData->collect(end) = self.association()->collect(connection))
```

[2] The association ends of the link end data must have *targetScope* of *instance*.

```
self.endData->forall(end.targetScope = #instance)
```

[3] The input pins of the action are the same as the pins of the link end data, qualifier values, and insertion pins.

```
self.inputPin->asSet() =
    let ledpins : Set = self.endData->collect(value)->union(self.endData.qualifier.value) in
        if self.oclIsKindOf(LinkEndCreationData)
        then ledpins->union(self.endData.oclAsType(LinkEndCreationData).insertAt)
        else ledpins
```

Additional operations

[1] *association* operates on *LinkAction*. It returns the association of the action.

```
association();
association = self.endData->asSequence().first().end.association
```

2.21.7.14 LinkEndCreationData

LinkEndCreationData is not an action. It is part of the metamodel that identifies links. It comprises a set of values that identifies one end of a link to be created by *CreateLinkAction* or *CreateLinkObjectAction*. It is required for specifying ordered association ends and for replacing all links at an end. See also CreateLinkAction.

Attributes

- isReplaceAll : Boolean [1..1]
  Specifies whether existing classifiers of the object should be removed before adding the new classifiers.

Associations

- insertAt : InputPin [0..1]
  Specifies where the new link should be inserted for ordered association ends, or where an existing link should be moved to. The type of the input is UnlimitedInteger. This pin is omitted for association ends that are not ordered.

Well-formedness rules

[1] LinkEndCreationData can only be end data for *CreateLinkAction* or one of its specializations.

```
self.LinkAction.oclIsKindOf(CreateLinkAction)
```

[2] Link end data for ordered association ends must have a single input pin for the insertion point with type *UnlimitedInteger* and multiplicity of 1..1, otherwise the link end data has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.end.ordering = #unordered
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin = insertAts->asSequence()->first() in
            insertAtPins->size() = 1
            and insertAtPin.type = UnlimitedInteger
            and insertAtPin.multiplicity.is(1,1))
    endif
```

2.21.7.15 LinkEndData

LinkEndData is not an action. It is part of the metamodel that identifies links. It identifies one end of a link to be read by a *ReadLinkAction* or written by the children of *WriteLinkAction*.

Associations

- end : AssociationEnd [1..1]
  Association end for which this link-end data specifies values.
- value : InputPin [0..1]
  Input pin that provides the specified object for the given end. This pin is omitted if the link-end data specifies an “open” end for reading.
- qualifier : QualifierValue [0..*]
  Specifies qualifier attribute/value pairs of the given end.

Well-formedness rules

[1] The qualifiers include all and only the qualifiers of the association end.

```
self.qualifier->collect(qualifier) = self.end.qualifier
```

[2] The type of the end object input pin is the same as the type of the association end.

```
self.value.type = self.end.participant
```

[3] The multiplicity of the end object input pin must be “1..1”.

```
self.value.multiplicity.is(1,1)
```

[4] The end object input pin is not also a qualifier value input pin.

```
self.value->excludesAll(self.qualifier.value)
```

2.21.7.16 QualifierValue

QualifierValue is not an action. It is part of the metamodel that identifies links. It gives a single qualifier within a link end data specification. See LinkEndData.

Associations

- qualifier : Attribute [1..1]
  Attribute representing the qualifier for which the value is to be specified.
- value : InputPin [1..1]
  Input pin from which the specified value for the qualifier is taken.

Well-formedness Rules

[1] The qualifier attribute must be a qualifier of the association end of the link-end data.

```
self.LinkEndData.end->collect(qualifier)->includes(self.qualifier)
```

[2] The type of the qualifier value input pin are the same as the type of the qualifier attribute.

```
self.value.type = self.qualifier.type
```

[3] The multiplicity of the qualifier value input pin is “1..1”.

```
self.value.multiplicity.is(1,1)
```

2.21.7.17 ReadAttributeAction

This action reads the values of an attribute, in order if the attribute is ordered.

Associations

- result: OutputPin [1..1]
  (Derived from Action:outputPin) Gives the output pin on which the result is put.

Inputs

- object : T [1..1], where T is a self.attribute.owner
  (Inherited from AttributeAction:object) Object whose attribute is to be read. The type of the runtime object is the same as the type of the owner of the attribute.

Outputs

- result : U [1..1], where U is self.attribute.type
  Value of the attribute. It is of the same type as the attribute. The multiplicity of the attribute must be compatible with the multiplicity of this pin.

Well-formedness Rules

[1] The type and ordering of the result output pin are the same as the type and ordering of the attribute.

```
self.result.type = self.attribute.type
and self.result.ordering = self.attribute.ordering
```

[2] The multiplicity of the attribute must be compatible with the multiplicity of the output pin.

```
self.attribute.multiplicity.compatibleWith(self.result.multiplicity)
```

2.21.7.18 ReadExtentAction

This action reads the runtime objects of any classifier that may have instances. It reads all instances, direct and indirect.

Association

- classifier : Classifier [1..1]
  The classifier whose extent is to be read.

Inputs

None.

Outputs

- result : T [1..1], where T is self.classifier
  The runtime objects of the classifier.

Well-formedness Rules

[1] The action has no input pins.

```
self.pinValue→size() = 0
```

[2] The type of the result output pin is the classifier.

```
self.result.type = self.classifier
```

[3] The multiplicity of the result output pin is “0..*”.

```
self.result.multiplicity.is(0,#unlimited)
```

2.21.7.19 ReadIsClassifiedObjectAction

This action tests an object’s classification against a statically specified class. It returns true if the object input to the action is classified by the specified classifier with no intervening classes between the object and the specified classifier. It returns true if the isDirect attribute is false and the object input to the action is classified by the specified classifier, either directly or with intervening classifiers. Otherwise, it returns false.

Attributes

- isDirect : Boolean [1..1]Indicates whether the classifier must directly classify the input object.

Associations

- classifier : Classifier [1..1]
  The classifier for testing classification of the input object.
- input : InputPin [1..1]
  (Derived from Action:inputPin) The input pin on which to test classification.
- result : OutputPin [1..1]
  (Derived from Action:ouputPin) The output pin on which the result is put.

Inputs

- input : T [1..1], where T is any class
  The object on which to test classification. There is no restriction on its type, other than it must be a class.

Outputs

- result : Boolean [1..1]
  The result of testing the classification of the input object.

Well-formedness rules

[1] The multiplicity of the input pin is 1..1.

self.input.multiplicity.is(1,1)

[2] The input pin has no type.

self.input.type->size() = 0

[3] The multiplicity of the output pin is 1..1.

self.result.multiplicity.is(1,1)

[4] The type of the output pin is Boolean.

self.result.type = Boolean

[5] If isDirect is false, then generalization between classifiers must be statically defined.

This rule is not formalized.

2.21.7.20 ReadLinkAction

This action navigates links of an association towards one end. For example, it navigates the link of a binary association from a source object to the objects at the other end of links of the association in which that source object participates. The end towards which navigation occurs is the one that does not have an input pin to take its object (the “open” end). The objects put on the result output pin are the ones participating in the

association at the open end, conforming to the specified qualifiers, in order if the end is ordered. The semantics is undefined for reading a link that violates the navigability or visibility of the open end.

Associations

- result : OutputPin [0..*]
  (Derived from Action:outputPin) The pin on which are put the objects participating in the association at the end not specified by the inputs.

Inputs

- endData.value : T [1..*], where T are self.endData.end.participant
  (Inherited from LinkAction) Gives the object at an association end. It is of the same type as the end. See LinkEndData.
- endData.qualifier.value : U [0..*], where U are self.endData.end.qualifier.type
  (Inherited from LinkAction) Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier attribute. See LinkEndData.

Outputs

- result : V [0..*], where V are self.endData.end.association.connection[open end].participant, where [open end] designates the end of the association not included in the inputs.
  The objects participating in the association at the end not specified by the inputs. The type of the identities are the same as the type of the open association end. The multiplicity of the association end must be compatible with the multiplicity of this pin.

Well-formedness Rules

[1] Exactly one link-end data specification (the “open” end) must not have an end object input pin.

```
self.endData->select(ed | ed.value->size() = 0)->size() = 1
```

[2] The type and ordering of the result output pin are same as the type and ordering of the open association end.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)-
>asSequence()->first().end in
    self.result.type = openend.type
    and self.result.ordering = openend.ordering
```

[3] The multiplicity of the open association end must be compatible with the multiplicity of the result output pin.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)-
>asSequence()->first().end in
    openend.multiplicity.compatibleWith(self.result.multiplicity)
```

[4] The open end must be navigable.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)-
>asSequence()->first().end in
    openend.isNavigable = true
```

[5] Visibility of the open end must allow access to the object performing the action.

```
let host : Classifier = self.procedure().hostClassifier() in
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)-
>asSequence()->first().end in
    openend.visibility = #public
    or self.endData->exists(oed | not oed.end = openend
                                  and (host = oed.end.participant
                                      or (openend.visibility = #protected
                                          and host.allSupertypes-
>includes(oed.end.participant))))
```

2.21.7.21 ReadLinkObjectEndAction

This action reads the object on an end of a link object.

Associations

- end : AssociationEnd [1..1]
  Link end to be read.
- object : InputPin [1..1]
  (Derived from Action:inputPin) Gives the input pin from which the link object is obtained.

Inputs

- object : T [1..1], where T is self.end.association
  Link object being read. The type of the runtime object is the same as the association owning the association end being read.

Outputs

- result : U [1..1], where U is self.end.participant
  Object participating in the link at the specified end.

Well-formedness Rules

[1] The association of the association end must be an association class.

```
self.end.Association.oclIsKindOf(AssociationClass)
```

[2] The ends of the association must all have instance targetScope.

```
self.end.Association.connections->forall(targetscope = #instance)
```

[3] The type of the object input pin is the association class that owns the association end.

```
self.object.type = self.end.Association
```

[4] The multiplicity of the object input pin is “1..1”.

self.object.multiplicity.is(1,1)

[5] The type of the result output pin is the same as the type of the association end.

self.result.type = self.end.participant

[6] The multiplicity of the result output pin is 1..1.

self.result.multiplicity.is(1,1)

2.21.7.22 ReadLinkObjectQualifierAction

This action reads a qualifier value on an end of a link object.

Associations

- qualifier : Attribute [1..1]
  The attribute representing the qualifier to be read.
- object : InputPin [1..1]
  (Derived from Action:inputPin) Gives the input pin from which the link object is obtained.

Inputs

- object : T [1..1], where T is self.end.association
  The link object being read. The type of the runtime object is the same as the association owning the association end of the qualifier attribute being read.

Outputs

- result : U [0..1], where U is self.qualifier.type
  Value of the qualifier attribute on its end of the link, if any. The value has the same type as the qualifier attribute.

Well-formedness Rules

[1] The qualifier attribute must be a qualifier attribute of an association end.

self.qualifier.associationEnd->size() = 1

[2] The association of the association end of the qualifier attribute must be an association class.

self.qualifier.associationEnd.Association.oclIsKindOf(AssociationClass)

[3] The ends of the association must all have instance targetScope.

self.qualifier.associationEnd.Association.connections->forall(targetscope = #instance)

[4] The type of the object input pin is the association class that owns the association end that has the given qualifier attribute.

self.object.type = self.qualifier.associationEnd.Association

[5] The multiplicity of the object input pin is “1..1”.

self.object.multiplicity.is(1,1)

[6] The type of the result output pin is the same as the type of the qualifier attribute.

self.result.type = self.qualifier.type

[7] The multiplicity of the result output pin is “1..1”.

self.result.multiplicity.is(1,1)

2.21.7.23 ReadSelfAction

This action reads the host object of an action. The semantics is undefined for usage in a procedure that does not have a host object.

Associations

- result : OutputPin [1..1]
  (Derived from Action:outputPin) Gives the output pin on which the hosting object is placed.

Inputs

None.

Outputs

- result : T [1..1], where T is the class that owns the procedure containing the action Object hosting the action.

Well-formedness Rules

[1] The action must be contained in a procedure that has a host classifier.

self.procedure().hostClassifier()->size() = 1

[2] If the action is contained in a procedure that is acting as the body of a method, then the operation of the method must have an ownerScope of instance.

```
let hostelement : Element = self.procedure().hostElement() in
    not hostelement.oclIsKindOf(Method)
    or hostelement.oclAsType(Method).specification.ownerScope = #instance
```

[3] The type of the result output pin is the host classifier.

self.result.type = self.procedure().hostClassifier()

[4] The multiplicity of the result output pin of a read-self action is “1..1”.

self.result.multiplicity.is(1,1)

2.21.7.24 ReadVariableAction

This action reads the values of a variable, in order if the variable is ordered.

Associations

- result : OutputPin [1..1]
  (Derived from Action:outputPin) Gives the output pin on which the values of the variable are placed.

Inputs

None.

Outputs

- result : T [0..*], where T is self.variable.type
  Value of the variable. The type of the value is the same as the type of the variable. The multiplicity of the variable must be compatible with the multiplicity of this pin.

Well-formedness Rules

[1] The type and ordering of the result output pin of a read-variable action are the same as the type and ordering of the variable.

```
self.result.type =self.variable.type
and self.result.ordering = self.variable.ordering
```

[2] The multiplicity of the variable must be compatible with the multiplicity of the output pin.

```
self.variable.multiplicity.compatibleWith(self.result.multiplicity)
```

2.21.7.25 ReclassifyObjectAction

This action changes classifiers for an object. The object input to the action is classified by its existing classifiers plus the new classifiers and minus the old classifiers statically specified by the action. It also supports the removal of existing classifiers of the object before the new classifiers are added. The action has no other effect. In particular, the identity of the object is preserved, no constructors or destructors are executed and no initial expressions are evaluated. New classifiers replace existing classifiers in one action, so that attribute values and links are not lost by intermediate stages of classification when the old and new classifiers have attributes and associations in common.

Adding a classifier that duplicates one already existing, or removing a classifier that is not there, has no effect. Adding and removing the same classifiers has no effect.

States are preserved for state machines that are in common before and after the action. New state machines are not started. Removed state machines behave as if the object were deleted.

The semantics is undefined if any of the new classifiers are abstract. The semantics is undefined if all classifiers are removed from a runtime object.

Attributes

isReplaceAll : Boolean [1..1]
Specifies whether existing clasifiers of the object should be removed before adding the new classifiers.

Associations

- input : InputPin [1..1]
  (Derived from Action:inputPin) Gives the object to be reclassified.
- newClassifier : Classifier [0..*]
  Classifiers to add to the classifiers of the object.
- oldClassifiers : Classifier [0..*]
  Classifiers to remove from classes of the object.

Inputs

- input : T [1..1], where T is any class
  Object to be reclassified.

Outputs

None.

Well-formedness rules

[1] None of the new classifiers may be abstract.

```
not self.newClassifier->exists(isAbstract = true)
```

[2] The multiplicity of the input pin is 1..1.

```
self.input.multiplicity.is(1,1)
```

[3] The input pin has no type.

```
self.input.type->size() = 0
```

2.21.7.26 RemoveAttributeValueAction

This action removes values from attributes. Attributes are potentially multi-valued. Removing a value succeeds even when it violates the minimum multiplicity. Removing a value that does not exist has no effect.

The semantics is undefined for removing an existing value for an attribute with changeability addonly. The semantics is undefined for removing an existing value of an attribute with changeability frozen after initialization of the owning object.

Inputs

- value : T [1..1], where T is self.attribute.type
  (Inherited from WriteAttributeAction) Value of attribute to remove or remove. Its type is the same as the type of the attribute.

- object : U [1..1], where U is self.attribute.owner
  (Inherited from AttributeAction) Object from which to remove the attribute value. Its type is the same as the type of the owner of the attribute being modified.

Outputs

None.

2.21.7.27 RemoveVariableValueAction

This action removes values from variables. Variables are potentially multi-valued. Removing a value succeeds even when that violates the minimum multiplicity. Removing a value that does not exist has no effect.

Inputs

- value : T [1..1], where T is self.variable.type
  (Inherited from WriteVariableAction) Value to remove. Its type is the same as the type of the variable.

Outputs

None.

2.21.7.28 StartObjectStateMachineAction

This action puts the state machines of an object in their top states, if they have not been there already. This can only be used once per object. The action has no effect if the object does not have a state machine.

Associations

- input : InputPin [1..1]
  (Derived from Action:inputPin) The object on which to start the state machines.

Inputs

- input : T [1..1], where T is any user class that has a state machine
  Object on which to start the state machines.

Outputs

None.

Well-formedness rules

None

2.21.7.29 VariableAction (abstract)

A variable action operates on a statically specified variable.

Associations

- variable : Variable [1..1]
  Variable being accessed.

Well-formedness rules

[1] The action must be in the scope of the variable.

```
self.variable.isAccessibleBy(self)
```

2.21.7.30 WriteAttributeAction (abstract)

A write attribute action operates on an attribute of an object to modify its values. It has an input pin on which the value that will be added or removed is put. Other aspects of write attribute actions are inherited from *AttributeAction*.

Associations

- value : InputPin [1..1]
  (Derived from Action:inputPin) Value to be added or removed from the attribute.

Well-formedness rules

[1] The type input pin is the same as the owner of the attribute.

```
self.value.type = self.attribute.owner
```

[2] The multiplicity of the input pin is 1..1.

```
self.value.multiplicity.is(1,1)
```

2.21.7.31 WriteLinkAction (abstract)

A write link action creates or destroys links.

Well-formedness rules

[1] All end data must have exactly one input object pin.

```
self.endData.forall(value->size() = 1)
```

2.21.7.32 WriteVariableAction (abstract)

A write variable action modifies a statically specified variable.

Associations

- value : InputPin [1..1]
  (Derived from Action:inputPin) Value to be added or removed from the variable.

Well-formedness rules

[1] The type of the input pin is the same as the type of the variable.

self.value.type =self.variable.type

[2] The multiplicity of the input pin is 1..1.

self.value.multiplicity.is(1,1)

2.22 Computation Actions

These actions transform a set of input values to a set of output values. These actions do not read or write attribute or link values, nor do they otherwise interact with object memory or other objects, so their control is entirely self-contained. Consequently, they embody mathematical functions. These actions supply the primitive functions out of which computations are constructed.

2.22.1 Computation actions

Computation actions evaluate various mathematical functions. They take input values and produce output values. The output values depend only on the input values and not on the state of the memory or the state of the control.

This specification does not define a set of primitive functions. Rather, we assume that any particular implementation of the action semantics will define a set of primitive functions, presumably using a profile. For modeling purposes, users must be able to define new primitive functions, but the mechanisms of the definition are outside of the UML and action semantics. This specification does require each primitive function to have a name and lists of input and output types.

This approach has the drawback that users must agree on the set of primitive functions, but different groups of users will prefer different sets of functions anyway, so little is lost in not providing a default set of functions.

The following model shows the computation action classes.

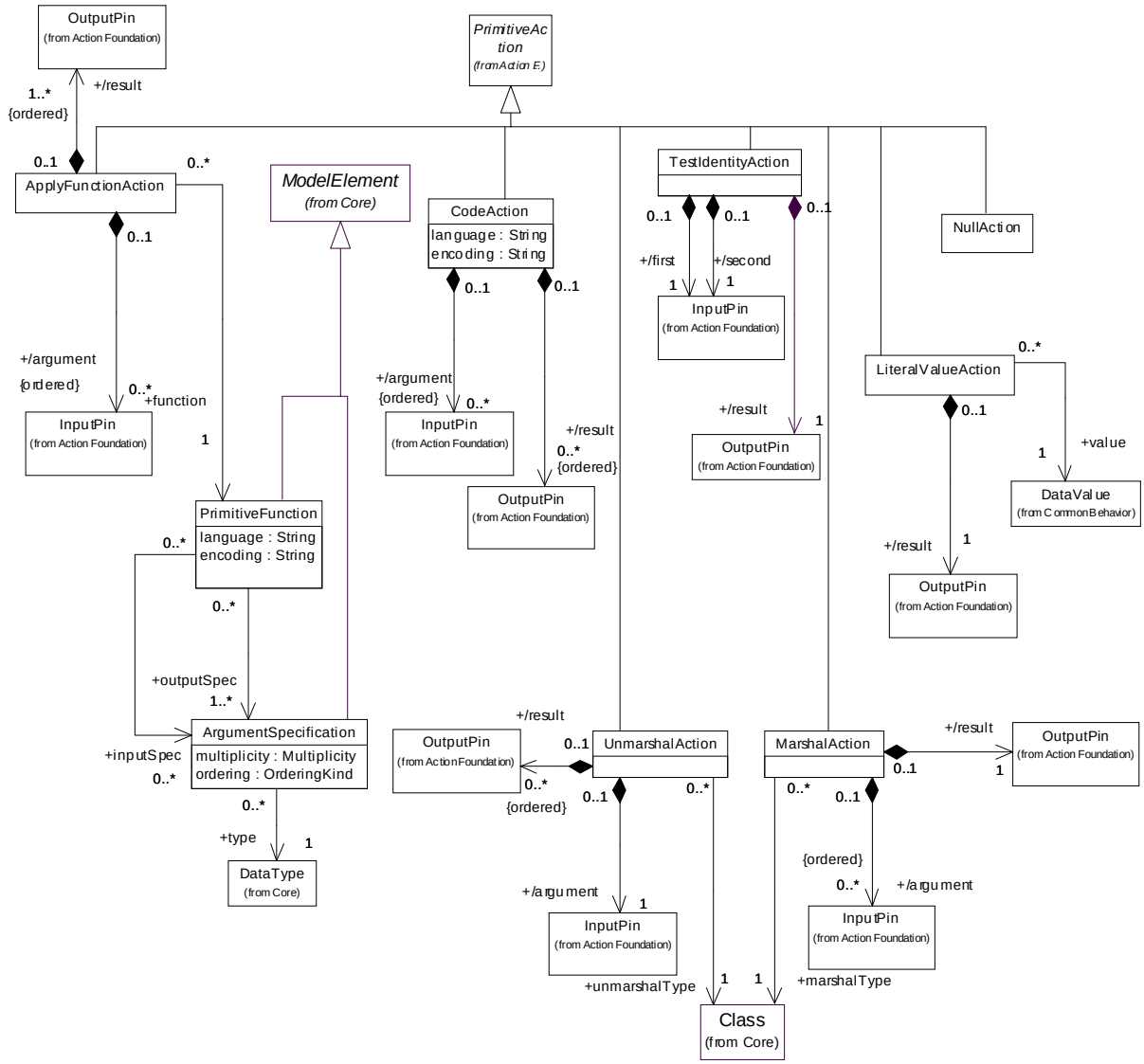


Figure 2-54 Computation classes metamodel

Literal value actions are broken out as a separate kind of action. These are to be regarded as special kinds of primitive functions, with zero inputs and one output.

Code actions are included here, although they might possibly have external effects and therefore not be pure mathematical functions. Because they are explicitly implementation-dependent, it is hard to say much more about them within UML itself.

2.22.2 Computation Classes

2.22.2.1 ApplyFunctionAction

This action computes a primitive predefined mathematical function that depends only on the input values, with no access to object memory or to any objects. The execution and results of this action depend only on the function and the input values. There are absolutely no side effects of this action and it therefore cannot conflict with anything. All it does is produce result values using a mathematical function.

New primitive functions may be defined (outside of UML) as mathematical functions of input values to output values. All usual primitive operations should be considered as primitive functions (e.g., addition, nand, square root, finding a substring, Bessel function, etc.).

A list of defined primitive functions may be supplied as part of a modeling profile. UML does not provide a mechanism to define primitive functions, as their definition is outside its scope. It is expected that users will agree on environments containing such definitions.

Attributes

None

Associations

- function: PrimitiveFunction[1..1]
  The function to execute

Inputs

- argument: T [0..*], where T = self.function.inputSpec.type
  The input values

Outputs

- result: U[0..*], where U = self.function.outputSpec.type
  The output values

Well-formedness rules

[1] The number and types of the input argument and output result pins must be compatible with the number and types of the parameters of the function.

```
self.input_argument()->size( ) = self.function.inputType->size( ) and
Sequence {1..self.input_argument()->size( )} -> forAll (i:Integer |
        let argumenti = self.input_argument (i)
            let inparameteri = self.function.inputType->at(i)
                argumenti.type.isCompatibleWith (inparameteri.type)) and
self.output_result()->size( ) = self.function.outputType->size( ) and
Sequence {1..self.output_result()->size( )} -> forAll (i:Integer |
    let resulti = self.output_result (i) in
    let outparameteri = self.function.outputType->at(i) in
    outparameteri.type.isCompatibleWith (resulti.type))
```

Therefore there are no functions on collections, but operations on collections can be constructed as actions at a higher level out of functional pieces.

Functions have no effect on and may not access object state.

The definition of the mathematical functions is outside of UML.

Semantics

1. When all of the control and data flow prerequisites of the action have been satisfied, the argument values are obtained from the input pins and made available to the computation.
2. The result values are computed from the input values according to the given function. During the execution of the computation, there is no communication or interaction with the rest of the system. The amount of time to compute the results is unspecified. Some primitive functions may raise exceptions for certain input values, in which case the computation is terminated.
3. The result values are placed on the output pins of the action execution and the execution of the primitive function action is complete; or, the primitive function execution may raise an exception, in which case no output values are produced.

2.22.2.2 ArgumentSpecification

(Not an action) Specification of an input or output argument of a primitive function.

Attributes

- multiplicity: Multiplicity
  The range of allowed cardinality of the values on the argument.
- ordering: OrderingKind
  Whether and how multiple values on an argument are arranged.

Associations

- type: DataType [1..1]
  The data type that values on the argument must conform to.

Well-formedness rules

None

2.22.2.3 CodeAction

A code action performs an action that is defined outside of UML. This may involve external interactions, so it is not a mathematical transformation like a primitive function action. The action may have inputs and outputs. Code actions must not alter object memory state, although it is hard to prevent such things. The semantics of a code action that alters object memory are undefined, together with the semantics of all subsequently executed actions.

Attributes

- language: String-- the language in which the code action is specified.
- encoding: String-- a string that identifies the action in the given language. Not meaningful to UML.

Associations

none

Inputs

- argument: T [*], where T is implementation dependent on self.encoding
  The input values

Outputs

- result: U [*], where U is implementation dependent on self.encoding
  The output values

Well-formedness rules

Clearly each kind of code action has constraints on its input and output values, but as the whole purpose of the code action is to do something that is outside of UML, it is not possible to specify the constraints within UML.

Semantics

1. When all of the control and data flow prerequisites of the action have been satisfied, the input values are obtained from the input pins and made available to the computation.

2. During the execution of the code action, the execution may access or change values in the rest of the system or communicate with the run-time execution engine or devices in the real world. Such behavior is inherently implementation dependent and may be different or meaningless in different implementation environments.

3. At such time as specified by the implementor of the code action, the output values (if any) are placed on the output pins of the action execution and the execution is complete.

2.22.2.4 LiteralValueAction

Generates a literal value on the output pin. The value can be of any pure data type.

Attributes

None

Associations

- value: DataValue
  The literal value produced by the action.

Inputs

none

Outputs

- result: T, where T = self.value.type
  The output value, equal to the literal value.

Well-formedness rules

self.output_result().type = self.value.type

Semantics

1. When all of the control prerequisites of the action have been satisfied, the value specified by the literal is placed on the output pin of the action execution, and the action execution satisfies the appropriate control and data flow prerequisites.

2.22.2.5 MarshalAction

Creates an object whose attribute values are initialized from the inputs.

Attributes

None

Associations

- marshalType: Class[1.1]
  The type of object to create.

Inputs

- argument: T [0..*], where T = self.marshalType.attribute.type for attribute
  The initial attribute values of the newly created object.

Outputs

- result: U [1..1], where U = self.marshalType
  The newly created, initialized object.

Well-formedness rules

[1] The argument types must match the attribute types of the marshal type.

```
self.input_argument()->size( ) = self.marshalType.allAttribute->size( ) and
Sequence {1..self.input_argument()->size( )} -> forAll (i:Integer |
        let argumenti = self.input_argument (i) in
            let inparameteri = self.marshalType.allArgument()->at(i) in
                argumenti.type.isCompatibleWith (inparameteri.type))
```

Semantics

1. When all of the control and data flow prerequisites of the action have been satisfied, the input values are obtained from the input pins and made available to the computation. An object of the type specified by *marshalType* is created and its attributes are initialized with the input values. The identity of the object is placed on the output pin of the action execution, and the execution of the action is complete and satisfies appropriate control and data flow prerequisites.

2.22.2.6 NullAction

An action that has no effect.

Attributes

none

Associations

none

Inputs

none

Outputs

none

Well-formedness rules

none

Semantics

1. When all of the control prerequisites of the action have been satisfied, the execution of the null action is complete, and the action execution satisfies any control prerequisites for which it is a predecessor.

2.22.2.7 PrimitiveFunction

(Not an action) Describes the signature of a primitive function, that is, a mathematical function that produces output values from input values without any internal action semantics substructure. The manner of specifying functions is outside the scope of action semantics and must be expressed in some external language.

Attributes

- name: Name (inherited from ModelElement)
  The name of the function.
- language: String
  The language in which the function is specified.
- encoding: String
  The specification of the function in the given language.

Associations

- inputSpec: ArgumentSpecification [0..*]
  Specification of the input values of the function.
- outputSpec: ArgumentSpecification [1..*]
  Specification of the output values of the function.

Well-formedness rules

None

2.22.2.8 TestIdentityAction

Produces true if the two input values are the same identity, false if they are not. Defined only on object identities.

Attributes

None

Associations

none

Inputs

- first: T, where T is any class-- one object identity
- second: U, where U is any class-- another object identity

Outputs

- result: Boolean
  True if first and second input values are the same identity.

Well-formedness rules

None

Semantics

1. When all of the control and data flow prerequisites of the action have been satisfied, the input values are obtained from the input pins and made available to the computation. If the two input values represent the same object identity (regardless of any implementation-level encoding), the value *true* is placed on the output pin of the action execution; otherwise the value *false* is placed on the output pin. The execution of the action is complete and satisfies appropriate control and data flow prerequisites.

2.22.2.9 UnmarshalAction

Breaks an object of a known type into outputs, each of which is equal to the value of one of the object's attribute values.

Attributes

None

Associations

- unmarshalType: Class
  The type of object accepted by the action.

Inputs

- argument: T, where T = self.unmarshalType-
  The identity of an input object, which must be of the given unmarshalType or a descendant of it.

Outputs

- result: U [0..*], where U = self.unmarshalType.attribute.type
  Values equal to the attribute values of the object (according to the unmarshalType).

Well-formedness rules

[1] The result types must match the attributes of the unmarshal type.

```
self.output_result()->size( ) = self.unmarshalType.attribute->size( ) and
Sequence {1..self.output_result()->size( )} -> forAll (i:Integer |
    let resulti = self.output_result (i) in
    let outparameteri = self.function.outputType->at(i) in
    outparameteri.type.isCompatibleWith (resulti.type))
```

Semantics

1. When all of the control and data flow prerequisites of the action have been satisfied, the input value is obtained from the input pins and made available to the computation. An object of the type specified by *marshalType* is required as the input value. The values of the various attributes of the object are placed on the respective output pins of the action execution, one attribute value per pin. The execution of the action is complete and satisfies appropriate control and data flow prerequisites.

2.23 Collection Actions

A collection action applies another action (a “subaction”) to collections of elements. Collection actions avoid explicit indexing and extracting of elements from collections, and the consequent overspecification of control.

There are four kinds of collection actions, as follows:

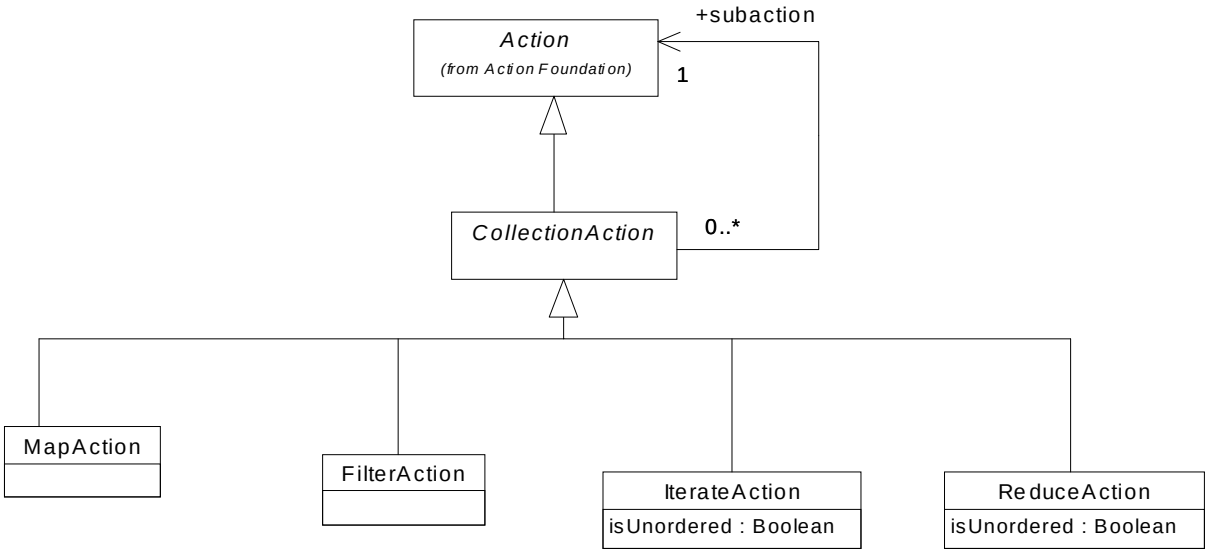


Figure 2-55 Collection actions

- Filter - The *filter* action applies a subaction that determines whether to include an element of a collection in a new output collection, effectively filtering the input collection according to some criterion. The subaction can be applied to the elements in parallel. An example is a subaction that determines whether an account is above a certain balance.
- Iterate - The *iterate* action applies a subaction repeatedly to each of the elements in a collection, accumulating the effects in "loop variables." Because the result of each subaction is accumulated, the subaction must be applied to each element in the collection in sequence. An example is paying creditor accounts in order of precedence until funds are exhausted.
- Map - The *map* action applies a subaction in parallel to each of the elements of a collection of data resulting in a collection with the same number of elements. An example is paying interest on each account.
- Reduce - The *reduce* action repeatedly applies a subaction to pairs of adjacent elements in a collection, replacing a pair of elements by the result, until the final result, which is a scalar of the same type. An example is summing up balances of all accounts.

2.23.1 General Rules for Collection Actions

A number of elements, all of the same type, comprise a collection. The *element type* is the type of the collection's elements. The number of elements in a collection is its *size*. Collection actions repeat an action (the *subaction*) for the elements in a collection.

Because the number of repetitions is governed by the size of the collection, rather than a programmed test condition required in a loop action, when a collection action takes more than one collection as input, they must all have the same size. Similarly, multiple collections produced by the same action must have the same size as each other.

Multiple collections are conceptually equivalent to a single collection of tuples. A *slice* is a tuple containing one element, at the same position, from each collection. Each subaction has a single pin for each scalar element in the slice, while each pin of a collection action holds collections. That is, the input and output pins to collection actions hold collections while the pins to the subactions hold the corresponding elements.

If there is more than one input collection in a slice, the elements in each collection must be ordered so that the corresponding elements in the different collections can be correctly matched (hence the word *slice*). If the input collection or collections are ordered, then the output collection or collections are ordered also. (The concept of ordering can be extended to more complex data structures, such as bags, trees, graphs, and so on. In such a case, all collections in an action must have the same form.)

Subactions may have other scalar inputs from outside the collection action. Such values will be the same for all the subaction executions during one execution of a collection action. Subaction outputs are never available outside the collection action.

2.23.2 Collection Action Classes

2.23.2.1 CollectionAction (abstract)

An action that operates on a collection of values.

Attributes

none

Associations

- subaction: Action[1..1]The action applied repeatedly or concurrently to the elements of the collection.

2.23.2.2 FilterAction

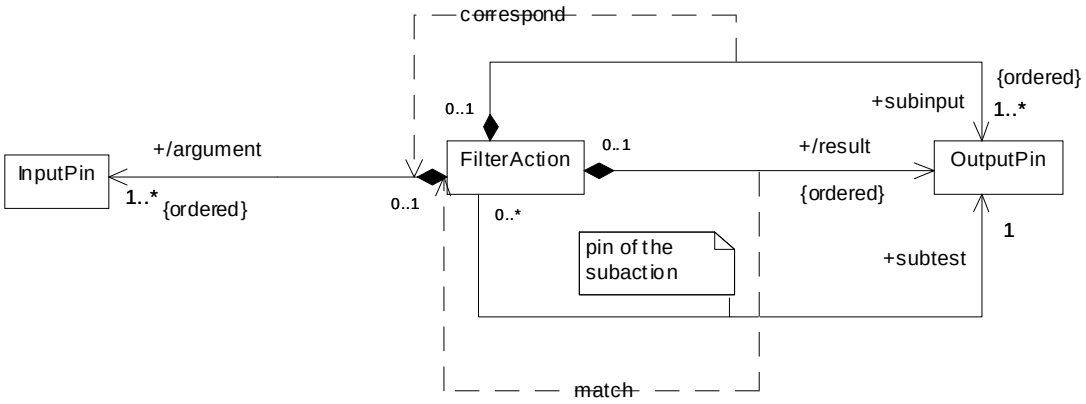


Figure 2-56 Filter action

The filter action applies a subaction to every slice of inputs. The subaction must yield a boolean output whose value is used to decide whether each slice of input values is passed through to the output collections. The result of the filter action is a set of collections, whose size is equal to or less than the size of the input collections, comprising all the slices of the input collections for which the subaction yielded a true value. The gaps caused by the subaction yielding a false value are closed up. All executions of the subaction can be concurrent.

In the simplest case, the subaction takes an input (a single element from the collection input to the filter action), and produces a boolean data value on the output pin. The filter action has a collection of elements as input and an output that is a collection of a subset of the input collection, in the same order, whether the collection is ordered or not.

The subaction may also take inputs that are not inputs to the filter action, such as a constant or a ‘variable’ used in every execution of the subaction. For example, if the subaction tests whether an element is less than *x*, then the inputs to the subaction are the element and *x*, while the input to the filter action is the collection of elements. The output of the subaction remains a boolean scalar.

The filter action may also take multiple input collections, each of which must have the same number of elements. When each slice is presented successively as inputs to the subaction, the subaction determines whether to pass the slice through to the corresponding output collections. The number of the input and output collections must therefore be the same. The type of each output collection matches the type of its corresponding input collection. In addition, the number of elements in each output collection will be the same.

Attributes

None

Associations

- subaction: Action[1..]
  An action that tests whether to keep the input slice in the result. It must have an output pin that yields a Boolean value. For each execution yielding a true value, the input slice is copied to the output collections of the filter action.
- subinput : OutputPin [1..*]
  An ordered list of collections. During each execution of the subaction, a value from each input collection (a *slice*) is copied to the corresponding subinput pin. The subaction is executed once for each input slice.
- subtest: OutputPin - The boolean result of the testing subaction. During each execution of the subaction, the value on this output pin determines whether the corresponding slice is copied to the output collections. If the value is false, the slice does not appear in the output collections and the gap is closed up. If the input collection(s) are ordered, then the output collections are ordered, otherwise the output is unordered.

Inputs

- argument: U [1..*], where U[i] = Collection of T[i], in which T is from *subinput* *[These input pins are owned by the filter action itself.]* An ordered list of collections. All the collections must have the same size, but each collection may contain a different type of element. The subaction is executed once per slice. If there is more than one collection, they must be ordered so that the element values can be matched.

Outputs

- result: U[1..*], where U is the same as on *argument*
  [These output pins are owned by the filter action itself.] At the completion of execution of the filter action, the value of each result pin in the ordered list is a collection of values, equal in number to the number of input argument collections.

The type of each collection and the type of elements in it is identical to that of the corresponding argument collection. The size of each collection is less than or equal to the size of the corresponding input collection. The list of values in each collection is the same as the list of values in the corresponding input collection, after removing elements for which the subtest value was false and closing up the gaps. If the collections are ordered, the ordering of elements is preserved.

- subtest: Boolean[1..]*[This output pin is owned by an action embedded within the subaction.]* At the completion of an execution of the subaction, this pin holds a Boolean value. If the value is true, the value of the subinput values are passed through to the corresponding result collections. If the value is false, the result collections lack elements at the appropriate position.
- subinput: T [1..*], where T are user classes
[These output pins are owned by the FilterAction and accessible only to the subaction and its embedded actions.] The number of pins is equal to the number of input collections of the filter action. All the input collections must be the same size. During each execution of the subaction, the value on each subinput pin is equal to the value of an element of the collection on the corresponding input pin of the filter action. The position of the element in each collection is identical, but different for each execution of the subaction.

Semantics

1. When all the control and dataflow prerequisites of the action execution have been satisfied, it begins execution. The argument values must all be collections of the same size and kind, otherwise the model is ill formed and its subsequent behavior is undefined. A subordinate action execution is created for each position in the group of collections. The element value at the given position in each collection is copied to the respective input pin of the subordinate action execution corresponding to the position in the collections.

2. The subordinate action executions execute concurrently. When a subordinate action execution completes, the Boolean value on the designated *subtest* pin determines whether the slice of input values will be present in the collection of output values of the filter action.

3. When all of the subordinate action executions have completed, a group of output collections are created, each of the same kind and type as the corresponding input collection of the filter action. The elements of the output collections are the same as the elements of the input collections, except that positions corresponding to false test values of the subordinate action executions are removed from the collections. When an element is filtered out from an output collection, the remaining values “close up” the missing position, as follows: For a set, the missing element is simply absent; for a list, the subsequent elements move forward one position for each missing element; for other kinds of collections that may be defined in the future, the specifier must define the effect of removing an element. The filtered collections are placed on the output pins of the filter action and its execution is complete.

2.23.2.3 IterateAction

The iterate action applies an action repeatedly, once for each input slice. A bank of loop variables accumulates the result of the iteration and is eventually passed to the output of the iterate action. This action is a special case of a loop in which the number of iterations is equal to the number of elements in a collection and the elements of the collection are made available to the loop body on successive iterations.

The iterate action executes a subaction once for each input slice. The slices are presented from first to last in the scan order for the collection. When the order of computation does not affect the result, for example if the input collection is a set, or if the *isUnordered* attribute is true, then the slices are presented in an indeterminate (and not necessarily repeatable) order. Like a loop, an iterate action has loop variables that accumulate the effects of the iteration. The subaction accesses the previous values of the loop variables and computes new values for the next execution. The initial values of the loop variables are supplied by inputs to the iterate action, and the final values of the loop variables become the results of the iterate action. If there are no loop variables, the action can have an effect only by writing memory values.

An iterate action has two kinds of input pins: the input collections, and scalar values used to initialize the loop variables. The iterate action has one kind of output pins, whose number and types match the loop variable input pins.

The iterate action owns internal OutputPins, matching the loop variable output pins. On the initial execution of the subaction, these pins get the values from the loop variable input pins. The iterate action also owns a bank (labeled subinput) of internal OutputPins, equal in number to the collection input pins. The type of each subinput pin matches the type of element containing in the corresponding collection input pin. During each execution of the subaction, these pins hold one slice. The iterate action also designates (as suboutput) a bank of OutputPins owned by the subaction. At the conclusion of the execution of the subaction, the values on these pins become the new values of the loop variable pins.

The subaction has access to the subinput values and the loop variable values that change during each execution of the subaction. It may also access available OutputPins in the containing scope. Such values are fixed during the executions of the subactions for any one execution of the iterate action. During one execution, the subaction computes values for the suboutput pins. The values on the suboutput pins become the new values of the loop variable pins on the next iteration of the subaction. When all slices of the collections have been processed, the final values of the loop variables become the values on the result output pins of the overall iterate action. No outputs of the subaction are available outside of it, except for the explicit suboutput pins designated by the iterate action, which are available only to the iterate action itself.

The *isUnordered* attribute states that the order of execution of slices is irrelevant, even though the ordering of elements in each collection is still used to match corresponding elements into slices. The purpose of this attribute would be to remove overspecification of ordering and permit optimization within an implementation, especially if values are not all computed at the same time (such as lazy evaluation). If the input collection shape is a set, then the slices are processed in an indeterminate order.

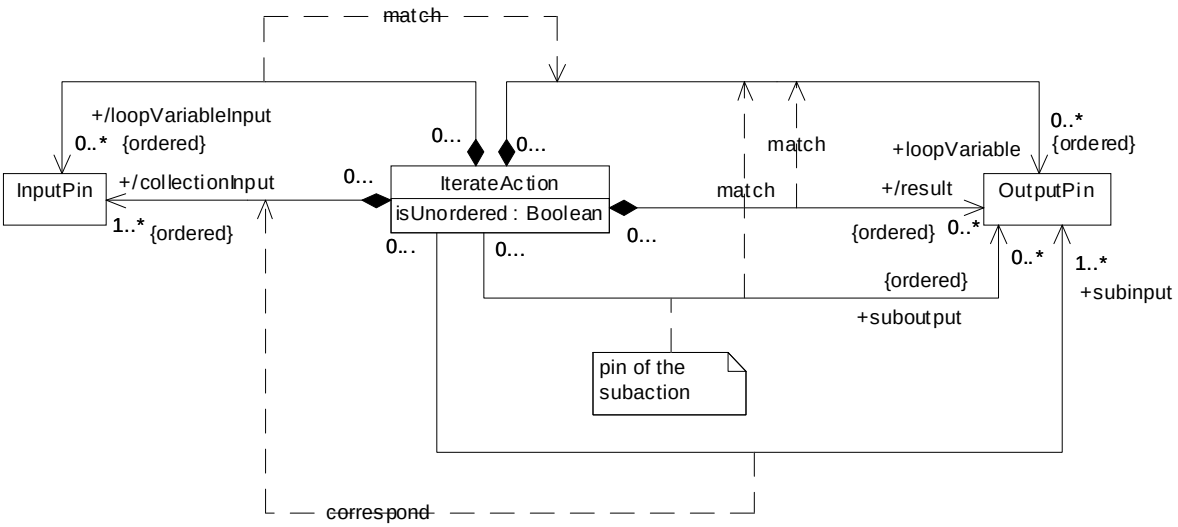


Figure 2-57 Iterate action

Attributes

- isUnordered: Boolean[1..1]
  If true, indicates that the slices may be given to the successive executions of the subaction in any order. This should be set only if the final result is insensitive to execution order. If false, the subaction is executed on slices of the collections in order from first to last, in accord with the scan order of the particular kind of collection. If the collection is a set, then the iteration order is automatically unordered and this flag has no further effect.

Associations

- subaction: Action[1..1]
  An action that is executed repeatedly and sequentially, once for each slice of values in a list of input collections. During each execution of the action, one slice of values is made available to the execution, from the first to last position in the collections. Like a loop action, this action has a list of loop variables that make the results of one execution of the subaction available to the next sequential execution of the subaction. On the first execution of the subaction, the loop variables are set by the loopValueInput values that are input to the overall IterateAction. The iterate action designates a list of output pins from the subaction as the updated values of the loop variables. During each execution of the subaction, the previous values of the loop variables are available within the subaction. After each execution, the designated output values in the subaction are copied to the loop variables. After the subaction has been executed once for each sliced of the input collections, the final values of the loop variables are copied to the output pins of the iterate action.

- loopVariable: OutputPin[0..*]
  A (possibly empty) list of output pins (owned by the iterate action) whose values represent the accumulated result of executing the action so far. During the first execution of the subaction, each pin holds a copy of the corresponding loopVariableInput pin (among the input pins of the iterate action). During each subsequent execution of the subaction, each pin holds a copy of the value on the corresponding suboutput pin within the previous execution of the subaction.
- subinput:OutputPin[1..*]
  A nonempty list of output pins (owned by the iterate action) whose values represent one slice of the input collections during one execution of the subaction. During each execution of the subaction, a different slice of values appears on the pins. The pins are available to the subaction. The type of each element pin must match the type of element held by each collection on the corresponding input pin.
- suboutput: OutputPin[0..*]
  A (possibly empty) list of available output pins owned by the subaction. They represent updated values of the loop variables computed by the subaction. After each execution of the subaction, the values on these pins are copied to the loop variable pins for the next execution of the subaction or the output result of the overall iterate action. This list of pins must match in number and types the loopVariable pins.

Inputs

- collectionInput: V[1..*], where V[i] = Collection of T[i], in which T are from *subinput*
  [These input pins are owned by the iterate action itself.] Each input pin holds a collection of values. All the collections must have the same shape and number of elements, but the types of elements in the various pins may differ. During each execution of the subaction, one value from the same position of each collection constitutes a slice of values available to that execution of the subaction as the values of the subinput pins. The subaction is executed once for each position in the input collections, in order from first to last position.
- loopVariableInput: U [0..*], where U are any user classes
  [These input pins are owned by the iterate action itself.] The number of pins must equal the number of loop variable pins and the type of value in each pin must match the type of element in the corresponding loop variable pin. The values on these pins represent the initial values for the loop variables. During the first execution of the subaction, the loop variable pins hold copies of the values on the corresponding loop variable input pins.

Outputs

- loopVariable: U [0..*], where U are the same as on *loopVariableInput*
  [These output pins are owned by the iterate action and accessible only within the subaction. The values are not accessible outside the iterate action.] During the first execution of the subaction, each loopVariable pin has the value of the corresponding loopVariableInput pin. On each subsequent execution, each loopVariable pin has the value of the corresponding suboutput pin after the previous execution of the subaction.

- result: U [0..*], where U are the same as on *loopVariableInput*
[These output pins are owned by the iterate action itself.] The number of pins must equal the number of loop variable input pins and the number of loop variable pins, and the types of pins in each corresponding position must match. After the execution of the subaction on the final slice of values from the input collections, the result pins have values equal to the values of the suboutput pins on the final execution of the subaction. If the input collections are of size zero and the subaction is consequently not executed, the result pins get copies of the values on the loop variable input pins.
- subinput: T [1..*], where T are user classes
[These output pins are owned by the iterate action itself but are accessible only within the subaction, not accessible outside the iterate action.] During the execution of a subaction, each subinput pin has a value equal to the value of the element in a given position of the corresponding collectionInput collection. During each subsequent execution of the subaction, the position moves through the collections from first element to last element.
- suboutput: U [1..*], where U are the same as on *loopVariableInput*
[These output pins are owned by actions nested within the subaction. They are accessible only within the iterate action.] The number and types of the subinput pins must match the loopVariable pins. At the completion of an execution of the subaction, the list of suboutputs will have values. The value of each suboutput pin becomes the value of the corresponding loopVariable pin during the next execution of the subaction.

Semantics

1. When all control flow and data flow prerequisites of an iterate action are satisfied, the execution of the iteration begins. All of the values on *loopVariableInput* pins are copied into a newly created set of *loopVariable* values owned by the iteration execution.
2. For each position with the tuple of *inputCollection* collections, the subaction is executed once. The executions are sequential, in the same order as the elements in the input collections. If the collections are unordered or if the *isUnordered* flag is true, then the order of processing elements is undefined and nondeterministic. For each execution of the subaction, the *subinput* pins of the execution receive the values corresponding to the given element position in the respective input collections. The execution also has access to the loopVariable values created by the iteration execution (for the first iteration) or updated by the previous subaction execution (for subsequent iterations).
3. When the execution of a subaction is complete, the values of its *suboutput* pins are copied to the corresponding *loopVariable* pins for the next iteration.
4. When the subaction has been executed once for each slice of the input collections, the value on each *loopVariable* pin is copied to the corresponding *result* pin of the iteration action. The execution of the iteration action is complete.

2.23.2.4 MapAction

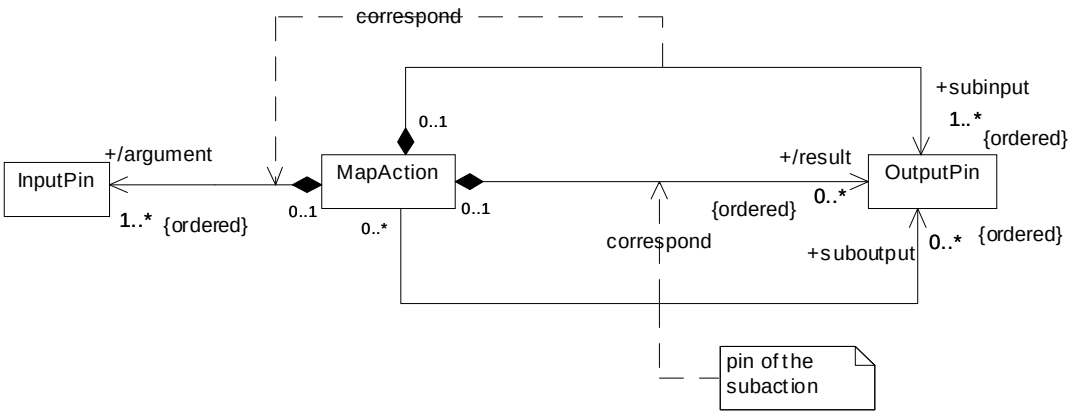


Figure 2-58 Map action

The map action applies a subaction to each input slice. If the subaction has output pins, then the map action has that same number of outputs, each of which is a collection with the same size as the input collections. Any values produced on the suboutput pins are formed into collections, each containing one value from each execution.

The subaction is executed concurrently, once for each input slice. If executions of the subaction conflict (because they write shared objects), then the result is indeterminate.

The map action has zero or more output pins, each of which holds a collection. The output collections from the map action have the same size as the input collections, and each output value occupies the corresponding position in its collection, but the types of the collections may differ from each other and the input collections. The number of output collections need not equal the number of input collections.

The subaction may access available scalar inputs from outside the map action, but no output pin of the subaction is available outside the map action.

Attributes

None

Associations

- subaction: Action[1..1]An action to be executed once for each input slice. During each execution of the subaction, one element value from each input collection (i.e., a slice) is made available to the subaction. The subaction must produce one output slice (i.e., one value for each output collection).

- subinput: OutputPin[1..*] Each output pin has a type that matches the element type. The subaction is executed once for each slice in the input collections. During each execution of the subaction, a value from each slice is copied to the corresponding subinput pin.
- suboutput: OutputPin[0..*] The number of suboutput pins matches the number of output pins of the map action, and each suboutput pin has a type that matches the element type. The value from each suboutput pin is copied to the corresponding map action result pin at the same position in the collection as the input slice to the subaction.

Inputs

- argument: T [1..*], where T[i] = Collection of U[i], in which U is on *subinput [These input pins are owned by the map action itself.)* An ordered list of collections. The collections may have different types but all of them must have the same number of elements.

Outputs

- result: W [0..*], where W[i] = Collection of V[i], in which V is on *suboutput [These output pins are owned by the map action.]* An ordered list of collections. Each collection has the same collection type and element type as the corresponding argument collection, and all of them have the same number of elements as the input collections. After completion of execution of the subaction for each slice of input values, the value of each element in each output collection is equal to the value of the corresponding suboutput pin after the execution of the subaction for the same position in the input collections.
- subinput: U [1..*], where U are user classes
  [These output pins that are owned by the map action. The pins are accessible within the subaction but are not accessible outside the map action.] The subaction is executed once for each element position in the input collections. During each execution of the subaction, each subinput pin has a value equal to the value of the element in the given position of the corresponding input collection.
- suboutput: V [0..*], where V are user classes
  [These output pins are owned by actions nested within the subaction and are accessible only within the map action.] At the completion of each execution of the subaction, the suboutput pins have values computed during that execution of the subaction. The result values of the map action in a given element position are copied from the suboutput values of the subaction execution for that position of input elements.

Semantics

1. When all the control and dataflow prerequisites of the action execution have been satisfied, it begins execution. The argument values must all be collections of the same size and kind, otherwise the model is ill formed and its subsequent behavior is undefined. A subordinate action execution is created for each position in the group of collections. The element value at the given position in each collection is copied

to the respective input pin of the subordinate action execution corresponding to the position in the collections. (Each subordinate action has an input pin corresponding to each input pin of the map action.)

2. The subordinate action executions execute concurrently. When a subordinate action execution completes, it has values available on its designated suboutput pins (if any).

3. When all of the subordinate action executions have completed, a group of output collections are created, each containing elements of the same type as one of the suboutput pins of the subaction. Each collection kind is the same as the (common) collection kind of the input collections (i.e., set, list, etc.). For each subordinate action execution, the value of each suboutput pin is copied to the position in the respective output collection corresponding to the position of the input values for that execution.The output collections are placed on the output pins of the map action and its execution is complete.

2.23.2.5 ReduceAction

The reduce action applies an associative binary subaction repeatedly to adjacent pairs of slices from the input collections, until the (intermediate working) collection is reduced to a single slice *of scalar values*, which together constitute the output values of the reduce action. The order in which the subaction is applied to pairs of values is indeterminate, and does not affect the ultimate result if the action is associative, unless the subactions are not isolated from each other, in which case the result is unpredictable.

As an example, consider a collection comprising four integers in order: 2, 7, 5 and -3. The result of applying the reduce action to this collection with the binary associative subaction Addition is the scalar 9. This can be computed by any of the following orderings: (((2+7) + 5) + -3) = 11; (2 + (7 + (5 + -3))) = 11; ((2 + 7) + (5 + -3)) = 11.

When the subaction is symmetric, as with addition, the order of the elements in the collection is not important. However, some associative operations are not symmetric, such as matrix multiplication, so A × B is not the same as B × A. In these cases, the concept of adjacency of the elements and the order in which they appear is critical.

The reduce action requires a subaction, which must be a binary associative operator. Each of the two inputs to the subaction is a slice from the input collections to the reduce action. For example, to sum all the balances for a customer's account, the reduce action has a single input collection of account balances, and a single scalar output, the sum of those balances, which must necessarily be of the same type. Each of the two inputs to the subaction, Addition, takes "a slice from the input collections to the reduce action," which in this case is a single account balance on each input. The output is a tuple with the same structure: a balance. When the reduce action has several input collections, then one slice across all collections will be one input to the subaction and another slice will constitute the other input.

The reduce action executes the subaction one fewer time than the size of the input collections., because it operates on adjacent pairs of slices from the input collections. The output of the subaction conceptually replaces the two input slices in the collection

of tuples, so the subaction can be applied repeatedly to pairs of slices until a single slice remains. Its value is place on the result output pins of the overall reduce action *as scalars*. In other words, the reduce action serves to reduce a collection of values to a single value by repeated application of a binary function.

If the subaction accesses values from outside the reduce action, such values will be the same for all the concurrent subaction executions during a single execution of the reduce action. No output pins of the subaction may be connected outside the reduce action.

The *isUnordered* attribute states that the reduction can be applied to the slices in any order, even though the ordering of elements in each collection is still used to match corresponding elements into slices. This will be mathematically valid if the subaction is symmetric and the actions are isolated.

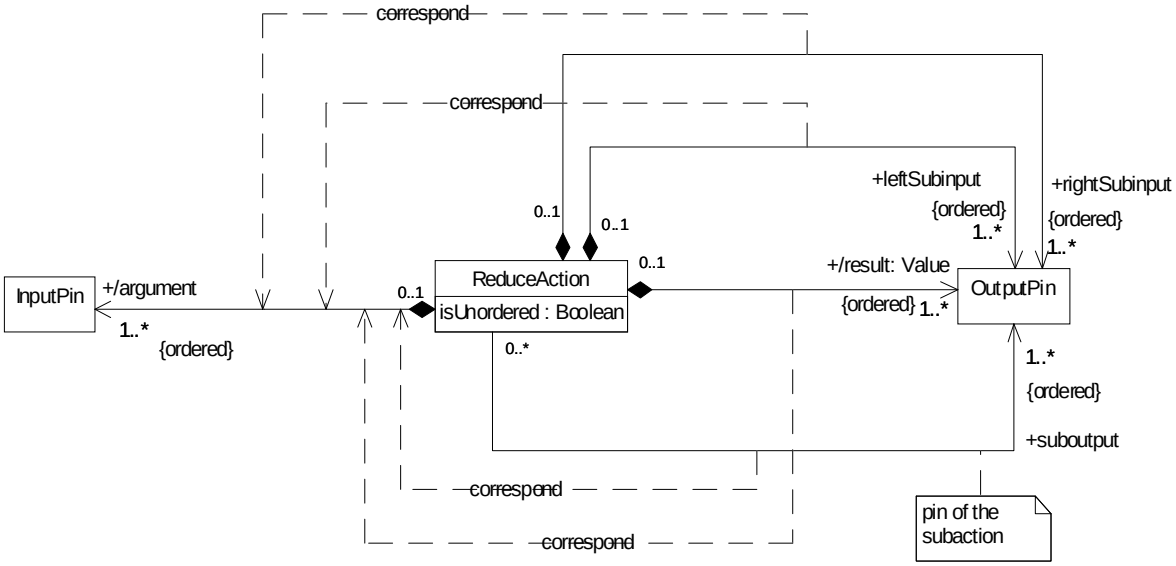


Figure 2-59 Reduce action

Attributes

- isUnordered: Boolean
  If true, indicates that the slices of values may be given to the successive executions of the subaction in any order. This should be set only if the final result is insensitive to execution order. If false, the subaction is executed on slices of the collections in order from first to last, in accord with the scan order of the particular kind of collection. If the collection is a set, then the iteration order is automatically unordered and this flag has no further effect.

Associations

- subaction: Action
  An action that is executed repeatedly on adjacent pairs of slices. The action produces a slice with the same size.
- leftSubinput: OutputPin[1..*]
  A nonempty list of output pins (owned by the reduce action), which represents the first of two tuples that are input to the subaction. On each execution of the subaction, these pins hold the values in the first of two adjacent slices in the working collection of the reduce action. The number of pins and their types must match the output pins of the reduce action.
- rightSubinput: OutputPin[1..*]
  A nonempty list of output pins (owned by the reduce action), which represents the second tuples input to the subaction. On each execution of the subaction, these pins hold the values in the second of two adjacent slices in the working collection of the reduce action. The number of pins and their types must match the output pins of the reduce action.
- suboutput: OutputPin[1..*]
  A nonempty list of available output pins owned by the subaction, which represents the results of executing the subaction. After each execution of the subaction, the slice of values on these pins replaces the two adjacent slices in the working collection that served as subinputs to the subaction. The number of pins and their types must match the output pins of the reduce action.

Inputs

- argument: U [1..*], where U[i] = Collection of T[i], in which T are from *result* *[These input pins are owned by the reduce action.]* The input collections to the reduce action.

Outputs

- result: T [1..*], where T are user classes
  These output pins are owned by the reduce action. The corresponding result, leftSubinput, rightSubinput, *and* suboutput *pins all have the same type, equal to the element type of the corresponding argument collection.]* After the final execution of the subaction, each result pin has the value equal to the value of the corresponding suboutput pin after completion of the final execution of the subaction.
- leftSubinput: T [1..*], where T are the same as in *result*
  [These l output pins are owned by the reduce action. The values are accessible within the subaction but are not available outside the reduce action.] During each execution of the subaction, each pin has the value of the corresponding suboutput pin on a previous execution of the subaction.
- rightSubinput: T [1..*], where T are the same as in *result*
  [These output pins are owned by the reduce action. The values are accessible within the subaction but are not available outside the reduce action.] During each execution of the subaction, each pin has the value of the corresponding suboutput

pin on a previous execution of the subaction. The leftSubinput and the rightSubinput come from adjacent positions in the implicit intermediate collections that start with the argument collections and end up as collections of size one.

- suboutput: T [1..*], where T are the same as in *result*
  [These output pins are owned by actions nested within the subaction. These are accessible only within the reduce action.] After the completion of each execution of the subaction, the suboutput pins have values computed during that execution. Each suboutput value conceptually replaces the adjacent pair of values that supplied the left and right subinput values from the implicit intermediate collection, thereby reducing the size of the intermediate collection by one element. When the size of the intermediate collections is one, the values of the suboutput pins on the final execution of the subaction become the results of the reduce action.

Semantics

1. When all control flow and data flow prerequisites of an iterate action are satisfied, the execution of the reduce action begins. The tuple of *argument* collections is conceptually copied to a temporary working store of collections that accumulates intermediate results of the action. The original input collections are not modified by the action.

2. Two contiguous positions in the intermediate working store are selected nondeterministically and a subordinate execution of the subaction is created. The subaction execution receives the first slice of intermediate collection values as a tuple of leftSubinput pin values, and it receives the second slice of intermediate collection values as a tuple of rightSubinput pin values. If the collections are unordered or if the isUnordered flag is true, then any two elements may be selected nondeterministically.

3. Additional pairs of contiguous positions may be selected nondeterministically for concurrent execution of the subaction, provided they do not include positions already selected for execution.

4. When a subordinate action execution completes, the tuple of values on its suboutput pins replaces the pair of slices of values within the intermediate working store. For collections more complicated than lists, the specifier must define what it means to replace two elements by a single element.

5. Step 2 is repeated as long as the working store contains more than one element position. At any point, more than one execution of a subordinate action may be working on a pair of element positions.

6. When the size of each collection in the working store has been reduced to one element, the value of the element from each collection in the working store is copied to the corresponding result position of the reduce action. The execution of the reduce action is complete.

The execution order of the reduce action is nondeterministic. If, however, the subaction represents an associative operator (i.e., *(x op y) op z = x op (y op z)*) and multiple executions of the subaction do not conflict, the result value will be insensitive to execution order, so the result value will be deterministic for ordered collections. If, in

addition, the subaction represents a commutative operator (i.e., *x op y = y op x*), then the result value will be insensitive to the order of elements, so the result value will be deterministic for unordered collections (including the use of the *isUnordered* flag). Many common operations (e.g., sum, maximum value, union) satisfy these properties, and the normal intent of the reduce action is for use in such cases.

2.24 Messaging Actions

These actions exchange messages among objects. An initial message from one object to another is called a *request*. The sender of a request may simply continue execution immediately without concern for the behavior invoked by the request (an *asynchronous* invocation), or it may choose to suspend execution until the activity invoked by the request reaches a well-defined point and sends a reply message back to the requestor, with optional return values (a synchronous invocation). If the request is synchronous, the behavior of the receiver must have a well-defined reply point; if the request is asynchronous, a reply is optional. The receiver may handle a request in various ways based on its organization, including procedure execution and triggering a state machine. The requestor need not be aware of how the request will be handled. The messaging model covers a wide range of ways to match behavior to requests, including state machine triggers, fixed procedures, class-based method lookup, method combination (such as before-after methods), object-based delegation (as in *self*), and so on. In all cases, the effect is processed by a distinct context from the context of the requestor and the messaging information is transmitted among requestor and target by value. This messaging model fully supports distributed processing without special mechanisms. This model unifies operations and signals into a single concept.

The actions *CallOperationAction*, *SendSignalAction*, and *BroadcastSignalAction* provide the functionality found in traditional programming languages without the generality of the full messaging mechanism. Logically, they could be considered special cases of the unified messaging actions, but they may be implemented directly without using the unified messaging mechanisms.

2.24.1 Request

A *request* represents a request for service made by a *requestor* object to a *target* object. It includes both the kind of service to be performed (such as a particular operation or signal) as well as the parameters of the service request (the parameters of the operation or signal). A request is modeled as an object. The class of the request object represents the specific kind of service requested, that is, the operation to be performed or the signal to be handled. The attributes of the object represent the parameters of the service request, that is, the parameters of the operation or signal. The target object is specified separately from the request information. A class used for request objects is called a *request* class. There is one request class corresponding to each operation and signal. Request classes may be organized into generalization hierarchies (subject to constraints by the modeler), and request resolution mechanisms may use such hierarchies in matching requests to behavior. Request classes are defined in user models explicitly or implicitly from other elements, such as operations.

There is nothing special about request or reply classes. A class is known to be a request or reply class because it is used in invocation actions, not because it has any particular structure, therefore no Request or Reply metaclasses are defined in the model.

2.24.2 Asynchronous Invocation

An *asynchronous invocation* is the transmission of a request from a requestor to a target object in which the requestor continues execution immediately, without waiting for a reply. The target object might later communicate to the requestor, but any such communication must be explicitly programmed and it is not part of the asynchronous invocation action itself. Sending a signal and asynchronously calling a procedure are examples of asynchronous invocations. The requestor continues after the invocation without waiting for a response, so the invoked execution need not have a definite termination point. It is permissible to asynchronously invoke a request to a procedure that eventually issues a reply; the reply message is simply discarded. The reverse is not allowed: a synchronous invocation that activates behavior without a distinct reply will leave the invocation hanging forever (although implementations can attempt to detect this and provide some kind of exception handling to deal with it as a programming error).

The execution of an asynchronous invocation has the following structure:

- the requestor creates a request object and sends it to a target object by means of an asynchronous invocation action;
- once the invocation is set up, the requestor continues execution without further direct interaction with the invoked execution;
- the request object is transmitted to the target object, which might take some time in a distributed system;
- the request object arrives at the target object and is kept until the receiver is ready to handle it. Real-time extensions to UML might define queuing orders, priority mechanisms, and so on, but they are beyond the scope of the basic action semantics, which leave the order of handling requests undefined;
- the receiver begins handling the request;
- the type of the request object is matched against information in the target object, and the request is *resolved* into a behavioral effect, usually involving the execution of a procedure;
- a procedure may be executed.

This definition of asynchronous invocation is meant to encompass both asynchronous calls as well as traditional signals that trigger state machines, as well as other models of computation. From the requestor's viewpoint, these are all requests that do not wait for a reply. The modeler may change the implementation of the request without changing the invocation.

The action semantics provides a general framework and does not limit or specify the exact resolution mechanism.

2.24.3 Synchronous invocation

A *synchronous invocation* is the transmission of a request from a requestor to a target object in which the requestor waits for a reply from the invoked execution. The invoked execution may supply return values, but even if there are no return values, the requestor waits for a reply indicating that the invoked execution has completed. A behavior invoked by a synchronous request must therefore have a definite reply point, even if there are no return values, because the reply point indicates the point at which the requestor may continue execution, possibly concurrently with the remainder of the invoked execution. With no loss of generality, we may think of the invoked execution as sending a *reply object* back to the requestor. The reply object is an object whose attributes represent the return values, but even if there are no return values, the sending of the reply message and its receipt by the requestor allow the requestor to continue execution. In the most general case, the requestor may use the type of the reply message as well as its values (for example, to distinguish different kinds of values or indicate exceptional situations), although many programming languages may choose to constrain the return type to a predetermined type.

The execution of a synchronous invocation has the following structure:

- the requestor creates a request message object and sends it to a target object by means of a synchronous invocation action, at which point the execution of the requestor is blocked;
- the request message is transmitted to the target object, which might take some time in a distributed system;
- the message arrives at the target object and is kept until the receiver is ready to handle it. Real-time extensions might define queuing orders, priority mechanisms, and so on, but they are beyond the scope of the basic action semantics, which leave the order of handling requests undefined;
- the receiver begins handling the request;
- the type of the message is matched against information in the target object, and it is *resolved* into a behavioral effect, usually involving the execution of a procedure;
- the procedure is executed and at some point a reply message is generated;
- the reply message is transmitted back to the requestor;
- the requestor is given the reply message and execution of the requestor is allowed to proceed.

This definition of synchronous invocation is meant to encompass both traditional operation calls and also calls that trigger state machines, as well as other models of computation. From the requestor’s viewpoint, these are all requests that wait for a reply. The modeler may change the implementation of the request without changing the invocation.

The action semantics provides a general framework and does not limit or specify the exact resolution mechanism.

2.24.4 Request Handling

To support a wide range of behavior in mainstream and more innovative languages, the skeleton description of invocation may be parameterized in the following ways, which must be specified as part of a UML extension or profile defining the given system environment. These represent different semantic variation points on the behavior of the invocation mechanism:

- The handling of requests by an object may be sequential or concurrent, according to the specification of the target object (not the requestor) and the type of request.
- Sequential handling corresponds to a state machine on the target object, the guarded execution of a procedure, or the direct inline receipt of a request by an executing procedure. Such execution cannot proceed until previous executions by the target object or the resolved procedure have completed, and there is sharing of control state across subsequent executions.
- Concurrent handling corresponds to the execution of a procedure in its own context, regardless of other executions; for example, a traditional procedure call. Such execution can proceed immediately on receipt because it automatically generates a new execution context, and sharing of information is only through the attributes of the target object.
- The order in which requests are handled may be specified by the received object. This permits the definition of queuing and priority mechanisms. This document does not specify such possible mechanisms, but we would expect them to be specified in a real-time profile, for example.
- The transmission of messages may be subject to delays and errors. We do not elaborate this possibility in this document, but we expect it to be pursued in real-time profiles.
- The way in which a request type determines the execution of one or more procedures (and other behavioral effects) is called *resolution*. It represents a more general form of method lookup and is expanded in more detail later. Triggering state machine transitions, selecting a method attached to an ancestor of the target object type, and object-based delegation are all varieties of resolution.
- Sending a reply message can be an explicit action or it can be implicit in the structure of the procedures.

All messages are transmitted “by value.” That is, a message object itself has no identity; its attribute values can be freely copied without loss of information. However, the values of the attributes within a message may be object references, that is, they may reference instances. In other words, a value may be a simple data value or a reference to an identity.

An “in-out” parameter includes a value in both the request and reply objects.

A send is similar to a call, except that the sender continues execution immediately and has no further interaction with the invoked execution. If the invoked execution attempts to send a reply, the reply message is immediately discarded.

2.24.5 Reply Handling

A traditional procedure expects a request of a specific type and generates a reply of a specific type. The messaging model presented here permits variants on both the request to and the reply from a single procedure. Variant replies would permit a procedure to return objects of different types depending on circumstances. This would permit handling unusual or exceptional situations without extra flags or special data values.

Inline exceptions are considered as simply another type of reply value (see Section 2.25.3, “Exceptions,” on page 2-335). If the execution of a called procedure generates an exception that cannot be handled within the called procedure, the exception object is returned to the caller as a reply object. No special indication is necessary that it is an exception. In fact, the caller is free to treat it as an ordinary reply value (presumably within a case statement on reply types). If the reply type violates constraints on the expected reply type, then it is raised as an exception in the scope of the execution that made the call, permitting the exception handling mechanisms to come into play.

This approach unifies exception handling with variant reply types and allows the caller and the executed procedure to choose independently between explicit exceptions and multiple reply types.

2.24.6 Procedures

A *procedure* is a structured set of actions that can be invoked as a behavior entity. An action can only be executed as part of a procedure execution; individual actions may not be referenced outside their procedures. A procedure represents a distinct, closed context. No variables are shared among caller and called procedure. A procedure also does not have “global variables” or “static variables” that hold state. Any such state must be implemented using objects. Because it has no state, many executions of the same procedure are conceivable on the same object, each with its own execution context. This does not mean that all procedures must support such simultaneous access, and many will not, but simply that the structure of procedures does not prevent it.

The target object can be different from the requesting object and they can have different classes. We do not distinguish “local calls” from “remote procedure calls;” semantically, there is no difference. There is an implicit chain of call executions during execution, but no need to assume that they are organized into a stack or owned by an explicit task, thread, or process. Such things are implementation choices that can be implemented in different ways.

While the execution of a procedure invoked by a synchronous invocation is progressing, the requesting action execution is “blocked.” This does not require any special mechanisms; the synchronous action execution is simply in the “executing” state until the invoked procedure execution terminates and returns. The called procedure execution has, as part of its state, sufficient information to identify the calling action execution. There is no special “return” action available to a procedure. A procedure execution returns when it completes the execution of its action, and the

output values of the procedure are used as the return values. If the invocation was asynchronous and the request was repliable, any output values are formed into a reply object which is transmitted asynchronously to the invoking object.

A procedure has an input value and an output value, both represented as objects without identity (or snapshots). In the most usual case, the input value is immediately unmarshalled into its constituent attribute values and is not available in the procedure as an object, but other variations are possible.

2.24.7 Performing requests

A request has a request type and a list of argument values. A request is modeled as an object. The type of the object represents the request type. The attributes of the request class represent the parameters. Because the request type and its parameters are modeled as a single class, there is no possibility of an incorrect argument list or missing arguments.

Making a request requires a list of argument values that have been marshalled into a request object. The action semantics does not restrict the manner in which request objects are produced. For a typical operation call, all the argument values could be generated concurrently and then used to generate a request object. However, request objects can also be produced in several stages, starting with a raw object of the correct type and then initializing attribute values one at a time. All that matters is that a valid request object be available when the request action is performed. Note that a request object is transmitted “by value,” therefore the identity of the request object supplied to the request action is irrelevant, and the contents of the object can be copied if convenient for the semantics or the implementation. The attribute values themselves may include object identities, each of which must continue to identify the same object in spite of any implementation encodings due to transmission.

Requests may be sent to any object. Each object determines how it handles requests that it receives. All requests to the same object need not be handled the same way. Some requests may invoke methods and some may trigger transitions. The manner of handling a request is discussed later under *Resolution*.

The execution of a method creates a new procedure execution and does not affect an existing state machine or running action execution. The activation of a method creates, from the point of view of the receiver, a new “thread of control” that executes alongside existing “threads of control” on the same object, but without sharing control or variables. Different executions interact only indirectly, through the attributes of the target object, which are shared by all executions on it.

The triggering of a transition is possible only if the target object has a state machine execution attached to it. Triggering a transition does not create a new thread of control. The execution proceeds in the context of the existing state machine execution, when the state machine is ready to handle the request.

2.24.8 Effect Resolution

A *request* is an unsolicited message sent from a requestor object to a target object that causes behavior when it is received by the target object. (The other kind of message is a *reply*, which goes directly to a blocked execution, not to an object, and is expected by it.) The mechanism of turning a received request into execution is called *resolution*. The requestor need not know how the request will be resolved, and therefore need not change even if the manner of handling the request is changed. Resolution is a framework within which different kinds of behavior can be specified. UML extensions or profiles would be needed to specify new kinds of behavior.

Here are 3 common mechanisms for resolving requests that have appeared in various programming languages. These are not meant to be comprehensive, and others might be proposed in the future.

1. The request type is used to determine a procedure to be executed. The procedure execution receives the request object as its input value. When the procedure execution completes, if the invocation was synchronous, its output value is a reply message that is transmitted back to the requesting execution. If the request was asynchronous and repliable, then its output value is a reply message which is transmitted to the invoking object as an asynchronous request. If the request was asynchronous and not repliable, the reply message is not generated. The request type corresponds to an operation, the attributes of the request to the arguments of the operation, and the attributes of the reply message to the return values. This mechanism corresponds to a traditional operation call (synchronous or asynchronous, as the case may be). In some variations, multiple procedures can be executed, and the resolution mechanism must indicate their order of execution and which one supplies the reply value.

2. The request type is used to trigger a transition within the state machine execution attached to the object. Requests are stored until the state machine execution is quiescent and chooses to handle one of them. If the request type matches an event on a transition leaving the current state (subject to the full rules of transition firing), the transition fires. Firing a transition updates the state of the object. If there is a procedure attached to the transition, it is executed. When the procedure execution completes, its output value is a reply message that is transmitted back to the calling execution (for a synchronous request) or to the calling object (for an asynchronous, repliable request). Typically more than one transition can execute on a single firing (e.g., entry and exit transitions). The resolution mechanism must indicate which procedure supplies the reply value. If the request is asynchronous and not repliable, then any output values are ignored and no reply message is sent. In traditional terms, the request type corresponds to a signal, its attributes to the attributes of the signal, and any attributes of the reply message to return values on a call event.

3. The request is explicitly read by an execution using an explicit action. In this case, requests are held in a queue until the execution explicitly reads them. When the execution reads a request, the execution gains access to the request data and may take whatever actions it likes. The execution also gets a handle to the return information, which it can later use implicitly to send a reply message. It may not

manipulate the return information, however. This mechanism corresponds to traditional interprocess communication and would require a UML extension or profile in which executions are directly manipulable at run time.

In practice, the resolution mechanisms will often be specified for a particular kind of system or a particular request type, but we define them in a general way that can cover many different kinds of systems. We do not imply that the actual implementation of a system must be identical to this logical model; only the final effect must be the same.

The following sections describe each kind of resolution mechanism.

2.24.9 Operation Lookup

The phrase *operation lookup* refers to mapping a request type (e.g., an operation) to a procedure (e.g., a method) in the context of an object. In Smalltalk or C++, operation lookup is based on the class of an object and the class hierarchy, but other languages support other kinds of operation lookup, which we wish to encompass in the action semantics. For example, *self* supports an object-based lookup mechanism and CLOS supports before- and after-methods.

The following list describes the possible kinds of operation lookup:

1. An operation map (a set of operation-method pairs) is attached to an object. If a request type matches an operation in the map, the corresponding method is executed. If the operation is not found, then a link points to another object that is then searched. This is the concept of *delegation*.

2. An operation map is attached to a class. The map on the type of the object is examined. If a request type matches an operation in the map, the corresponding method is executed. If the operation is not found, then a link points to another class that is then searched. This is the concept of *inheritance*.

3. An operation map is defined globally. If a request type matches an operation in the map, the corresponding method is executed. This is the concept of a direct procedure call (i.e., traditional).

A further variation allows the execution of more than one matching procedure from different maps. The order in which the procedures are executed is part of the particular operation lookup mechanism. Another variation would allow several kinds of maps on each element, with matching methods to be executed in a designated order. For example, CLOS supports 3 kinds of maps: a *before* map, a *main* map, and an *after* map. First, all the matching methods on the before map are executed, starting at the top of the class hierarchy; then the most specific match in the main map is executed, then all the matches in the after map, ending at the top of the class hierarchy. Only the return value from the main match is returned to the caller.

Section 2.24.16, “Optional Profile for Resolution of Operations and Signals,” on page 2-330 describes one way that traditional operation calls as in variation 2 above could be implemented.

2.24.10 Transition Triggering

The phrase *transition triggering* refers to mapping a request type (a signal) to a procedure (a procedure) in the context of an object with a state machine. A state machine may be attached to the object or, more usually, to the type of the object. The object has a current state (which may decompose into a set of primitive states). The state of the object is a state (or set of states) in the attached state machine. The following rules, summarized from the state machine package, are suitable for implementation of request resolution with no change in semantics:

The following item describes the event lookup:

1. A transition map (a set of event-procedure-next state tuples, with some additional information) is attached to a state on the state machine attached to the type of the target object. If a request type matches an event in the map, subject to various qualifying information (such as conditions), the corresponding procedure is executed and the current state is updated. The request object is delivered to the procedure execution. If the event is not found, then a link points to another state that is then searched. This is the concept of *nested states*. If any entries or exits are triggered, the request object is delivered to any triggered procedures.

2. Same as above, except the state machine is attached directly to the object. This idea pushes the state of the art but it seems compatible with object-level method lookup.

The normal nested-state variation allows multiple matches, somewhat similar in spirit to CLOS before-after methods, although not identical: exit transitions and entry transitions are executed on the path from the old state to the new state. Many variations are possible.

The state machine packages appear s in a form directly amenable to resolution, provided Signals are regarded as request objects.

A “synchronous signal” (formerly “call event”) is a request sent by a synchronous invocation action that is resolved into a transition trigger rather than a method. The manner of handling this is not restricted by the action semantics, but one variation that implements the correct semantics is as follows: If a state machine has a transition triggered by a synchronous invocation object, the request object is delivered to each procedure attached to a triggered transition. When a transition is triggered by a request, the procedure is executed. When the execution of the procedure attached to the main transition (that is, the one directly triggered by the request) is complete, its output values are formed into a reply message and passed back to the action that issued the call request, and it becomes the output of the call action. If the request triggers exit or entry transitions, only the main procedure attached to the main transition is used to determine the reply. (Other rules are possible for determining the return values.)

2.24.11 Direct Communication among Executions

Usually invocations are handled implicitly by some underlying execution machinery that manages the gathering of packets by a target object, their storage until the object is ready to process one of them, and the selection and resolution of a request into a behavioral effect. This approach characterizes method invocation and state machine

triggering. It is also possible, however, for executions to explicitly receive requests and generate replies. This kind of usage is characteristic of kinds of real-time computation. The mechanisms for such interaction are not specified in this document, but they could be defined in a UML extension or profile.

2.24.12 Strong Typing

The action semantics attempts to define actions in a general way that accommodates both mainstream, strongly-typed languages as well as alternative languages. The messaging model presented here does not require strong typing but is compatible with it. It can operate on an object basis (as in *self*), in which case typing is irrelevant. It can operate on a class basis in which run-time types are used for request resolution (as in Smalltalk). It can also operate on a class basis with predefined types for requests and replies, to support strong typing as in C++ and other languages.

In general, in this messaging model, a request is an object whose type is determined at run time. This model is usual for signals. However, most languages assume that an operation is specified as part of a call at compile time. This may be modeled in the Action Semantics as a constraint on the type of the request and reply objects, corresponding to fixing the operation.

This model can also accommodate pointers to operations. For example, an *integrate* procedure takes as one of its parameters a pointer to the function to integrate. This is easily handled in this model because the request is an object representing the function. A constraint can be placed on the type of request object that might be passed. For example, the integration function must be an operation with one numerical input and one numerical output. Rather than fixing the exact operation, any operation that meets the constraints could be used.

Variant reply types in a strong typing system are interpreted as jumps within the scope of the calling action and trigger the jump handling mechanisms (Section 2.25, “Jump Actions,” on page 2-332).

2.24.13 Transmitting messages

The execution of request invocations implies “information in transit.” The action semantics in this document are not affected by the presence of a request in transit, because there is no way for an action to access this state.

UML extensions or profiles could be specified in which the format of transmitted messages is defined and actions are provided to access it at run time. Similarly, profiles could be specified in which transmission time or the possibility of errors during transmission are present. All such profiles are likely to be implementation dependent.

2.24.14 Return information

During the execution of a synchronous invocation, the invoked procedure execution must have sufficient information to be able to awaken the invoking action execution when execution of the invoked procedure is complete. The manner in which this

information is stored is part of the implementation and is not defined here. An invoked procedure execution has no explicit access to such information; it is used implicitly on the completion of the procedure execution.

UML extensions or profiles could be specified in which the format of return information is defined and actions are provided to access it at run time.

2.24.15 Messaging Classes

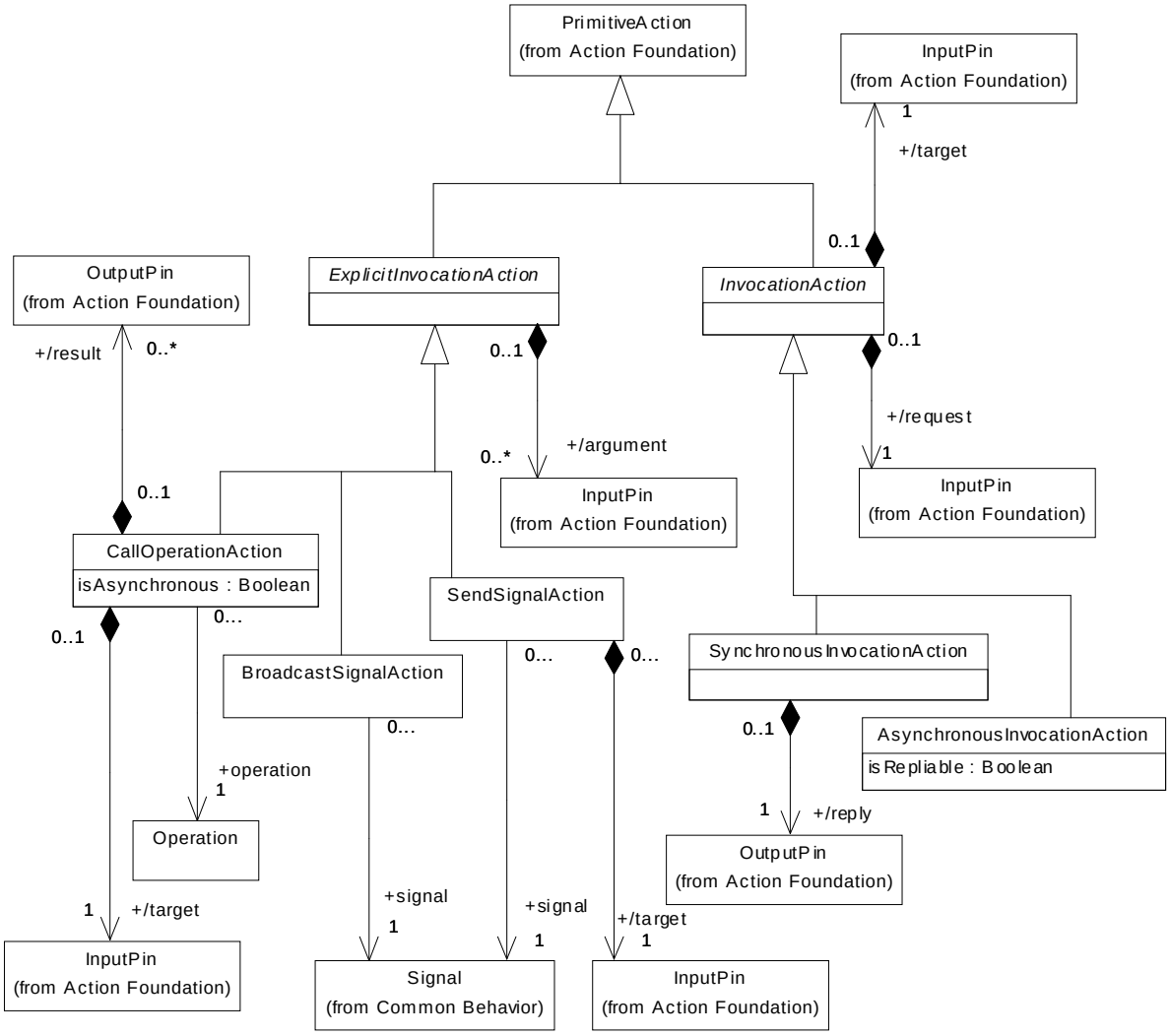


Figure 2-60 Messaging classes metamodel

2.24.15.1 AsynchronousInvocationAction

Creates a request that is transmitted to the target object, where it causes the execution of an effect, such as a method or the triggering of a transition. The request object is available to the execution of invoked procedures. The requestor continues execution without waiting for the request to be delivered or handled. If the invocation is repliable, a subsequent reply by the invoked execution is transmitted to the requesting object as an asynchronous request. If the invocation is not repliable, any attempt by the invoked execution to issue a reply is ignored.

Attributes

- isRepliable: Boolean
  If true, the identity of the invoking object is transmitted as part of the request. If the receiver later reaches a reply point within a synchronous context, a reply message is transmitted to the invoking object as an asynchronous message.
  If false, the identity of the invoker is not transmitted and any attempt to later issue a reply is ignored.

Associations

None

Inputs

- target: T, where T is any user class
  The target object. Information in the object or the object's class is used to determine a method procedure or procedures to execute the operation. To send to a set of objects, use a map action around an invocation action.
- request: U, where U is any user class
  An object whose type represents the kind of operation or signal sent and whose attributes are the arguments of the operation or signal.

Outputs

None

Well-formedness rules

None.

Semantics

1. When all the control and data flow prerequisites of the action execution are satisfied, a copy of the *request* object is transmitted to the target object. The identity of the request object is not preserved in the transmission, but the identities of its attributes are preserved. (In other words, the request is just a collection of values and has no significance as an object.) If the invocation is repliable, the identity of the invoking object is transmitted as part of the request. The target object may be

local or remote. The manner of encoding the transmission, the amount of time required to transmit it, and the path for reaching the target object are undefined. (They are appropriate topics for a runtime implementation profile.)

2. When the transmission arrives at the target object, it causes a behavioral effect on the target object as specified in the target object itself or in the class of the target object. A copy of the request object is available to the behavioral effect. The manner of specifying behavior effects depends on the particular effect. For example, if the request type appears as a reception for the target class, the effect is a signal event for the state machine of the target object; the request object is available as an argument to procedures invoked by transitions caused by the signal. If the request type appears as the signature of an operation for the target class, the effect is the execution of the procedure for the method realizing the operation.

3. If the behavioral effect attempts to return control to a repliable invocation, the reply object is transmitted to the invoking object as an asynchronous request. If the invocation is not repliable, the attempt to reply is simply ignored.

2.24.15.2 BroadcastSignalAction

Creates a request signal that is transmitted to all the potential target objects in the system, where it may cause the firing of state machine transitions and the execution of attached procedures. The argument values are available to the execution of attached procedures. The requestor continues execution immediately. Any attempt to issue a reply is ignored.

The definition of the set of target objects in the system requires an implementation-dependent specification, as any practical implementation of this action obviously requires a finite set of objects.

Attributes

None

Associations

- signal: Signal
  The kind of signal transmitted to the target object.

Inputs

- argument: T [0..*], where T = self.signal.attribute.type
  A tuple of values that become the attributes of the signal.

Outputs

None

Well-formedness rules

None.

Semantics

1. When all the control and data flow prerequisites of the action execution are satisfied, the argument values are formed into a request object that is transmitted concurrently to each of the target objects in the system. The target objects may be local or remote. The signal type is encoded into the transmission. The manner of encoding the transmission, the amount of time required to transmit it, and the path for reaching the target objects are undefined. (They are appropriate topics for a runtime implementation profile.)

2. When a transmission arrives at a target object, it causes a signal event in the target object. The signal has the type specified by the action and its attributes are the arguments of the action. The effect of receiving a signal event is specified elsewhere; normally it involves causing the firing of a state machine transmission and the execution of an attached procedure. If the isList value of the procedure is true, the original argument values of the broadcast signal action are made available to the procedure execution as the separate values of its argument pins.

3. A broadcast signal action provides no return information to the invoked behavioral effect. If the invoked procedure attempts to return control to the broadcast signal action execution, the attempt is simply ignored.

2.24.15.3 CallOperationAction

Assembles the call arguments into an operation call request that is transmitted to the target object, where it causes the selection of a method and the execution of its procedure. The argument values are available to the execution of the invoked procedure as predefined OutputPin values. (They are output pins because they represent values available within the procedure.) The action execution waits until the effect invoked by the request completes and returns to the caller. When the execution of a procedure is complete, its result values are returned to the calling execution. When a return message is received, execution of the action is complete and the return values are used as the result values of the call operation action execution.

Note – The semantics of this action could be mapped onto the SynchronousRequestAction and the AsynchronousRequestAction (depending on the isSynchronous flag), so that the semantics of this action are purely derivative. There is no requirement that this be done, however, either in the logical model or in an actual implementation; this action may be implemented directly.

Attributes

- isSynchronous: Boolean
  If true, the call is synchronous and the caller waits for completion and a reply.
  If false, the call is asynchronous and the caller proceeds immediately and does not expect a reply.

Associations

- operation: Operation
  The operation to be invoked by the action execution.

Inputs

- target: T, where T is any user type
  The target object. The object's class is used to determine a method to execute the operation.
- argument: U [0..*], where U = self.operation.parameter.type for which self.operation.parameter.kind = in or inout
  A tuple of input values whose types must match the input parameters of the operation (including any in-out parameters).

Outputs

- result:V [0..*], where V = self.operation.parameter.type for which self.operation.parameter.kind = return or out or inout
  A tuple of output values whose types must match the output parameters of the operation (including any in-out parameters).

Semantics

1. When all the control and data flow prerequisites of the action execution are satisfied, information comprising the operation and the argument pin values of the action execution is created and transmitted to the target object. The target object may be local or remote. The manner of encoding the transmission, the amount of time required to transmit it, and the path for reaching the target object are undefined. (They are appropriate topics for a runtime implementation profile.) If the call is asynchronous, the caller proceeds immediately. If the call is synchronous, the caller is blocked from further execution until it receives a reply as a consequence of the call.

2. When the transmission arrives at the target object, it causes the selection of a method realizing the operation in the class of the target object. The method is selected according to the inheritance rules for inheritance of methods. The procedure implementing this method is executed. The argument values from the call invocation are made available to the procedure execution as the predefined values of its argument pins. If the call is synchronous, the runtime environment encodes (in an unspecified manner) return information sufficient to identify the invoking action execution. The return information is bound to the invoked procedure execution, but is not accessible to it.

3. If the call is synchronous, when the execution of the invoked procedure completes, the values of its *result* pins are formed into reply information that is transmitted to the calling action execution using the return information bound to the procedure execution. The manner of encoding the reply information for transmission, the time required for transmission, and the transmission path are unspecified.

4. When the reply information arrives at the invoking action execution, copies of the values of its attributes are placed on its result pins, and the execution of the CallOperationAction is complete.

2.24.15.4 ExplicitInvocationAction (abstract)

Abstract action that indicates sending a request object to a target object using an explicit argument list. Creates a request that is transmitted to the target object. The request is resolved into a behavioral effect by the target object or its class based on the type of the request. Depending on the kind of action, the requestor may or may not wait for a reply.

Attributes

None

Associations

None

Inputs

- argument: T [0..*], where T is determined by the specific subclass of action
  A list of values that are the arguments to the invocation.

Outputs

None -- Depends on subclass.

Well-formedness rules

None.

Semantics

See the particular subclass.

2.24.15.5 InvocationAction (abstract)

Abstract action that indicates sending a request object to a target object. Creates a request that is transmitted to the target object. The request is resolved into a behavioral effect by the target object or its class based on the type of the request. Depending on the kind of action, the requestor may or may not wait for a reply.

Attributes

None

Associations

None

Inputs

- target: T, where T is any user class
  The target object. Information in the object or the object's class is used to determine a method procedure or procedures to execute the operation.
- request: U, where U is any user class
  An object whose type represents the kind of operation or signal sent and whose attributes are the arguments of the operation or signal.

Outputs

None -- Depends on subclass.

Action Description: The request is sent to the target object, where it causes the execution of a method or the triggering of a transition. The request object is available to the execution of invoked procedures. The behavior of the requestor depends on the specific concrete action.

Well-formedness rules

None.

Semantics

See the particular subclass.

2.24.15.6 SendSignalAction

Creates a request signal that is transmitted to the target object where it may cause the firing of a state machine transition and the execution of an attached procedure. The argument values are available to the execution of attached procedures. The requestor continues execution without waiting for the request to be delivered or handled. Any attempt by the state machine to issue a reply is ignored.

Attributes

None

Associations

- signal: Signal
  The signal transmitted to the target object.

Inputs

- target: T, where T is any user class
  The target object. The signal will be sent to this object and processed by its state machine. To send to a set of objects, use a map action around a send signal action or use a BroadcastSignalAction to send indiscriminately to the entire available system.

- argument: U [0..*], where U = self.signal.attribute.type
  A tuple of values that become the attributes of the signal.

Outputs

None

Well-formedness rules

None.

Semantics

1. When all the control and data flow prerequisites of the action execution are satisfied, the argument values are formed into a request object that is transmitted to the target object. The target object may be local or remote. The signal type is encoded into the transmission. The manner of encoding the transmission, the amount of time required to transmit it, and the path for reaching the target object are undefined. (They are appropriate topics for a runtime implementation profile.)

2. When the transmission arrives at the target object, it causes a signal event in the target object. The signal has the type specified by the action and its attributes are the arguments of the action. The effect of receiving a signal event is specified elsewhere; normally it involves causing the firing of a state machine transmission and the execution of an attached procedure. If a procedure is executed, the original argument values of the send signal action are made available to it as values of its argument pins (the isList value of the procedure must be true).

3. A send signal action provides no return information to the invoked behavioral effect. If the invoked procedure attempts to return control to the send signal action execution, the attempt is simply ignored.

2.24.15.7 SynchronousInvocationAction

Creates a request packet that is transmitted to the target object where it causes an effect, such as the execution of a method or the triggering of a transition. The request object is available to the execution of invoked procedures. The action execution waits until the effect invoked by the request completes and returns a reply message to the caller. When a reply message is received, execution of the action is complete and the reply message is used as the output of the call action execution.

Attributes

None

Associations

None

Inputs

- target: T, where T is any user class
  The target object. Information in the object or the object's class is used to determine an effect to execute the operation.
- request: U, where U is any user class
  An object whose type represents the kind of operation or signal sent and whose attributes are the arguments of the operation or signal.

Outputs

- reply: V, where V is any user class
  The results produced by executing the selected effect with the given arguments. The number and types of the results is determined by the executed effect. The model may place a constraint on the type of value returned (as with an ordinary operation).

Well-formedness rules

None.

Semantics

1. When all the control and data flow prerequisites of the action execution are satisfied, a copy of the *request* object is transmitted to the target object. The identity of the request object is not preserved in the transmission, but the identities of its attributes are preserved. (In other words, the request is just a collection of values and has no significance as an object.) The target object may be local or remote. The manner of encoding the transmission, the amount of time required to transmit it, and the path for reaching the target object are undefined. (They are appropriate topics for a runtime implementation profile.) The invoking action execution is blocked until it subsequently receives a return object as a result of the behavioral effect caused by the request.

2. When the transmission arrives at the target object, it causes a behavioral effect on the target object as specified in the target object itself or in the class of the target object. A copy of the request object is available to the behavioral effect. The manner of specifying behavior effects depends on the particular effect. For example, if the request type appears as a reception for the target class, the effect is a signal event for the state machine of the target object; the request object is available as an argument to procedures invoked by transitions caused by the signal. If the request type appears as the signature of an operation for the target class, the effect is the execution of the procedure for the method realizing the operation. The runtime environment encodes (in an unspecified manner) return information sufficient to identify the invoking action execution. The return information is bound to the invoked behavioral effect, but is not accessible to it.

3. A behavioral effect invoked by a synchronous invocation must eventually reach a return point defined by the behavior effect. For a procedure invoked as the method realizing an operation or as a consequence of the firing of a state machine transition, the return point is the completion of execution of the procedure. The definition of the return point includes a set of return values. For an invoked

procedure, the return values are the values on its *result* pins when it completes execution. Behavioral effects that do not invoke procedure executions must define how their return points and return values are determined. When control within a behavioral effect reaches the return point, a return object of a type specified by the behavioral effect is created, and its attribute values are the return values of the behavior effect. Using the return information bound to the behavior effect, the return object is transmitted to the invoking action execution. The manner of encoding the return object for transmission, the time required for transmission, and the transmission path are unspecified.

4. When the return object arrives at the invoking action execution, a copy of it becomes the value of the *reply* pin of the action execution, and the execution of the SynchronousInvocationAction is complete.

If the invoked behavioral effect never reaches a reply point, either because the effect is an asynchronous effect or the effect gets caught in an endless loop, then the invoking action execution will remain blocked forever, which is probably not a good thing for the designer but does not violate any rules.

2.24.16 Optional Profile for Resolution of Operations and Signals

Traditional operations and signals can be modeled using the *CallOperationAction* and the *SendSignalAction*, but this does not provide the ability to mix operations and signals and hide the implementation from the requestor. This section defines an optional *UML Profile for Messaging of Operations and Signals*, or *ResolvedFeature* for short. This profile is not part of the basic action semantics specification. It is an optional compliance point and need not be implemented to achieve UML compliance. It is expected that other profiles will be proposed to provide other kinds of resolution.

This kind of resolution can be implemented using the profile shown in Tabl e. The only additions in this profile are the two stereotyped tags of the *OperationResolution* stereotype of BehavioralFeature named *inputSignature* and *outputSignature*, the values of which are of type Classifier. Each distinct resolved Operation must define two attached Classifiers: one for its arguments (inputSignature) and one for its results (outputSignature). (For an asynchronous operation, there is only one classifier, the inputSignature.) These classes are derivable from the operation specification, namely, the list of parameters, and would not have to be explicitly defined by modelers. The same two classes are attached to the Methods for each Operation.

Each distinct resolved Reception must define one attached Classifier, the inputSignature. This Classifier is the Signal classifier itself. If the Reception generates a reply suitable for a synchronous invocation, the Reception must also define an outputSignature classifier.

On an invocation action, the type of the request object is matched against the inputSignatures attached to the behavioral features of the class of the target object. If the matching behavioral feature is an Operation, the normal inheritance rules are used to find a Method, whose Procedure is executed with the request object as its single argument. If the matching behavioral feature is a Signal, a signal event occurs in the

target object. If the handling of the event results in the firing of a transition that invokes a Procedure, the procedure is executed with the request object as its single argument.

If an invocation matches more than one operation and/or signal, then the model is ill formed.

If a method is executed on a synchronous invocation, a reply is generated when the execution of the invoked procedure is complete. The output values of the operation are formed into a reply object, whose type is the outputSignature class and whose attribute values are the result values of the operation. The reply object is returned to the calling action execution.

If a transition fires as a result of the reception of a synchronous request, the procedure (if any) attached to the transition itself (rather than any entry/exit procedures) is regarded as the main procedure. The reply point occurs on the completion of execution of this procedure. A reply object is created whose type is the outputSignature for the Reception. The output values of this procedure become the attributes of the reply object. The reply object is returned to the calling action execution. Any entry procedures caused by the firing of the transition will execute concurrently with the invoker of the request.

If more than one transition is triggered by a synchronous request or if more than one procedure is attached to a triggered transition, the model is ill formed. A more complicated profile could provide for such possibilities by specifying how to choose or assemble the eventual reply object if multiple procedure executions are caused by a transition. In any case, eventually one reply object must be returned to a synchronous requester that triggers a state machine.

If an exception or other jump propagates to the top level of a procedure, the jump object becomes the reply object that is returned to the caller. If the type of this object is not the type specified as the output type of the SynchronousInvocationAction, a jump is raised in the context of the action execution and may be handled by a handler on the invocation action execution or may propagate upward.

These semantics implement traditional operations and signals in the context of the unified messaging model. Note that the request object can be dynamically created; therefore, this profile supports dynamic operation determination not possible with CallOperationAction. Other profiles could be defined that would implement other semantics, such as object-based delegation (as in *self*), operations with variant return types, before-after methods as in CLOS, Ada-style rendezvous, etc.

Stereotype	Base Class	Parent	Tags	Constraints	Description
Resolved-Feature	BehavioralFeature	N/A	inputSignature, outputSignature	none	Operations with this stereotype have a resolution mechanism as defined here..

Tag	Stereotype	Type	Multiplicity	Description
inputSignature	ResolvedFeature	Class	0..1	Specifies the class that serves as the request type for a request that resolves to the operation.
outputSignature	ResolvedFeature	Class	0..1	Specifies the class that serves as the reply type for a request that resolves to the operation.

2.25 Jump Actions

All flow of control for a procedure *could* use Dijkstra-style, fully nested flow-of-control constructs, but this style can be awkward and obscure when dealing with unusual or secondary conditions that do not follow the main line. Programming languages include constructs such as *break*, *continue*, and *exceptions* for dealing with these situations. When a non-mainline situation occurs, the normal flow of control is abandoned and a different flow of control, specified in the program, is taken. The UML jump construct unifies these nonlinear flow-of-control mechanisms while providing the functionality found in most modern programming languages.

2.25.1 Jumps

A *jump* is a condition that occurs synchronously during the execution of an action or a sequence of actions that causes the abandonment of the normal execution sequence (at a definite known location) and the execution of an alternate action (sequence) that brings the execution to a known state compatible with the successors of the action that would have been executed had the jump not occurred. In other words, under certain conditions, execution of the current action is aborted and control jumps to some enclosing level at which a handler action cleans things up and allows control to resume after the higher level action. A jump is used to handle a situation that is inconvenient to handle using linear flow of control. Jumps are often used to handle unexpected inputs or situations regarded as errors, such as exceptions, but they are not restricted to such use and they may be considered another control mechanism, albeit one that should be used with restraint. The traditional *break* and *continue* statements are a typical use of jumps as a static control mechanism. The traditional exception handling mechanism is a typical use of jumps as a more dynamic control mechanism.

A *jump type* is a classifier, in the same way that a signal type is declared as a classifier. There is no jump type metaclass; any class may be used as a jump type. Jumps may be explicitly caused by a jump action. For example, a traditional break statement can be modeled as a jump action with a predefined Break class as the jump type. A primitive action may also cause jumps as part of its behavior for given input values. Such a usage of a jump to handle an abnormal or non-mainline situation is often called an *exception*. For example, a *squareRoot* action might specify that the Irrational jump occurs if the input argument is less than zero, and the jump object has one attribute whose value is the square root of the absolute value of the argument. An arrayIndex action might specify that an OutOfBounds jump is caused when an index argument is not legal for an array; the jump type would have the index value and the array bounds as attributes.

The occurrence of a jump is manifested as an instance—a jump object—of the given jump type. The object types indicate what kind of situation caused the jump. The attribute values (if any) describe the situation in more detail. The occurrence of a jump terminates the current action and transfers control to a jump handler attached to the current action. If the current action lacks a jump handler for the given jump type, the jump propagates to enclosing levels of actions.

A jump handler may be attached to an action (primitive or composite). A jump handler is a map: a set of jump-type-to-action pairs, similar to the signal-type-to-transition pairs attached to states in state machines. If a jump occurs during the execution of an action and the jump type or one of its ancestor types appears in the jump handler attached to the action, then the execution of the action is abandoned and the handler action corresponding to the jump type is executed instead. A jump object is created with the argument values specified by the action. The jump object is the input to the handler action. The handler action may also access output values in its accessible scope, like any action. The accessible values in a handler are the same as the accessible values in the action that it protects. When the execution of a handler action is completed, the execution of the original action is deemed to be complete and successors are enabled. If the original action has any data flow outputs, each of its handlers must have a list of data flow outputs whose number, order, and types match the output list of the action (otherwise the successors will not have needed values available), therefore an action with handlers has much the form of a conditional. There is one exception permitted: A handler action that always causes a jump during its execution must have zero output pins and need not match the outputs of the protected action (because it will never complete normally). A handler action is ill formed if it can possibly complete normally but the pins do not match.

Actions within a jump handler may have jump handlers attached to them. If a jump occurs during the execution of a jump handler action, the execution of the handler action causing the jump occurrence is terminated, and its jump handler (if any) is started. If there is no jump handler, the jump propagates upward. If the propagated jump reaches the top of the jump handler without being handled, it is propagated to the action enclosing the action having the jump handler. If a second jump occurs during the processing of another jump, the original jump is lost.

If a jump is handled directly by a handler on the action causing the jump, then nothing else need be done. However, if an action does not have a handler for a jump, then the jump occurrence propagates to the immediate enclosing composite action, where it may trigger a jump handler on the wider scope. If jump propagation occurs, the execution of any successors to the original action will not have begun and the execution of any predecessors of the original action will have completed, so the state of execution of a linear sequence of actions is determinate. Concurrent executions within the composite scope must complete before propagation of the jump may continue. Eventually the addition of an interrupt mechanism to UML would permit concurrent executions to be aborted due to the occurrence of an unhandled jump in another action execution.

If a jump is not handled by the outermost action in a procedure, then the execution of the procedure terminates and a reply object is returned to the caller of the procedure. Such a reply object is just an object of a particular reply type, different from the reply

type expected for a normal return. This permits programming languages to treat jump returns (such as exceptions) in two different, but essentially equivalent, ways: as multiple variant return types handled inline as ordinary values, or as a deviation from an expected default return type in which the receipt of any non-default types cause jumps within the calling scope. This approach permits a procedure to have multiple return types and have them treated either as variant returns or as exceptions, at the option of the caller. See Section 2.24.5, “Reply Handling.

Note that jump handling does not add any fundamental power to the semantics. Each action *could* instead output status values that would be checked by conditionals, with the jump handlers being clauses of the conditionals executed if jump types were returned. Experience with programming languages has shown that such a program organization obscures its fundamental structure and makes changes difficult. Jump handling permits separation of concerns between normal control flow and unusual situations that require special handling.

2.25.2 Break and Continue Statements

In many programming languages, a *break* statement terminates the execution of the current composite flow-of-control construct, such as a block, a case statement, or a conditional. Some languages have the ability to break out of several nested levels of control by supplying an argument to designate the level to break to. A *continue* statement terminates the execution of the body of a loop construct and advances to the next iteration. There are many variations in the way these statements work in different languages. The UML jump construct can represent these statements with flexibility. Each of these static flow-of-control constructs can be represented in UML as a JumpAction. The differences come in the type of jump object supplied and the place where the matching jump handler is placed.

A simple break statement can be modeled as a JumpAction whose argument type is a special predefined class, such as *BreakJump*. This class has no attributes. A break handler for this type is placed on the immediately enclosing GroupAction. The break handler body is the null action. Execution of the JumpAction simply terminates execution of the current group of actions and resumes control beyond the group action. The same *BreakJump* type can be used throughout the entire model to represent all simple break statements. Each one will be caught by its immediately enclosing group action. The entire structure is statically determined. On a loop, the break handler would be placed on the LoopAction itself to terminate execution of the loop.

If the enclosing GroupAction has data flow outputs, the handler must generate them. This is a situation that does not occur in traditional programming languages, but it is straightforward to model and does not present any difficulties in concept or implementation. In some situations, the modeler may choose to define a special jump type whose attributes include the output values of the group action or data needed to compute them. This is a powerful but straightforward extension of the traditional break statement involving parameters.

To represent a break statement that jumps over several nested levels, use a special jump type caught only by the desired enclosing composite action. Actions in any embedded nesting level can potentially cause a jump of the given type.

A simple *continue* statement can be modeled as a JumpAction whose argument type is a special predefined class, such as *ContinueJump*. This class has no arguments. A break handler for this type is placed on the (not necessarily immediately) enclosing GroupAction that represents the body of the loop. The break handler body is a null action. Execution of the JumpAction terminates execution of the current group of actions, and the jump propagates through any additional levels of nesting until it is caught by the appropriate level representing the loop body, whose execution is terminated. Control will then resume with the next iteration of the loop. The same *ContinueJump* type can be used throughout the entire model to represent all simple continue statements.

If the enclosing loop body has data flow outputs, the handler must generate them. In particular, if the loop has dataflow loop variables, their values must be generated by the handler. Often the use of a continue statement represents a situation in which the loop variables take on default values. In complicated situations, it may be necessary to define a special jump type for the particular loop so that the information needed can be passed as part of the jump. This becomes a parameterized continue statement.

2.25.3 Exceptions

In many programming languages, exceptions may occur during the execution of certain statements. In most languages, an exception can also be raised explicitly by a program statement. The occurrence of an exception terminates the current statement and causes control to jump to an enclosing statement that has an exception handler to catch the particular type of exception. The UML jump construct can represent traditional exception handling with flexibility. (Note that UML includes a metaclass called Exception. This metaclass represents interobject error conditions handled using state machines. The kind of exceptions found in traditional programming languages represent syntactic control constructs within a single thread of control, and are not related to or well modeled by the Exception metaclass. This section discusses the syntactic programming-language kind of exceptions, not the Exception metaclass.)

The occurrence of an exception is modeled in UML as a jump object. The type of the jump object represents the kind of exception. The arguments of the jump object represent the parameters of the exception. Explicitly raising an exception is modeled using the JumpAction.

In addition, primitive functions may be predefined to raise exceptions for particular arguments. For example, the divide function may be defined to cause a DivideByZero jump if the quotient is zero. The possibility of causing jumps is built into the definition of the function and represents part of its overall input-output mapping. Most users will not define their own primitive functions; usually those are predefined in a particularly computing environment, including their possible exceptions. However, any definition of a primitive function must specify the types of jumps that may occur, the input values for which they occur, and the mapping from input values to jump object attribute values.

An exception handler is modeled as a jump handler that catches the given jump type. Because the jump object may have attributes, the jump handler has access to relevant information about the circumstances that caused the exception and can use the information in dealing with the situation.

2.25.4 Jumps with Concurrent Executions

If a jump propagates to a composite action, the execution of actions concurrent with the original action must be dealt with before the execution of the composite action can be abandoned, and their state at the time of the jump occurrence is indeterminate. The simplest approach is to wait for all concurrent executions to complete normally, under the assumption that they are unaffected by an error in a separate region. This approach is supported by the current semantics.

Another possibility is that the concurrent executions are made irrelevant by an exception and should be terminated, but this decision must be made by the modeler in each specific situation. An *interrupt* is a condition that occurs asynchronously during the execution of an action or action sequence that may force a change in execution without waiting for normal completion. Dealing with the execution of concurrent actions because of the propagation of a jump to an enclosing scope is an interrupt situation. It might occur anywhere in the execution sequence of the concurrent action. In the simplest case, the execution of concurrent action could simply be allowed to complete, but in many situations an exception in a concurrent execution means that their results are worthless. Alternately, all concurrent executions could be abandoned, but there are many situations in which the affected concurrent action should be given a chance to clean itself up before terminating (or possibly even ignore the interrupt and complete normally). Past experience with operating systems has shown that there are various ways to deal with interrupts. It is anticipated that interrupt handling will be provided in a future update to UML. It is anticipated that the propagation of a jump to an action execution with concurrent executions would be a situation in which an interrupt would occur or could be made to occur. There is a likelihood that interrupt mechanisms will depend on the implementation and might therefore be defined in layers built on top of basic UML.

If two jumps occur concurrently in different concurrent actions, it is indeterminate which jump (or possibly a third jump) will be raised at the level of the composite action. In any case, only one jump will be propagated upward and the others are lost. When a jump propagates upward, the execution of the composite action is complete and no activity or information (such as additional jump occurrences) remains.

Without the use of interrupts, a jump may propagate upward only after concurrent executions complete.

2.25.5 Jump Classes

The metamodel in Figure 2-61 shows the jump classes.

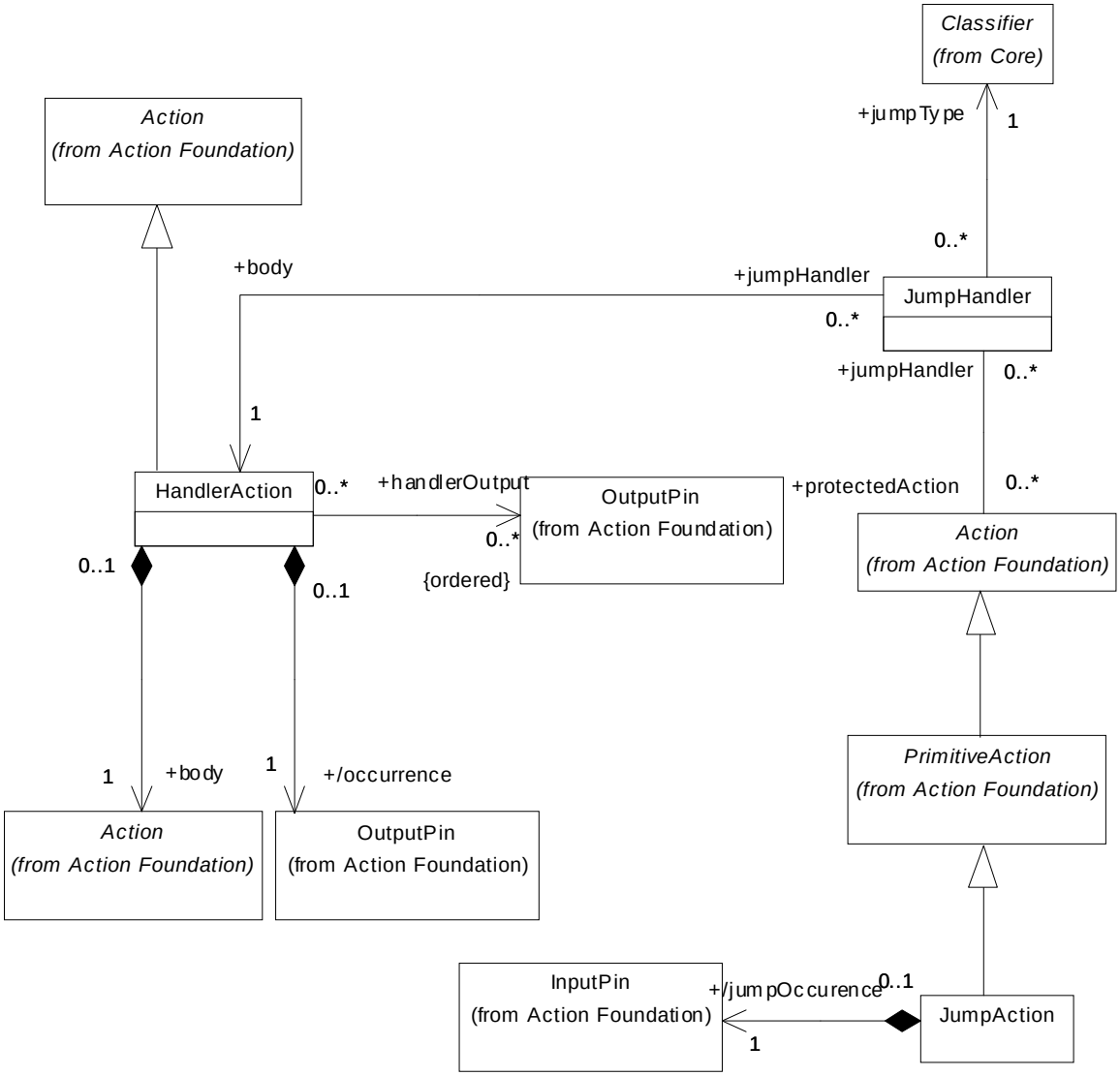


Figure 2-61 Jump handling classes

Note that jump handling extends the definitions of several other action classes to provide the ability to attach jump handlers or to define the behavior of the action when a jump occurs.

2.25.5.1 Action

(in addition to the original definition in Action Foundation)

Associations

- jumpHandler: JumpHandler [0..*]
  Designates a set of JumpHandlers whose jump types apply to the action. If a jump of the given jump type or a descendant type occurs during the execution of the owning action, the HandlerAction attached to the JumpHandler is executed. If the jump type is not present on any jump handler attached to the action, the jump propagates to the enclosing action.

2.25.5.2 HandlerAction

An action may have zero or more handlers attached to it. A handler deals with jumps that occur during the execution of its underlying action. A jump handler associates actions and jump types with the handler actions that replace the actions if a jump of the given type (or a descendant of the jump type) occurs. An action used as a handler is a kind of composite action. It must have one internal output pin; the output pin receives the jump occurrence object and makes it available within the handler action. A jump handler action must have a list of output pins that matches in number, order, and types the outputs of the action for which it handles jumps. If it does not have a matching list of output pins, it may still catch the jump, but it must re-raise the same or another jump rather than completing normally (otherwise the outputs of the original action would not be defined). The handler action has a body action (often a group action) that it executes. The occurrence pin is available to the body action as an available input.

Associations

- jumpHandler: JumpHandler [1..1]
  The handler catches any jump of the given jump type or a descendant type during an execution of its attached action. The handler action must own one internal output pin that receives the jump occurrence object. It must have a list of output pins that matches in number, order, and types those of the action it protects.
- body: Action [1..1]
  The action that is triggered by the jump handler. The occurrence output pin of the handler action is available to the body action. All inputs available to the protected action are also available to the body action. No output pins of the body action are accessible outside the jump handler except the designated handlerOutput pins.

Inputs

none

Outputs

- occurrence: T [1..1], where T = self.jumpHandler.jumpType
  [*An output pin owned by the HandlerAction itself and available to the body action but not outside the HandlerAction.]* During the execution of the handler action, the value of the jump occurrence object is available on this pin.
- handlerOutput: U [0..*], where U = self.jumpHandler.protectedAction..availableOutput.type, for all combinations of jumpHandler and protectedAction

[*A list of output pins owned by the body action but designated by the handler action. The number and types of the pins in the ordered list must match the number and types of the pins that are outputs of the action protected by the handler action.*] At the completion of execution of the body action for a jump handler, the values on these pins are copied to the output pins of the protected action, and the successors of the protected action are enabled as if it had completed normally.

Well-formedness rules

[1] An action used as a jump handler must have output pins that match the output pins of the protected action.

self.body.outputPin.type = self.jumpHandler.protectedAction.outputPin.type

[2] The occurrence input pin must have a type matching the corresponding jump type.

self.occurrence.type = self.jumpHandler.jumpType

2.25.5.3 JumpAction

An action whose execution causes the occurrence of a specified jump with specified arguments. The execution of this action has the remarkable property that it cannot complete normally. Although a jump handler could (somewhat perversely) be attached to the jump action itself, usually the purpose of a jump action is to terminate a group of actions containing the raise action and escape to a jump handler at some enclosing level. The use of this action is the normal way to represent *break* and *continue* statements from programming languages.

Inputs

- jumpOccurrence: T [1..1], where T is any user class
  The jump occurrence object caused by the action. When the action is executed, a jump occurs and the object becomes its the jump occurrence object. The execution of the jump action is terminated (!) and the normal process of jump handling occurs using the given jump object.

Outputs

none

2.25.5.4 JumpHandler

(not an action) Essentially the reification of a qualified association relating an Action and a jump type to the HandlerAction that is invoked if the jump occurs during the execution of the action.

Associations

- body: HandlerAction [1..1]
  The action executed if an occurrence of the given jump type occurs during an execution of the attached action.

- jumpType: Classifier [1..1]
  The type of jump caught by the jump handler.
- protectedAction [0..*]
  The action whose executions are protected by the jump handler. The same handler can protect multiple actions.

Well-formedness rules

none

2.25.6 Additional Jump Semantics for Actions Defined Elsewhere

2.25.6.1 ConditionalAction

Semantics

(Additional semantics for jump handling)

If a jump fails to be handled by an executing body action, reaches the propagating status, the jump propagates to the conditional execution action itself. The execution of the conditional is terminated. Normal jump handling occurs.

If jump fails to be handled by the execution of a test action of a clause and no other clauses are executing, the jump propagates to the conditional execution itself. The execution of the conditional is terminated. Normal jump handling occurs. Note that if some clause returns true, execution of its body proceeds in spite of a jump in the test action from another clause.

2.25.6.2 FilterAction

Semantics

(Additional semantics for jump handling)

If a jump is propagated from a subordinate subaction, the jump is considered raised in the filter execution. If there are other active subordinate subactions, they must complete before jump handling can continue. When there are no active subordinate subactions, the filter execution is terminated and the jump is raised on it. If more than one subordinate action propagates a jump, it is indeterminate which jump will be propagated to the filter execution.

2.25.6.3 GroupAction

Semantics

(Additional semantics for jump handling)

If a jump fails to be handled by one of the concurrently executing subactions of the group, the jump is considered raised in the group action execution. Any concurrently executing actions in the group must complete before the jump propagates to the enclosing action.

If more than one jump is propagated concurrently to the group action execution, one of them will be propagated when all of the other concurrent executions are inactive, but it is indeterminate which one.

When interrupts are added to UML, it is expected that the occurrence of an unhandled jump within a group action would be a situation that would generate an interrupt.

2.25.6.4 IterateAction

Semantics

(Additional semantics for jump handling)

If a jump is propagated from the subordinate execution, the iterate execution is terminated and the jump handler of the iterate action handles the jump.

2.25.6.5 LoopAction

Semantics

(Additional semantics for jump handling)

If a jump fails to be handled by either the test action or the body action of a loop, the jump propagates to the loop action itself. The execution of the loop is terminated. Normal jump handling occurs.

2.25.6.6 MapAction

Semantics

(Additional semantics for jump handling)

If a jump is propagated from a subordinate subaction, the jump is considered raised in the map execution. If there are other active subordinate subactions, they must complete before the jump propagates to the map execution itself. When there are no active subordinate subactions, the map execution is terminated and the normal jump handling occurs at the map execution action. If more than one subordinate action propagates a jump, it is indeterminate which jump will be propagated to the map execution.

2.25.6.7 Procedure

Semantics

(Additional semantics for jump handling)

If a jump is propagated from the subordinate action, execution of the procedure is terminated. The jump object is treated as the reply and it is passed to the caller in lieu of the normal reply object. The procedure execution is considered complete. If the invocation was asynchronous, the reply is ignored and no further propagation occurs.

The jump is returned to the caller as the reply object. This permits the caller to handle a jump inline, if desired. If the call result is strongly typed and the jump type does not match the return type, the jump is re-raised in the calling procedure. Therefore no special mechanism is needed to propagate jumps to callers.

2.25.6.8 ReduceAction

Semantics

(Additional semantics for jump handling)

If a jump is propagated from a subordinate execution, execution of the reduce action is terminated and the jump is raised on the reduce action itself.

2.25.7 Jump Value Classes

These classes may be used as jump types to provide certain traditional control flow capabilities.

2.25.7.1 BreakJump

As the type of a jump, represents a traditional break statement. It has no attributes.

Attributes

none

Associations

none

2.25.7.2 ContinueJump

As the type of a jump, represents a traditional continue statement. It has no attributes.

Attributes

none

Associations

none

UML Notation Guide 3

This guide describes the notation for the visual representation of the Unified Modeling Language (UML). This notation document contains brief summaries of the semantics of UML constructs, but the UML Semantics chapter must be consulted for full details.

Contents

This chapter contains the following topics.

Topic	Page
"Part 1 - Background"	
"Introduction"	3-5
Part 2 - Diagram Elements	
"Graphs and Their Contents"	3-6
"Drawing Paths"	3-7
"Invisible Hyperlinks and the Role of Tools"	3-7
"Background Information"	3-8
"String"	3-8
"Name"	3-9
"Label"	3-10
"Keywords"	3-11
"Expression"	3-11
"Type-Instance Correspondence"	3-14
Part 3 - Model Management	

Topic	Page
“Package”	3-16
“Subsystem”	3-19
“Model”	3-24
Part 4 - General Extension Mechanisms	
“Constraint and Comment”	3-26
“Element Properties”	3-29
“Stereotypes”	3-31
Part 5 - Static Structure Diagrams	
“Class Diagram”	3-34
“Object Diagram”	3-35
“Classifier”	3-35
“Class”	3-35
“Name Compartment”	3-38
“List Compartment”	3-38
“Attribute”	3-41
“Operation”	3-44
“Nested Class Declarations”	3-48
“Type and Implementation Class”	3-49
“Interfaces”	3-50
“Parameterized Class (Template)”	3-52
“Bound Element”	3-54
“Utility”	3-56
“Metaclass”	3-57
“Enumeration”	3-57
“Stereotype Declaration”	3-57
“Powertype”	3-61
“Class Pathnames”	3-61
“Accessing or Importing a Package”	3-62
“Object”	3-64
“Composite Object”	3-67
“Association”	3-68
“Binary Association”	3-68

Topic	Page
“Association End”	3-71
“Multiplicity”	3-75
“Qualifier”	3-76
“Association Class”	3-77
“N-ary Association”	3-79
“Composition”	3-81
“Link”	3-84
“Generalization”	3-86
“Dependency”	3-90
“Derived Element”	3-93
“InstanceOf”	3-93
Part 6 - Use Case Diagrams	
“Use Case Diagram”	3-94
“Use Case”	3-96
“Actor”	3-97
“Use Case Relationships”	3-97
“Actor Relationships”	3-99
Part 7 - Interaction Diagrams	
“Collaboration”	3-101
“Sequence Diagram”	3-102
“Object Lifeline”	3-108
“Activation”	3-110
“Message and Stimulus”	3-111
“Transition Times”	3-113
Part 8 - Collaboration Diagrams	
“Collaboration Diagram”	3-114
“Pattern Structure”	3-117
“Collaboration Contents”	3-121
“Interactions”	3-123
“Collaboration Roles”	3-124
“Multiobject”	3-127
“Active object”	3-128

Topic	Page
“Message and Stimulus”	3-111
“Creation/Destruction Markers”	3-134
Part 9 - Statechart Diagrams	
“Statechart Diagram”	3-136
“State”	3-137
“Composite States”	3-140
“Events”	3-142
“Simple Transitions”	3-145
“Transitions to and from Concurrent States”	3-146
“Transitions to and from Composite States”	3-147
“Factored Transition Paths”	3-150
“Submachine States”	3-152
“Synch States”	3-154
Part 10 - Activity Diagrams	
“Activity Diagram”	3-155
“Action state”	3-158
“Subactivity state”	3-159
“Decisions”	3-159
“Call States”	3-161
“Swimlanes”	3-161
“Action-Object Flow Relationships”	3-163
“Control Icons”	3-165
“Synch States”	3-154
“Dynamic Invocation”	3-168
“Conditional Forks”	3-169
Part 11 - Implementation Diagrams	
“Component Diagram”	3-169
“Deployment Diagram”	3-171
“Node”	3-173
“Component”	3-174

Part 1 - Background

3.1 Introduction

This chapter is arranged in parts according to semantic concepts subdivided by diagram types. Within each diagram type, model elements that are found on that diagram and their representation are listed. Note that many model elements are usable in more than one diagram. An attempt has been made to place each description where it is used the most, but be aware that the document involves implicit cross-references and that elements may be useful in places other than the section in which they are described. Be aware also that the document is nonlinear: there are forward references in it. It is not intended to be a teaching document that can be read linearly, but a reference document organized by affinity of concept.

Each part of this chapter is divided into sections, roughly corresponding to important model elements and notational constructs. Note that some of these constructs are used within other constructs; do not be misled by the flattened structure of the chapter. Within each section the following subsections may be found:

- Semantics: Brief summary of semantics. For a fuller explanation and discussion of fine points, see the *UML Semantics* chapter in this specification.
- Notation: Explains the notational representation of the semantic concept ("forward mapping to notation").
- Presentation options: Describes various options in presenting the model information, such as the ability to suppress or filter information, alternate ways of showing things, and suggestions for alternate ways of presenting information within a tool.

  Dynamic tools need the freedom to present information in various ways and the authors do not want to restrict this excessively. In some sense, we are defining the "canonical notation" that printed documents show, rather than the "screen notation." The ability to extend the notation can lead to unintelligible dialects, so we hope this freedom will be used in intuitive ways. The authors have not sought to eliminate all the ambiguity that some of these presentation options may introduce, because the presence of the underlying model in a dynamic tool serves to easily disambiguate things. Note that a tool is not supposed to pick just one of the presentation options and implement it. Tools should offer users the options of selecting among various presentation options, including some that are not described in this document.
- Style guidelines: Include suggestions for the use of stylistic markers, such as fonts, naming conventions, arrangement of symbols that are not explicitly part of the notation, but that help to make diagrams more readable. These are similar to text indentation rules in C++ or Smalltalk. Not everyone will choose to follow these suggestions, but the use of some consistent guidelines of your own choosing is recommended in any case.
- Example: Shows samples of the notation. String and code examples are given in the following font: This is a string sample.

- Mapping: Shows the mapping of notation elements to metamodel elements (“reverse mapping from notation”). This indicates how the notation would be represented as semantic information. Note that, in general, diagrams are interpreted in a particular context in which semantic and graphic information is gathered simultaneously. The assumption is that diagrams are constructed by an editing tool that internalizes the model as the diagram is constructed. Some semantic constructs have no graphic notation and would be shown to a user within a tool using a form or table.

Part 2 - Diagram Elements

3.2 Graphs and Their Contents

Most UML diagrams and some complex symbols are graphs containing nodes connected by paths. The information is mostly in the topology, not in the size or placement of the symbols (there are some exceptions, such as a sequence diagram with a metric time axis). There are three kinds of visual relationships that are important:

1. connection (usually of lines to 2-d shapes),
2. containment (of symbols by 2-d shapes with boundaries), and
3. visual attachment (one symbol being “near” another one on a diagram).

These visual relationships map into connections of nodes in a graph, the parsed form of the notation.

UML notation is intended to be drawn on 2-dimensional surfaces. Some shapes are 2-dimensional projections of 3-d shapes (such as cubes), but they are still rendered as icons on a 2-dimensional surface. In the near future, true 3-dimensional layout and navigation may be possible on desktop machines; however, it is not currently practical.

There are basically four kinds of graphical constructs that are used in UML notation:

1. Icons - An icon is a graphical figure of a fixed size and shape. It does not expand to hold contents. Icons may appear within area symbols, as terminators on paths or as standalone symbols that may or may not be connected to paths.
2. 2-d Symbols - Two-dimensional symbols have variable height and width and they can expand to hold other things, such as lists of strings or other symbols. Many of them are divided into compartments of similar or different kinds. Paths are connected to two-dimensional symbols by terminating the path on the boundary of the symbol. Dragging or deleting a 2-d symbol affects its contents and any paths connected to it.
3. Paths - Sequences of line segments whose endpoints are attached. Conceptually a path is a single topological entity, although its segments may be manipulated graphically. A segment may not exist apart from its path. Paths are always attached to other graphic symbols at both ends (no dangling lines). Paths may have *terminators*; that is, icons that appear in some sequence on the end of the path and that qualify the meaning of the path symbol.

4. Strings - Present various kinds of information in an “unparsed” form. UML assumes that each usage of a string in the notation has a syntax by which it can be parsed into underlying model information. For example, syntaxes are given for attributes, operations, and transitions. These syntaxes are subject to extension by tools as a presentation option. Strings may exist as singular elements of symbols or compartments of symbols, as elements in lists (in which case the position in the list conveys information), as labels attached to symbols or paths, or as stand-alone elements on a diagram.

3.3 Drawing Paths

A path consists of a series of line segments whose endpoints coincide. The entire path is a single topological unit. Line segments may be orthogonal lines, oblique lines, or curved lines. Certain common styles of drawing lines exist: all orthogonal lines, or all straight lines, or curves only for bevels. The line style can be regarded as a tool restriction on default line input. When line segments cross, it may be difficult to know which visual piece goes with which other piece; therefore, a crossing may optionally be shown with a small semicircular jog by one of the segments to indicate that the paths do not intersect or connect (as in an electrical circuit diagram).

In some relationships (such as aggregation and generalization) several paths of the same kind may connect to a single symbol. In some circumstances (described for the particular relationship) the line segments connected to the symbol can be combined into a single line segment, so that the path from that symbol branches into several paths in a kind of tree. This is purely a graphical presentation option; conceptually the individual paths are distinct. This presentation option may not be used when the modeling information on the segments to be combined is not identical.

3.4 Invisible Hyperlinks and the Role of Tools

A notation on a piece of paper contains no hidden information. A notation on a computer screen may contain additional invisible hyperlinks that are not apparent in a static view, but that can be invoked dynamically to access some other piece of information, either in a graphical view or in a textual table. Such dynamic links are as much a part of a *dynamic* notation as the visible information, but this guide does not prescribe their form. We regard them as a tool responsibility. This document attempts to define a *static* notation for the UML, with the understanding that some useful and interesting information may show up poorly or not at all in such a view. On the other hand, we do not know enough to specify the behavior of all dynamic tools, nor do we want to stifle innovation in new forms of dynamic presentation. Eventually some of the dynamic notations may become well enough established to standardize them, but we do not feel that we should do so now.

3.5 Background Information

3.5.1 Presentation Options

Each appearance of a symbol for a class on a diagram or on different diagrams may have its own presentation choices. For example, one symbol for a class may show the attributes and operations and another symbol for the same class may suppress them. Tools may provide style sheets attached either to individual symbols or to entire diagrams. The style sheets would specify the presentation choices. (Style sheets would be applicable to most kinds of symbols, not just classes.)

Not all modeling information is presented most usefully in a graphical notation. Some information is best presented in a textual or tabular format. For example, much detailed programming information is best presented as text lists. The UML does not assume that all of the information in a model will be expressed as diagrams; some of it may only be available as tables. This document does not attempt to prescribe the format of such tables or of the forms that are used to access them, because the underlying information is adequately described in the UML metamodel and the responsibility for presenting tabular information is a tool responsibility. It is assumed that hidden links may exist from graphical items to tabular items.

3.6 String

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters.

3.6.1 Semantics

Diagram strings normally map underlying model strings that store or encode information about the model, although some strings may exist purely on the diagrams. UML assumes that the underlying character set is sufficient for representing multibyte characters in various human languages; in particular, the traditional 8-bit ASCII character set is insufficient. It is assumed that the tool and the computer manipulate and store strings correctly, including escape conventions for special characters, and this document will assume that arbitrary strings can be used without further fuss.

3.6.2 Notation

A string is displayed as a text string graphic. Normal printable characters should be displayed directly. The display of nonprintable characters is unspecified and platform-dependent. Depending on purpose, a string might be shown as a single-line entity or as a paragraph with automatic line breaks.

Typeface and font size are graphic markers that are normally independent of the string itself. They may code for various model properties, some of which are suggested in this document and some of which are left open for the tool or the user.

3.6.3 Presentation Options

Tools may present long strings in various ways, such as truncation to a fixed size, automatic wrapping, or insertion of scroll bars. It is assumed that there is a way to obtain the full string dynamically.

3.6.4 Examples

BankAccount

integrate (f: Function, from: Real, to: Real)

{ author = “Joe Smith”, deadline = 31-March-1997, status = analysis }

The purpose of the shuffle operation is nominally to put the cards into a random configuration. However, to more closely capture the behavior of physical decks, in which blocks of cards may stick together during several riffles, the operation is actually simulated by cutting the deck and merging the cards with an imperfect merge.

3.6.5 Mapping

A graphic string maps into a string within a model element. The mapping depends on context. In some circumstances, the visual string is parsed into multiple model elements. For example, an operation signature is parsed into its various fields. Further details are given with each kind of symbol.

3.7 Name

3.7.1 Semantics

A name is a string that is used to identify a model element uniquely within some scope. A pathname is used to find a model element starting from the root of the system (or from some other point). A name is a selector (qualifier) within some scope—the scope is made clear in this document for each element that can be named.

A pathname is a series of names linked together by a delimiter (such as ‘::’). There are various kinds of pathnames described in this document, each in its proper place and with its particular delimiter.

3.7.2 Notation

A name is displayed as a text string graphic. Normally a name is displayed on a single line and will not contain nonprintable characters. Tools and languages may impose reasonable limits on the length of strings and the character set they use for names, possibly more restrictive than those for arbitrary strings, such as comments.

3.7.3 Example

Names:

BankAccount

integrate

controller

abstract

this_is_a_very_long_name_with_underscores

Pathname:

MathPak::Matrices::BandedMatrix

3.7.4 Mapping

Maps to the name of a model element. The mapping depends on context, as with String. Further details are given with the particular element.

3.8 Label

A label is a string that is attached to a graphic symbol.

3.8.1 Semantics

A label is a term for a particular use of a string on a diagram. It is purely a notational term.

3.8.2 Notation

A label is a string that is attached graphically to another symbol on a diagram. Visually the attachment normally is by containment of the string (in a closed region) or by placing the string near the symbol. Sometimes the string is placed in a definite position (such as below a symbol) but most of the time the statement is that the string must be “near” the symbol. A tool maintains an explicit internal graphic linking between a label and a graphic symbol, so that the label drags with the symbol, but the final appearance of the diagram is a matter of aesthetic judgment and should be made so that there is no confusion about which symbol a label is attached to. Although the attachment may not be obvious from a visual inspection of a diagram, the attachment is clear and unambiguous at the graphic level (and poses no ambiguity in the semantic mapping).

3.8.3 Presentation Options

A tool may visually show the attachment of a label to another symbol using various aids (such as a line in a given color, flashing of matched elements, etc.) as a convenience.

3.8.4 Example

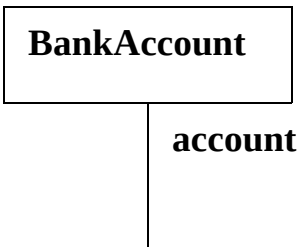


Figure 3-1 Attachment by Containment and Attachment by Adjacency

3.9 Keywords

The number of easily-distinguishable visual symbols is limited. The UML notation makes use of text keywords in places to distinguish variations on a common theme, including metamodel subclasses of a base class, stereotypes of a metamodel base class, and groups of list elements. From the user's perspective, the metamodel distinction between metamodel subclasses and stereotypes is often unimportant, although it is important to tool builders and others who implement the metamodel.

The general notation for the use of a keyword is to enclose it in guillemets («»):

«*keyword***»**

Certain predefined keywords are described in the text of this document. These must be treated as reserved words in the notation. Others are available for users to employ as stereotype names. The use of a stereotype name that matches a predefined keyword is ill formed.

3.10 Expression

3.10.1 Semantics

Various UML constructs require expressions, which are linguistic formulas or procedures that yield values when evaluated at run-time. These include expressions for types, boolean values, and numbers. UML does not include an explicit linguistic analyzer for expressions. Rather, expressions are expressed as strings in a particular

language or using procedures, or both. The OCL constraint language is used within the UML semantic definition and may also be used at the user level; other languages (such as programming languages) may also be used.

UML avoids specifying the syntax for constructing type expressions because they are so language-dependent. It is assumed that the name of a class or simple data type will map into a simple Classifier reference, but the syntax of complicated language-dependent type expressions, such as C++ function pointers, is the responsibility of the specification language.

3.10.2 Notation

An expression is displayed as a string defined in a particular language. The syntax of the string is the responsibility of a tool and a linguistic analyzer for the language. The assumption is that the analyzer can evaluate strings at run-time to yield values of the appropriate type, or can yield a procedure to capture the meaning of the expression. For example, a type expression evaluates to a Classifier reference, and a boolean expression evaluates to a true or false value. The language itself is known to a modeling tool but is generally implicit on the diagram, under the assumption that the form of the expression makes its purpose clear.

3.10.3 Examples

BankAccount

BankAccount * (*) (Person*, int)

array [1..20] of reference to range (-1.0..1.0) of Real

[i > j and self.size > i]

3.10.4 Mapping

An expression string maps to an Expression element (possibly a particular subclass of Expression, such as BooleanExpression or TimeExpression). If an analyzer yields a procedure for calculating the value of the expression, then the body association from Expression to Procedure is used to record this.

3.10.5 OCL Expressions

UML includes a definition of the OCL language, which is used to define constraints within the UML metamodel itself. The OCL language may be supported by tools for user-written expressions as well. Other possible languages include various computer languages as well as plain text (which cannot be parsed by a tool, of course, and is therefore only for human information). The OCL language is defined in the “Object Constraint Language Specification” chapter.

3.10.6 Selected OCL Notation

Syntax for some common navigational expressions are shown below. These forms can be chained together. The leftmost element must be an expression for an object or a set of objects. The expressions are meant to work on sets of values when applicable.

item '.' *selector*	The *selector* is the name of an attribute in the item or the name of the target end of a link attached to the item. The result is the value of the attribute or the related object(s). The result is a value or a set of values depending on the multiplicities of the item and the association.
item '.' *selector* '[' *qualifier-value* ']'	The *selector* designates a qualified association that qualifies the *item*. The *qualifier-value* is a value for the qualifier attribute. The result is the related object selected by the qualifier. Note that this syntax is applicable to array indexing as a form of qualification.
set '->' 'select' '(' *boolean-expression* ')'	The *boolean-expression* is written in terms of objects within the set. The result is the subset of objects in the set for which the boolean expression is true.

3.10.7 Examples

```
flight.pilot.training_hours > flight.plane.minimum_hours
```

```
company.employees–>select (title = "Manager" and self.reports–>size > 10)
```

3.11 Note

A note is a graphical symbol containing textual information (possibly including embedded images). It is a notation for rendering various kinds of textual information from the metamodel, such as constraints, comments, method bodies, and tagged values.

3.11.1 Semantics

A note is a notational item. It shows textual information within some semantic element.

3.11.2 Notation

A note is shown as a rectangle with a "bent corner" in the upper right corner. It contains arbitrary text. It appears on a particular diagram and may be attached to zero or more modeling elements by dashed lines.

3.11.3 Presentation Options

A note may have a stereotype.

A note with the keyword “constraint” or a more specific stereotype of constraint (such as the code body for a method) designates a constraint that is part of the model and not just part of a diagram view. Such a note is the view of a model element (the constraint).

3.11.4 Example

See also Figure 3-17 on page 3-28 for a note symbol containing a constraint.

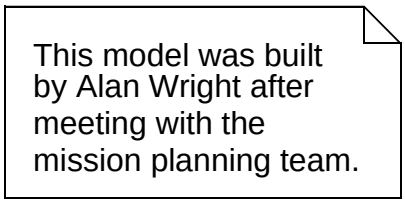


Figure 3-2 Note

3.11.5 Mapping

A note may represent the textual information in several possible metamodel constructs; it must be created in context that is known to a tool, and the tool must maintain the mapping. The string in the note maps to the body of the corresponding modeling element. A note may represent:

- a constraint,
- a tagged value,
- the body of a procedure of a method, or
- other string values within modeling elements.

It may also represent a comment attached directly to a diagram element.

3.12 Type-Instance Correspondence

A major purpose of modeling is to prepare generic descriptions that describe many specific items. This is often known as the *type-instance dichotomy.* Many or most of the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one represents the generic descriptor and the other the individual items that it describes. Examples of such pairs in UML include: Class-Object, Association-Link, UseCase-UseCaseInstance, Message-Stimulus, and so on.

Although diagrams for type-like elements and instance-like elements are not exactly the same, they share many similarities. Therefore, it is convenient to choose notation for each type-instance pair of elements such that the correspondence is visually apparent immediately. There are a limited number of ways to do this, each with advantages and disadvantages. In UML, the type-instance distinction is shown by employing the same geometrical symbol for each pair of elements and by underlining

the name string (including type name, if present) of an instance element. This visual distinction is generally easily apparent without being overpowering even when an entire diagram contains instance elements.

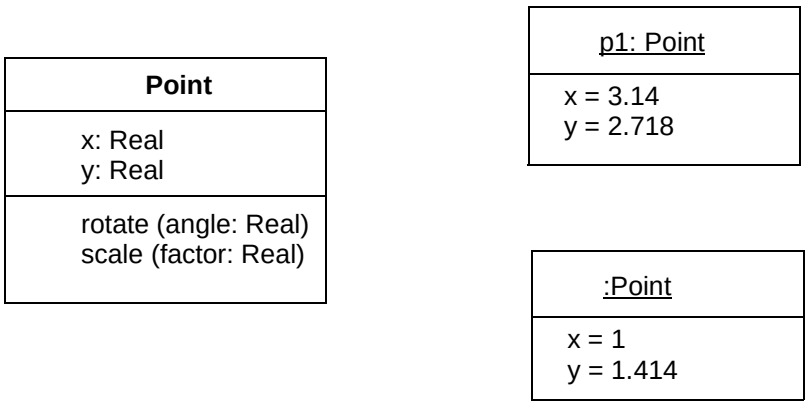


Figure 3-3 Classes and Objects

A tool is free to substitute a different graphic marker for instance elements at the user's option, such as color, fill patterns, or so on.

Roles (in collaborations) are somewhat between types and instances. Like instances, they identify distinct occurrences of a single classifier. Like types, they describe a reusable element that can have many distinct instances. A role is a distinguishable use of a classifier, but one that is still part of a general description (a collaboration) that can be used to create many instances. A run-time object may correspond to zero or more classes and to zero or more roles. The notation for a role permits indication of its base classifiers. The notation for an instance permits specification of its classifiers, its roles, or both.

A role is indicated by a name, colon, and type, not underlined and part of a collaboration. An instance is indicated by an optional name, optional slash followed by list of roles, colon, and list of types.

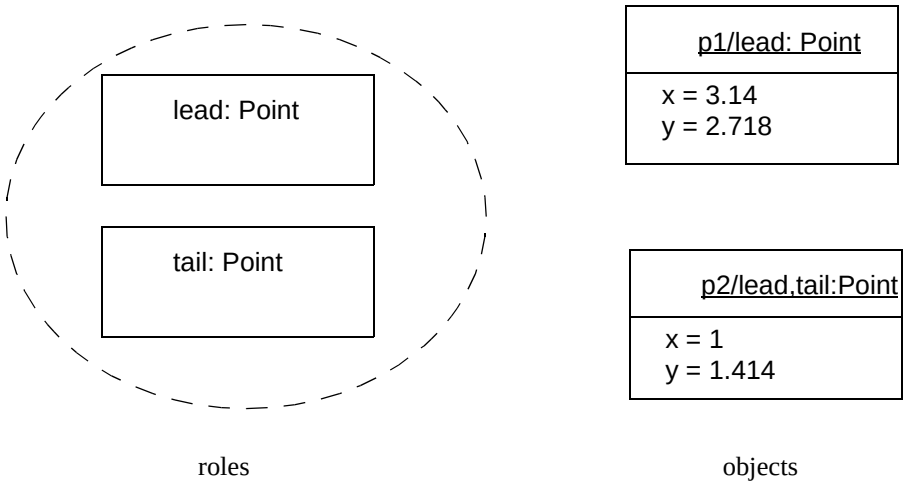


Figure 3-4 Roles and objects

Part 3 - Model Management

3.13 Package

3.13.1 Semantics

A *package* is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain subordinate packages as well as other kinds of model elements. All kinds of UML model elements can be organized into packages.

Note that packages *own* model elements and are the basis for configuration control, storage, and access control. Each element can be directly owned by a single package, so the package hierarchy is a strict tree. However, packages can reference other packages, modeled by using one of the stereotypes «import» and «access» of Permission dependency, so the usage network is a graph. Other kinds of dependencies between packages usually imply that one or more dependencies among the elements exists.

3.13.2 Notation

A package is shown as a large rectangle with a small rectangle (a “tab”) attached to the left side of the top of the large rectangle. It is the common folder icon.

The contents of the package may be shown within the large rectangle. Contents may also be shown by branching lines to contained elements, drawn outside of the package (see Figure 3-5 on page 3-18). A plus sign (+) within a circle is drawn at the end attached to the container.

- If the contents of the package are not shown within the large rectangle, then the name of the package may be placed within the large rectangle.
- If the contents of the package are shown within the large rectangle, then the name of the package may be placed within the tab.

A keyword string may be placed above the package name. The predefined stereotypes *facade, framework, stub,* and *topLevel* are notated within guillemets.

A list of properties may be placed in braces after or below the package name. Example: {abstract}. See Section 3.17, “Element Properties,” on page3-29 for details of property syntax.

The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol (‘+’ for public, ‘-’ for private, ‘#’ for protected, ‘~’ for package).

Relationships may be drawn between package symbols to show relationships between some of the elements in the packages. An import or access relationship between two packages is drawn as a dashed arrow with open arrowhead, labeled with the string «import» or «access», respectively.

Elements from imported or accessed packages may be shown outside the package symbol. As (public) elements in imported packages are added to the client namespace, they may alternatively be drawn inside the package symbol.

3.13.3 Presentation Options

A tool may show visibility by a graphic marker, such as color or font.

A tool may also show visibility by selectively displaying those elements that meet a given visibility level; for example, all of the public elements only.

A diagram showing a package with contents must not necessarily show all its contents; it may show a subset of the contained elements according to some criterion.

The contents of a package may also be shown using tree notation. The namespace ownership relationships between the package and its elements are marked with a circle with a cross in it at the owning end.

3.13.4 Style Guidelines

It is expected that packages with large contents will be shown as simple icons with names, in which the contents may be dynamically accessed by “zooming” to a detailed view.

3.13.5 Example

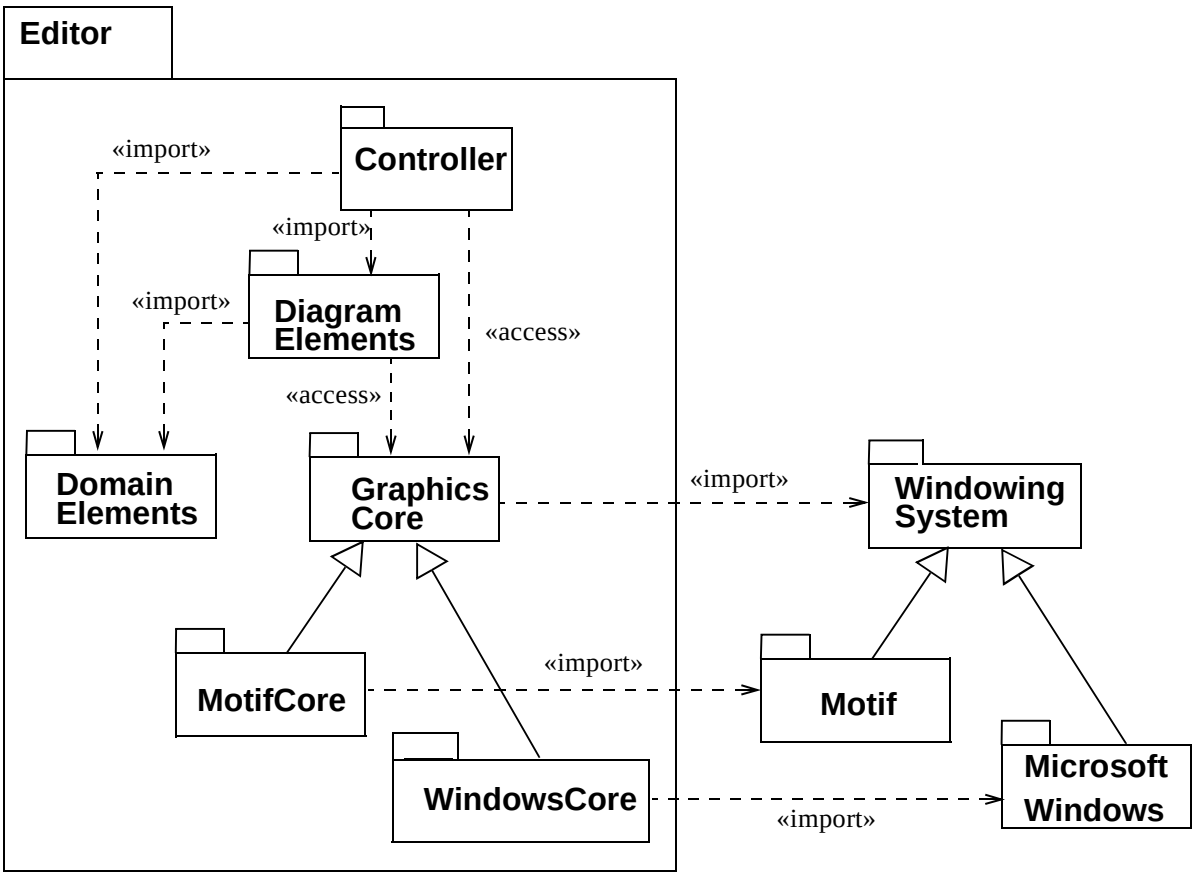


Figure 3-5 Packages and their access and import relationships.

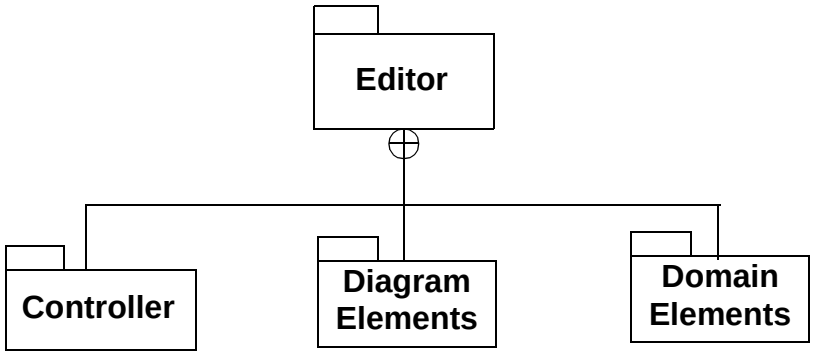


Figure 3-6 Some of the contents of the Editor package shown in a tree structure.

3.13.6 Mapping

A package symbol maps into a Package element. The name on the package symbol is the name of the Package element. If there is a string above the package name other than «model» or «subsystem», then it maps into a Package element with the corresponding stereotype. If there is a string «model» or «subsystem», then it maps into a Model or Subsystem element, respectively.

A relationship icon drawn from the package symbol boundary to another package symbol maps into a corresponding relationship to the other package element.

A symbol directly contained within the package symbol; that is, not contained within another symbol maps into a model element either owned or referenced by the package element. The alias used for a referenced element is often its pathname, in which case it is directly visible from the diagram that the element is not owned by the package. Only the reference is owned by the current package. Alternatively, a symbol shown outside the package symbol, attached to one of the symbols within the package symbol, denotes a referenced model element.

Symbols connected to the package symbol by branching lines with a plus sign at the end attached to the package symbol, map to elements in the package.

3.14 Subsystem

3.14.1 Semantics

Whereas a package is a generic mechanism for organizing model elements, a *subsystem* represents a behavioral unit in the physical system, and hence in the model. A subsystem offers interfaces and has operations, and its contents are partitioned into specification and realization elements. The specification of the subsystem consists of operations on the subsystem, together with specification elements such as use cases, state machines.

Apart from defining a namespace, a subsystem serves as a specification unit for the behavior of its contained model elements. A subsystem may or may not be instantiable.

3.14.2 Notation

A subsystem is notated basically in the same way as a package, with the addition of a fork symbol placed in the upper right corner of the large rectangle. The name of the subsystem (together with optional keyword, stereotype) is placed within the large rectangle. Optionally, especially if contents of the subsystem are shown within the large rectangle, the subsystem name and the fork are placed within the tab (the small rectangle).

An instantiable subsystem has the string «instantiable» above its name.

The large rectangle has three compartments, one for operations and one for each of the subsets specification elements and realization elements. These are usually shown by dividing the rectangle by a vertical line, and then dividing the area to the left of this

line into two compartments by a horizontal line. The operations are shown in the upper left compartment, the specification elements in the compartment below, and the realization elements in the right compartment. The latter two compartments are labeled ‘Specification Elements’ and ‘Realization Elements,’ respectively, to avoid potential ambiguity. The operations compartment is unlabeled. This is the general pattern for subsystem notation, although there are many different ways to customize it in a particular diagram, see Section 3.14.3, “Presentation Options,” on page 3-20 and Section 3.14.4, “Example,” on page 3-21.

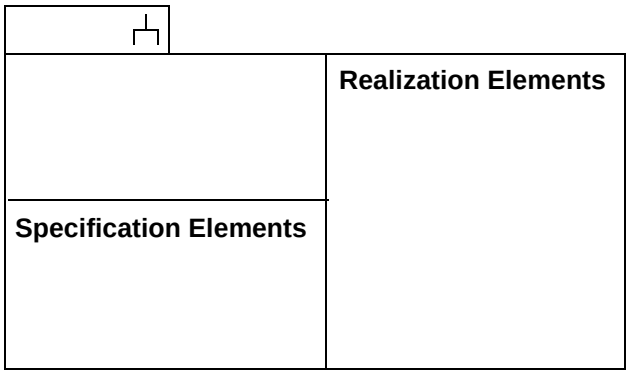


Figure 3-7 The general pattern for subsystem notation, with three compartments.

The mapping from the realization part to the specification part; that is, to operations and specification elements, is drawn using dashed arrows with closed, hollow arrowheads. For collaborations, the mapping may also be expressed textually.

When a subsystem is shown together with other, peer elements in a diagram, it is often shown without contents, in which case there are no compartments in the large rectangle. See Section 3.14.4, “Example,” on page 3-21.

3.14.3 Presentation Options

The fork symbol may be replaced by the keyword «subsystem» placed above the name of the subsystem.

The compartments may be rearranged within the subsystem symbol.

One or more of the compartments may be collapsed or suppressed. In cases where more than one diagram is used to show all information about a particular subsystem, each diagram shows a subset of the subsystem’s features and/or contents. Hence, compartments not relevant in a particular diagram are suppressed.

All contained elements in a subsystem may be shown together in one, non-labeled compartment; that is, no visual differentiating between specification elements and realization elements is done.

Tools may provide alternative ways to differentiate specification elements from realization elements, such as different colors, using the keyword «specification» for specification elements, etc.

As with packages, the contents of a subsystem may be shown using tree notation. Distinction between specification and realization elements may then be done; for example, by having two separate, labeled branches, or by showing the category separately for each element in the tree as suggested above.

3.14.4 Example

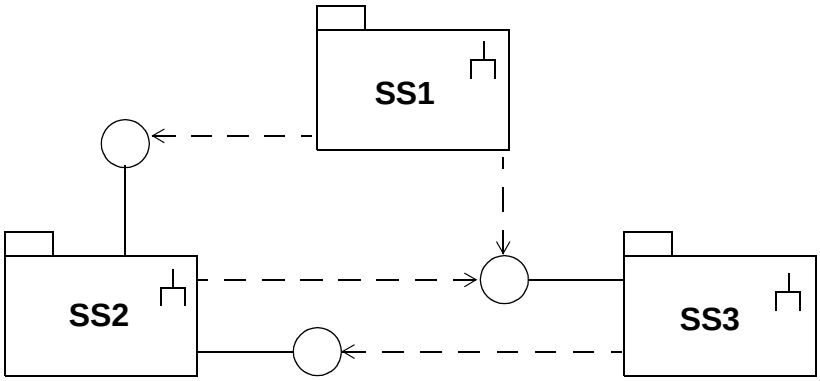


Figure 3-8 An overview diagram showing subsystems with interfaces and their dependencies.

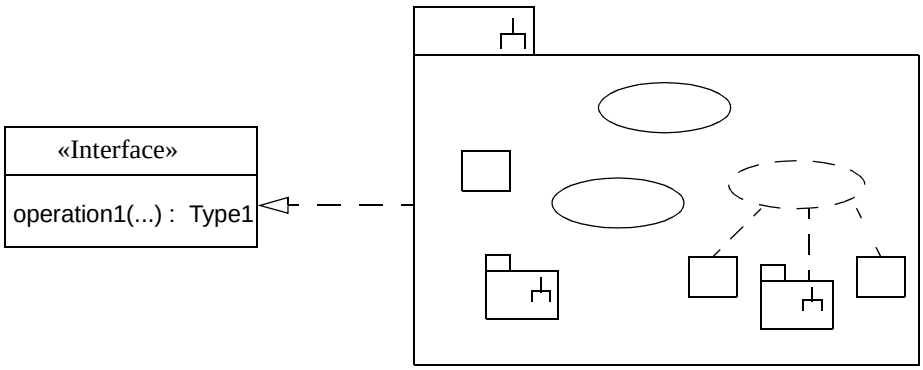


Figure 3-9 All contained elements of a subsystem shown together without division into compartments. Here, the subsystem offers operation1(...) although this is not explicitly shown.

In Figure 3-9 no visual separation between specification and realization elements is made. The following three figures are schematic examples where the specification/realization distinction is explicit. Together these figures constitute an example of how the basic notation for subsystem can be used to show different “views” of a subsystem in different diagrams, together giving the whole picture of the subsystem.

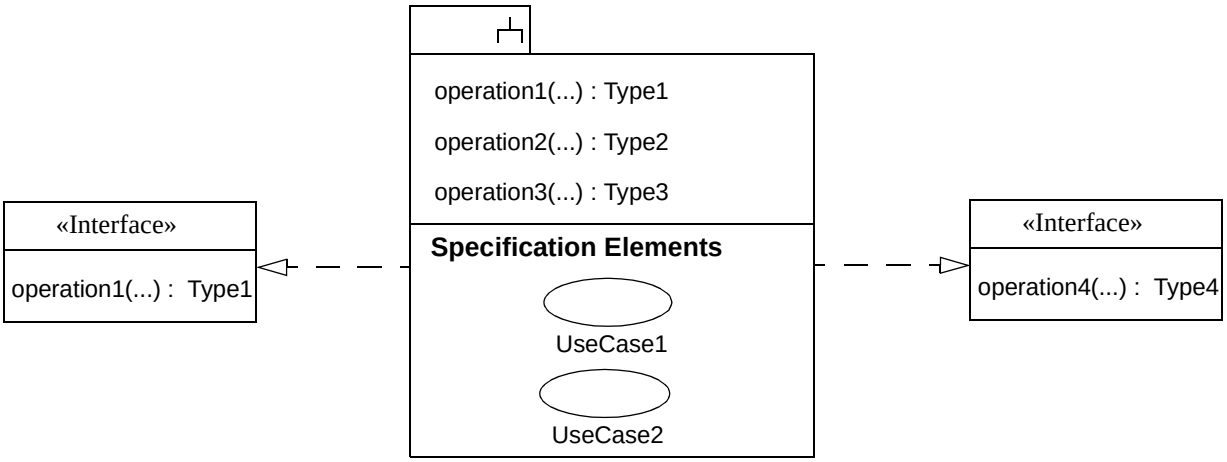


Figure 3-10 The specification part of a subsystem; compartment for realization part is suppressed. Implicit from the diagram is that the operation4(...) is either an operation of a specification element (UseCase1 or UseCase2) or of the subsystem itself. Furthermore, in cases where no operations are used for the specification but only contained specification elements, there is no operations compartment, and *vice versa*.

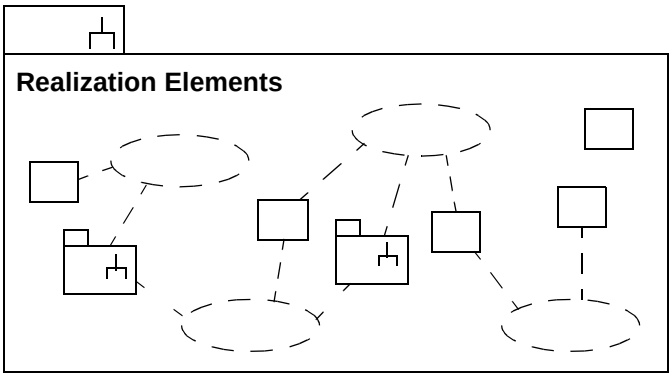


Figure 3-11 The realization part of a subsystem; compartments for specification part; that is, operations and specification elements are suppressed. Alternatively, collaborations could be shown in a separate diagram.

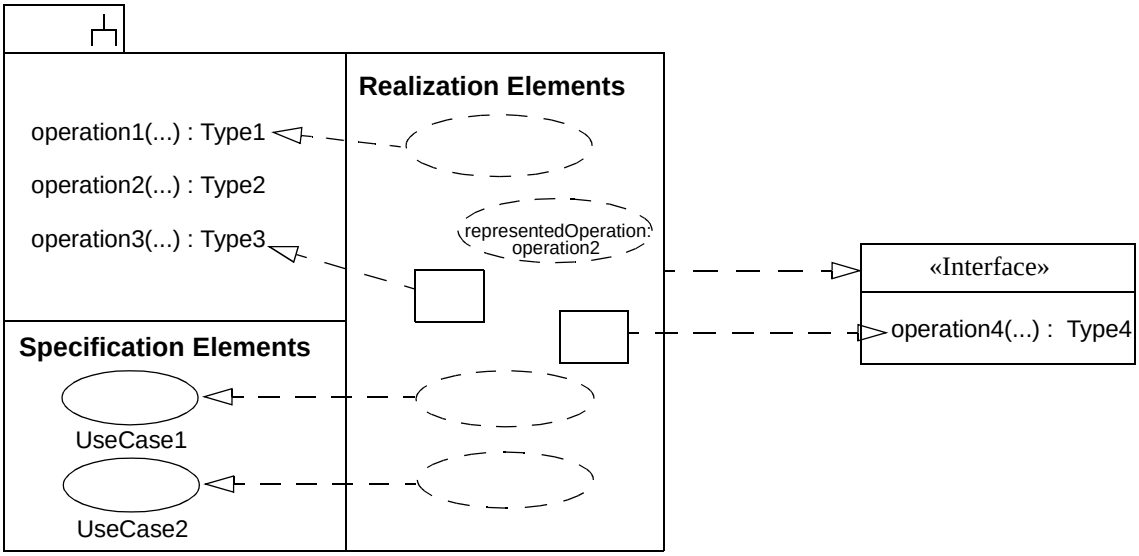


Figure 3-12 The mapping between specification part and realization part shown using all three compartments, but only those realization elements with relevance to the mapping are shown. The figure also shows examples of different ways to express the mapping.

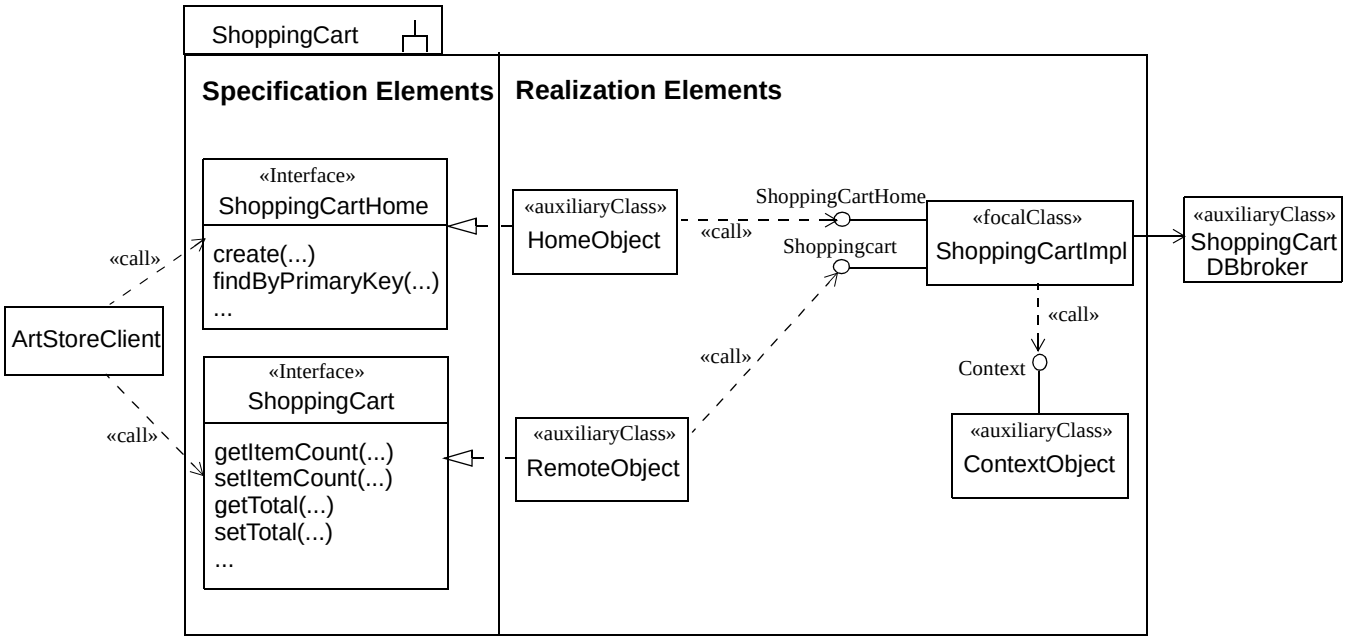


Figure 3-13 A component modeled using a subsystem and classes stereotyped «focalClass» or «auxiliaryClass», respectively.

3.14.5 Mapping

A subsystem symbol maps into a Subsystem with the given name. The mapping is analogous to that of package symbols, with the following addition:

A symbol within a compartment of the large rectangle labeled ‘Specification Elements’ or ‘Realization Elements’ is mapped to a specification or realization element of the subsystem, respectively. An operation signature string within a non-labeled compartment maps to an operation of the subsystem. Note that a compartment may coincide with the whole rectangle.

A symbol, that is not an operation signature string, within a non-labeled compartment maps to an element contained in the subsystem.

A dashed arrow with closed, hollow arrowhead from a symbol denoting a realization element to a symbol denoting a specification element or an operation maps to a «realize» relationship between the corresponding elements.

3.15 Model

3.15.1 Semantics

A model captures a view of a physical system. Hence, it is an abstraction of the physical system with a certain purpose; for example, to describe behavioral aspects of the physical system to a certain category of stakeholders. A model contains all the model elements needed to represent a physical system completely according to the purpose of this particular model. The model elements in a model are organized into a package/subsystem hierarchy, where the top-most package/subsystem represents the boundary of the physical system.

Different models of the same physical system show different aspects of the system. The pre-defined stereotype «systemModel» can be applied to a model containing the entire set of models for a physical system.

Relationships between elements in different models have no semantic impact on the contents of the models because of the self-containment of models. However, they are useful for tracing refinements and for keeping track of requirements between models.

Relationships between models express refinement, import, etc.

3.15.2 Notation

A model is notated using the ordinary package symbol with a small triangle in the upper right corner of the large rectangle. Optionally, especially if contents of the model is shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.

Relationships between models as well as relationships between elements in different models are shown using the notation for the given kind of relationship. In particular, trace dependencies are notated with a dashed line, with an optional open arrowhead, and the keyword «trace».

3.15.3 Presentation Options

A model may be notated as a package, using the ordinary package symbol with the keyword «model» placed above the name of the model.

3.15.4 Example

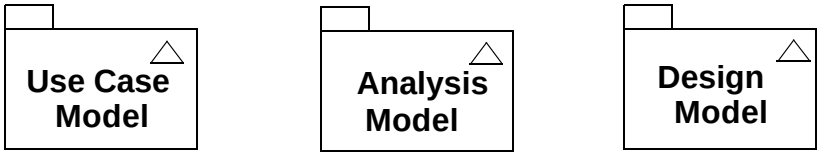


Figure 3-14 Three views of a physical system, each represented by a model.

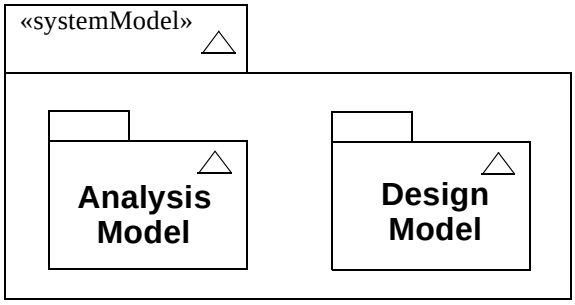


Figure 3-15 A «systemModel» containing an analysis model and a design model.

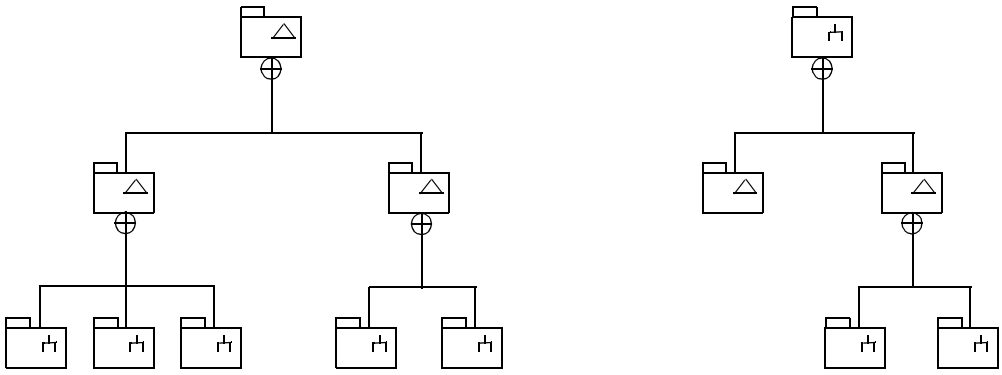


Figure 3-16 Two examples of containment hierarchies with models and subsystems shown using branching lines. The left hierarchy is based on Model, whereas the right one is based on Subsystem.

3.15.5 Mapping

A model symbol maps to a Model with the given name. The mapping is analogous to that of package symbols.

Part 4 - General Extension Mechanisms

The elements in this section are general purpose mechanisms that may be applied to any modeling element. The semantics of a particular use depends on a convention of the user or an interpretation by a particular constraint language or programming language; therefore, they constitute an extensibility device for UML.

3.16 Constraint and Comment

3.16.1 Semantics

A *constraint* is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true; otherwise, the system described by the model is invalid (with consequences that are outside the scope of UML). Certain kinds of constraints (such as an association “xor” constraint) are predefined in UML, others may be user-defined. A user-defined constraint is described in words in a given language, whose syntax and interpretation is a tool responsibility. A constraint represents semantic information attached to a model element, not just to a view of it.

A *comment* is a text string (including references to human-readable documents) attached directly to a model element. A comment can attach arbitrary textual information to any model element of presumed general importance but it has no semantic force. Comments may be used for explaining the reasons for decisions, among other things.

3.16.2 Notation

A constraint is shown as a text string in braces ({ }). There is an expectation that individual tools may provide one or more languages in which formal constraints may be written. One predefined language for writing constraints is OCL (see the Object Constraint Language Specification chapter); otherwise, the constraint may be written in natural language. Each constraint is written in a specific language, although the language is not generally displayed on the diagram (the tool must keep track of it, however).

For an element whose notation is a text string (such as an attribute, etc.), the constraint string may follow the element text string in braces.

For a list of elements whose notation is a list of text strings (such as the attributes within a class), a constraint string may appear as an element in the list. The constraint applies to all succeeding elements of the list until another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraint, but may augment or modify individual constraints within the constraint string.

For a single graphical symbol (such as a class or an association path), the constraint string may be placed near the symbol, preferably near the name of the symbol, if any.

For two graphical symbols (such as two classes or two associations), the constraint is shown as a dashed arrow from one element to the other element labeled by the constraint string (in braces). The direction of the arrow is relevant information within the constraint. The client (tail of the arrow) is mapped to the first position and the supplier (head of the arrow) is mapped to the second position in the constraint.

For three or more graphical symbols, the constraint string is placed in a note symbol and attached to each of the symbols by a dashed line. This notation may also be used for the other cases. For three or more paths of the same kind (such as generalization paths or association paths), the constraint may be attached to a dashed line crossing all of the paths.

A comment is shown as a text string (not enclosed in braces) within a note icon. Syntax for including comments within other elements (such as expressions or constraints) are not specified by UML but may be provided by a tool as part of the expression syntax for a particular language.

3.16.3 Example

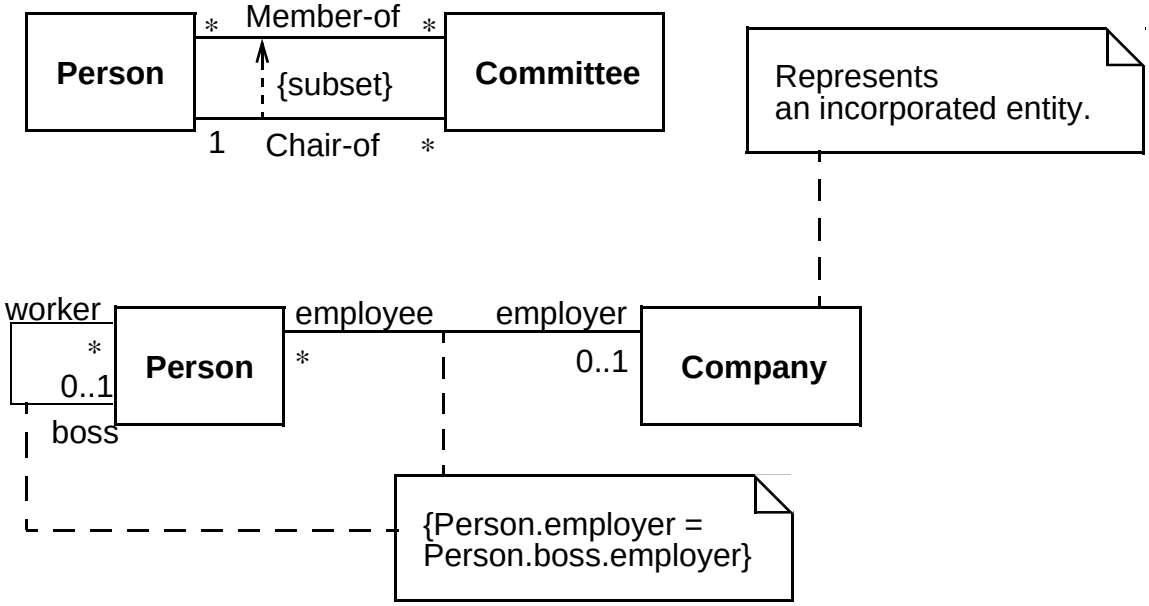


Figure 3-17 Constraints and comment

3.16.4 Mapping

A constraint string is a string enclosed in braces ({ }).

The constraint string maps into the *body* expression in a Constraint element. The mapping depends on the language of the expression, which is known to a tool but generally not displayed on a diagram.

A constraint string following a list entry maps into a Constraint attached to the element corresponding to the list entry.

A constraint string represented as a stand-alone list element maps into a separate Constraint attached to each succeeding model element corresponding to subsequent list entries (until superseded by another constraint or property string).

A constraint string placed near a graphical symbol must be attached to the symbol by a hidden link by a tool operating in context. The tool must maintain the graphical linkage implicitly. The constraint string maps into a Constraint attached to the element corresponding to the symbol.

A constraint string attached to a dashed arrow maps into a constraint attached to the two elements corresponding to the symbols connected by the arrow.

A string enclosed in braces in a note symbol maps into a Constraint attached to the elements corresponding to the symbols connected to the note symbol by dashed lines.

A string (not enclosed in braces) in a note attached to the symbol for an element maps into a Comment attached to the corresponding element.

3.17 Element Properties

Many kinds of elements have detailed properties that do not have a visual notation. In addition, users can define new element properties using the *tagged value* mechanism.

A string may be used to display properties attached to a model element. This includes properties represented by attributes in the metamodel as well as both predefined and user-defined tagged values.

3.17.1 Semantics

Note that we use *property* in a general sense to mean any value attached to a model element, including attributes, associations, and tagged values. In this sense it can include indirectly reachable values that can be found starting at a given element. Some kinds of properties would have syntax within expressions (not specified by UML) but no explicit UML notation.

A *tagged value* is a keyword-value pair that may be attached to any kind of model element (including diagram elements as well as semantic model elements). The keyword is called a *tag*. Each tag represents a particular kind of property applicable to one or many kinds of model elements. Both the tag and the value are encoded as strings. Tagged values are an extensibility mechanism of UML permitting arbitrary information to be attached to models. It is expected that most model editors will provide basic facilities for defining, displaying, and searching tagged values as strings but will not otherwise use them to extend the UML semantics. It is expected, however, that back-end tools such as code generators, report writers, and the like will read tagged values to guide their semantics in flexible ways.

3.17.2 Notation

A property (either a metamodel attribute or a tagged value) is displayed as a comma-delimited sequence of *property specifications* all inside a pair of braces ({ }).

A *property specification* has the form

name = *value*

where *name* is the name of a property (metamodel attribute or arbitrary tag) and *value* is an arbitrary string that denotes its value. If the type of the property is Boolean, then the default value is **true** if the value is omitted. That is, to specify a value of true you may include just the keyword. To specify a value of false, you omit the name completely. Properties of other types require explicit values. The syntax for displaying the value is a tool responsibility in cases where the underlying model value is not a string or a number.

Note that property strings may be used to display built-in attributes as well as tagged values.

Boolean properties frequently have the form is*Name*, where *name* is the name of some condition that may be true or false. In these cases, the form "*name*" may usually appear by itself, without a value, to mean "is*Name* = true." For example, {abstract} is the same as {isAbstract = true}.

Tagged values can sometimes refer to other model elements (see Section 2.6.2.5, "TaggedValue," on page 2-79). In that case, the usual tagged value format is used except that the value is the name of the model element that is referenced. Alternatively, it may be represented graphically using a «taggedValue» relationship, which uses the dependency notation. The direction of the dependency arrow is towards the referenced element. These two cases are illustrated in Figure 3-18

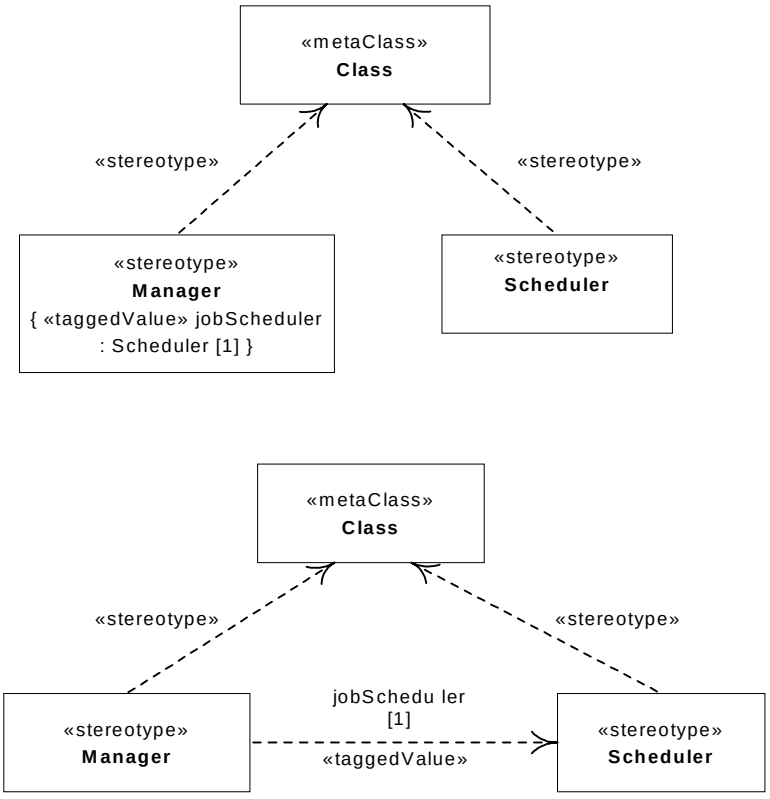


Figure 3-18 Alternative notations for tagged values as references

3.17.3 Presentation Options

A tool may present property specifications on separate lines with or without the enclosing braces, provided they are marked appropriately to distinguish them from other information. For example, properties for a class might be listed under the class name in a distinctive typeface, such as italics or a different font family.

3.17.4 Style Guidelines

It is legal to use strings to specify properties that have graphical notations; however, such usage may be confusing and should be used with care.

3.17.5 Example

{ author = “Joe Smith”, deadline = 31-March-1997, status = analysis }

{ abstract }

3.17.6 Mapping

Each term within a string maps to either a built-in attribute of a model element or a tagged value (predefined or user-defined). A tool must enforce the correspondence to built-in attributes.

3.18 Stereotypes

3.18.1 Semantics

A stereotype is, in effect, a new class of metamodel element that is introduced at modeling time. It represents a subclass of an existing metamodel element with the same form (attributes and relationships) but with a different intent. Generally a stereotype represents a usage distinction. A stereotyped element may have additional constraints on it from the base metamodel class. It may also have required tagged values that add information needed by elements with the stereotype. It is expected that code generators and other tools will treat stereotyped elements specially. Stereotypes represent one of the built-in extensibility mechanisms of UML.

3.18.2 Notation

The general presentation of a stereotype is to use the symbol for the metamodel base element but to place a keyword string above the name of the element (if any). The keyword string (Section 3.9, “Keywords,” on page 3-11) is the name of the stereotype within matched *guillemets*, which are the quotation mark symbols used in French and certain other languages (for example, «foo»).

Note – A guillemet looks like a double angle-bracket, but it is a single character in most extended fonts. Most computers have a Character Map utility. Double angle-brackets may be used as a substitute by the typographically challenged.

The keyword string is generally placed above or in front of the name of the model element being described. If multiple stereotypes are defined for the same model element, they are placed vertically one below the other. The keyword string may also be used as an element in a list, in which case it applies to subsequent list elements until

another stereotype string replaces it, or an empty stereotype string («») nullifies it. Note that a stereotype name should not be identical to a predefined keyword applicable to the same element type.

To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype. The UML does not specify the form of the graphic specification, but many bitmap and stroked formats exist (and their portability is a difficult problem). The icon can be used in one of two ways:

1. It may be used instead of, or in addition to, the stereotype keyword string as part of the symbol for the base model element that the stereotype is based on. For example, in a class rectangle it is placed in the upper right corner of the name compartment. In this form, the normal contents of the item can be seen.

2. The entire base model element symbol may be "collapsed" into an icon containing the element name or with the name above or below the icon. Other information contained by the base model element symbol is suppressed. More general forms of icon specification and substitution are conceivable, but we leave these to the ingenuity of tool builders, with the warning that excessive use of extensibility capabilities may lead to loss of portability among tools.

If multiple stereotypes are defined, the graphical icons or markers are omitted.

UML avoids the use of graphic markers, such as color, that present challenges for certain persons (the color blind) and for important kinds of equipment (such as printers, copiers, and fax machines). None of the UML symbols *require* the use of such graphic markers. Users *may* use graphic markers freely in their personal work for their own purposes (such as for highlighting within a tool) but should be aware of their limitations for interchange and be prepared to use the canonical forms when necessary.

The classification hierarchy of the stereotypes themselves can be displayed on a class diagram, as described in Section 3.35, "Stereotype Declaration," on page 3-57. This capability is not required by many modelers who must use existing stereotypes but not define new kinds of stereotypes.

3.18.3 Examples

Figure 3-19 on page 3-33 illustrates various notational forms of the stereotype notation. Note that the top four shapes are alternatives of each other. The next one shows how a dependency can be stereotyped and the bottom example illustrates a model element with multiple stereotypes.

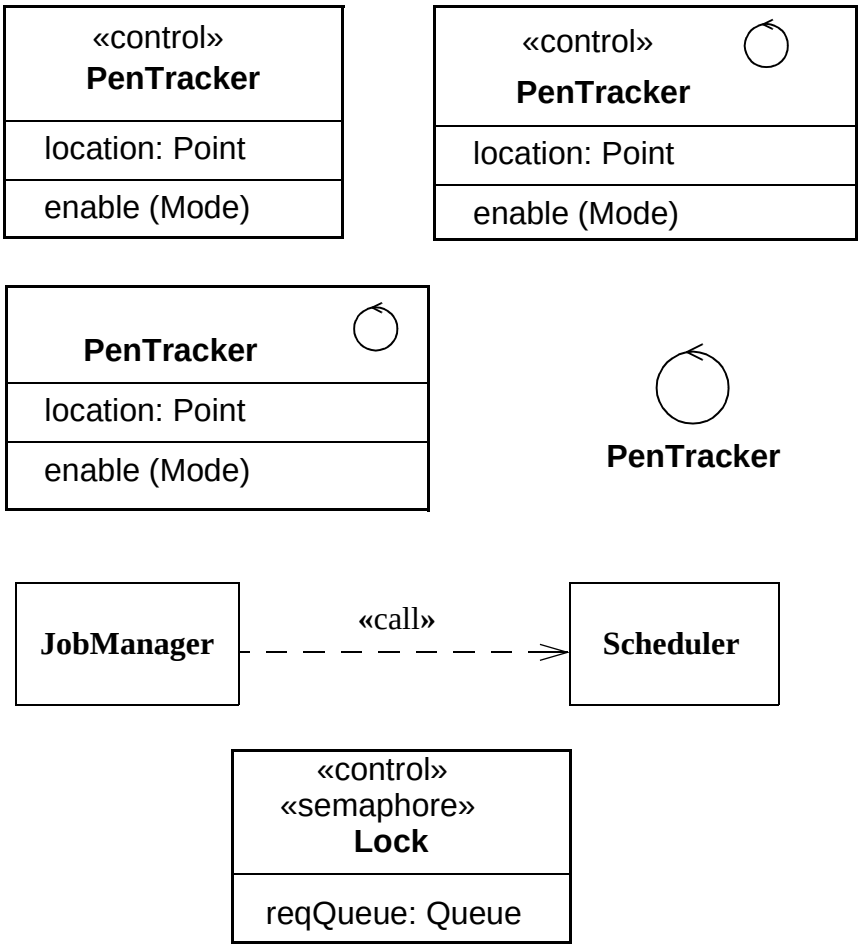


Figure 3-19 Varieties of Stereotype Notation

3.18.4 Mapping

The use of a stereotype keyword maps into the stereotype relationship between the Element corresponding to the symbol containing the name and the Stereotype of the given name. The use of a stereotype icon within a symbol maps into the stereotype relationship between the Element corresponding to the symbol containing the icon and the Stereotype represented by the symbol. A tool must establish the connection when the symbol is created and there is no requirement that an icon represent uniquely one stereotype. The use of a stereotype icon, instead of a symbol, must be created in a context in which a tool implies a corresponding model element and a Stereotype represented by the icon. The element and the stereotype have the stereotype relationship.

Part 5 - Static Structure Diagrams

Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things. Class diagrams do not show temporal information, although they may contain reified occurrences of things that have or things that describe temporal behavior. An object diagram shows instances compatible with a particular class diagram.

This section discusses classes and their variations, including templates and instantiated classes, and the relationships between classes (association and generalization) and the contents of classes (attributes and operations).

3.19 Class Diagram

A class diagram is a graph of Classifier elements connected by their various static relationships. Note that a “class” diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be “static structural diagram” but “class diagram” is shorter and well established.

3.19.1 Semantics

A class diagram is a graphic view of the static structural model. The individual class diagrams do not represent divisions in the underlying model.

3.19.2 Notation

A class diagram is a collection of static declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents. Class diagrams may be organized into packages either with their underlying models or as separate packages that build upon the underlying model packages.

3.19.3 Mapping

A class diagram does not necessarily match a single semantic entity. A package within the static structural model may be represented by one or more class diagrams. The division of the presentation into separate diagrams is for graphical convenience and does not imply a partitioning of the model itself. The contents of a diagram map into elements in the static semantic model. If a diagram is part of a package, then its contents map into elements in the same package (including possible references to elements accessed or imported from other packages).

3.20 Object Diagram

An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, mainly to show examples of data structures.

Tools need not support a separate format for object diagrams. Class diagrams can contain objects, so a class diagram with objects and no classes is an “object diagram.” The phrase is useful, however, to characterize a particular usage achievable in various ways.

3.21 Classifier

Classifier is the metamodel superclass of *Class*, *DataType*, and *Interface*. All of these have similar syntax and are therefore all notated using the rectangle symbol with keywords used as necessary. Because classes are most common in diagrams, a rectangle without a keyword represents a class, and the other subclasses of *Classifier* are indicated with keywords. In the sections that follow, the discussion will focus on *Class*, but most of the notation applies to the other element kinds as semantically appropriate and as described later under their own sections.

3.22 Class

A *class* is the descriptor for a set of objects with similar structure, behavior, and relationships. The model is concerned with describing the intension of the class, that is, the rules that define it. The run-time execution provides its extension, that is, its instances. UML provides notation for declaring classes and specifying their properties, as well as using classes in various ways. Some modeling elements that are similar in form to classes (such as interfaces, signals, or utilities) are notated using keywords on class symbols; some of these are separate metamodel classes and some are stereotypes of Class. Classes are declared in class diagrams and used in most other diagrams. UML provides a graphical notation for declaring and using classes, as well as a textual notation for referencing classes within the descriptions of other model elements.

3.22.1 Semantics

A class represents a concept within the system being modeled. Classes have data structure and behavior and relationships to other elements.

The name of a class has scope within the package in which it is declared and the name must be unique (among class names) within its package.

3.22.2 Basic Notation

A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class (including stereotype); the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations.

See Section 3.23, “Name Compartment,” on page 3-38 and Section 3.24, “List Compartment,” on page 3-38 for more details.

3.22.2.1 References

By default a class shown within a package is assumed to be defined within that package. To show a reference to a class defined in another package, use the syntax

Package-name::*Class-name*

as the name string in the name compartment. A full pathname can be specified by chaining together package names separated by double colons (::).

3.22.3 Presentation Options

Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary (Section3.24, “ List Compartment,” on page 3-38).

Additional compartments may be supplied as a tool extension to show other predefined or user-defined model properties (for example, to show business rules, responsibilities, variations, events handled, exceptions raised, and so on). Most compartments are simply lists of strings. More complicated formats are possible, but UML does not specify such formats; they are a tool responsibility. Appearance of each compartment should preferably be implicit based on its contents. Compartment names may be used, if needed.

Tools may provide other ways to show class references and to distinguish them from class declarations.

A class symbol with a stereotype icon may be “collapsed” to show just the stereotype icon, with the name of the class either inside the class or below the icon. Other contents of the class are suppressed.

3.22.4 Style Guidelines

- Center class name in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above class name.

- For those languages that distinguish between uppercase and lowercase characters, capitalize class names; that is, begin them with an uppercase character.
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show the names of abstract classes or the signatures of abstract operations in italics.

As a tool extension, boldface may be used for marking special list elements; for example, to designate candidate keys in a database design. This might encode some design property modeled as a tagged value, for example.

Show full attributes and operations when needed and suppress them in other contexts or references.

3.22.5 Example

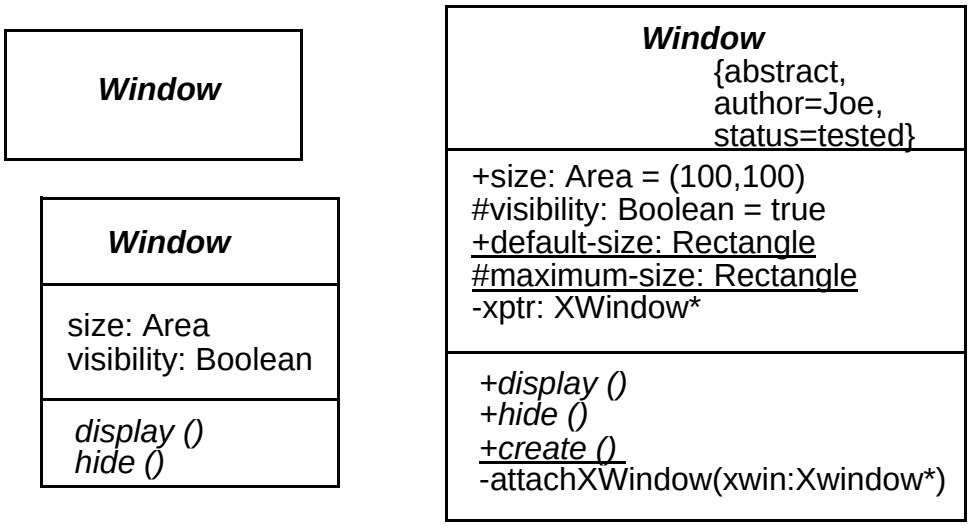


Figure 3-20 Class Notation: Details Suppressed, Analysis-level Details, Implementation-level Details

3.22.6 Mapping

A class symbol maps into a Class element within the package that owns the diagram. The name compartment contents map into the class name and into properties of the class (built-in attributes or tagged values). The attribute compartment maps into a list of Attributes of the Class. The operation compartment maps into a list of Operations of the Class.

The property string {location=*name*} maps into an implementationLocation association to a Component. The *name* is the name of the containing Component.

3.23 Name Compartment

3.23.1 Notation

The name compartment displays the name of the class and other properties in up to three sections:

An optional stereotype keyword may be placed above the class name within guillemets, and/or a stereotype icon may be placed in the upper right corner of the compartment. The stereotype name must not match a predefined keyword.

The name of the class appears next. If the class is abstract, this can be indicated by italicizing its name (for those languages that support italicization) or by placing the keyword *abstract* in a property list below or after the name; for example, Invoice {abstract}. Note that any explicit specification of generalization status takes precedence over the name font.

A list of strings denoting properties (metamodel attributes or tagged values) may be placed in braces below the class name. The list may show class-level attributes for which there is no UML notation and it may also show tagged values. The presence of a keyword for a Boolean type without a value implies the value *true*. For example, a leaf class shows the property "{leaf}".

The stereotype and property list are optional.

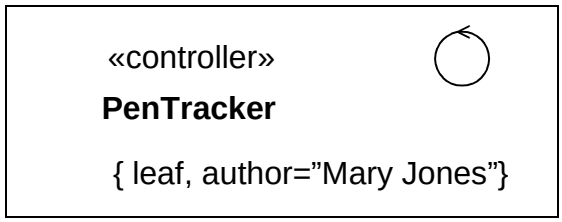


Figure 3-21 Name Compartment

3.23.2 Mapping

The contents of the name compartment map into the name, stereotype, and various properties of the Class represented by the class symbol.

3.24 List Compartment

3.24.1 Notation

A list compartment holds a list of strings, each of which is the encoded representation of a feature, such as an attribute or operation. The strings are presented one to a line with overflow to be handled in a tool-dependent manner. In addition to lists of

attributes or operations, optional lists can show other kinds of predefined or user-defined values, such as responsibilities, rules, or modification histories. UML does not define these optional lists. The manipulation of user-defined lists is tool-dependent.

The items in the list are ordered and the order may be modified by the user. The order of the elements is meaningful information and must be accessible within tools (for example, it may be used by a code generator in generating a list of declarations). The list elements may be presented in a different order to achieve some other purpose (for example, they may be sorted in some way). Even if the list is sorted, the items maintain their original order in the underlying model. The ordering information is merely suppressed in the view.

An ellipsis (. . .) as the final element of a list or the final element of a delimited section of a list indicates that additional elements in the model exist that meet the selection condition, but that are not shown in that list. Such elements may appear in a different view of the list.

3.24.1.1 Group properties

A property string may be shown as an element of the list, in which case it applies to all of the succeeding list elements until another property string appears as a list element. This is equivalent to attaching the property string to each of the list elements individually. The property string does not designate a model element. Examples of this usage include indicating a stereotype and specifying visibility. Keyword strings may also be used in a similar way to qualify subsequent list elements.

3.24.1.2 Compartment name

A compartment may display a name to indicate which kind of compartment it is. The name is displayed in a distinctive font centered at the top of the compartment. This capability is useful if some compartments are omitted or if additional user-defined compartments are added. For a Class, the predefined compartments are named **attributes** and **operations**. An example of a user-defined compartment might be **requirements**. The name compartment in a class must always be present; therefore, it does not require or permit a compartment name.

3.24.2 Presentation Options

A tool may present the list elements in a sorted order, in which case the inherent ordering of the elements is not visible. A sort is based on some internal property and does not indicate additional model information. Example sort rules include:

- alphabetical order,
- ordering by stereotype (such as constructors, destructors, then ordinary methods),
- ordering by visibility (public, then package, then protected, then private).

The elements in the list may be filtered according to some selection rule. The specification of selection rules is a tool responsibility. The absence of items from a filtered list indicates that no elements meet the filter criterion, but no inference can be

drawn about the presence or absence of elements that do not meet the criterion. However, the ellipsis notation is available to show that invisible elements exist. It is a tool responsibility whether and how to indicate the presence of either local or global filtering, although a stand-alone diagram should have some indication of such filtering if it is to be understandable.

If a compartment is suppressed, no inference can be drawn about the presence or absence of its elements. An empty compartment indicates that no elements meet the selection filter (if any).

Note that attributes may also be shown by composition (see Figure 3-45 on page 3-83).

3.24.3 Example

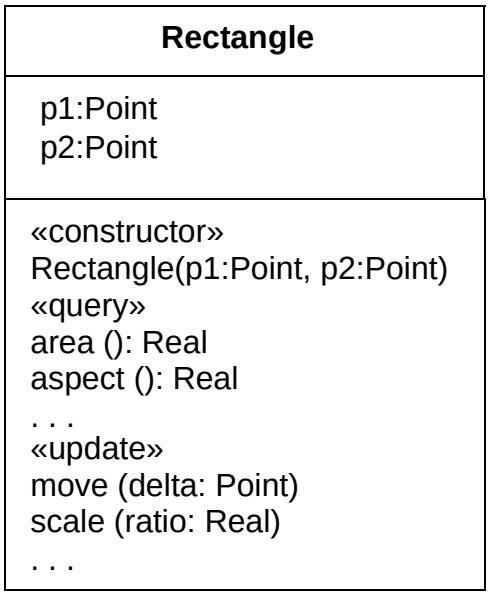


Figure 3-22 Stereotype Keyword Applied to Groups of List Elements

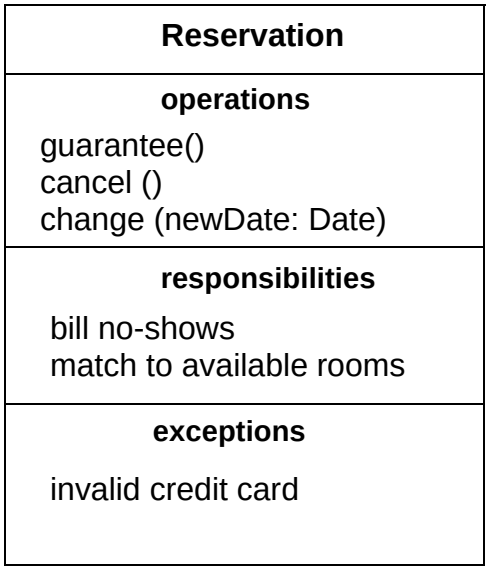


Figure 3-23 Compartments with Names

3.24.4 Mapping

The entries in a list compartment map into a list of ModelElements, one for each list entry. The ordering of the ModelElements matches the list compartment entries (unless the list compartment is sorted in some way). In this case, no implication about the ordering of the Elements can be made (the ordering can be seen by turning off sorting). However, a list entry string that is a stereotype indication (within guillemets) or a property indication (within braces) does not map into a separate ModelElement. Instead, the corresponding property applies to each subsequent ModelElement until the appearance of a different stand-alone stereotype or property indicator. The property specifications are conceptually duplicated for each list Element, although a tool might maintain an internal mechanism to store or modify them together. The presence of an ellipsis ("...") as a list entry implies that the semantic model contains at least one Element with corresponding properties that is not visible in the list compartment.

3.25 Attribute

Strings in the attribute compartment are used to show attributes in classes. A similar syntax is used to specify qualifiers, template parameters, operation parameters, and so on (some of these omit certain terms).

3.25.1 Semantics

Note that an attribute is semantically equivalent to a composition association; however, the intent and usage is normally different.

The type of an attribute is a Classifier.

3.25.2 Notation

An attribute is shown as a text string that can be parsed into the various properties of an attribute model element. The default syntax is:

visibility name : *type-expression* [*multiplicity ordering*] = *initial-value* { *property-string* }

- Where *visibility* is one of:

  \+ public visibility

  # protected visibility

  - private visibility

  ~ .package visibility

  The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public). A tool should assign visibilities to new attributes even if the visibility is not shown. The visibility marker is a shorthand for a full *visibility* property specification string.

  Visibility may also be specified by keywords (*public, protected, private, package*). This form is used particularly when it is used as an inline list element that applies to an entire block of attributes.

  Additional kinds of visibility might be defined for certain programming languages, such as C++ *implementation* visibility (actually all forms of nonpublic visibility are language-dependent). Such visibility must be specified by property string or by a tool-specific convention.

- Where *name* is an identifier string that represents the name of the attribute.
- Where [*multiplicity ordering*] shows the multiplicity and the ordering of the attribute (Section 3.44, “Multiplicity,” on page 3-75). The term may be omitted, in which case the multiplicity is 1..1 (exactly one).
- The *ordering* property is meaningful if the multiplicity upper bound is greater than one. It may be one of:
  - *(absent)* — the values are unordered
  - unordered — the values are unordered
  - ordered — the values are ordered
- Where *type-expression* is either
  - if it is a simple word, the name of a classifier, or
  - a language-dependent string that maps into a ProgrammingLanguageDataType.
- Where *initial-value* is a language-dependent expression for the initial value of a newly created object. The initial value is optional (the equal sign is also omitted). An explicit constructor for a new object may augment or modify the default initial value.

- Where *property-string* indicates property values that apply to the element. The property string is optional (the braces are omitted if no properties are specified).

A class-scope attribute is shown by underlining the name and type expression string; otherwise, the attribute is instance-scope.

class-scope-attribute

The notation justification is that a class-scope attribute is an instance value in the executing system, just as an object is an instance value, so both may be designated by underlining. An instance-scope attribute is not underlined; that is the default.

There is no symbol for whether an attribute is changeable (the default is changeable). A nonchangeable attribute is specified with the property "{frozen}".

In the absence of a multiplicity indicator, an attribute holds exactly 1 value. Multiplicity may be indicated by placing a multiplicity indicator in brackets after the classifier name, for example:

colors : Color [3]
points : Point [2..* ordered]

Note that a multiplicity of 0..1 provides for the possibility of null values: the absence of a value, as opposed to a particular value from the range. For example, the following declaration permits a distinction between the *null* value and the empty string:

name : String [0..1]

A stereotype keyword in guillemets precedes the entire attribute string, including any visibility indicators. A property list in braces follows the rest of the attribute string.

3.25.3 Presentation Options

The type expression may be suppressed (but it has a value in the model).

The initial value may be suppressed, and it may be absent from the model. It is a tool responsibility whether and how to show this distinction.

A tool may show the visibility indication in a different way, such as by using a special icon or by sorting the elements by group.

A tool may show the individual fields of an attribute as columns rather than a continuous string.

If the type-expression string is not a word, then it is assumed to be expressed in the syntax of a particular programming language, such as C++ or Smalltalk. This form is assumed if the string is not a word. Specific tagged properties may be included in the string. The programming language must be known from the general context of the diagram or a tool supporting it. In this case, the type-expression maps into a ProgrammingLanguageDataType whose expression attribute specifies the language name and the string representation of the data type in that language.

Particular attributes within a list may be suppressed (see Section 3.24, "List Compartment," on page 3-38).

3.25.4 Style Guidelines

Attribute names typically begin with a lowercase letter. Attribute names are in plain face.

3.25.5 Example

+size: Area = (100,100)
#visibility: Boolean = invisible
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindowPtr

3.25.6 Mapping

A string entry within the attribute compartment maps into an Attribute within the Class corresponding to the class symbol. The properties of the attribute map in accord with the preceding descriptions. If the visibility is absent, then no conclusion can be drawn about the Attribute visibilities unless a filter is in effect; for example, only public attributes shown. Likewise, if the type or initial value are omitted. The omission of an underline always indicates an instance-scope attribute. The omission of multiplicity denotes a multiplicity of 1.

Any properties specified in braces following the attribute string map into properties on the Attribute. In addition, any properties specified on a previous stand-alone property specification entry apply to the current Attribute (and to others).

3.26 Operation

Entries in the operation compartment are strings that show operations defined on classes and methods supplied by classes.

3.26.1 Semantics

An operation is a service that an instance of the class may be requested to perform. It has a name and a list of arguments.

3.26.2 Notation

An operation is shown as a text string that can be parsed into the various properties of an operation model element. The default syntax is:

visibility name (*parameter-list*) : *return-type-expression* { *property-string* }

- Where *visibility* is one of:

  \+ public visibility

  \# protected visibility

- private visibility

~ package visibility

The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public). The visibility marker is a shorthand for a full *visibility* property specification string.

Visibility may also be specified by keywords (*public*, *protected*, *private*, *package*). This form is used particularly when it is used as an inline list element that applies to an entire block of operations.

Additional kinds of visibility might be defined for certain programming languages, such as C++ *implementation* visibility (actually all forms of nonpublic visibility are language-dependent). Such visibility must be specified by property string or by a tool-specific convention.

- Where *name* is an identifier string.
- Where *return-type-expression* is a language-dependent specification of the implementation type or types of the value returned by the operation. The colon and the return-type are omitted if the operation does not return a value (as for C++ void). A list of expressions may be supplied to indicate multiple return values.
- Where *parameter-list* is a comma-separated list of formal parameters, each specified using the syntax:

  kind name : *type-expression* = *default-value*

  - where *kind* is **in, out,** or **inout**, with the default **in** if absent.
  - where *name* is the name of a formal parameter.
  - where *type-expression* is the (language-dependent) specification of an implementation type.
  - where *default-value* is an optional value expression for the parameter, expressed in and subject to the limitations of the eventual target language.
- Where *property-string* indicates property values that apply to the element. The property string is optional (the braces are omitted if no properties are specified).

A class-scope operation is shown by underlining the name and type expression string. An instance-scope operation is the default and is not marked.

An operation that does not modify the system state (one that has no side effects) is specified by the property “{query}”; otherwise, the operation may alter the system state, although there is no guarantee that it will do so.

The concurrency semantics of an operation are specified by a property string of the form “{concurrency = *name*}, where *name* is one of the names: *sequential, guarded, concurrent*. As a shorthand, one of the names may be used by itself in a property string to indicate the corresponding concurrency value. In the absence of a specification, the concurrency semantics are unspecified and must therefore be assumed to be sequential in the worst case.

The top-most appearance of an operation signature declares the operation on the class (and inherited by all of its descendents). If this class does not implement the operation; that is, does not supply a method, then the operation may be marked as "{abstract}" or the operation signature may be italicized to indicate that it is abstract. A subordinate appearance of the operation signature without the {abstract} property indicates that the subordinate class implements a method on the operation.

The actual text or procedure of a method may be indicated in a note attached to the operation.

If the objects of a class accept and respond to a given signal, an operation entry with the keyword «signal» indicates that the class accepts the given signal. The syntax is identical to that of an operation. The response of the object to the reception of the signal is shown with a state machine. Among other uses, this notation can show the response of objects of a class to error conditions and exceptions, which should be modeled as signals.

The specification of operation behavior is given as a note attached to the operation. The text of the specification should be enclosed in braces if it is a formal specification in some language (a semantic Constraint); otherwise, it should be plain text if it is just a natural-language description of the behavior (a Comment).

A stereotype keyword in guillemets precedes the entire operation string, including any visibility indicators. A property list in braces follows the entire operation string.

3.26.3 Presentation Options

The argument list and return type may be suppressed (together, not separately).

A tool may show the visibility indication in a different way, such as by using a special icon or by sorting the elements by group.

The syntax of the operation signature string can be that of a particular programming language, such as C++ or Smalltalk. Specific tagged properties may be included in the string.

A procedure body for a method may be shown in a note attached to the operation entry within the compartment (Figure 3-24 on page 3-47). The line is drawn to the string within the compartment. This approach is useful mainly for showing small method bodies.

.

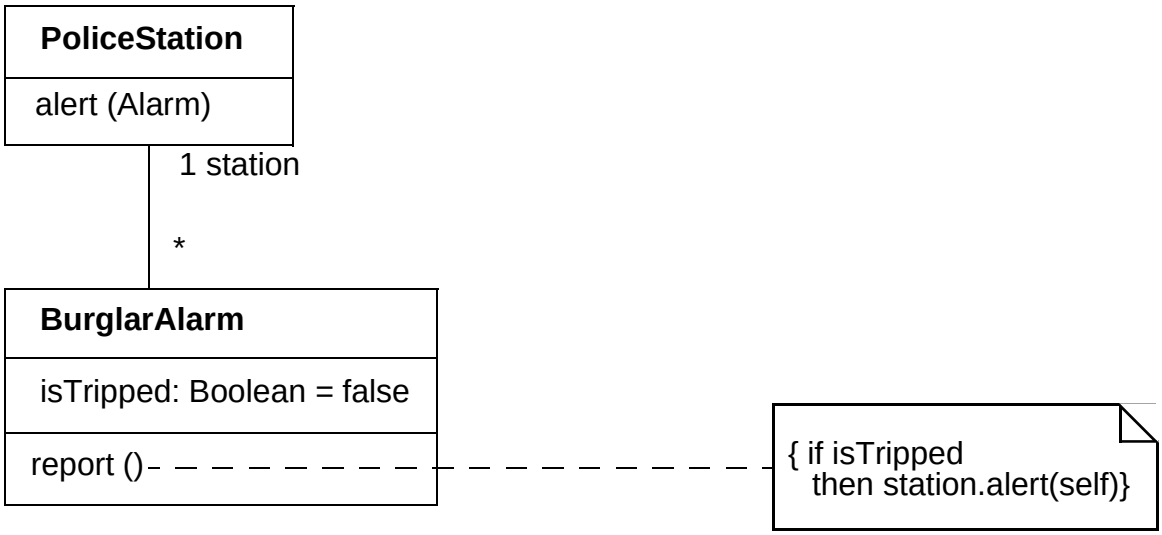


Figure 3-24 Note showing method body

3.26.4 Style Guidelines

Operation names typically begin with a lowercase letter. Operation names are in plain face. An abstract operation may be shown in italics.

3.26.5 Example

+display (): Location
+hide ()
+create ()
-attachXWindow(xwin:Xwindow*)

Figure 3-25 Operation List with a Variety of Operations

3.26.6 Mapping

A string entry within the operation compartment maps into an Operation or a Method within the Class corresponding to the class symbol. The properties of the operation map in accordance with the preceding descriptions. See the description of Section3.25, “Attribute,” on page 3-41 for additional details. Parameters without keywords map into Parameters with kind=in, otherwise according to the keyword. Return value names map into Parameters with kind=return.

If the entry has the keyword «signal», then it maps into a Reception on the Class instead.

The topmost appearance of an operation specification in a class hierarchy maps into an Operation definition in the corresponding Class or Interface. Interfaces do not have methods. In a Class, each appearance of an operation entry maps into the presence of a Method in the corresponding Class, unless the operation entry contains the {abstract} property (including use of conventions such as italics for abstract operations). If an abstract operation entry appears within a hierarchy in which the same operation has already been defined in an ancestor, it has no effect but is not an error unless the declarations are inconsistent.

Note that the operation string entry does not specify the body of a method.

3.27 Nested Class Declarations

3.27.1 Semantics

A class declared within another class belongs to the namespace of the other class and may only be used within it. This construct is primarily used for implementation reasons and for information hiding.

3.27.2 Notation

A declaring class and a class in its namespace may be connected by a line, with an “anchor” icon on the end connected to a declaring class (Figure 3-26 on page 3-48). An anchor icon is a cross inside a circle. The contents of the package are declared within the class and belong to its namespace.

3.27.3 Mapping

If Class B is attached to Class A by an “anchor” line with the “anchor” symbol on Class A, then Class B is declared within the Namespace of Class A. That is, the relationship between Class A and Class B is the namespace-ownedElement association.

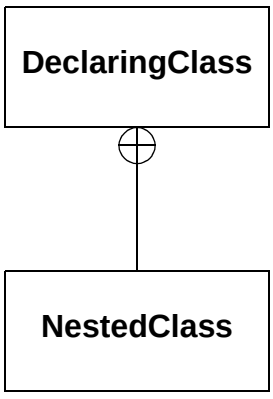


Figure 3-26 Nested class declaration

3.28 Type and Implementation Class

3.28.1 Semantics

Classes can be stereotyped as Types or Implementation Classes (although they can be left undifferentiated as well). A Type is used to specify a domain of objects together with operations applicable to the objects without defining the physical implementation of those objects. A Type may not include any methods, but it may provide behavioral specifications for its operations. It may also have attributes and associations that are defined solely for the purpose of specifying the behavior of the type's operations.

An Implementation Class defines the physical data structure (for attributes and associations) and methods of an object as implemented in traditional languages (C++, Smalltalk, etc.). An Implementation Class is said to *realize* a Type if it provides all of the operations defined for the Type with the same behavior as specified for the Type's operations. An Implementation Class may realize a number of different Types.

3.28.2 Notation

An undifferentiated class is shown with no stereotype. A type is shown with the stereotype "«type»." An implementation class is shown with the stereotype "«implementationClass»." A tool is also free to allow a default setting for an entire diagram, in which case all of the class symbols without explicit stereotype indications map into Classes with the default stereotype. This might be useful for a model that is close to the programming level.

The implementation of a type by a class is modeled as the Realization relationship, shown as a dashed line with a solid triangular arrowhead (a dashed "generalization arrow"). This symbol implies the realizing class provides at least all the operations of the Type, with conforming behavior, but it does not imply inheritance of structure (attributes or associations). The generalization hierarchy of a set of classes frequently parallels the generalization hierarchy of a set of types that they realize, but this is not mandatory, as long as each class provides the operations of the types that it realizes.

3.28.3 Example

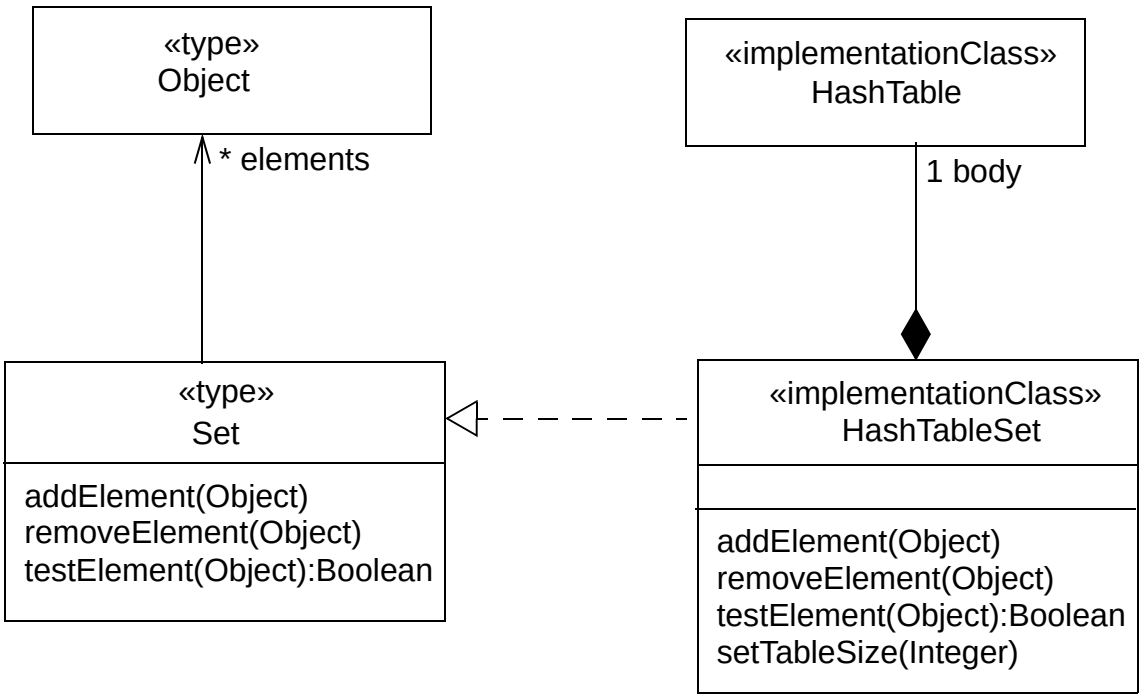


Figure 3-27 Notation for Types and Implementation Classes

3.28.4 Mapping

A class symbol with a stereotype (including “type” and “implementationClass”) maps into a Class with the corresponding stereotype. A class symbol without a stereotype maps into a Class with the default stereotype for the diagram (if a default has been defined by the modeler or tool); otherwise, it maps into a Class with no stereotype. The realization arrow between two symbols maps into an Abstraction relationship, with the «realize» stereotype, between the Classifiers corresponding to the two symbols. Realization is usually used between a class and an interface, but may also be used between any two classifiers to show conformance of behavior.

3.29 Interfaces

3.29.1 Semantics

An interface is a specifier for the externally-visible operations of a class, component, or other classifier (including subsystems) without specification of internal structure. Each interface often specifies only a limited part of the behavior of an actual class. Interfaces do not have implementation. They lack attributes, states, or associations; they only have operations. (An interface may be the target of a one-way association,

however, but it may not have an association that it can navigate.) Interfaces may have generalization relationships. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations, but Interface is a peer of Class within the UML metamodel (both are Classifiers).

3.29.2 Notation

An interface is a Classifier and may be shown using the full rectangle symbol with compartments and the keyword «interface». A list of operations supported by the interface is placed in the operation compartment. The attribute compartment may be omitted because it is always empty.

An interface may also be displayed as a small circle with the name of the interface placed below the symbol. The circle may be attached by a solid line to classifiers that support it. This indicates that the class provides all of the operations in the interface type (and possibly more). The operations provided are not shown on the circle notation; use the full rectangle symbol to show the list of operations. A class that uses or requires the operations supplied by the interface may be attached to the circle by a dashed arrow pointing to the circle. The dashed arrow implies that the class requires no more than the operations specified in the interface; the client class is not required to actually use *all* of the interface operations.

The Realization relationship from a classifier to an interface that it supports is shown by a dashed line with a solid triangular arrowhead (a “dashed generalization symbol”). This is the same notation used to indicate realization of a type by an implementation class. In fact, this symbol can be used between any two classifier symbols, with the meaning that the client (the one at the tail of the arrow) supports at least all of the operations defined in the supplier (the one at the head of the arrow), but with no necessity to support any of the data structure of the supplier (attributes and associations).

3.29.3 Example

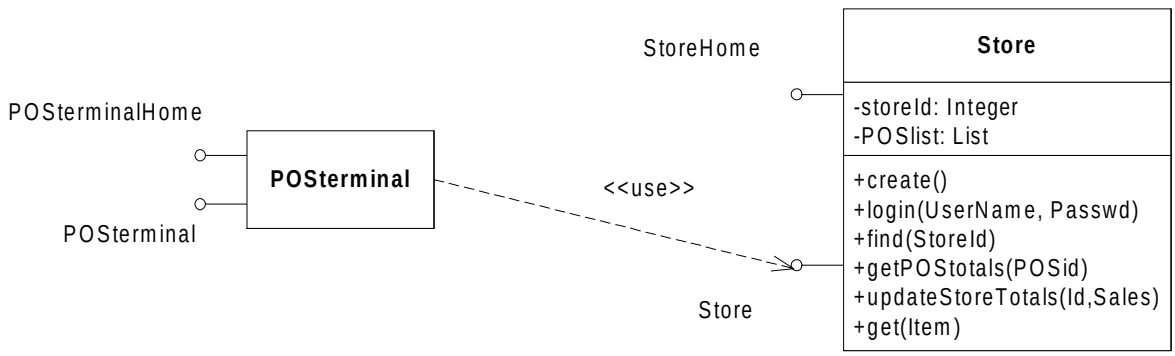


Figure 3-28 Shorthand Version of Interface Notation

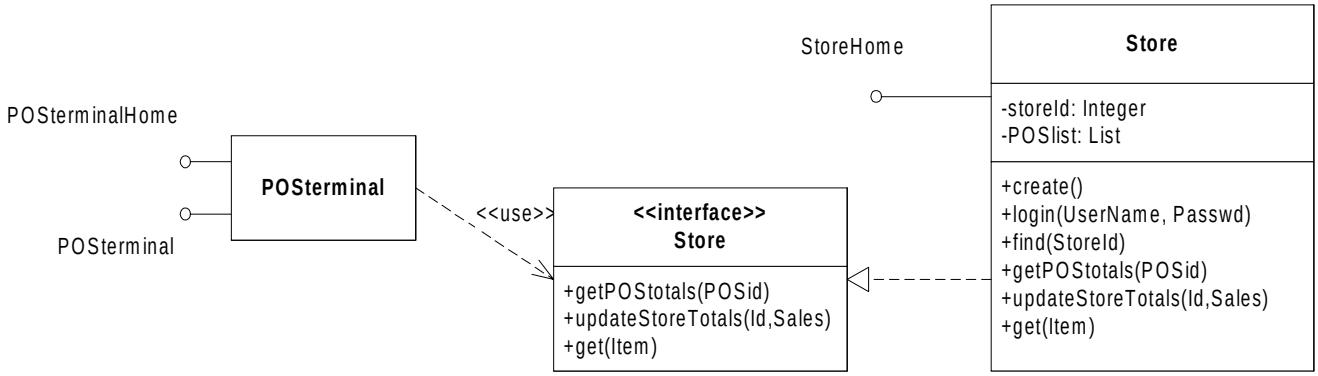


Figure 3-29 Longhand Version of Interface Notation

3.29.4 Mapping

A class rectangle symbol with stereotype «interface», or a circle on a class diagram, maps into an Interface element with the name given by the symbol. The operation list of a rectangle symbol maps into the list of Operation elements of the Interface.

A dashed generalization arrow from a class symbol to an interface symbol, or a solid line connecting a class symbol and an interface circle, maps into an Abstraction dependency with the «realize» stereotype between the corresponding Classifier and Interface elements. A dependency arrow from a class symbol to an interface symbol maps into a Usage dependency between the corresponding Classifier and Interface.

3.30 Parameterized Class (Template)

3.30.1 Semantics

A template is the descriptor for a class with one or more unbound formal parameters. It defines a family of classes, each class specified by binding the parameters to actual values. Typically, the parameters represent attribute types; however, they can also represent integers, other types, or even operations. Attributes and operations within the template are defined in terms of the formal parameters so they too become bound when the template itself is bound to actual values.

A template is not a directly usable class because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a class. Only a class can be a superclass or the target of an association (a one-way association *from* the template *to* another class is permissible, however). A template may be a subclass of an ordinary class. This implies that all classes formed by binding it are subclasses of the given superclass.

Parameterization can be applied to other ModelElements, such as Collaborations or even entire Packages. The description given here for classes applies to other kinds of modeling elements in the obvious way.

3.30.2 Notation

A small dashed rectangle is superimposed on the upper right-hand corner of the rectangle for the class (or to the symbol for another modeling element). The dashed rectangle contains a parameter list of formal parameters for the class and their implementation types. The list must not be empty, although it might be suppressed in the presentation. The name, attributes, and operations of the parameterized class appear as normal in the class rectangle; however, they may also include occurrences of the formal parameters. Occurrences of the formal parameters can also occur inside of a context for the class, for example, to show a related class identified by one of the parameters.

3.30.3 Presentation Options

The parameter list may be comma-separated or it may be one per line.

Parameters are restricted attributes, shown as strings with the syntax:

name : *type* = *default-value*

- Where *name* is an identifier for the parameter with scope inside the template.
- Where *type* is a string designating a *Classifier* for the parameter. If it is a simple word, it must be the name of a Classifier. Otherwise it is a programming-language dependent string that maps into a ProgrammingLanguageDataType according to the programming language (if any) for the diagram context or specified in a support tool.
- Where *default-value* is a string designating an Expression for a default value that is used when the corresponding argument is omitted in a Binding. The equal sign and expression may be omitted, in which case there is no default value and the argument must be supplied in a Binding.

If the type name is omitted, the parameter type is assumed to be Classifier. The value supplied for an argument in a Binding must be the name of a Classifier (including a class or a data type). Other parameter types (such as Integer) must be explicitly shown. The value supplied for an argument in a Binding must be an actual instance value of the given kind.

3.30.4 Example

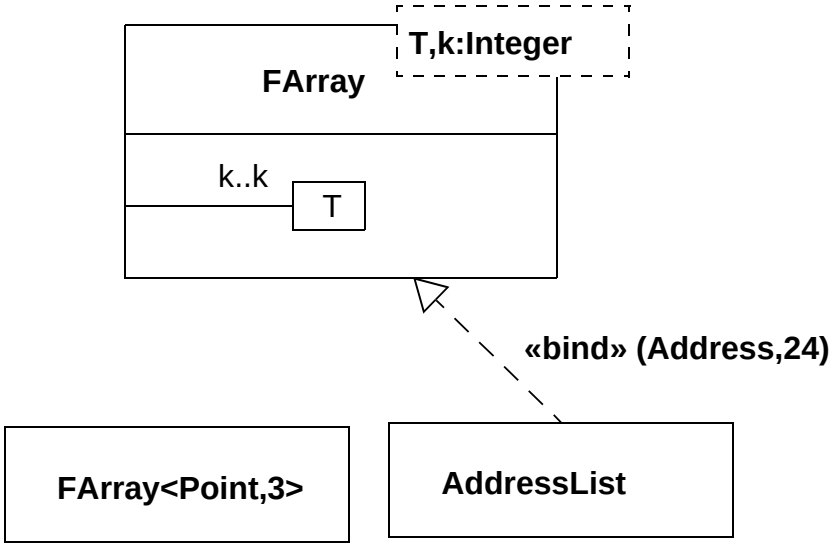


Figure 3-30 Template Notation with Use of Parameter as a Reference

3.30.5 Mapping

The addition of the template dashed box to a symbol causes the addition of the parameter names in the list as ModelElements within the Namespace of the ModelElement corresponding to the base symbol (or to the Namespace containing a ModelElement that is not itself a Namespace). Each of the parameter ModelElements has the templateParameter association to the base ModelElement.

3.31 Bound Element

3.31.1 Semantics

A template cannot be used directly in an ordinary relationship such as generalization or association, because it has a free parameter that is not meaningful outside of a scope that declares the parameter. To be used, a template's parameters must be *bound* to actual values. The actual value for each parameter is an expression defined within the scope of use. If the referencing scope is itself a template, then the parameters of the referencing template can be used as actual values in binding the referenced template. The parameter names in the two templates cannot be assumed to correspond because they have no scope outside of their respective templates.

3.31.2 Notation

A bound element is indicated by a text syntax in the name string of an element, as follows:

Template-name ‘<‘ *value-list* ‘>’

- Where *value-list* is a comma-delimited non-empty list of value expressions.
- Where *Template-name* is identical to the name of a template.

For example, VArray<Point,3> designates a class described by the template Varray.

The number and type of values must match the number and type of the template parameters for the template of the given name.

The bound element name may be used anywhere that an element name of the parameterized kind could be used. For example, a bound class name could be used within a class symbol on a class diagram, as an attribute type, or as part of an operation signature.

Note that a bound element is fully specified by its template; therefore, its content may not be extended. Declaration of new attributes or operations for classes is not permitted, for example, but a bound class could be subclassed and the subclass extended in the usual way.

The relationship between the bound element and its template alternatively may be shown by a Dependency relationship with the keyword «bind». The arguments are shown in parentheses after the keyword. In this case, the bound form may be given a name distinct from the template.

3.31.3 Style Guidelines

The attribute and operation compartments are normally suppressed within a bound class, because they must not be modified in a bound template.

3.31.4 Example

See Figure 3-30 on page 3-54.

3.31.5 Mapping

The use of the bound element syntax for the name of a symbol maps into a Binding dependency between the dependent ModelElement (such as Class) corresponding to the bound element symbol and the provider ModelElement (again, such as Class) whose name matches the name part of the bound element without the arguments. If the name does not match a template element or if the number of arguments in the bound element does not match the number of parameters in the template, then the model is ill formed. Each argument position in the bound element maps into a TemplateArgument bearing a binding link to the Binding dependency and a modelElement link to the

ModelElement that is implicitly substituted for the template parameter in the corresponding position in the template definition. An explicitly drawn «bind» dependency symbol maps to a Binding dependency with arguments as described above.

3.32 Utility

A utility is a grouping of global variables and procedures in the form of a class declaration. This is not a fundamental construct, but a programming convenience. The attributes and operations of the utility become global variables and procedures. A utility is modeled as a stereotype of a classifier.

3.32.1 Semantics

The instance-scope attributes and operations of a utility are interpreted as global attributes and operations. It is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope.

3.32.2 Notation

A utility is shown as the stereotype «utility» of Class. It may have both attributes and operations, all of which are treated as global attributes and operations.

3.32.3 Example

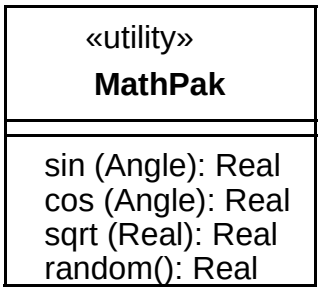


Figure 3-31 Notation for Utility

3.32.4 Mapping

This is not a special symbol. It simply maps into a Class element with the «utility» stereotype.

3.33 Metaclass

3.33.1 Semantics

A metaclass is a class whose instances are classes.

3.33.2 Notation

A metaclass is shown as the stereotype «metaclass» of Class.

3.33.3 Mapping

This is not a special symbol. It simply maps into a Class element with the «metaclass» stereotype.

3.34 Enumeration

3.34.1 Semantics

An Enumeration is a user-defined data type whose instances are a set of user-specified named enumeration literals. The literals have a relative order but no algebra is defined on them.

3.34.2 Notation

An Enumeration is shown using the Classifier notation (a rectangle) with the keyword «enumeration». The name of the Enumeration is placed in the upper compartment. An ordered list of enumeration literals may be placed, one to a line, in the middle compartment. Operations defined on the literals may be placed in the lower compartment. The lower and middle compartments may be suppressed.

3.34.3 Mapping

Maps into an Enumeration with the given list of enumeration literals.

3.35 Stereotype Declaration

3.35.1 Semantics

A Stereotype is a user-defined metaelement whose structure matches an existing UML metaelement (its “base class”). Because it is user defined, a stereotype declaration is an element that appears at the “model” layer of the UML four-layer metamodeling hierarchy although it conceptually belongs in the layer above, the metamodel layer.

3.35.2 Notation

Because stereotypes span two different metamodeling layers, a special notation is required to clearly indicate the crossover between the two layers. Specifically, it is necessary to show how a model-level element (the stereotype) relates to its metaelement (its UML base class). This is denoted using a special stereotype of Dependency called «stereotype» as shown in Figure 3-32 on page 3-59.

The Stereotype itself is shown using the Classifier notation (a rectangle) with the keyword «stereotype» (Figure 3-32). The name of the Stereotype is placed in the upper compartment. Constraints on elements described by the stereotype may be placed in a named compartment called **Constraints**. Required tags may be placed in a named compartment called **Tags**. Individual items (tags) in the list are defined according to the following format:

```
tagDefinitionName : String [multiplicity]
```

where `string` can be either a string matching the name of a data type representing the type of the values of the tag, or it is a reference to a metaclass or a stereotype. In the latter case, the string has the form:

«metaclass» metaclassName

or

«stereotype» stereotypeName

where `metaclassName` is the name of the referenced metaclass and is the name of the references stereotype. The multiplicity element is optional and conforms to standard rules for specifying multiplicities. In case of a range specification, a lower bound of zero indicates an optional tag.

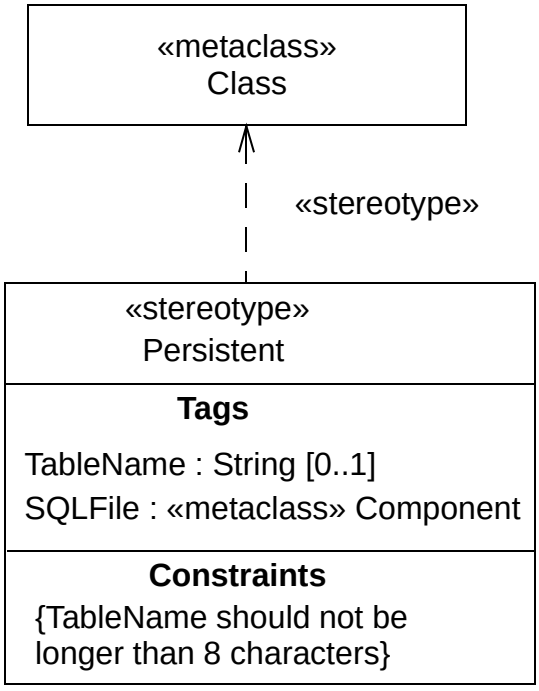


Figure 3-32 Notational form for declaring a stereotype

In the example diagram in Figure 3-32, the stereotype Persistent is a stereotype of the UML metaelement Class. TableName is an optional tag whose type is a model type called String while SQLFile is a reference to an instance of Component in the model.

An icon can be defined for the stereotype, but its graphical definition is outside the scope of UML and must be handled by an editing tool.

An alternative and usually more compact way of specifying stereotypes and tags using tables is shown in Figure 3-33 and Figure 3-34, respectively.

Stereotype	Base Class	Parent	Tags	Constraints	Description
Architectural Element	Generalizable Element	N/A	N/A	N/A	A generic stereotype that is the parent of all other stereotypes used for architectural modeling .
Capsule	Class	Architectural Element	isDynamic	self.isActive = true	Indicates a class that is used to model the structural components of an architecture specification.

Figure 3-33 Tabular form for specifying stereotypes

Tag	Stereotype	Type	Multiplicity	Description
isDynamic	Capsule	UML::Datatypes::Boolean	1	Used to identify if the associated capsule class may be created and destroyed dynamically.

Figure 3-34 Tabular form for specifying tags

Each row of the stereotype specification table in Figure 3-33 defines one stereotype and each row in the tag specification table in Figure 3-34 contains one tag definition.

The columns of the stereotype specification table are defined as follows:

- *Stereotype* - the name of the stereotype.
- *Base Class* - the UML metamodel element that serves as the base for the stereotype.
- *Parent* - the direct parent of the stereotype being defined (NB: if one exists, otherwise the symbol “N/A” is used).
- *Tags* - a list of all tags of the tagged values that may be associated with this stereotype (or N/A if none are defined).
- *Constraints* - a list of constraints associated with the stereotype.
- *Description* - an informal description with possible explanatory comments.

The columns of the tag specification table are defined as follows:

- *Tag* - the name of the tag.
- *Stereotype* - the name of the stereotype that owns this tag, or “N/A” if it is a stand alone tag.
- *Type* - the name of the type of the values that can be associated with the tag.
- *Multiplicity* - the maximum number of values that may be associated with one tag instance.
- *Description* - an informal description with possible explanatory comments.

In the case of both the stereotype specification table and the tag specification table, columns that are not applicable may be omitted.

In the example stereotype specification table of Figure 3-34, two related stereotypes are defined. The first row declares the stereotype ArchitecturalElement, which is a stereotype of GeneralizableElement, while the second row declares the stereotype Capsule, which is a specialization of the ArchitecturalElement stereotype, but which applies only to instances of Class, which is a subclass of GeneralizableElement in the metamodel.

The equivalent declaration as the one table in Figure 3-34, less the constraints and the informal descriptions, is shown graphically in Figure 3-35.

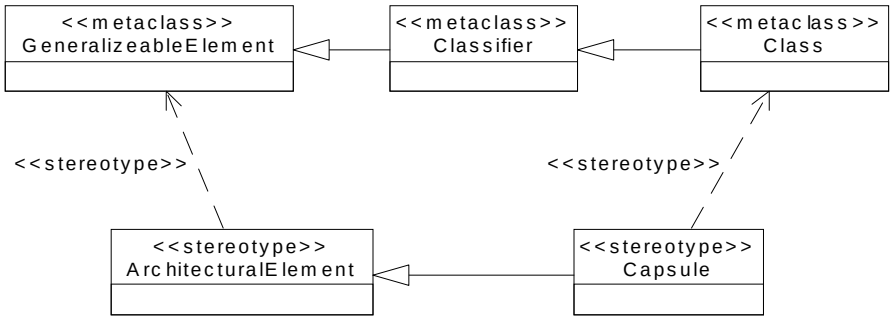


Figure 3-35 Graphical equivalent of the stereotype declarations shown in Figure 3-34

3.35.3 Mapping

A classifier with a stereotype «metaclass» maps into a UML metaelement and a classifier with a stereotype «stereotype» maps into a Stereotype. The «stereotype» dependency maps to the baseClass attribute definition of the stereotype. The constraints listed in the **Constraints** compartment map to stereotype constraints and the items in the **Tags** compartment map to the defined tags of the stereotype. Each item in the **Tags** list maps to a TagDefinition. The string before the colon separator maps to the name of the tag definition while the string following the colon maps to an instance of Name. If a multiplicity specification is included in the item, it maps to the multiplicity attribute of the tag definition.

3.36 Powertype

3.36.1 Semantics

A Powertype is a user-defined metaelement whose instances are classes in the model.

3.36.2 Notation

A Powertype is shown using the Classifier notation (a rectangle) with the stereotype keyword «powertype». The name of the Powertype is placed in the upper compartment. Because the elements are ordinary classes, attributes and operations on the powertype are usually not defined by the user.

The instances of the powertype may be indicated by placing a dashed line across the parent lines of the classes with the syntax

```
discriminatorName: powertypeName,
```

where the powertype name on the line implicitly defines a powertype if one is not explicitly defined.

3.36.3 Mapping

Maps into a Class with the «powertype» stereotype with the given classes as instances.

3.37 Class Pathnames

3.37.1 Notation

Class symbols (rectangles) serve to define a class and its properties, such as relationships to other classes. A reference to a class in a different package is notated by using a pathname for the class, in the form:

package-name :: *class-name*

References to classes also appear in text expressions, most notably in type specifications for attributes and variables. In these places a reference to a class is indicated by simply including the name of the class itself, including a possible package name, subject to the syntax rules of the expression.

3.37.2 Example

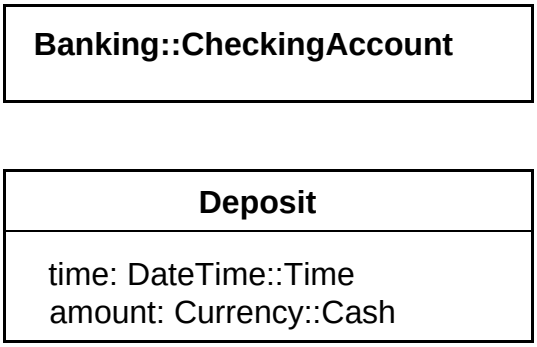


Figure 3-36 Pathnames for Classes in Other Packages

3.37.3 Mapping

A class symbol whose name string is a pathname represents a reference to the Class with the given name inside the package with the given name. The name is assumed to be defined in the target package; otherwise, the model is ill formed. A Relationship from a symbol in the current package; that is, the package containing the diagram and its mapped elements to a symbol in another package is part of the current package.

3.38 Accessing or Importing a Package

3.38.1 Semantics

An element may reference an element contained in a different package. On the package level, the «access» dependency indicates that the contents of the target package may be referenced by the client package or packages recursively embedded within it. The target references must have visibility sufficient for the referents: public visibility for an unrelated package, public or protected visibility for a descendant of the target package, or any visibility for a package nested inside the target package (an access dependency is not required for the latter case). A package nested inside the package making the access gets the same access.

Note that an access dependency does not modify the namespace of the client or in any other way automatically create references; it merely grants permission to establish references. Note also that a tool could automatically create access dependencies for users if desired when references are created.

An import dependency grants access and also loads the names with appropriate visibility in the target namespace into the accessing package; that is, a pathname is not necessary to reference them. Such names are not automatically re-exported; however; a name must be explicitly re-exported (and may be given a new name and visibility at the same time).

3.38.2 Notation

The access dependency is displayed as a dependency arrow from the referencing (client) package to the target (supplier) package containing the target of the references. The arrow has the stereotype keyword «access». This dependency indicates that elements within the client package may legally reference elements within the supplier. The references must also satisfy visibility constraints specified by the supplier. Note that the dependency does not automatically create any references. It merely grants permission for them to be established.

The import dependency has the same notation as the access dependency except it has the stereotype keyword «import».

3.38.3 Example

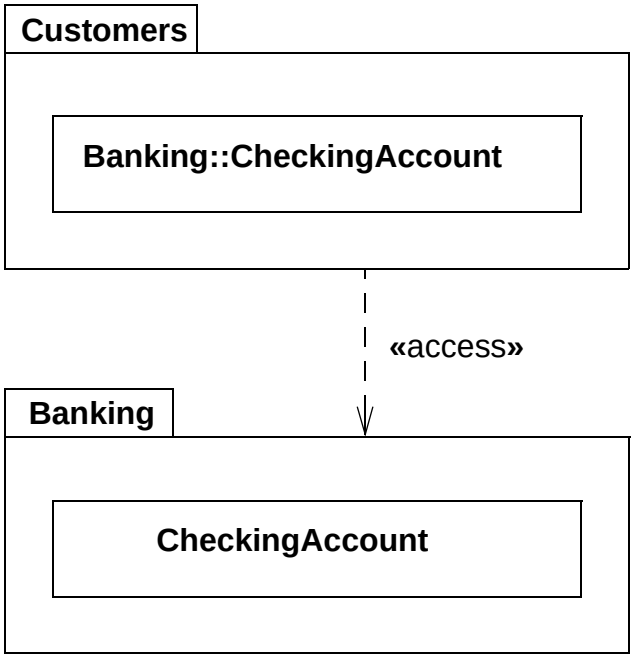


Figure 3-37 Access Dependency Among Packages

3.38.4 Mapping

This is not a special symbol. It maps into a Permission dependency with the stereotype «access» or «import» between the two packages.

3.39 Object

3.39.1 Semantics

An object represents a particular instance of a class. It has identity and attribute values. A similar notation also represents a role within a collaboration because roles have instance-like characteristics.

3.39.2 Notation

The object notation is derived from the class notation by underlining instance-level elements, as explained in the general comments in Section 3.12, “Type-Instance Correspondence,” on page 3-14.

An object shown as a rectangle with two compartments.

The top compartment shows the name of the object and its class, all underlined, using the syntax:

objectname : *classname*

The classname can include a full pathname of enclosing package, if necessary. The package names precede the classname and are separated by double colons. For example:

```
display_window: WindowingSystem::GraphicWindows::Window
```

A stereotype for the class may be shown textually (in guillemets above the name string) or as an icon in the upper right corner. The stereotype for an object must match the stereotype for its class.

To show multiple classes that the object is an instance of, use a comma-separated list of classnames. These classnames must be legal for multiple classification; that is, only one implementation class permitted, but multiple types permitted.

To show the presence of an object in a particular state of a class, use the syntax:

objectname : *classname* '[' *statename-list* ']'

The list must be a comma-separated list of names of states that can legally occur concurrently.

The second compartment shows the attributes for the object and their values as a list. Each value line has the syntax:

attributename : *type* = *value*

The type is redundant with the attribute declaration in the class and may be omitted.

The value is specified as a literal value. UML does not specify the syntax for literal value expressions; however, it is expected that a tool will specify such a syntax using some programming language.

The flow relationship between two values of the same object over time can be shown by connecting two object symbols by a dashed arrow with the keyword «become». If the flow arrow is on a collaboration diagram, the label may also include a sequence number to show when the value changes. Similarly, the keyword «copy» can be used to show the creation of one object from another object value.

3.39.3 Presentation Options

The name of the object may be omitted. In this case, the colon should be kept with the class name. This represents an anonymous object of the given class given identity by its relationships.

The class of the object may be suppressed (together with the colon).

The attribute value compartment as a whole may be suppressed.

Attributes whose values are not of interest may be suppressed.

Attributes whose values change during a computation may show their values as a list of values held over time. In an interactive tool, they might even change dynamically. An alternate notation is to show the same object more than once with a «becomes» relationship between them.

3.39.4 Style Guidelines

Objects may be shown on class diagrams. The elements on collaboration diagrams are not objects, because they describe many possible objects. They are instead roles that may be held by object. Objects in class diagrams serve mainly to show examples of data structures.

3.39.5 Variations

For a language such as *Self* in which operations can be attached to individual objects at run time, a third compartment containing operations would be appropriate as a language-specific extension.

3.39.6 Example

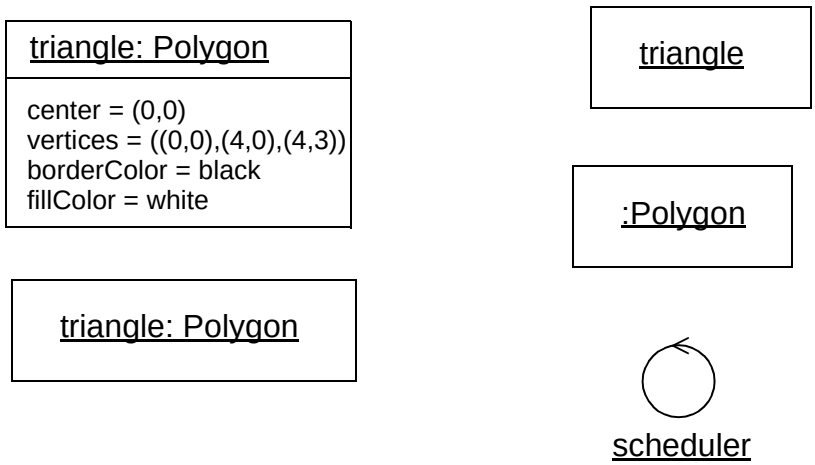


Figure 3-38 Objects

3.39.7 Mapping

In an object diagram, or within an ordinary class diagram, an object symbol maps into an Object of the Class (or Classes) given by the *classname* part of the name string. The attribute list in the symbol maps into a set of AttributeLinks attached to the Object, with values given by the value expressions in the attribute list in the symbol. If a list of states in brackets follows the class name, then this maps into a ClassifierInState with the named Class as its type and the named States as the states. The ClassfierInState classifies the Object.

3.40 Composite Object

3.40.1 Semantics

A composite object represents a high-level object made of tightly-bound parts. This is an instance of a composite class, which implies the composition aggregation between the class and its parts. A composite object is similar to (but simpler and more restricted than) a collaboration; however, it is defined completely by composition in a static model. See Section 3.48, “Composition,” on page 3-81.

3.40.2 Notation

A composite object is shown as an object symbol. The name string of the composite object is placed in a compartment near the top of the rectangle (as with any object). The lower compartment holds the parts of the composite object instead of a list of attribute values. (However, even a list of attribute values may be regarded as the parts of a composite object, so there is not a great difference.) It is possible for some of the parts to be composite objects with further nesting.

3.40.3 Example

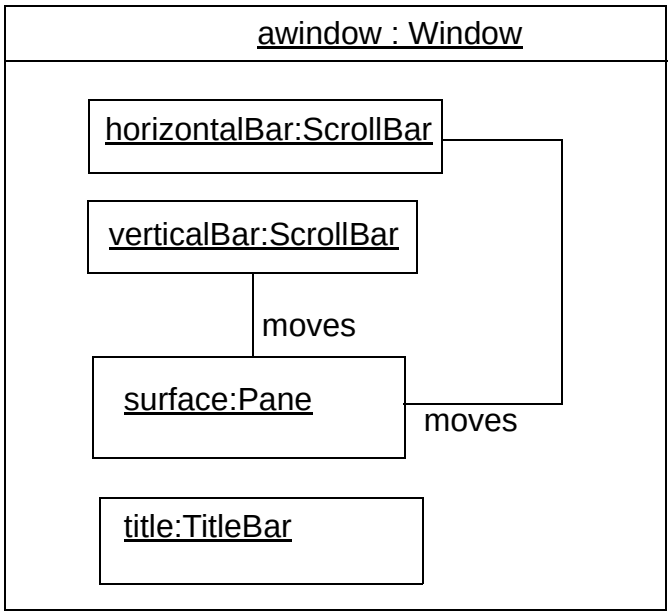


Figure 3-39 Composite Objects

3.40.4 Mapping

A composite object symbol maps into an Object of the given Class with composition links to each of the Objects and Links corresponding to the class box symbols and to association path symbols directly contained within the boundary of the composite object symbol (and not contained within another deeper boundary).

3.41 Association

Binary associations are shown as lines connecting two classifier symbols. The lines may have a variety of adornments to show their properties. Ternary and higher-order associations are shown as diamonds connected to class symbols by lines.

3.42 Binary Association

3.42.1 Semantics

A binary association is an association among exactly two classifiers (including the possibility of an association from a classifier to itself).

3.42.2 Notation

A binary association is drawn as a solid path connecting two classifier symbols (both ends may be connected to the same classifier, but the two ends are distinct). The path may consist of one or more connected segments. The individual segments have no semantic significance, but may be graphically meaningful to a tool in dragging or resizing an association symbol. A connected sequence of segments is called a *path*.

In a binary association, both ends may attach to the same classifier. The links of such an association may connect two different instances from the same classifier or one instance to itself. The latter case may be forbidden by a constraint if necessary.

The end of an association where it connects to a classifier is called an *association end*. Most of the interesting information about an association is attached to its ends.

The path may also have graphical adornments attached to the main part of the path itself. These adornments indicate properties of the entire association. They may be dragged along a segment or across segments, but must remain attached to the path. It is a tool responsibility to determine how close association adornments may approach an end so that confusion does not occur. The following kinds of adornments may be attached to a path.

3.42.2.1 association name

Designates the (optional) name of the association.

It is shown as a name string near the path (but not near enough to an end to be confused with a rolename). The name string may have an optional small black solid triangle in it. The point of the triangle indicates the direction in which to read the name. The name-direction arrow has no semantics significance, it is purely descriptive. The classifiers in the association are ordered as indicated by the name-direction arrow.

Note – There is no need for a *name direction* property on the association model; the ordering of the classifiers within the association *is* the name direction. This convention works even with n-ary associations.

A stereotype keyword within guillemets may be placed above or in front of the association name. A property string may be placed after or below the association name.

3.42.2.2 association class symbol

Designates an association that has class-like properties, such as attributes, operations, and other associations. This is present if, and only if, the association is an association class. It is shown as a class symbol attached to the association path by a dashed line.

The association path and the association class symbol represent the same underlying model element, which has a single name. The name may be placed on the path, in the class symbol, or on both (but they must be the same name).

Logically, the association class and the association are the same semantic entity; however, they are graphically distinct. The association class symbol can be dragged away from the line, but the dashed line must remain attached to both the path and the class symbol.

3.42.3 Presentation Options

When two paths cross, the crossing may optionally be shown with a small semicircular jog to indicate that the paths do not intersect (as in electrical circuit diagrams).

3.42.4 Style Guidelines

Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The choice of a particular set of line styles is a user choice.

3.42.5 Options

3.42.5.1 Xor-association

An xor-constraint indicates a situation in which only one of several potential associations may be instantiated at one time for any single instance. This is shown as a dashed line connecting two or more associations, all of which must have a classifier in

common, with the constraint string “{xor}” labeling the dashed line. Any instance of the classifier may only participate in one of the associations at one time. Each rolename must be different. (This is simply a predefined use of the constraint notation.)

3.42.6 Example

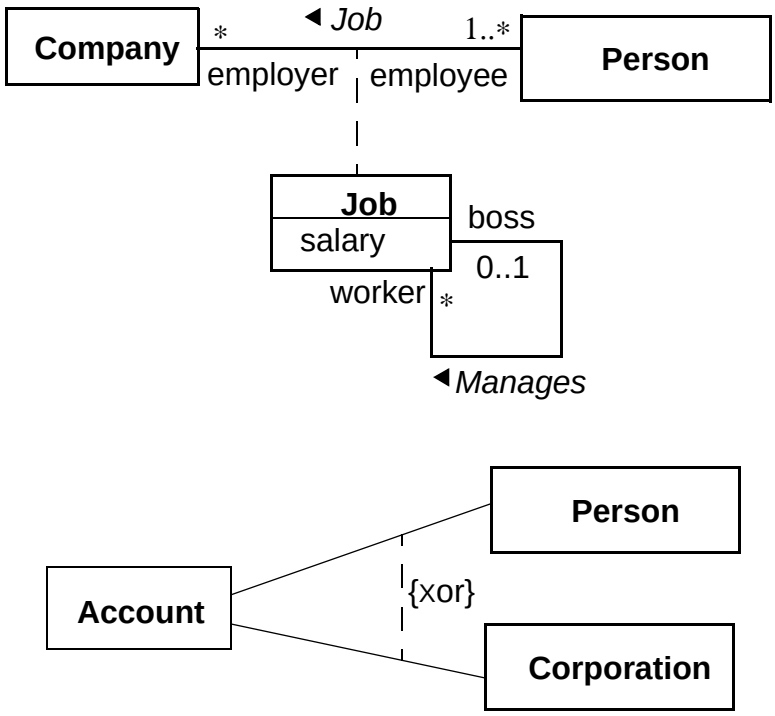


Figure 3-40 Association Notation

3.42.7 Mapping

An association path connecting two class symbols maps to an Association between the corresponding Classifiers. If there is an arrow on the association name, then the Class corresponding to the tail of the arrow is the first class and the Classifier corresponding to the head of the arrow is the second Classifier in the ordering of ends of the Association; otherwise, the ordering of ends in the association is undetermined. The adornments on the path map into properties of the Association as described above. The Association is owned by the package containing the diagram.

3.43 Association End

3.43.1 Semantics

An association end is simply an end of an association where it connects to a classifier. It is part of the association, not part of the classifier. Each association has two or more ends. Most of the interesting details about an association are attached to its ends. An association end is not a separable element, it is just a mechanical part of an association.

3.43.2 Notation

The path may have graphical adornments at each end where the path connects to the classifier symbol. These adornments indicate properties of the association related to the classifier. The adornments are part of the association symbol, not part of the classifier symbol. The end adornments are either attached to the end of the line, or near the end of the line, and must drag with it. The following kinds of adornments may be attached to an association end.

3.43.2.1 multiplicity

Specified by a text syntax. Multiplicity may be suppressed on a particular association or for an entire diagram. In an incomplete model the multiplicity may be unspecified in the model itself. In this case, it must be suppressed in the notation. See Section 3.44, “Multiplicity,” on page 3-75.

3.43.2.2 ordering

If the multiplicity is greater than one, then the set of related elements can be ordered or unordered. If no indication is given, then it is unordered (the elements form a set). Various kinds of ordering can be specified as a constraint on the association end. The declaration does not specify how the ordering is established or maintained. Operations that insert new elements must make provision for specifying their position either implicitly (such as at the end) or explicitly. Possible values include:

- unordered - the elements form an unordered set. This is the default and need not be shown explicitly.
- ordered - the elements of the set have an ordering, but duplicates are still prohibited. This generic specification includes all kinds of ordering. This may be specified by the keyword syntax “{ordered}.”

An ordered relationship may be implemented in various ways; however, this is normally specified as a language-specified code generation property to select a particular implementation. An implementation extension might substitute the data structure to hold the elements for the generic specification “ordered.”

At implementation level, sorting may also be specified. It does not add new semantic information, but it expresses a design decision:

- sorted - the elements are sorted based on their internal values. The actual sorting rule is best specified as a separate constraint.

3.43.2.3 qualifier

A qualifier is optional, but not suppressible. See Section 3.45, “Qualifier,” on page 3-76.

3.43.2.4 navigability

An arrow may be attached to the end of the path to indicate that navigation is supported toward the classifier attached to the arrow. Arrows may be attached to zero, one, or two ends of the path. To be totally explicit, arrows may be shown whenever navigation is supported in a given direction. In practice, it is often convenient to suppress some of the arrows and just show exceptional situations. See Section 3.22.3, “Presentation Options,” on page 3-36 for details.

3.43.2.5 aggregation indicator

A hollow diamond is attached to the end of the path to indicate aggregation. The diamond may not be attached to both ends of a line, but it need not be present at all. The diamond is attached to the class that is the aggregate. The aggregation is optional, but not suppressible.

If the diamond is filled, then it signifies the strong form of aggregation known as *composition*. See Section 3.48, “Composition,” on page 3-81.

3.43.2.6 rolename

A name string near the end of the path. It indicates the role played by the class attached to the end of the path near the rolename. The rolename is optional, but not suppressible.

3.43.2.7 interface specifier

The name of a Classifier with the syntax:

‘:’ *classifiername, . . .*

It indicates the behavior expected of an associated object by the related instance. In other words, the interface specifier specifies the behavior required to enable the association. In this case, the actual classifier usually provides more functionality than required for the particular association (since it may have other responsibilities).

The use of a rolename and interface specifier are equivalent to creating a small collaboration that includes just an association and two roles, whose structure is defined by the rolename and attached classifier on the original association. Therefore, the

original association and classifiers are a use of the collaboration. The original classifier must be compatible with the interface specifier (which can be an interface or a type, among other kinds of classifiers).

If an interface specifier is omitted, then the association may be used to obtain full access to the associated class.

3.43.2.8 changeability

If the links are changeable (can be added, deleted, and moved), then no indicator is needed. The property {frozen} indicates that no links may be added, deleted, or moved from an object (toward the end with the adornment) after the object is created and initialized. The property {addOnly} indicates that additional links may be added (presumably, the multiplicity is variable); however, links may not be modified or deleted.

3.43.2.9 visibility

Specified by a visibility indicator (‘+’, ‘#’, ‘-’ or explicit property name such as {public}) in front of the rolename. Specifies the visibility of the association traversing in the direction toward the given rolename. See Section 3.25, “Attribute,” on page3-41 for details of visibility specification.

Other properties can be specified for association ends, but there is no graphical syntax for them. To specify such properties, use the constraint syntax near the end of the association path (a text string in braces). Examples of other properties include mutability.

3.43.3 Presentation Options

If there are two or more aggregations to the same aggregate, they may be drawn as a tree by merging the aggregation end into a single segment. This requires that all of the adornments on the aggregation ends be consistent. This is purely a presentation option, there are no additional semantics to it.

Various options are possible for showing the navigation arrows on a diagram. These can vary from time to time by user request or from diagram to diagram.

- Presentation option 1: Show all arrows. The absence of an arrow indicates navigation is not supported.
- Presentation option 2: Suppress all arrows. No inference can be drawn about navigation. This is similar to any situation in which information is suppressed from a view.
- Presentation option 3: Suppress arrows for associations with navigability in both directions, show arrows only for associations with one-way navigability. In this case, the two-way navigability cannot be distinguished from no-way navigation; however, the latter case is normally rare or nonexistent in practice. This is yet another example of a situation in which some information is suppressed from a view.

3.43.4 Style Guidelines

If there are multiple adornments on a single association end, they are presented in the following order, reading from the end of the path attached to the classifier toward the bulk of the path:

- qualifier
- aggregation symbol
- navigation arrow

Rolenames and multiplicity should be placed near the end of the path so that they are not confused with a different association. They may be placed on either side of the line. It is tempting to specify that they will always be placed on a given side of the line (clockwise or counterclockwise), but this is sometimes overridden by the need for clarity in a crowded layout. A rolename and a multiplicity may be placed on opposite sides of the same association end, or they may be placed together (for example, "* employee").

3.43.5 Example

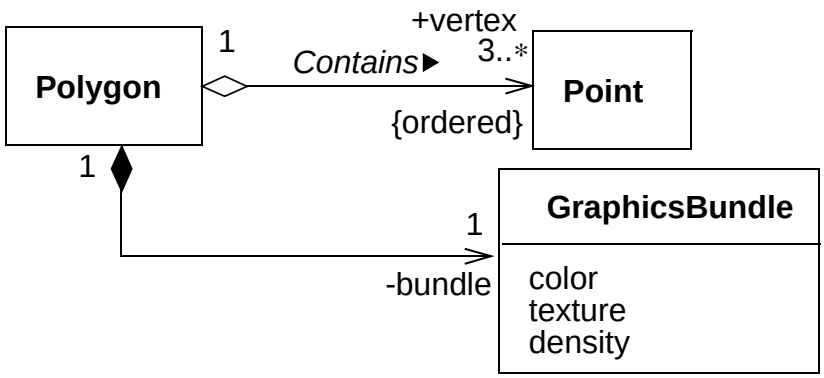


Figure 3-41 Various Adornments on Association Roles

3.43.6 Mapping

The adornments on the end of an association path map into properties of the corresponding role of the Association. In general, implications cannot be drawn from the absence of an adornment (it may simply be suppressed) but see the preceding descriptions for details. The interface specifier maps into the "specification" rolename in the AssociationEnd-Classifier association.

3.44 Multiplicity

3.44.1 Semantics

A multiplicity item specifies the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity specification is a subset of the open set of nonnegative integers.

3.44.2 Notation

A multiplicity specification is shown as a text string comprising a comma-separated sequence of integer intervals, where an interval represents a (possibly infinite) range of integers, in the format:

lower-bound .. upper-bound

where *lower-bound* and *upper-bound* are literal integer values, specifying the closed (inclusive) range of integers from the lower bound to the upper bound. In addition, the star character (*) may be used for the upper bound, denoting an unlimited upper bound. In a parameterized context (such as a template), the bounds could be expressions but they must evaluate to literal integer values for any actual use. Unbound expressions that do not evaluate to literal integer values are not permitted.

If a single integer value is specified, then the integer range contains the single integer value.

If the multiplicity specification comprises a single star (*), then it denotes the unlimited nonnegative integer range, that is, it is equivalent to 0..* (zero or more).

A multiplicity of 0..0 is meaningless as it would indicate that no instances can occur.

Expressions in some specification language can be used for multiplicities, but they must resolve to fixed integer ranges within the model; that is, no dynamic evaluation of expressions, essentially the same rule on literal values as most programming languages.

3.44.3 Style Guidelines

Preferably, intervals should be monotonically increasing. For example, “1..3,7,10” is preferable to “7,10,1..3”.

Two contiguous intervals should be combined into a single interval. For example, “0..1” is preferable to “0,1”.

3.44.4 Example

0..1

1

0..*

*

1..*

1..6

1..3,7..10,15,19..*

3.44.5 Mapping

A multiplicity string maps into a Multiplicity value with one or more MultiplicityRanges. Duplications or other nonstandard presentation of the string itself have no effect on the mapping. Note that Multiplicity is a value and not an object. It cannot stand on its own, but is the value of some element property.

3.45 Qualifier

3.45.1 Semantics

A qualifier is an attribute or list of attributes whose values serve to partition the set of instances associated with an instance across an association. The qualifiers are attributes of the association.

3.45.2 Notation

A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the classifier that it connects to. The qualifier rectangle is part of the association path, not part of the classifier. The qualifier rectangle drags with the path segments. The qualifier is attached to the source end of the association. An instance of the source classifier, together with a value of the qualifier, uniquely select a partition in the set of target classifier instances on the other end of the association; that is, every target falls into exactly one partition.

The multiplicity attached to the target end denotes the possible cardinalities of the set of target instances selected by the pairing of a source instance and a qualifier value. Common values include:

- “0..1” (a unique value may be selected, but every possible qualifier value does not necessarily select a value).
- “1” (every possible qualifier value selects a unique target instance; therefore, the domain of qualifier values must be finite).
- “*” (the qualifier value is an index that partitions the target instances into subsets).

The qualifier attributes are drawn within the qualifier box. There may be one or more attributes shown one to a line. Qualifier attributes have the same notation as classifier attributes, except that initial value expressions are not meaningful.

It is permissible (although somewhat rare), to have a qualifier on each end of a single association.

3.45.3 Presentation Options

A qualifier may not be suppressed (it provides essential detail whose omission would modify the inherent character of the relationship).

A tool may use a lighter line for qualifier rectangles than for class rectangles to distinguish them clearly.

3.45.4 Style Guidelines

The qualifier rectangle should be smaller than the attached class rectangle, although this is not always practical.

3.45.5 Example

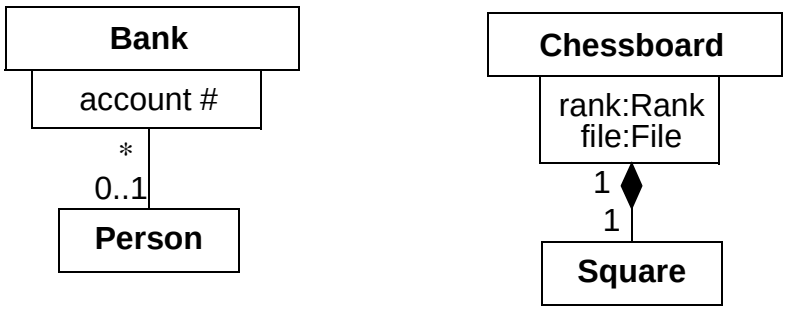


Figure 3-42 Qualified Associations

3.45.6 Mapping

The presence of a qualifier box on an end of an association path maps into a list of qualifier attributes on the corresponding Association Role. Each attribute entry string inside the qualifier box maps into an Attribute.

3.46 Association Class

3.46.1 Semantics

An association class is an association that also has class properties (or a class that has association properties). Even though it is drawn as an association and a class, it is really just a single model element.

3.46.2 Notation

An association class is shown as a class symbol (rectangle) attached by a dashed line to an association path. The name in the class symbol and the name string attached to the association path are redundant and should be the same. The association path may have the usual adornments on either end. The class symbol may have the usual contents. There are no adornments on the dashed line.

3.46.3 Presentation Options

The class symbol may be suppressed. It provides subordinate detail whose omission does not change the overall relationship. The association path may not be suppressed.

3.46.4 Style Guidelines

The attachment point should not be near enough to either end of the path that it appears to be attached to, the end of the path, or to any of the association end adornments.

Note that the association path and the association class are a single model element and have a single name. The name can be shown on the path, the class symbol, or both. If an association class has only attributes, but no operations or other associations, then the name may be displayed on the association path and omitted from the association class symbol to emphasize its “association nature.” If it has operations and other associations, then the name may be omitted from the path and placed in the class rectangle to emphasize its “class nature.” In neither case are the actual semantics different.

3.46.5 Example

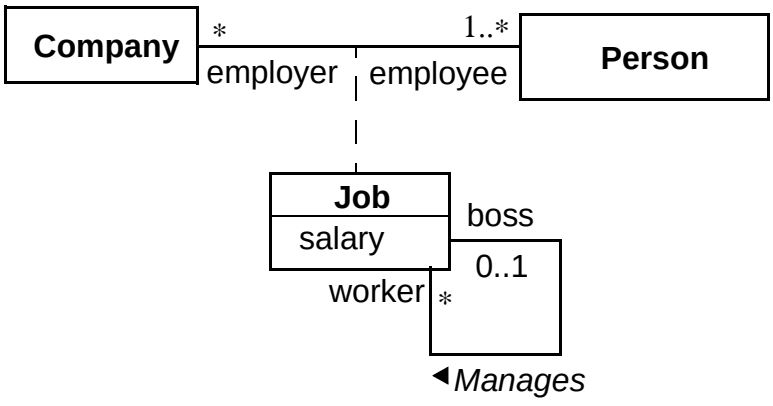


Figure 3-43 Association Class

3.46.6 Mapping

An association path connecting two class boxes connected by a dashed line to another class box maps into a single AssociationClass element. The name of the AssociationClass element is taken from the association path, the attached class box, or both (they must be consistent if both are present). The Association properties map from the association path, as specified previously. The Class properties map from the class box, as specified previously. Any constraints or properties placed on either the association path or attached class box apply to the AssociationClass itself; they must not conflict.

3.47 N-ary Association

3.47.1 Semantics

An n-ary association is an association among three or more classifiers (a single classifier may appear more than once). Each instance of the association is an n-tuple of values from the respective classifier. A binary association is a special case with its own notation.

Multiplicity for n-ary associations may be specified, but is less obvious than binary multiplicity. The multiplicity on a role represents the potential number of instance tuples in the association when the other N-1 values are fixed.

An n-ary association may not contain the aggregation marker on any role.

3.47.2 Notation

An n-ary association is shown as a large diamond (that is, large compared to a terminator on a path) with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. Role adornments may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted.

An association class symbol may be attached to the diamond by a dashed line. This indicates an n-ary association that has attributes, operations, and/or associations.

3.47.3 Style Guidelines

Usually the lines are drawn from the points on the diamond or the midpoint of a side.

3.47.4 Example

This example shows the record of a team in each season with a particular goalkeeper. It is assumed that the goalkeeper might be traded during the season and can appear with different teams.

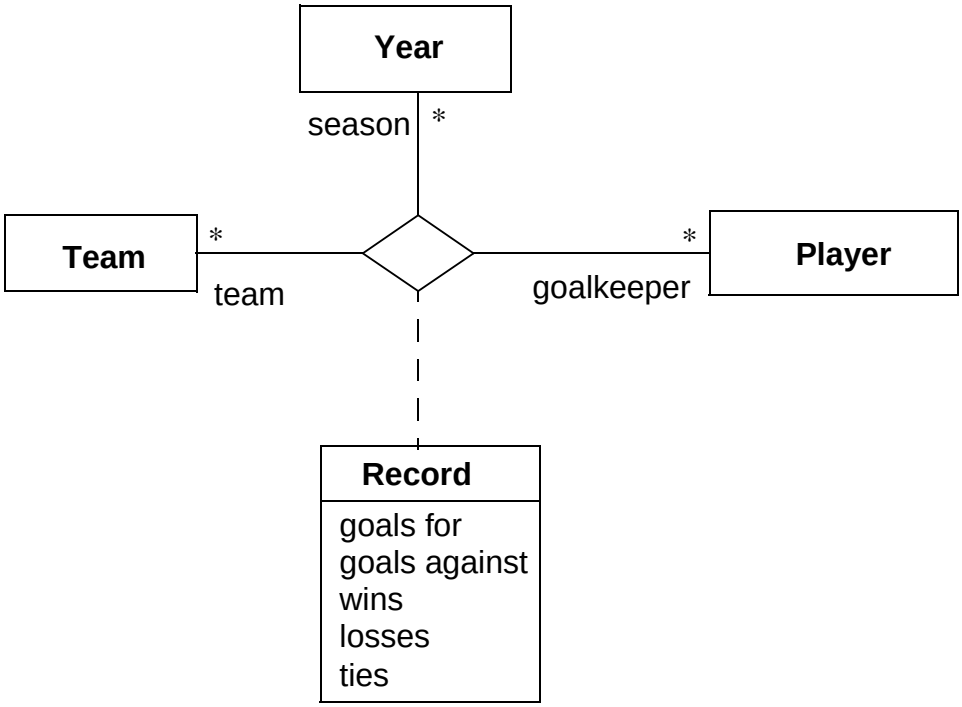


Figure 3-44 Ternary association that is also an association class

3.47.5 Mapping

A diamond attached to some number of class symbols by solid lines maps into an N-ary Association whose AssociationEnds are attached to the corresponding Classes. The ordering of the Classifiers in the Association is indeterminate from the diagram. If a class box is attached to the diamond by a dashed line, then the corresponding Classifier supplies the classifier properties for an N-ary AssociationClass.

3.48 Composition

3.48.1 Semantics

Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. The multiplicity of the aggregate end may not exceed one (it is unshared). See Section 3.43, “Association End,” on page 3-71 for further details.

The composite in a composition “projects” its identity onto the parts in the relationship. In other words, each part object in an object model can be identified with a unique composite object. It keeps its own identity as its primary identity. The point is that it can also be identified as being part of a unique composite. Composition is transitive. If object A is part of object B that is part of object C, then object A is also part of object C. A part may be identified with several composite objects in this way, each at a different level of detail.

The parts of a composition may include classes and associations (they may be formed into AssociationClasses if necessary). The meaning of an association in a composite object is that any tuple of objects connected by a single link must all belong to the *same* container object. In other words, the composite object projects its identity onto each link corresponding to the part end of a composition aggregation. If an association and two classes it relates are all related as parts to the same class as composite, a link that is an instance of the association is identified with an object that is an instance of the composite class; the objects connected by the link are also identified with the composite object; and they must all be associated with the same composite object.

3.48.2 Notation

Composition may be shown by a solid filled diamond as an association end adornment. Alternately, UML provides a graphically-nested form that is more convenient for showing composition in many cases.

Instead of using binary association paths using the composition aggregation adornment, composition may be shown by graphical nesting of the symbols of the elements for the parts within the symbol of the element for the whole. A nested class-like element may have a multiplicity within its composite element. The multiplicity is shown in the upper right corner of the symbol for the part. If the multiplicity mark is omitted, then the default multiplicity is many. This represents its multiplicity as a part within the composite classifier. A nested element may have a rolename within the composition; the name is shown in front of its type in the syntax:

rolename ‘:’ *classname*

This represents its rolename within its composition association to the composite.

Alternately, composition is shown by a solid-filled diamond adornment on the end of an association path attached to the element for the whole. The multiplicity may be shown in the normal way.

Note that attributes are, in effect, composition relationships between a classifier and the classifiers of its attributes.

An association drawn entirely within a border of the composite is considered to be part of the composition. Any instances on a single link of it must be from the same composite. An association drawn such that its path breaks the border of the composite is not considered to be part of the composition. Any instances on a single link of it may be from the same or different composites.

Note that the notation for composition resembles the notation for collaboration. A composition may be thought of as a collaboration in which all of the participants are parts of a single composite object.

Note that nested notation is not the correct way to show a class declared within another class. Such a declared class is not a structural part of the enclosing class but merely has scope within the namespace of the enclosing class, which acts like a package toward the inner class. Such a namescope containment may be shown by placing a package symbol in the upper right corner of the class symbol. A tool can allow a user to click on the package symbol to open the set of elements declared within it. The “anchor notation” (a cross in a circle on the end of a line) may also be used on a line between two class boxes to show that the class with the anchor icon declares the class on the other end of the line.

3.48.3 Design Guidelines

Note that a class symbol is a composition of its attributes and operations. The class symbol may be thought of as an example of the composition nesting notation (with some special layout properties). However, attribute notation subordinates the attributes strongly within the class; therefore, it should be used when the structure and identity of the attribute objects themselves is unimportant outside the class.

3.48.4 Example

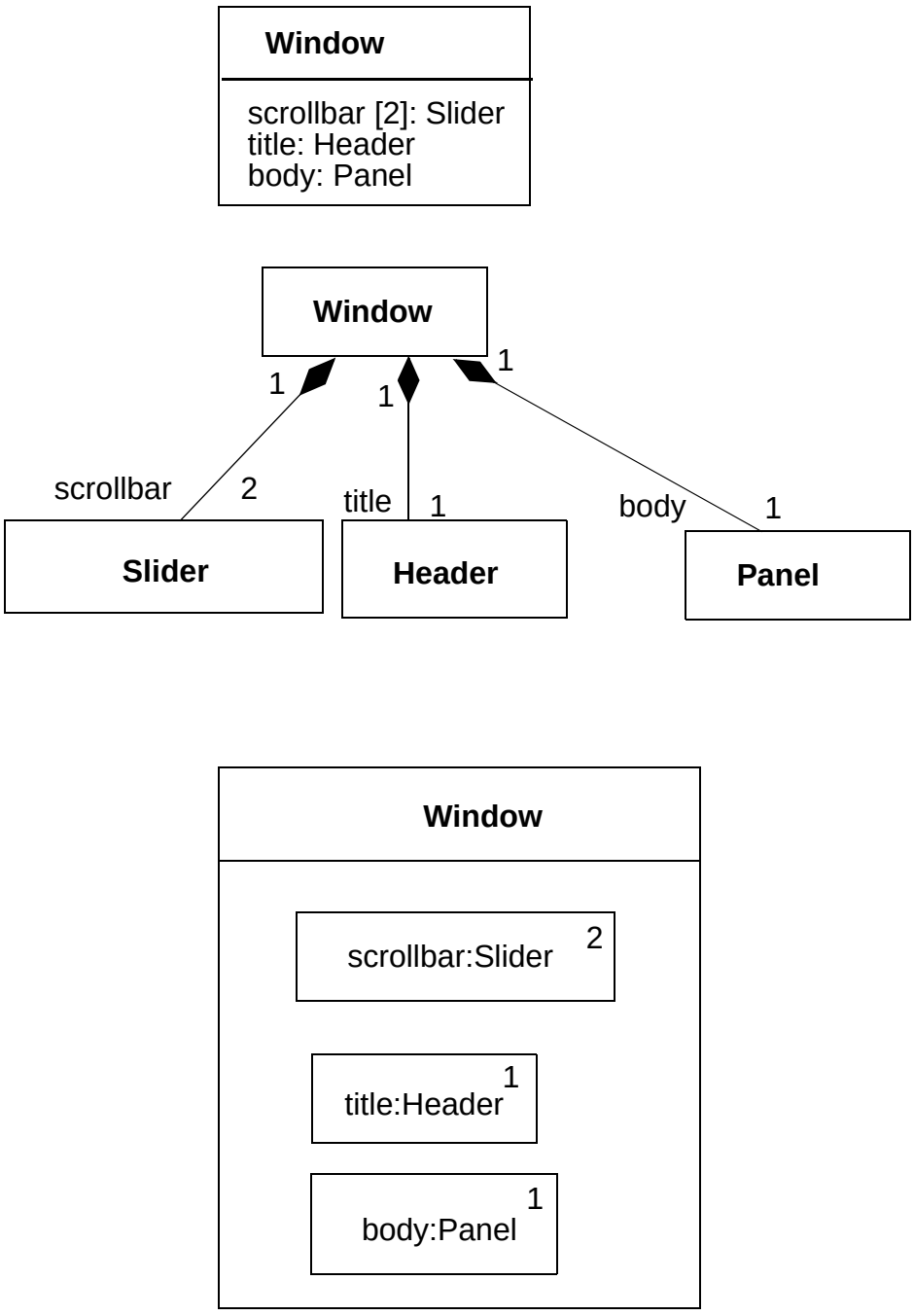


Figure 3-45 Different Ways to Show Composition

3.48.5 Mapping

A class box with an attribute compartment maps into a Class with Attributes. Although attributes may be semantically equivalent to composition on a deep level, the mapped model distinguishes the two forms.

A solid diamond on an association path maps into the aggregation-composition property on the corresponding Association Role.

A class box with contained class boxes maps into a set of composition associations; that is, one composition association between the Class corresponding to the outer class box and each of the Classes corresponding to the enclosed class boxes. The multiplicity of the composite end of each association is 1. The multiplicity of each constituent end is 1 if not specified explicitly; otherwise, it is the value specified in the corner of the class box *or* specified on an association path from the outer class box boundary to an inner class box.

3.49 Link

3.49.1 Semantics

A link is a tuple (list) of object references. Most commonly, it is a pair of object references. It is an instance of an association.

3.49.2 Notation

A binary link is shown as a path between two instances. In the case of a link from an instance to itself, it may involve a loop with a single instance. See “Association” on page 3-68 for details of paths.

A rolename may be shown at each end of the link. An association name may be shown near the path. If present, it is underlined to indicate an instance. Links do not have instance names, they take their identity from the instances that they relate. Multiplicity is *not* shown for links because they are instances. Other association adornments (aggregation, composition, navigation) may be shown on the link ends.

A qualifier may be shown on a link. The value of the qualifier may be shown in its box.

3.49.2.1 Implementation stereotypes

A stereotype may be attached to the link end to indicate various kinds of implementation. The following stereotypes may be used:

«association»	association (default, unnecessary to specify except for emphasis)
«parameter»	method parameter

«local»	local variable of a method
«global»	global variable
«self»	self link (the ability of an instance to send a message to itself)

3.49.2.2 N-ary link

An n-ary link is shown as a diamond with a path to each participating instance. The other adornments on the association, and the adornments on the association ends, have the same possibilities as the binary link.

3.49.3 Example

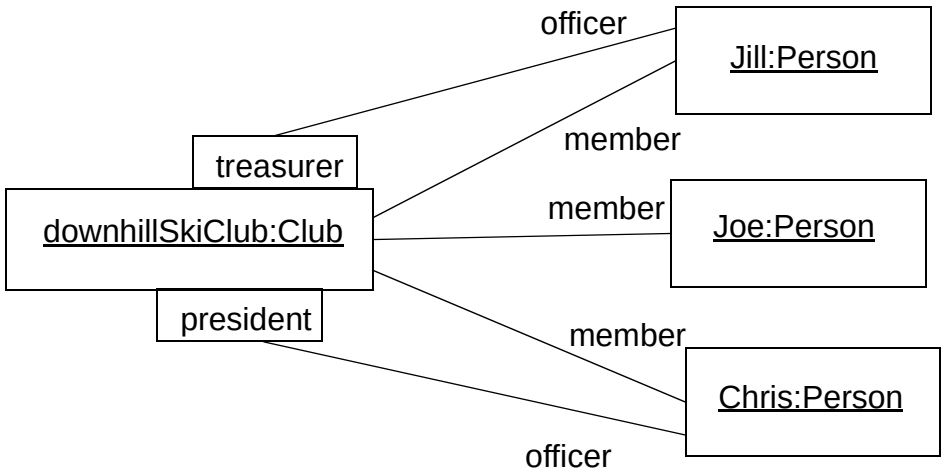


Figure 3-46 Links

3.49.4 Mapping

Within an object diagram, each link path maps to a Link between the Instances corresponding to the connected class boxes. If a name is placed on the link path, then it is an instance of the given Association (and the rolenames must match or the diagram is ill formed).

3.50 Generalization

3.50.1 Semantics

Generalization is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information. It is used for classes, packages, use cases, and other elements.

3.50.2 Notation

Generalization is shown as a solid-line path from the child (the more specific element, such as a subclass) to the parent (the more general element, such as a superclass), with a large hollow triangle at the end of the path where it meets the more general element.

A generalization path may have a text label called a discriminator that is the name of a partition of the children of the parent. The child is declared to be in the given partition. The absence of a discriminator label indicates the “empty string” discriminator which is a valid value (the “default” discriminator).

Generalization may be applied to associations as well as to classes. To notate generalization between associations, a generalization arrow may be drawn from a child association path to a parent association path. This notation may be confusing because lines connect other lines. An alternative notation is to represent each association as an association class and to draw the generalization arrow between the rectangles for the association classes, as with other classifiers. This approach can be used even if an association does not have any additional attributes, because a degenerate association class is a legal association.

The existence of additional children in the model that are not shown on a particular diagram may be shown using an ellipsis (. . .) in place of a child.

Note – This does not indicate that additional children may be added in the future. It indicates that additional children exist right now, but are not being seen. This is a notational convention that information has been suppressed, not a semantic statement.

Predefined constraints may be used to indicate semantic constraints among the children. A comma-separated list of keywords is placed in braces either near the shared triangle (if several paths share a single triangle) or near a dotted line that crosses all of the generalization lines involved. The following keywords (among others) may be used (the following constraints are predefined):

overlapping	An element may have two or more children from the set as ancestors. An instance may be a direct or indirect instance of two or more of the children.
disjoint	No element may have two children in the set as ancestors. No instance may be a direct or indirect instance of two of the children.
complete	All children have been specified (whether or not shown). No additional children are expected.
incomplete	Some children have been specified, but the list is known to be incomplete. There are additional children that are not yet in the model. This is a statement about the model itself. Note that this is not the same as the ellipsis, which states that additional children exist in the model but are not shown on the current diagram.

The *discriminator* must be unique among the attributes and association roles of the given parent. Multiple occurrences of the same discriminator name are permitted and indicate that the children belong to the same partition.

The use of multiple classification or dynamic classification affects the dynamic execution semantics of the language, but is not usually apparent from a static model.

3.50.3 Presentation Options

A group of generalization paths for a given parent may be shown as a tree with a shared segment (including the triangle) to the child, branching into multiple paths to each child.

If a text label is placed on a generalization triangle shared by several generalization paths to children, the label applies to all of the paths. In other words, all of the children share the given properties.

3.50.4 Example

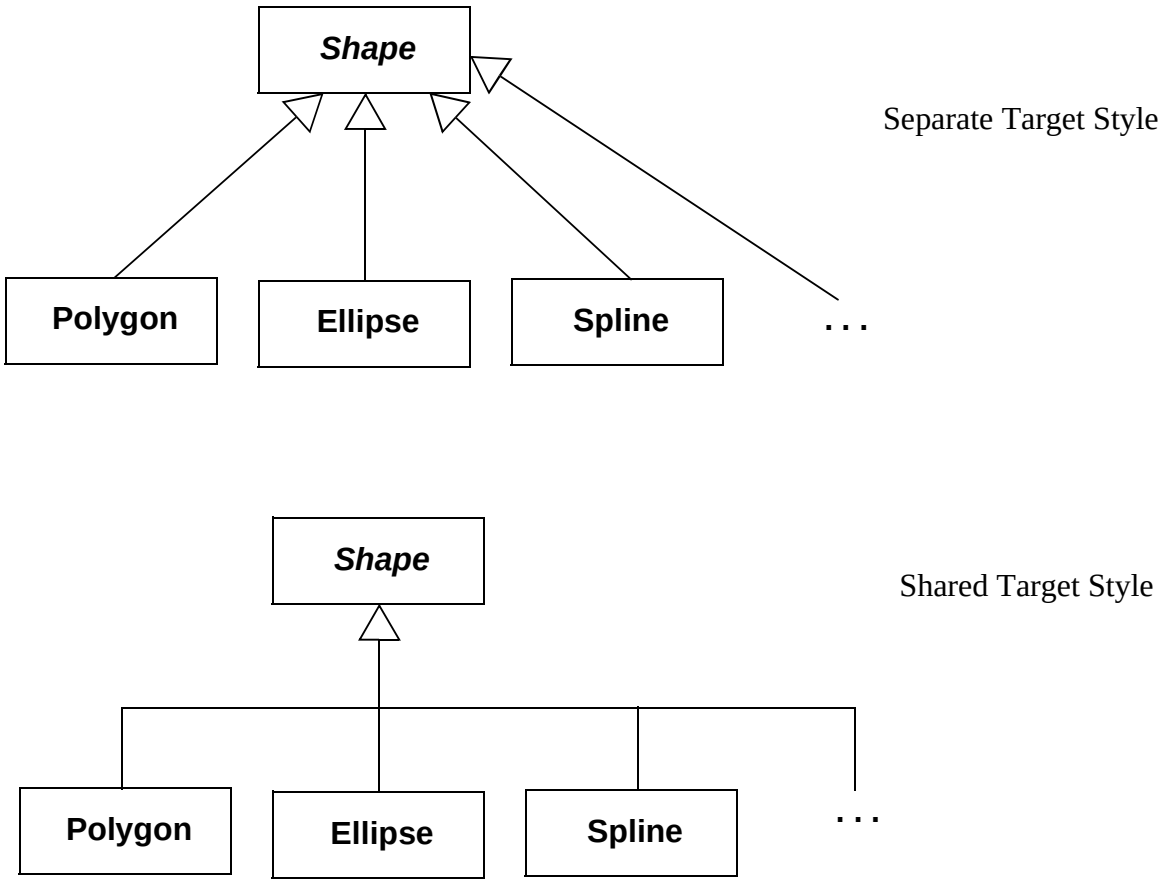


Figure 3-47 Styles of Displaying Generalizations

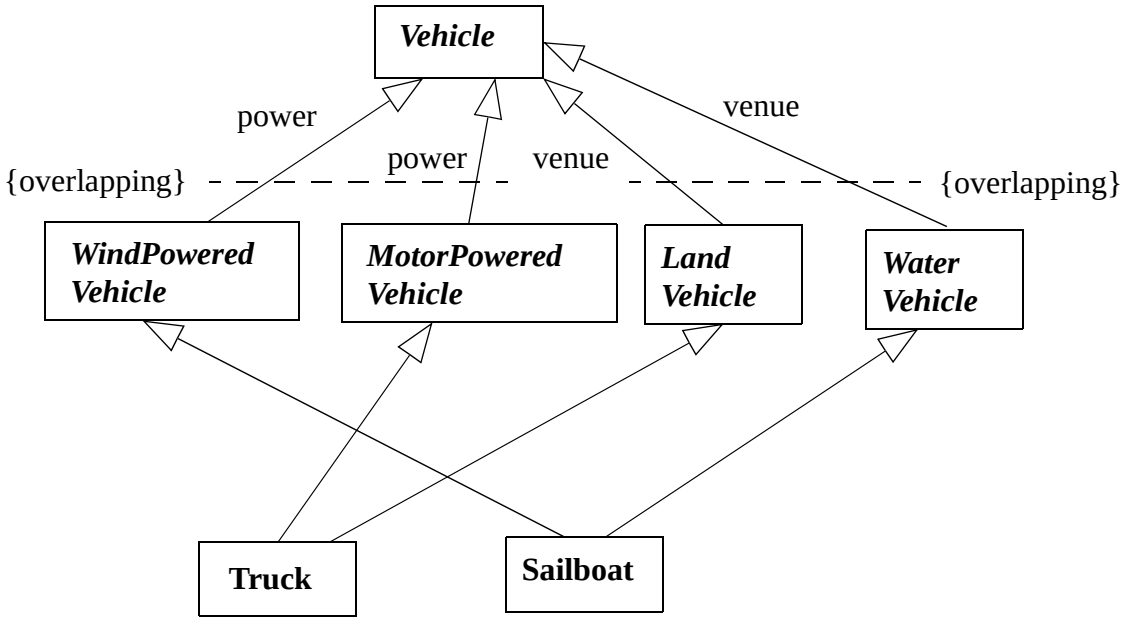


Figure 3-48 Generalization with Discriminators and Constraints, Separate Target Style

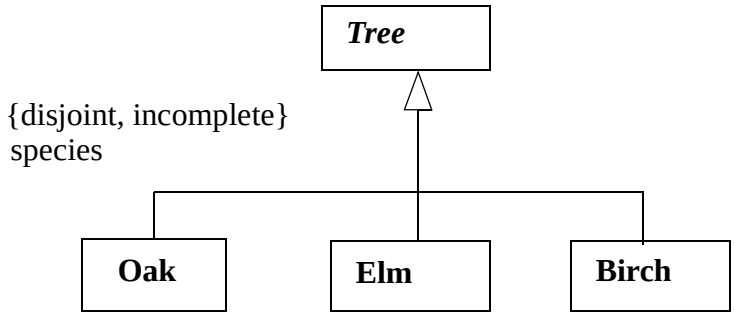


Figure 3-49 Generalization with Shared Target Style

3.50.5 Mapping

Each generalization path between two element symbols maps into a Generalization between the corresponding GeneralizableElements. A generalization tree with one arrowhead and many tails maps into a set of Generalizations, one between each

element corresponding to a symbol on a tail and the single GeneralizableElement corresponding to the symbol on the head. That is, a tree is semantically indistinguishable from a set of distinct arrows, it is purely a notational convenience.

Any property string attached to a generalization arrow applies to the Generalization. A property string attached to the head line segment on a generalization tree represents a (duplicated) property on each of the individual Generalizations.

The presence of an ellipsis ("...") as a child node of a given parent indicates that the semantic model contains at least one child of the given parent that is not visible on the current diagram. Normally, this indicator will be maintained automatically by an editing tool.

3.51 Dependency

3.51.1 Semantics

A dependency indicates a semantic relationship between two model elements (or two sets of model elements). It relates the model elements themselves and does not require a set of instances for its meaning. It indicates a situation in which a change to the target element may require a change to the source element in the dependency.

3.51.2 Notation

A dependency is shown as a dashed arrow between two model elements. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier). The arrow may be labeled with an optional stereotype and an optional individual name.

It is possible to have a set of elements for the client or supplier. In this case, one or more arrows with their tails on the clients are connected to the tails of one or more arrows with their heads on the suppliers. A small dot can be placed on the junction if desired. A note on the dependency should be attached at the junction point.

The following kinds of Dependency are predefined and may be indicated with keywords. Note that some of these correspond to actual metamodel classes and others to stereotypes of metamodel classes. All of these are shown as dashed arrows with keywords in guillemets. The name column shows the name of the metamodel class or the informal name of the class with the given keyword stereotype.

Keyword	Name	Description
access	Access	The granting of permission for one package to reference the public elements owned by another package (subject to appropriate visibility). Maps into a Permission with the stereotype access.
bind	Binding	A binding of template parameters to actual values to create a nonparameterized element. See Section 3.31, “Bound Element,” on page 3-54 for more details. Maps into a Binding.
derive	Derivation	A computable relationship between one element and another (one more than one of each). Maps into an Abstraction with the stereotype derivation.
import	Import	The granting of permission for one package to reference the public elements of another package, together with adding the names of the public elements of the supplier package to the client package. Maps into a Permission with the stereotype import.
refine	Refinement	A historical or derivation connection between two elements with a mapping (not necessarily complete) between them. A description of the mapping may be attached to the dependency in a note. Various kinds of refinement have been proposed and can be indicated by further stereotyping. Maps into an Abstraction with the stereotype refinement.
trace	Trace	A historical connection between two elements that represents the same concept at different levels of meaning. Maps into an Abstraction with the stereotype trace.
use	Usage	A situation in which one element requires the presence of another element for its correct implementation or functioning. May be stereotyped further to indicate the exact nature of the dependency, such as calling an operation of another class, granting permission for access, instantiating an object of another class, etc. Maps into a Usage. If the keyword is one of the stereotypes of Usage (call, create, instantiate, send), then it maps into a Usage with the given stereotype.

3.51.3 Presentation Options

Note – The connection between a note or constraint and the element it applies to is shown by a dashed line without an arrowhead. This is not a Dependency.

3.51.4 Example

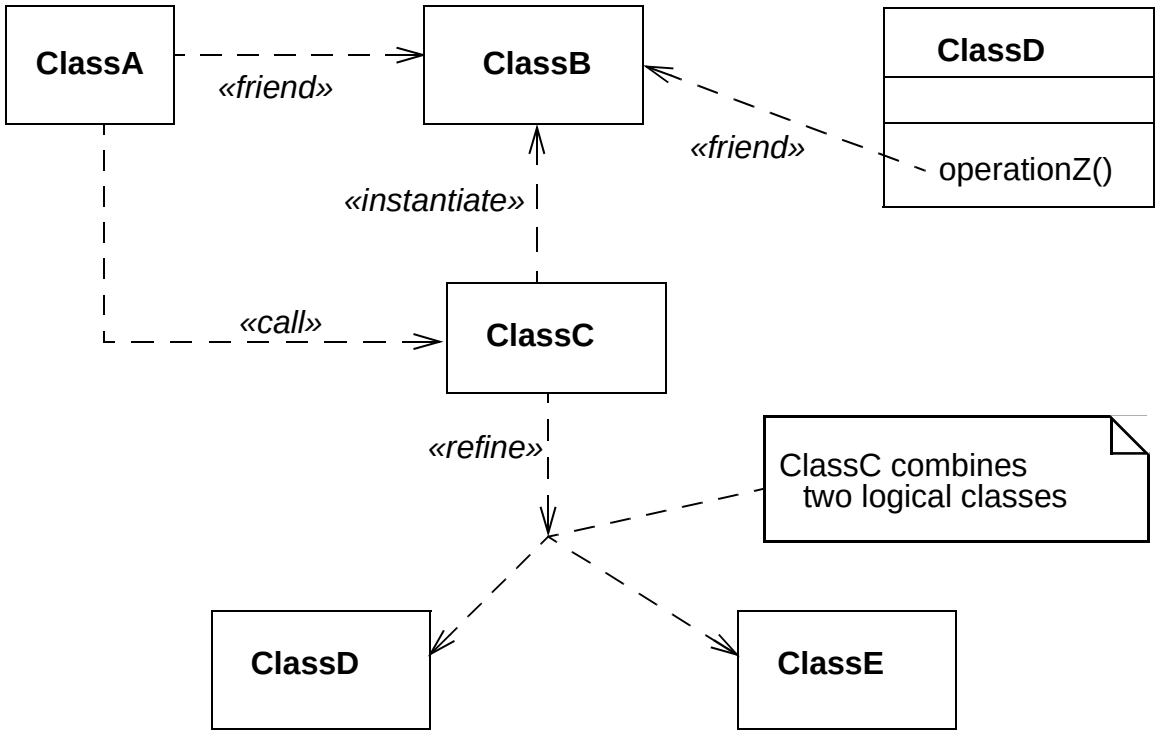


Figure 3-50 Various Dependencies Among Classes

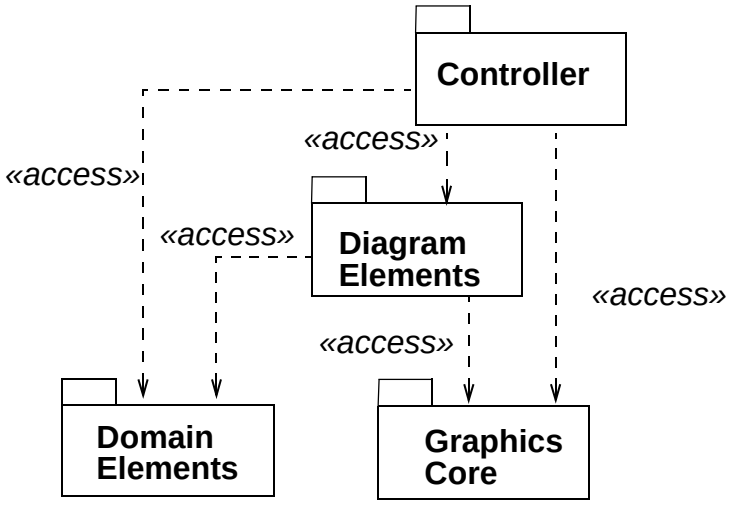


Figure 3-51 Dependencies Among Packages

3.51.5 Mapping

A dashed arrow maps into the appropriate kind of Dependency (based on keywords) between the Elements corresponding to the symbols attached to the ends of the arrow. The stereotype and the name (if any) attached to the arrow are the stereotype and name of the Dependency.

3.52 Derived Element

3.52.1 Semantics

A derived element is one that can be computed from another one, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

3.52.2 Notation

A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a rolename.

3.52.3 Style Guidelines

The details of computing a derived element can be specified by a dependency with the stereotype «derive». Usually it is convenient in the notation to suppress the dependency arrow and simply place a constraint string near the derived element, although the arrow can be included when it is helpful.

3.53 InstanceOf

3.53.1 Semantics

Shows the connection between an instance and its classifier.

3.53.2 Notation

Shown as a dashed arrow with its tail on the instance and its head on the classifier. The arrow has the keyword «instanceOf».

3.53.3 Mapping

Maps into an instance relationship from the instance to the classifier.

Part 6 - Use Case Diagrams

A use case diagram shows the relationship among use cases within a system or other semantic entity and their actors.

3.54 Use Case Diagram

3.54.1 Semantics

Use case diagrams show actors and use cases together with their relationships. The use cases represent functionality of a system or a classifier, like a subsystem or a class, as manifested to external interactors with the system or the classifier.

3.54.2 Notation

A use case diagram is a graph of actors, a set of use cases, possibly some interfaces, and the relationships between these elements. The relationships are associations between the actors and the use cases, generalizations between the actors, and generalizations, extends, and includes among the use cases. The use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system or classifier.

3.54.3 Example

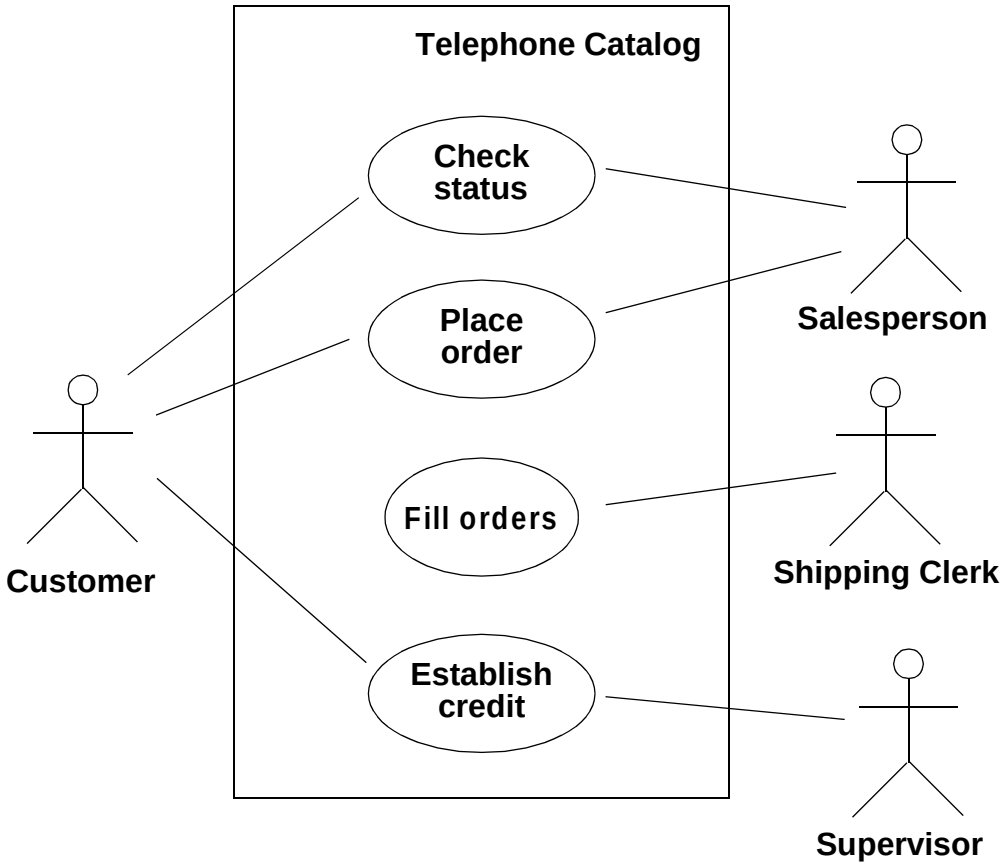


Figure 3-52 Use Case Diagram

3.54.4 Mapping

A set of use case ellipses with connections to actor symbols maps to a set of UseCases and Actors corresponding to the use case and actor symbols, respectively. The optional rectangle maps onto either a Model with the stereotype «useCaseModel» containing the set of UseCases and Actors, or to a Classifier, like Subsystem or Class, containing the set of UseCases. An interface in the diagram is mapped onto an Interface in the Model, and the connection between the interface and the actor or use case icons is mapped onto a realization Dependency (an Abstraction dependency being stereotyped «realize») between the Classifiers. Each generalization arrow maps onto a Generalization in the model, and each line between an actor symbol and a use case ellipse maps to an Association between the corresponding Classifiers. A dashed arrow with the keyword «include» or «extend» maps to an Include or Extend relationship between UseCases.

3.55 Use Case

3.55.1 Semantics

A *use case* is a kind of classifier representing a coherent unit of functionality provided by a system, a subsystem, or a class as manifested by sequences of messages exchanged among the system (subsystem, class) and one or more outside interactors (called *actors*) together with actions performed by the system (subsystem, class).

An *extension point* is a reference to one location within a use case at which action sequences from other use cases may be inserted. Each extension point has a unique name within a use case, and a description of the location within the behavior of the use case.

3.55.2 Notation

A use case is shown as an ellipse containing the name of the use case. An optional stereotype keyword may be placed above the name and a list of properties included below the name. As a classifier, a use case may also have compartments displaying attributes and operations.

Extension points may be listed in a compartment of the use case with the heading **extension points**. The description of the locations of the extension point is given in a suitable form, usually as ordinary text, but can also be given in other forms, like the name of a state in a state machine, or a precondition or a postcondition.

The behavior of a use case can be described in several different ways, depending on what is convenient. Plain text is used often, but state machines, operations, and methods are examples of other ways of describing the behavior of the use case. Sequence diagrams can be used for describing the interaction between use cases and their actors.

3.55.3 Presentation Options

The name of the use case may be placed below the ellipse. The name of an abstract use case may be shown in italics.

The ellipse may contain or suppress compartments presenting the attributes, the operations, and the extension points of the use case.

3.55.4 Style Guidelines

Use case names should follow capitalization and punctuation guidelines used for Classifiers in the model.

3.55.5 Mapping

A use case symbol maps to a UseCase with the given name. An extension point maps into an ExtensionPoint within the UseCase.

3.56 Actor

3.56.1 Semantics

An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.

3.56.2 Notation

The standard stereotype icon for an actor is a “stick man” figure with the name of the actor below the figure.

3.56.3 Presentation Options

An actor may also be shown as a class rectangle with the keyword «actor», with the usual notation for all compartments. Other icons that convey the kind of actor may also be used to denote an actor.

3.56.4 Style Guidelines

Actor names should follow capitalization and punctuation guidelines used for types and classes in the model.

3.56.5 Mapping

An actor symbol maps to an Actor with the given name. The names of abstract actors may be shown in italics.

3.57 Use Case Relationships

3.57.1 Semantics

There are several standard relationships among use cases or between actors and use cases.

- Association – The participation of an actor in a use case; that is, instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.

- Extend – An extend relationship from use case A to use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behavior specified by A. The behavior is inserted at the location defined by the extension point in B, which is referenced by the extend relationship.
- Generalization – A generalization from use case C to use case D indicates that C is a specialization of D.
- Include – An include relationship from use case E to use case F indicates that an instance of the use case E will also contain the behavior as specified by F. The behavior is included at the location which defined in E.

3.57.2 Notation

An association between an actor and a use case is shown as a solid line between the actor and the use case. It may have end adornments such as multiplicity.

An extend relationship between use cases is shown by a dashed arrow with an open arrow-head from the use case providing the extension to the base use case. The arrow is labeled with the keyword «extend». The condition of the relationship is optionally presented close to the key-word.

An include relationship between use cases is shown by a dashed arrow with an open arrow-head from the base use case to the included use case. The arrow is labeled with the keyword «include».

A generalization between use cases is shown by a generalization arrow; that is, a solid line with a closed, hollow arrow head pointing at the parent use case.

The relationship between a use case and its external interaction sequences is usually defined by an invisible hyperlink to sequence diagrams. The relationship between a use case and its realization may be shown as dashed arrow with the keyword «representedClassifier» to collaborations, but may also be defined as invisible hyperlinks.

3.57.3 Example

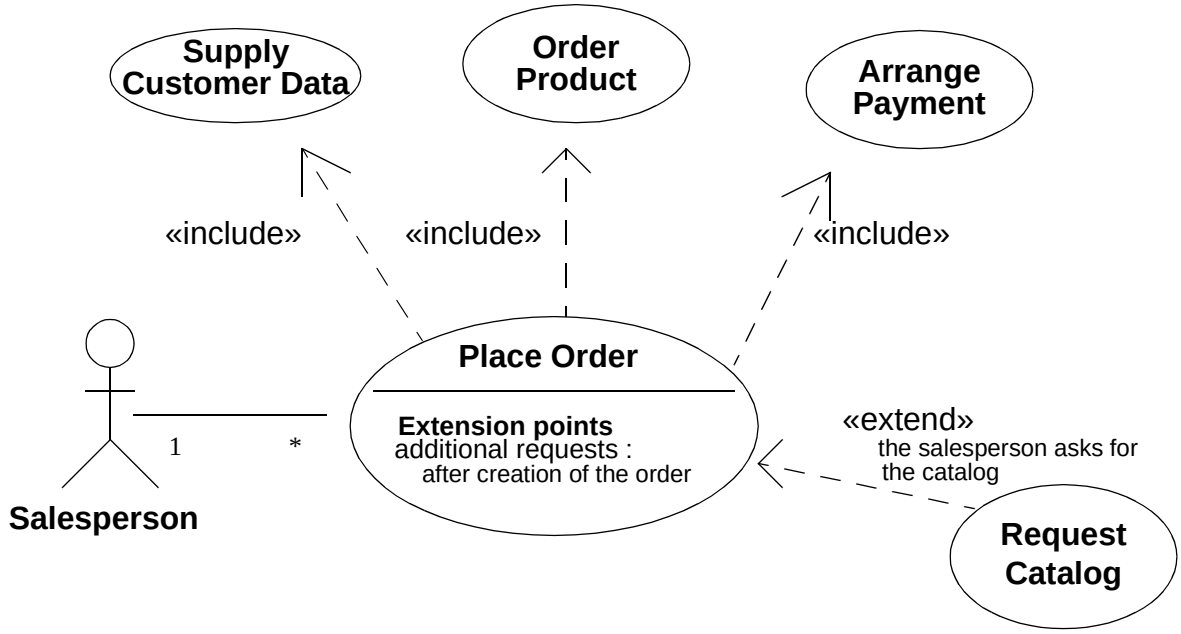


Figure 3-53 Use Case Relationships

3.57.4 Mapping

A path between use case and/or actor symbols maps into the corresponding relationship between the corresponding Elements, as described above.

3.58 Actor Relationships

3.58.1 Semantics

There is one standard relationship among actors and one between actors and use cases.

- Association – The participation of an actor in a use case; that is, instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.
- Generalization – A generalization from an actor A to an actor B indicates that an instance of A can communicate with the same kinds of use-case instances as an instance of B.

3.58.2 Notation

An association between an actor and a use case is shown as a solid line between the actor and the use case.

A generalization between actors is shown by a generalization arrow; that is, a solid line with a closed, hollow arrow head. The arrow head points at the more general actor.

3.58.3 Example

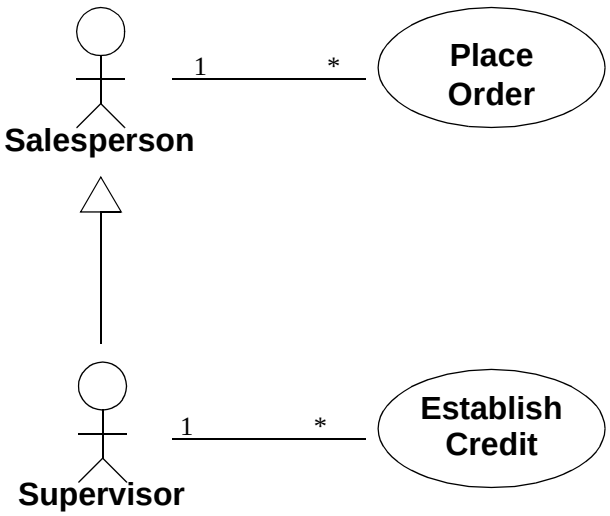


Figure 3-54 Actor Relationships

3.58.4 Mapping

A generalization between two actor symbols and an association between actor symbol and a use case symbol maps into the corresponding relationship between the corresponding Elements, as described above.

Part 7 - Interaction Diagrams

The description of behavior involves two aspects: 1) the structural description of the participants and 2) the description of the communication patterns. The structure of Instances playing roles in a behavior and their relationships is called a *Collaboration*. The communication pattern performed by Instances playing the roles to accomplish a specific purpose is called an *Interaction*. The two aspects of behavior are often described together on a single diagram, but at times it is useful to describe the structural aspects separately.

Interaction diagrams come in two forms based on the same underlying information, specified by a Collaboration and possibly by an Interaction, but each form emphasizes a particular aspect of it. The two forms are *sequence diagrams* and *collaboration diagrams*. A sequence diagram shows the explicit sequence of communications and is better for real-time specifications and for complex scenarios. A collaboration diagram shows an Interaction organized around the roles in the Interaction and their

relationships. It does not show time as a separate dimension, so the sequence of communications and the concurrent threads must be determined using sequence numbers.

3.59 Collaboration

3.59.1 Semantics

Behavior is implemented by ensembles of Instances that exchange Stimuli within an overall interaction to accomplish a task. To understand the mechanisms used in a design, it is important to see only those Instances and their cooperation involved in accomplishing a purpose or a related set of purposes, projected from the larger system of which they are part of. Such a static construct is called a *Collaboration*.

A Collaboration includes an ensemble of ClassifierRoles and AssociationRoles that define the participants needed for a given set of purposes. Instances conforming to the ClassifierRoles play the roles defined by the ClassifierRoles, while Links between the Instances conform to AssociationRoles of the Collaboration. ClassifierRoles and AssociationRoles define a usage of Instances and Links, and the Classifiers and Associations declare all required properties of these Instances and Links.

An *Interaction* is defined in the context of a Collaboration. It specifies the communication patterns between the roles in the Collaboration. More precisely, it contains a set of partially ordered *Messages*, each specifying one communication; for example, what Signal to be sent or what Operation to be invoked, as well as the roles to be played by the sender and the receiver, respectively.

A *CollaborationInstanceSet* references an ensemble of Instances that jointly perform the task specified by the CollaborationInstanceSet's Collaboration. These Instances play the roles defined by the ClassifierRoles of the Collaboration; that is, the Instances have all the properties declared by the ClassifierRoles (the Instances are said to *conform to* the ClassifierRoles). The Stimuli sent between the Instances when performing the task are participating in the *InteractionInstanceSet* of the CollaborationInstanceSet. These Stimuli conform to the Messages in one of the Interactions of the Collaboration. Since an Instance can participate in several CollaborationInstanceSets at the same time, all its communications are not necessarily referenced by only one InteractionInstanceSet. They can be interleaved.

A Collaboration may be attached to an Operation or a Classifier, like a UseCase, to describe the context in which their behavior occurs; that is, what roles Instances play to perform the behavior specified by the Operation or the UseCase. A Collaboration is used for describing the realization of an Operation or a Classifier. A Collaboration that describes a Classifier, like a UseCase, references Classifiers and Associations in general, while a Collaboration describing an Operation includes the arguments and local variables of the Operation, as well as ordinary Associations attached to the Classifier owning the Operation. The Interactions defined within the Collaboration specify the communication pattern between the Instances when they perform the behavior specified in the Operation or the UseCase. These patterns are presented in

sequence diagrams or collaboration diagrams. A Collaboration may also be attached to a Class to define the static structure of the Class; that is, how attributes, parameters, etc. cooperate with each other.

A parameterized Collaboration represents a design construct that can be used repeatedly in different designs. The participants in the Collaboration, including the Classifiers and Relationships, can be parameters of the generic Collaboration. The parameters are bound to particular ModelElements in each instantiation of the generic Collaboration. Such a parameterized Collaboration can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects). Whereas most Collaborations can be anonymous because they are attached to a named ModelElement, Collaboration patterns are free standing design constructs that must have names.

A Collaboration may be expressed at different levels of granularity. A coarse-grained Collaboration may be refined to produce another Collaboration that has a finer granularity.

3.60 Sequence Diagram

3.60.1 Semantics

A sequence diagram presents an Interaction, which is a set of Messages between ClassifierRoles within a Collaboration, or an InteractionInstanceSet, which is a set of Stimuli between Instances within a CollaborationInstanceSet to effect a desired operation or result.

3.60.2 Notation

A sequence diagram has two dimensions: the vertical dimension represents time, and the horizontal dimension represents different instances. Normally time proceeds down the page. (The dimensions may be reversed, if desired.) Usually only time sequences are important, but in real-time applications the time axis could be an actual metric. There is no significance to the horizontal ordering of the instances.

The different kinds of arrows used in sequence diagrams are described in Section 3.63, “Message and Stimulus,” on page 3-111. These are the same kinds as in collaboration diagrams; see Section 3.65, “Collaboration Diagram,” on page 3-114.

Note that much of this notation is drawn directly from the Object Message Sequence Chart notation of Buschmann, Meunier, Rohnert, Sommerlad, and Stal, which is itself derived with modifications from the Message Sequence Chart notation.

3.60.3 Presentation Options

The horizontal ordering of the lifelines is arbitrary. Often call arrows are arranged to proceed in one direction across the page; however, this is not always possible and the ordering does not convey information.

The axes can be interchanged, so that time proceeds horizontally to the right and different objects are shown as horizontal lines.

Various labels (such as timing constraints, descriptions of actions during an activation, and so on) can be shown either in the margin or near the transitions or activations that they label.

Timing constraints may be expressed using time expressions on message or stimuli names. The functions *sendTime* (the time at which a stimulus is sent by an instance) and *receiveTime* (the time at which a stimulus is received by an instance) may be applied to stimuli names to yield a time. The set of time functions is open-ended, so that users can invent new ones as needed for special situations or implementation distinctions (such as *elapsedTime*, *executionStartTime*, *queuedTime*, *handledTime*, etc.)

Construction marks of the kind found in blueprints can be used to indicate a time interval to which a constraint may be attached (see bottom right of Figure 3-55 on page 3-104). This notation is visually appealing but it is ambiguous if the arrow is horizontal, because the send time and the receive time cannot be distinguished. In many cases the transmission time is negligible, so the ambiguity is harmless, but a tool must nevertheless map such a notation unambiguously to an expression on message or stimuli names (as shown in the examples in the left of the diagram) before the information is placed in the semantic model. (A tool may adopt defaults for this mapping.) Similarly, a tool might permit the time function to be elided and use the stimulus name to denote the time of stimulus sending or receipt within a timing expression (such as “b.receiveTime - a.sendTime < 1 sec.” in Figure 3-55), but again this is only a surface notation that must be mapped to a proper time expression in the semantic model).

3.60.4 Example

Simple sequence diagram with concurrent objects.

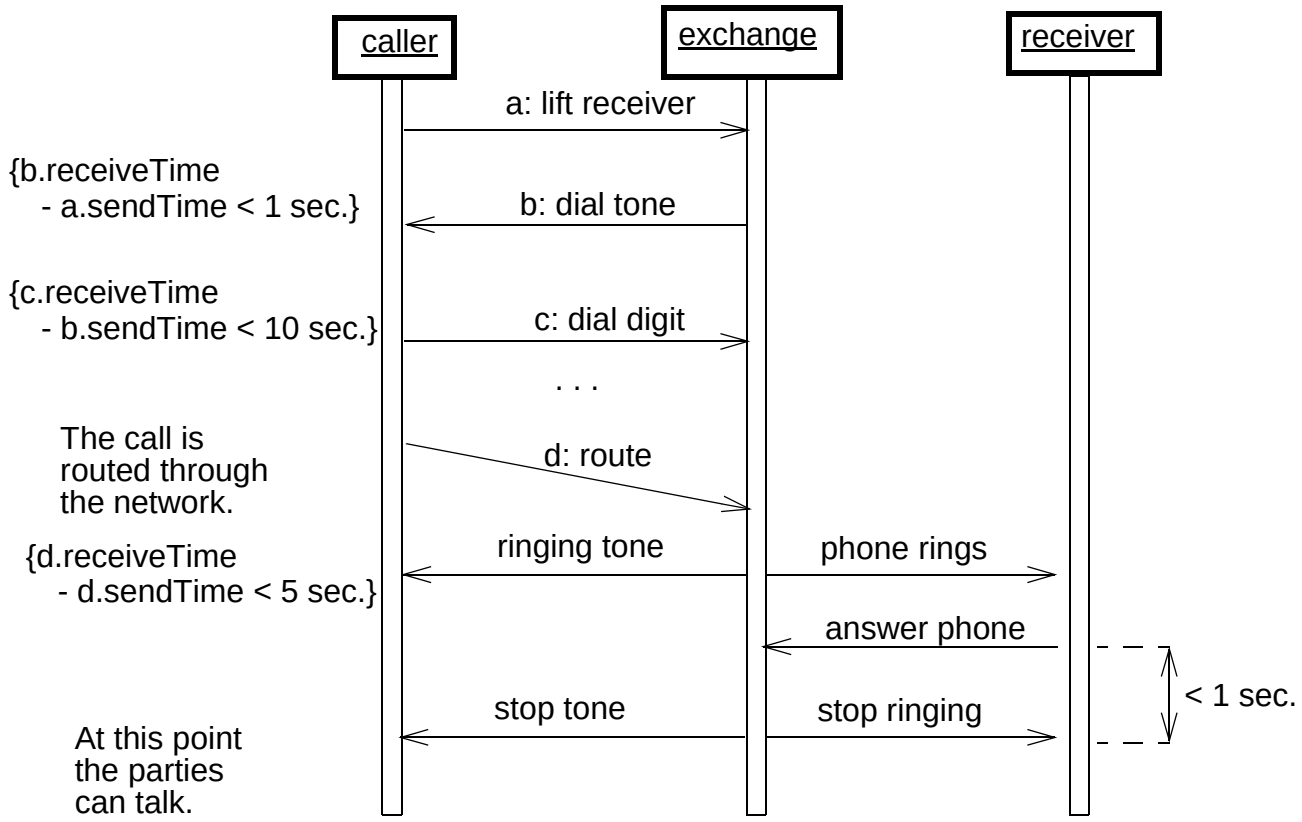


Figure 3-55 Simple Sequence Diagram with Concurrent Objects (denoted by boxes with thick borders).

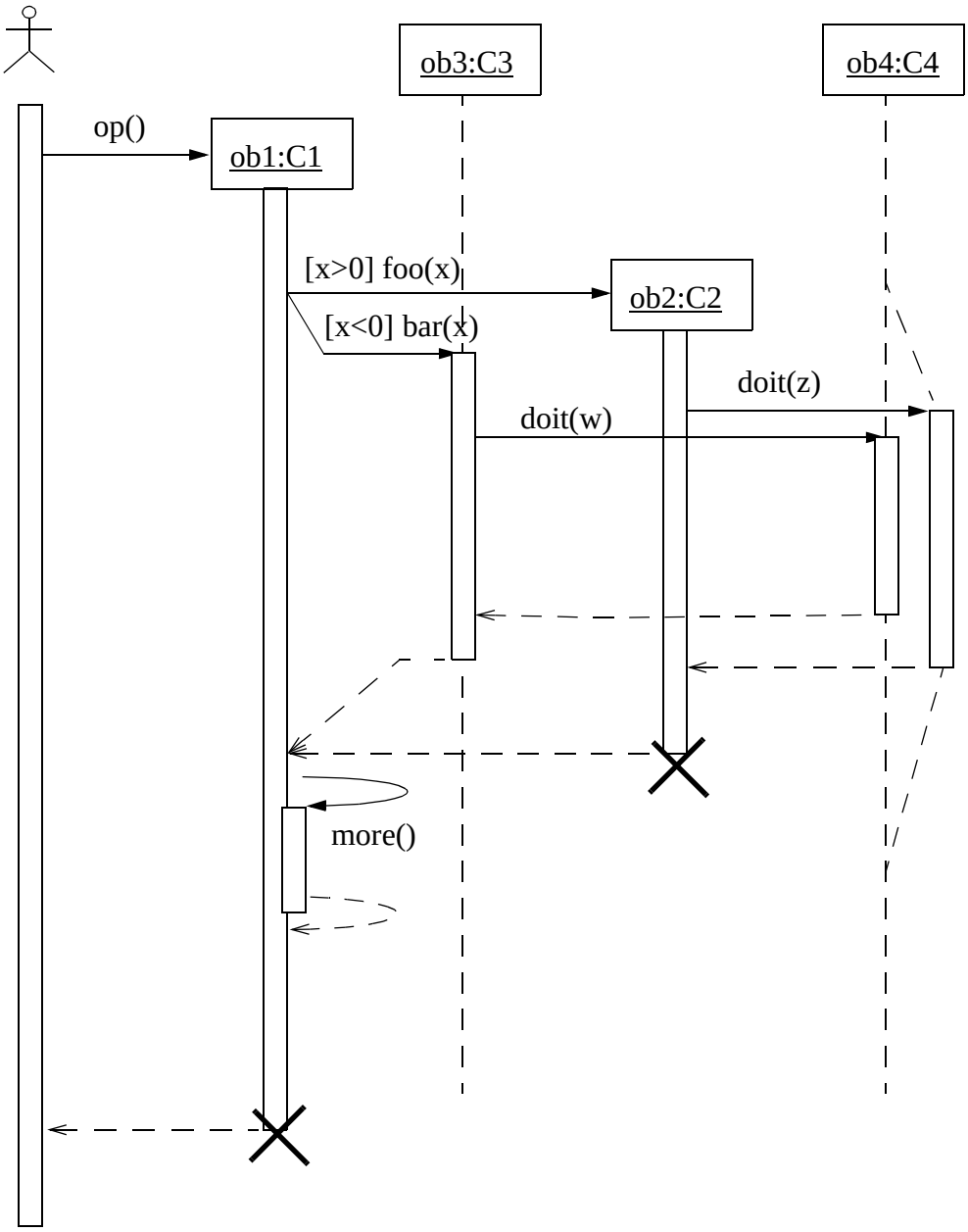


Figure 3-56 Sequence Diagram with Focus of Control, Conditional, Recursion, Creation, and Destruction.

3.60.5 Mapping

This section summarizes the mapping for the sequence diagram and the elements within it, some of which are described in subsequent sections.

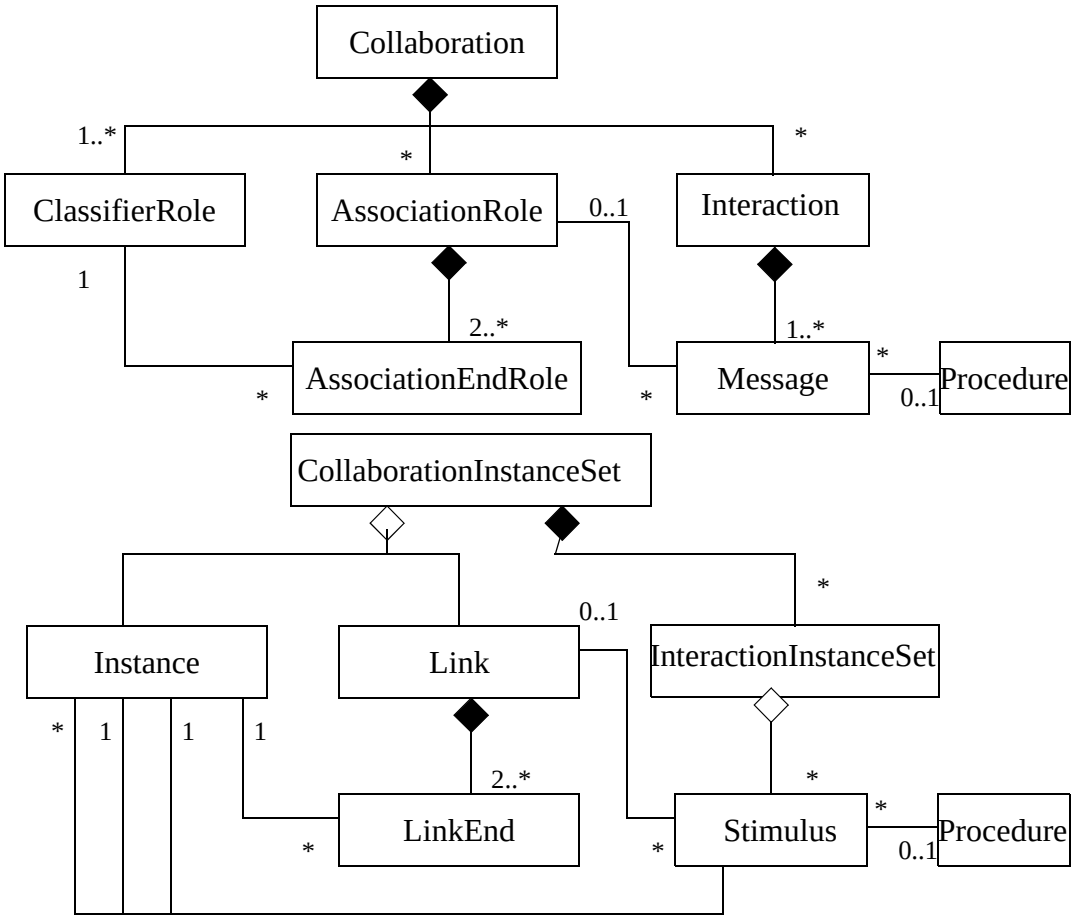


Figure 3-57 A summary of the UML constructs used in the section below.

3.60.5.1 Sequence diagram

A sequence diagram maps into an Interaction and an underlying Collaboration or an InteractionInstanceSet and an underlying CollaborationInstanceSet depending on whether the diagram shows Instances or ClassifierRoles. An Interaction specifies a sequence of communications; it contains a collection of partially ordered Messages, each specifying a communication between a sender role and a receiver role. A CollaborationInstanceSet references a collection of Instances that conform to the ClassifierRoles in the Collaboration owning the Interaction. These Instances communicate by dispatching Stimuli that conform to the Messages in the Interaction. The CollaborationInstanceSet has an InteractionInstanceSet that references these Stimuli. A sequence diagram presents either a collection of object symbols and arrows

mapping to Instances and Stimuli, or a collection of classifier-role symbols and arrows mapping to ClassifierRoles and Messages. The Instances and Stimuli conform to the ClassifierRoles and Messages.

The sequence diagram presents either a Collaboration or a CollaborationInstanceSet. In the former case, the classifier box with its lifeline maps onto a ClassifierRole in the Collaboration, and the arrows map onto the Messages in one of the Collaboration's Interactions. The name strings in the boxes map onto the names of the ClassifierRoles, while the classifier names map onto the ClassifierRole's *base* Classifiers. The AssociationRoles among the ClassifierRoles are not shown on the sequence diagram. They must be obtained in the model from a complementary collaboration diagram or other means.

If the sequence diagram presents a CollaborationInstanceSet, each object box with its lifeline maps into an Instance, which conforms to a ClassifierRole in the CollaborationInstanceSet's Collaboration. The name field maps into the name of the Instance, the role name into the ClassifierRole's name, and the class field maps into the names of the Classifiers being the *base* Classifiers of the ClassifierRole. An arrow maps into a Stimulus connected to two Instances: the sender and the receiver. The Link used for the communication of the Stimulus plays the role specified by the AssociationRole connected to the Message. Unless the correct Link can be determined from a complementary collaboration diagram or other means, the Stimulus is either not attached to a Link (not a complete model), or it is attached to an arbitrary Link or to a dummy Link between the Instances conforming to the AssociationRole implied by the two ClassifierRoles due to the lack of complete information.

The label of the arrow is mapped into either the body attribute of the Procedure, or into a detailed action model. For the action model, the name of the Operation to be invoked or Signal to be sent is mapped onto the name of the Operation or Signal invoked by the actions in the Procedure connected to the Message. Different alternatives exist for showing the arguments of the Stimulus. If references to the actual Instances being passed as arguments are shown, these are mapped onto the arguments of the Stimulus. If the argument expressions are shown instead, and a detailed action model is used, then these are mapped into CodeActions in the Procedure, or additional actions that compute the values of the expressions. Finally, if the types of the arguments are shown together with the name of the Operation or the Signal, these are mapped onto the parameter types of the Operation or the Attribute types of the Signal, respectively. A timing label placed on the level of an arrow endpoint maps into the name of the corresponding Message or Stimulus. A constraint placed on the diagram maps into a Constraint on the entire Interaction.

An arrow with the arrowhead pointing to an object symbol or role symbol within the frame of the diagram maps into a Stimulus (Message) dispatched by a CreateObjectAction. The interpretation is that an Instance is created by dispatching the Stimulus. If the target of the arrow is a classifier-role symbol, the Instance will conform to the ClassifierRole. (Note, that the diagram does not necessarily show from which Classifier the Instance originates; only that the newly created Instance conform to the ClassifierRole.) After the creation of the Instance, it may immediately start interacting with other Instances. This implies that the creation method (constructor, initializer) of the Instance dispatches these Stimuli. If an object termination symbol

("X") is the target of an arrow, the arrow maps into a Stimulus that will cause the receiving Instance to be removed. If the object termination symbol appears in the diagram without an incoming arrow, it maps into a Procedure containing a DestroyObjectAction.

The order of the arrows in the diagram maps onto pairs of associations between the Stimuli (Messages). A *predecessor* relationship is established between Stimuli (Messages) corresponding to successive arrows in the vertical sequence. In case of concurrent arrows preceding an arrow, the corresponding Stimulus (Message) has a collection of predecessors. Moreover, each Stimulus (Message) has an *activator* association to the Stimulus (Message) corresponding to the incoming arrow of the activation.

Procedural sequence diagram

On a procedural sequence diagram (one with focus of control and calls), subsequent arrows on the same lifeline map into Stimuli (Messages) obeying the *predecessor* association. An arrow to the head of a focus of control region establishes a nested activation. The arrow maps into a Stimulus (Message) with the dispatching Procedure containing a CallOperationAction. The Stimulus holds the sender and receiver Instance, as well as the argument Instances, to be supplied in the invocation and references the target Operation to be invoked. The expressions that evaluate to the arguments of the Operation are, in a detailed action model, mapped into CodeActions in the Procedure connected to the Stimulus, or additional actions that compute the values of the expressions. In the case the arrow maps onto a Message the sender and the receiver are specified by the *sender* and *receiver* ClassifierRoles of the Message. The sender and receiver Instances of a Stimulus conform to these ClassifierRoles. Any condition or iteration expression attached to the arrow becomes, in a detailed action model, the test clause action in a ConditionalAction or LoopAction in the dispatching Procedure. All arrows departing the nested activation map into Stimuli (Messages) with an *activation* Association to the Stimulus (Message) corresponding to the arrow at the head of the activation. A return arrow departing the end of the activation maps into a Stimulus (Message) with:

- an *activation* Association to the Stimulus (Message) corresponding to the arrow at the head of the activation, and
- a *predecessor* association to the previous Stimulus (Message) within the same activation; that is, the last Stimulus (Message) being sent in the activation.

A return must be the final Stimulus (Message) within a predecessor chain. It is not the predecessor of any Stimulus (Message).

3.61 Object Lifeline

3.61.1 Semantics

In a sequence diagram an object lifeline denotes an Instance playing a specific role. Arrows between the lifelines denote communication between the Instances playing those roles. Within a sequence diagram the existence and duration of the Instance in a

role is shown, but the relationships among the Instances are not shown. The role is specified by a ClassifierRole; it describes the properties of an Instance playing the role and describes the relationships an Instance in that role has to other Instances.

3.61.2 Notation

An Instance is shown as a vertical dashed line called the “lifeline.” The lifeline represents the existence of the Instance at a particular time. If the Instance is created or destroyed during the period of time shown on the diagram, then its lifeline starts or stops at the appropriate point; otherwise, it goes from the top to the bottom of the diagram. An object symbol is drawn at the head of the lifeline. If the Instance is created during the diagram, then the arrow, which maps onto the Stimulus that creates the Instance, is drawn with its arrowhead on the object symbol. If the Instance is destroyed during the diagram, then its destruction is marked by a large “X,” either at the arrow mapping to the Stimulus that causes the destruction or (in the case of self-destruction) at the final return arrow from the destroyed Instance. An Instance that exists when the transaction starts is shown at the top of the diagram (above the first arrow), while an Instance that exists when the transaction finishes has its lifeline continue beyond the final arrow.

The lifeline may split into two or more concurrent lifelines to show conditionality. Each separate track corresponds to a conditional branch in the communication. The lifelines may merge together at some subsequent point.

3.61.3 Presentation Options

In some cases, it is necessary to link sequence diagrams to each other; for example, it might not be possible to put all lifelines in one diagram, or a sub-sequence is included in several diagrams; hence, it is convenient to put the common sub-sequence in a separate diagram, which is referenced from the other diagrams. In these cases, the cut between the diagrams can be expressed in one of the diagrams with a dangling arrow leaving a lifeline but not arriving at another lifeline, and in the other diagram it is expressed with a dangling arrow arriving at a lifeline from nowhere. In both cases, it is recommended to attach a note stating which diagram the sequence originates from or continues in. This is purely notational. The different diagrams show different parts of the underlying Interaction.

3.61.4 Example

See also Figure 3-56 on page 3-105.

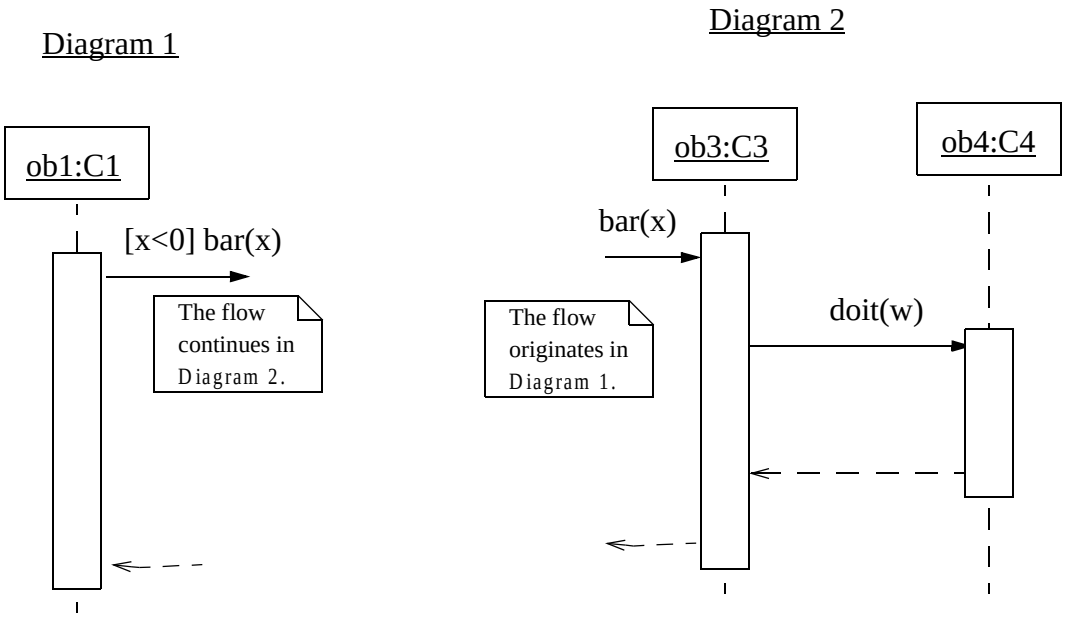


Figure 3-58 The flow shown in the sequence diagram to the left continues in the diagram to the right.

3.61.5 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

3.62 Activation

3.62.1 Semantics

An activation (focus of control) shows the period during which an Instance is performing a Procedure either directly or through a subordinate procedure. It represents both the duration of the performance of the Procedure in time and the control relationship between the activation and its callers (stack frame).

3.62.2 Notation

An activation is shown as a tall thin rectangle whose top is aligned with its initiation time and whose bottom is aligned with its completion time. The Procedure being performed may be labeled in text next to the activation symbol or in the left margin, depending on style. Alternately, the incoming arrow may indicate the Procedure, in which case it may be omitted on the activation itself. In procedural flow of control, the top of the activation symbol is at the tip of an incoming arrow (the one that initiates the procedure) and the base of the symbol is at the tail of a return arrow.

In the case of concurrent Instances each with their own threads of control, an activation shows the duration when each Instance is performing an Operation or transition in a state machine. Operations by other Instances are not relevant. If the distinction between direct computation and indirect computation (by a nested operation call) is unimportant, the entire lifeline may be shown as an activation.

3.62.3 Example

See Figure 3-55 on page 3-104 and Figure 3-56 on page 3-105.

3.62.4 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

3.63 Message and Stimulus

3.63.1 Semantics

A Stimulus is a communication between two Instances that conveys information with the expectation that action will ensue. A Stimulus will cause an Operation to be invoked, raise a Signal, or cause an Instance to be created or destroyed.

A Message is a specification of Stimulus, i.e. it specifies the roles that the sender and the receiver Instances must conform to, as well as the Procedure which will, when executed, dispatch a Stimulus that conforms to the Message.

3.63.2 Notation

In a sequence diagram a Stimulus as well as a Message is shown as a horizontal solid arrow from the lifeline of one Instance or ClassifierRole to the lifeline of another Instance or ClassifierRole. In case of a Stimulus from an Instance to itself, the arrow may start and finish on the same lifeline. The arrow is labeled with the name of the Operation to be invoked or the name of the Signal. Its argument values or argument expressions may be presented, as well.

The arrow may also be labeled with a sequence number to show the sequence of the Stimulus (Message) in the overall interaction. However, sequence numbers are often omitted in sequence diagrams, as the physical location of the arrow shows the relative sequences, but they are necessary in collaboration diagrams. Sequence numbers are useful on both kinds of diagrams for identifying concurrent threads of control. An arrow may also be labeled with a condition and/or iteration expression.

3.63.3 Presentation options

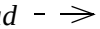
The following arrowhead variations may be used to show different kinds of communications.

filled solid arrowhead

Operation call or other nested flow of control. The entire nested sequence is completed before the outer level sequence resumes. The arrowhead may be used to denote ordinary operation calls, but it may also be used to denote concurrently active instances when one of them sends a Signal and waits for a nested sequence of behavior to complete before it continues.

stick arrowhead

Asynchronous communication; that is, no nesting of control. The sender dispatches the Stimulus and immediately continues with the next step in the execution.[1]

dashed arrow with stick arrowhead 

Return from operation call.

Variation:

In a procedural flow of control, the return arrow may be omitted (it is implicit at the end of an activation). It is assumed that every call has a paired return after any subordinate stimuli. The return value can be shown on the initial arrow. For nonprocedural flow of control (including parallel processing and asynchronous messages) returns should be shown explicitly.

Variation:

Normally message arrows are drawn horizontally. This indicates the duration required to send the stimulus is “atomic;” that is, it is brief compared to the granularity of the interaction and that nothing else can “happen” during the transmission of the stimulus. This is the correct assumption within many computers. If the stimulus requires some time to arrive, during which something else can occur (such as a stimulus in the opposite direction), then the arrow may be slanted downward so that the arrowhead is below the arrow tail.

Variation: Branching

A branch is shown by multiple arrows leaving a single point, each possibly labeled by a condition. Depending on whether the conditions are mutually exclusive, the construct may represent conditionality or concurrency.

---

1. UML 1.3 and previous versions included a half-stick arrowhead notation in addition to the stick arrowhead notation. This notation has been removed because the semantic distinction between the two was subtle and confusing.

Variation: Iteration

A connected set of arrows may be enclosed and marked as an iteration. For a generic sequence diagram, the iteration indicates that the dispatch of a set of stimuli can occur multiple times. For a procedure, the continuation condition for the iteration may be specified at the bottom of the iteration. If there is concurrency, then some arrows in the diagram may be part of the iteration and others may be single execution. It is desirable to arrange a diagram so that the arrows in the iteration can be enclosed together easily.

Variation:

A lifeline may subsume an entire set of objects on a diagram representing a high-level view.

Variation:

A distinction may be made between a period during which an Instance has a live activation and a period in which the activation is actually computing. The former (during which it has control information on a stack but during which control resides in something that it called) is shown with the ordinary double line. The latter (during which it is the top item on the stack) may be distinguished by shading the region.

3.63.4 Example

See Figure 3-56 on page 3-105.

3.63.5 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

3.64 Transition Times

3.64.1 Semantics

A Message may specify several different times; for example, a sending time and a receiving time. These are formal names that may be used within Constraint expressions. The set of different kinds of times is open-ended so that users can invent new ones as needed for special situations, such as *elapsedTime* and *startExecutionTime*. These expressions may be used in Constraints to designate specific time constraints valid for the Message.

3.64.2 Notation

A transition instance (such as a Stimulus or Message in a sequence diagram, a collaboration diagram, or a Transition in a state machine) may be given a name. A timing constraint is formed as an expression based on the name of the transition. For

example, if the name of a Stimulus is *stim*, its send-time is expressed by *stim.sendTime ()*, and its receive-time by *stim.receiveTime ()*. The timing constraint may be shown in the left margin aligned with the arrow (on a sequence diagram) or near the tail of the arrow (on a collaboration diagram). Constraints may be specified by placing Boolean expressions, possibly including time expressions, in braces on the sequence diagram.

3.64.3 Presentation Options

When it is clear from the context, the name of a Message or the name of a Stimulus may itself be used to denote the time at which the transition started. In cases where the performance of the transition is not instantaneous, the time at which the transition is ended may be indicated by the same name with a prime sign appended to the name.

3.64.4 Example

See Figure 3-55 on page 3-104.

3.64.5 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

Part 8 - Collaboration Diagrams

3.65 Collaboration Diagram

3.65.1 Semantics

A collaboration diagram presents either a Collaboration, which contains a set of roles to be played by Instances, as well as their required relationships given in a particular context, or it presents a CollaborationInstanceSet with a collection of Instances and their relationships. The diagram may also present an Interaction (InteractionInstanceSet), which defines a set of Messages (Stimuli) specifying the interaction between the Instances playing the roles within a Collaboration to achieve the desired result.

A Collaboration is used for describing the realization of an Operation or a Classifier. A Collaboration that describes a Classifier, like a UseCase, references Classifiers and Associations in general, while a Collaboration describing an Operation includes the arguments and local variables of the Operation, as well as ordinary Associations attached to the Classifier owning the Operation.

3.65.2 Notation

A collaboration diagram shows a graph of either Instances linked to each other, or ClassifierRoles and AssociationRoles; it may also include the communication stated by an Interaction or InteractionInstanceSet.

Because collaboration diagrams often are used to help design procedures, they typically show navigability using arrowheads on the lines representing Links or AssociationRoles. (An arrowhead on a line between boxes indicates a Link or AssociationRole with one-way navigability. An arrow next to a line indicates Stimuli or Message flowing in the given direction. Obviously such an arrow cannot point backwards over a one-way line.)

The order of the interaction is described with a sequence of numbers, usually starting with number *1*. For a procedural flow of control, the subsequent communication numbers are nested in accordance with call nesting. For a nonprocedural sequence of interactions among concurrent instances, all the sequence numbers are at the same level (that is, they are not nested).

A collaboration diagram without any interaction shows the *context* in which interactions can occur. It might be used to show the context for a single Operation or even for all of the Operations of a Class or group of Classes.

A collection of standard constraints may be used to show whether an Instance or a Link is created or destroyed during the execution:

- Instances and Links created during the execution may be designated as {new}.
- Instances and Links destroyed during the execution may be designated as {destroyed}.
- Instances and Links created during the execution and then destroyed may be designated as {transient}.

These changes in life state are derivable from the detailed interaction among the Instances, they are provided as notational conveniences.

3.65.2.1 Collaboration Instance

A collaboration diagram given at instance level shows a CollaborationInstanceSet; that is, a collection of object boxes and lines mapping to Instances and Links, respectively. These instances conform to the ClassifierRoles and AssociationRoles of the CollaborationInstanceSet's Collaboration. The diagram may also include arrows attached to the lines that correspond to Stimuli communicated over the Links. The diagram shows the Instances relevant to the realization of an Operation or Classifier, including Instances indirectly affected or accessed during the performance. The diagram also shows the Links among the Instances, including transient ones representing procedure arguments, local variables, and *self* links. Individual attribute values are usually not shown explicitly. If Stimuli must be sent to attribute values, the Attributes should be modeled using Associations instead.

3.65.2.2 Collaboration

A collaboration diagram given at specification level shows a Collaboration; that is, the roles defined within a Collaboration. Together, these roles form a realization of the attached Operation or Classifier of the Collaboration. The diagram contains a

collection of class boxes and lines corresponding to ClassifierRoles and AssociationRoles in the Collaboration. In this case the arrows attached to the lines map onto Messages.

3.65.3 Example

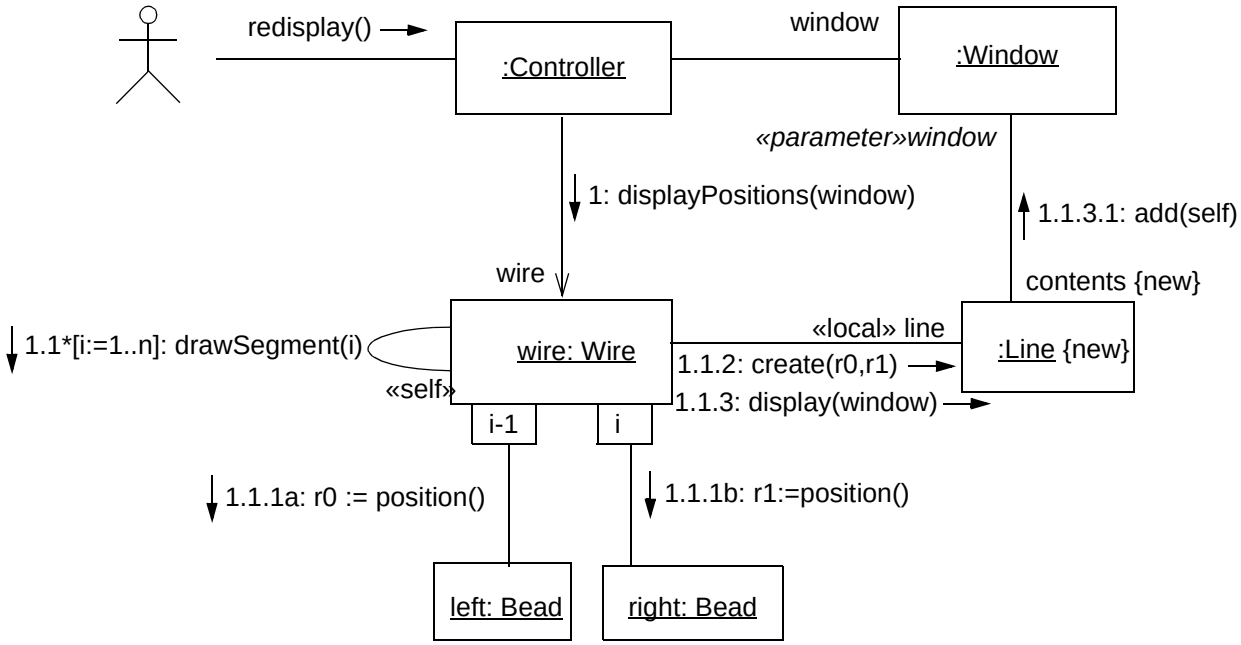


Figure 3-59 Collaboration Diagram at instance level, presenting Objects, Links, and Stimuli referenced by a CollaborationInstanceSet and its InteractionInstanceSet.

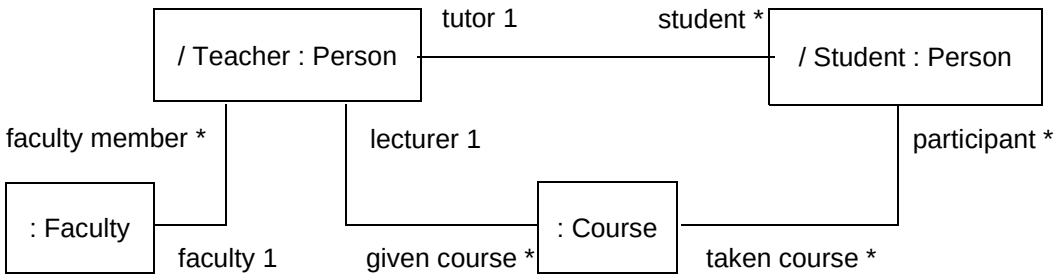


Figure 3-60 Collaboration Diagram at specification level, presenting the ClassifierRoles and the AssociationRoles that belong to the Collaboration.

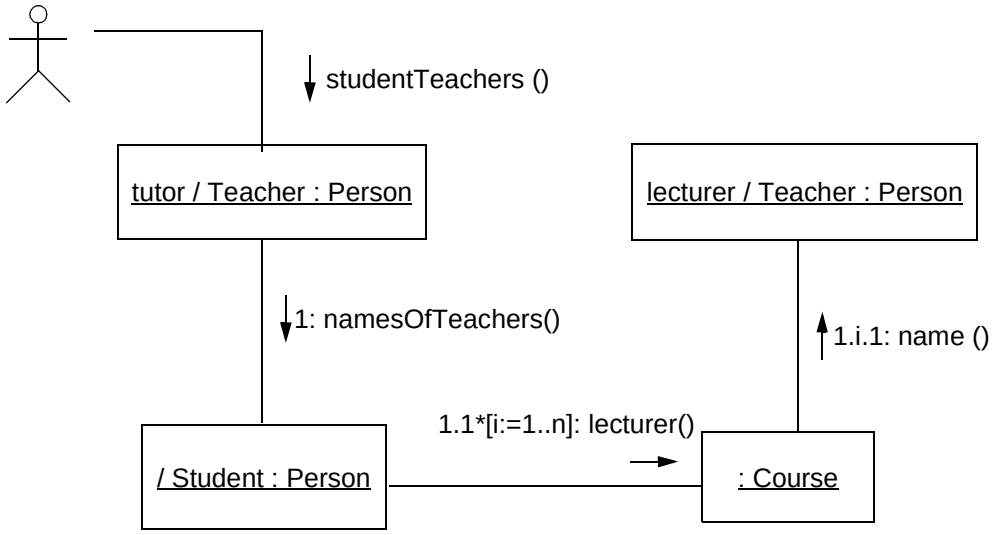


Figure 3-61 Collaboration Diagram presenting a CollaborationInstanceSet in which some of the Objects play the same role. The instances conform to the Collaboration shown in Figure 3-60 on page 3-116.

3.65.4 Mapping

A collaboration diagram maps either to a Collaboration, possibly together with an Interaction, or to a CollaborationInstanceSet possibly together with its InteractionInstanceSet. The mapping of each kind of icon is described in Section 3.69, “Collaboration Roles,” on page 3-124. The mapping of the stereotypes is explained in Section 3.49, “Link,” on page 3-84.

3.66 Pattern Structure

3.66.1 Semantics

A Collaboration can be used to specify the implementation of design constructs. For this purpose, it is necessary to specify its context and interactions. It is also possible to view a Collaboration as a single entity from the “outside.” For example, this could be used to identify the presence of design patterns within a system design. A pattern is a parameterized Collaboration; that is, a Collaboration template. In each use of the pattern, actual Classifiers are substituted for the parameters in the pattern definition.

Note that *patterns* as defined in *Design Patterns* by Gamma, Helm, Johnson, and Vlissides include much more than structural descriptions. UML describes the structural aspects and some behavioral aspects of design patterns; however, UML notation does not include other important aspects of patterns, such as usage trade-offs or examples. These must be expressed by other means, such as in text or tables.

A Collaboration can be defined in terms of other, so-called subordinate, Collaborations. Each role in the former Collaboration, the so-called superordinate Collaboration, is either a new role that is defined in the superordinate Collaboration or it is a role defined in one or several of the subordinate Collaborations and reused in the definition of the superordinate Collaboration. In the latter case, the role is often renamed so it better suits the purpose of the superordinate Collaboration. If so, the original names of the roles are shown within curly brackets after the name used within the superordinate Collaboration (see Figure 3-66 on page 3-120).

3.66.2 Notation

A use of a Collaboration is shown as a dashed ellipse containing the name of the Collaboration. A dashed line is drawn from the collaboration symbol to each of the symbols denoting Classifiers that participate in the Collaboration. Each line is labeled by the *role* of the participant. The roles correspond to the names of elements within the context for the Collaboration; such names in the Collaboration are treated as parameters that are bound to specify elements on each occurrence of the pattern within a model. Therefore, a collaboration symbol can show the use of a design pattern together with the actual Classifiers and Associations that occur in that particular use of the pattern.

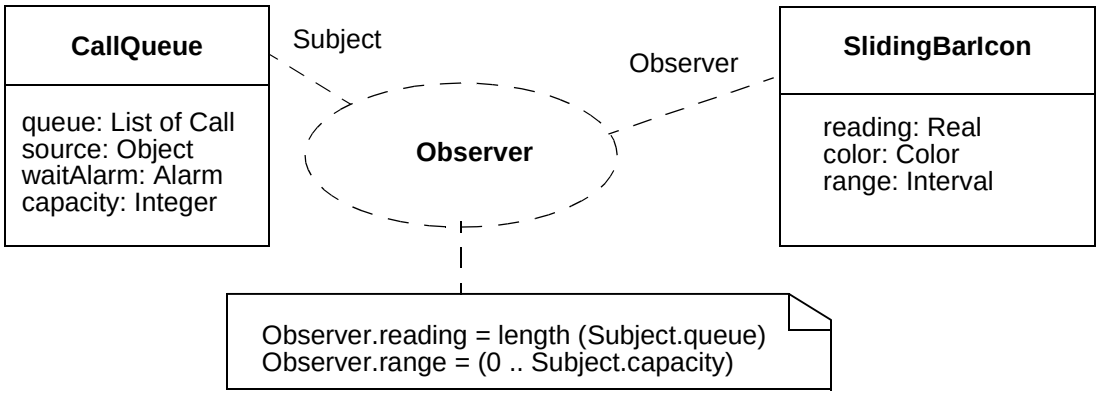


Figure 3-62 Use of a Collaboration.

As a Collaboration is a GeneralizableElement, it may have Generalization relationships to other Collaborations. In this way it is possible to define one Collaboration to be a specialization of another Collaboration. It is depicted by the ordinary Generalization arrow from the dashed ellipse representing the child Collaboration to the icon of the parent Collaboration. The roles of the child Collaborations may be specializations of roles in the parent Collaboration. This is shown by redefining the role name of the parent collaboration in the child collaboration.

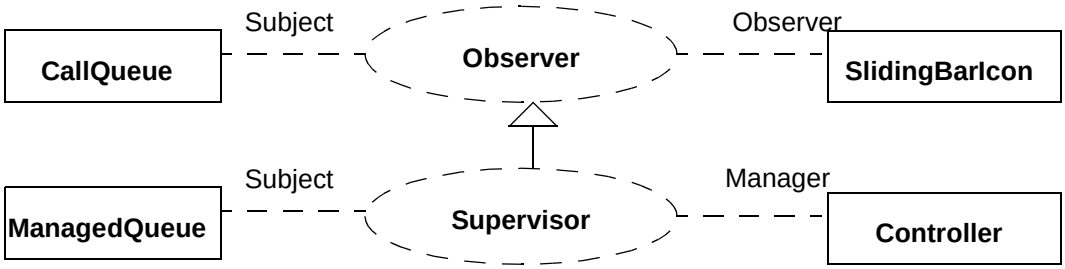


Figure 3-63 Specialization of a Collaboration. As the Subject role of the Supervisor collaboration is a specialization (an extension) of the Subject role defined in the Observer collaboration, the ManagedQueue class is used instead of the CallQueue class as the base of the Subject role.

A dashed arrow with a stick arrowhead is used to show that a Collaboration is a realization of an Operation or a Classifier. This relationship can also be presented in textual form within the Collaboration symbol.

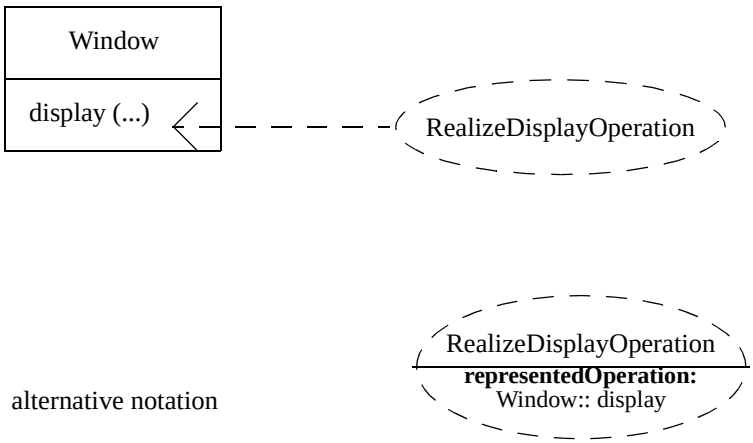


Figure 3-64 The relationship between a Collaboration and the element it is realizing can be shown either as a dashed arrow with a stick arrowhead from the Collaboration to the realized element, or in text.

The usual convention is used to show a CollaborationInstanceSet; that is, it is shown as a dashed ellipse with the name underlined. The Instances and the Links that participate in the CollaborationInstanceSet are connected to the ellipse with dashed lines. The name of the role an instance is playing is shown close to the line and the instance.

In some cases it is convenient to show the static structure of a Collaboration within the collaboration icon (the dashed ellipse).

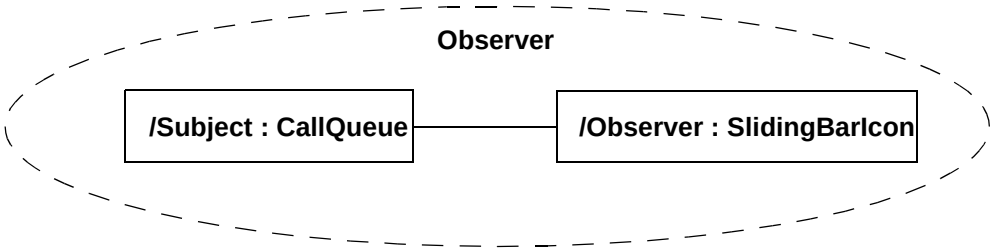


Figure 3-65 The static structure of a Collaboration shown within the collaboration icon.

It is possible to denote that a Collaboration is defined in terms of other Collaborations in two different ways, either using dashed ellipses showing the Collaborations and their relationships, or using ordinary collaboration diagrams. The former way has the advantage that it explicitly shows the relationship between the Collaborations, while the latter shows the structure of the new Collaboration.

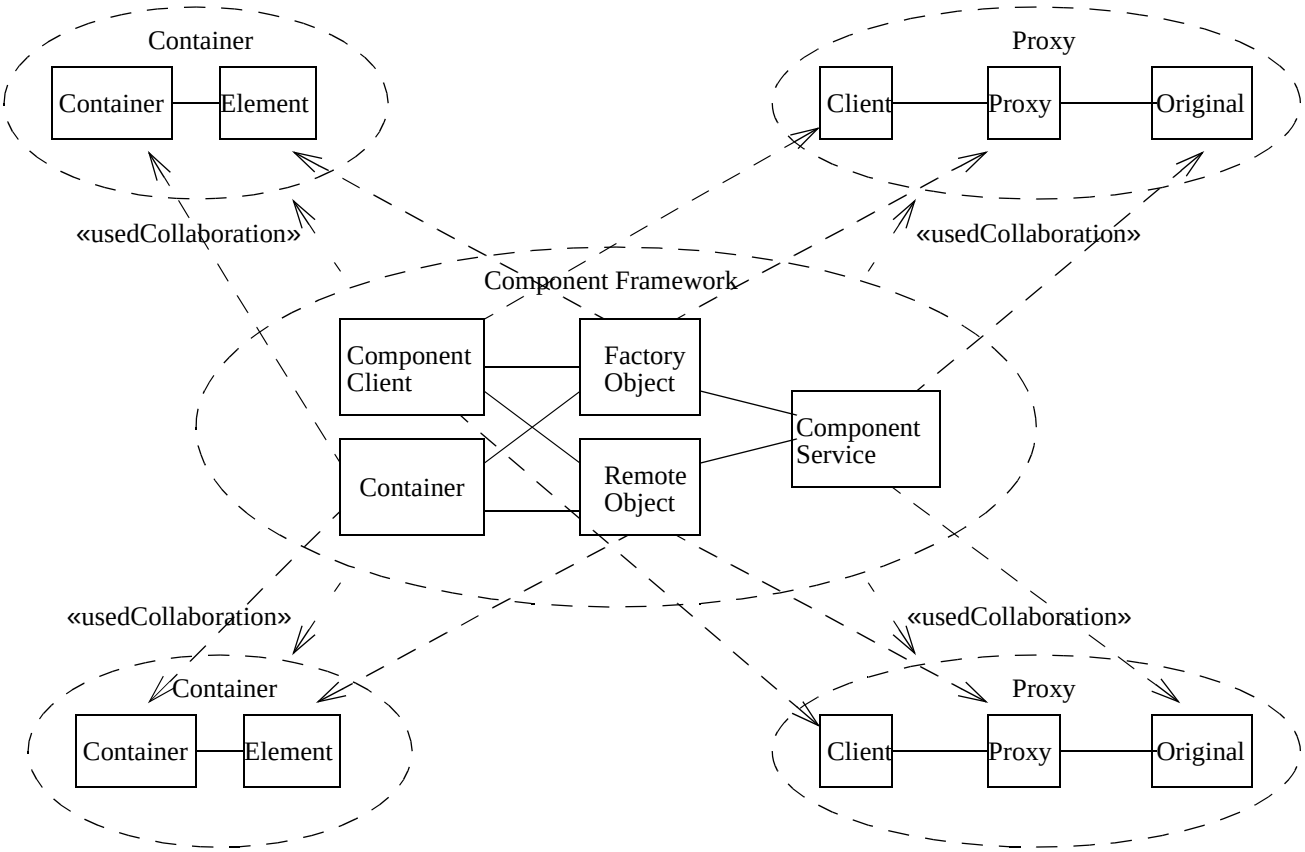


Figure 3-66 The ComponentFramework Collaboration uses two occurrences of the Proxy Collaboration and two occurrences of the Container Collaboration. Note that each role in the Component Framework corresponds to a role in two of the used Collaborations.

3.66.3 Mapping

A collaboration usage symbol maps into a Collaboration. For each class symbol and lines attached by a dashed line to the pattern occurrence symbol, the corresponding Classifier or Association is bound to the template parameter that is the *base* association target of the ClassifierRole or AssociationRole in the Collaboration template with the name equal to the name on the dashed line.

A dashed arrow with a closed hollow arrowhead from a Collaboration symbol to a Classifier or to an Operation is mapped onto the *representedClassifier* and onto the *representedOperation* association of the Collaboration, respectively.

A collaboration usage symbol with its name underlined is mapped onto a CollaborationInstanceSet. The object box symbols and the lines attached to the ellipse by dashed lines are mapped onto Instances and Links, respectively.

3.67 Collaboration Contents

The contents of a Collaboration is a collection of roles specifying how Instances and Links cooperate within a given context for a particular purpose, such as performing an Operation or a Use case. A Collaboration is a fragment of a larger complete model that is intended for a particular purpose.

3.67.1 Semantics

A *Collaboration diagram* shows either a Collaboration or a CollaborationInstanceSet. In the former case, the diagram shows one or more roles together with their contents, relationships, and neighbor roles, plus additional relationships and Classes as needed. When the diagram shows a CollaborationInstanceSet, it shows instances participating in the CollaborationInstanceSet, playing the roles defined in the Collaboration. To use a Collaboration, each role must be bound to an actual Classifier (or collection of Classifiers, if multiple classification is used) that (jointly) support the Features required by the role. The additional elements express additional requirements that cannot be modeled with roles, such as Generalizations between roles.

3.67.2 Notation

A collaboration diagram presents a graph of class boxes or object boxes together with connecting lines. These icons map onto ClassifierRoles and AssociationRoles, or Instance, and Links, respectively (see Section 3.69, “Collaboration Roles,” on page 3-124).

However, a collaboration diagram may also contain other elements, like different kinds of Classifiers, Generalizations, and Constraints, to express additional information. These elements are shown using their ordinary icons.

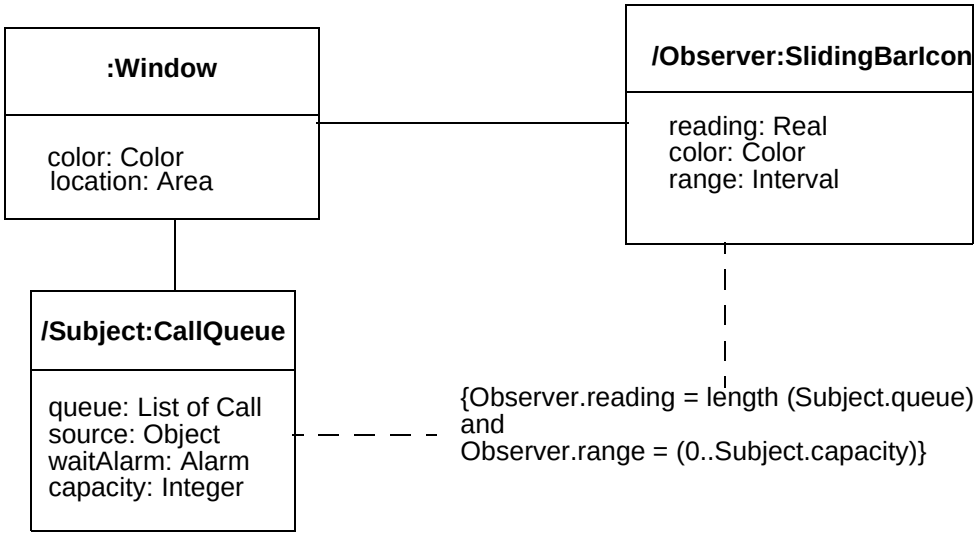


Figure 3-67 A collaboration diagram showing a Collaboration with a Constraint as a constraining element of the Collaboration.

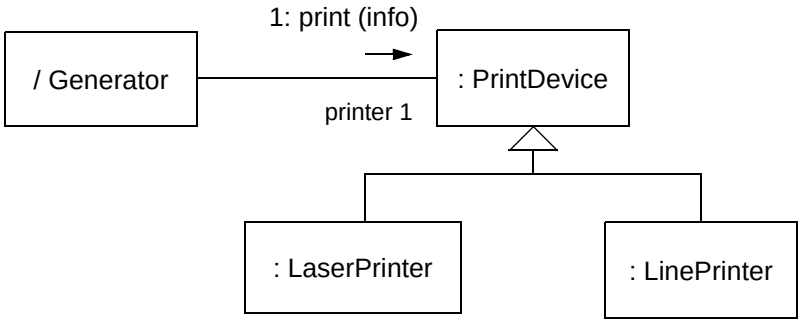


Figure 3-68 A collaboration diagram showing different roles, together with two additional Generalization relationships as constraining elements.

3.67.3 Mapping

The mapping of roles and instances are described in Section 3.69, “Collaboration Roles,” on page 3-124. Any constraining element, like a generalization arrow, is mapped onto its usual model element, such as Generalization. These elements are referenced by the Collaboration as its *constraining elements*.

3.68 Interactions

A collaboration of Instances interacts to accomplish a purpose (such as performing an Operation) by exchanging Stimuli. These may include both sending Signals and invocations of Operations, as well as more implicit interaction through conditions and time events. A specific pattern of communication exchanges to accomplish a specific purpose is called an *Interaction*. The collection of Stimuli sent between the Instances that participate in a CollaborationInstanceSet when they perform the task of the Collaboration is called an *InteractionInstanceSet*.

3.68.1 Semantics

An *Interaction* is a behavioral *specification* that comprises a sequence of communications exchanged among a set of Instances within a Collaboration to accomplish a specific purpose, such as the implementation of an Operation. To specify an Interaction, it is first necessary to specify a Collaboration; that is, to establish the roles that interact and their relationships. Then, the possible interaction sequences are specified. These can be specified in a single description containing conditionals (branches or conditional signals), or they can be specified by supplying multiple descriptions, each describing a particular path through the possible execution paths.

One communication is specified with a Message; it specifies the sender and the receiver roles, as well as the Procedure that will cause the communication to take place. The Procedure specifies what kind of communication that should take place, such as sending a Signal or invoking an Operation, and determines the actual arguments to be supplied. The Procedure may also state conditions or iterations of the communication.

When the Procedure is performed, a Stimulus is dispatched conforming to the Message. The Stimulus contains references to the sender and the receiver Instances playing the sender role and the receiver role of the Message, as well as a sequence of references to Instances being the actual arguments determined by the Procedure. An InteractionInstanceSet is a collection of Stimuli that conform to the Messages of an Interaction, i.e. the Stimuli are sent between the Instances participating an a CollaborationInstanceSet when they perform the task defined by the Collaboration.

3.68.2 Notation

Interactions are shown as sequence diagrams or as collaboration diagrams. Both diagram formats show the execution of collaborations. However, sequence diagrams do not show the relationships between the Instances or the Attribute values of the Instances; therefore, they do not fully show the context aspect of a Collaboration. Sequence diagrams do show the behavioral aspect of Collaborations explicitly, including the time sequence of Stimuli and explicit representation of method activations. Sequence diagrams are described in “Part 7 - Interaction Diagrams” on page 3-100. Collaboration diagrams show the full context of an interaction, including the Instances and their relationships relevant to a particular interaction. The sequencing of the Stimuli is done using sequence numbers, since distributing them along a time

axis, like in Sequence diagrams, is not possible in this kind of diagram. (In fact, in some cases it is convenient to use sequence numbers in combination with a time axis.) The contents of collaboration diagrams are described in the following section.

3.68.3 Mapping

The mapping of roles and instances are described below, while the mapping of messages and stimuli are described in Section 3.72, “Message and Stimulus,” on page 3-130.

3.68.4 Example

See Section 3.65, “Collaboration Diagram,” on page 3-114 for examples of Interactions and InteractionInstanceSets and their Collaborations and CollaborationInstanceSets, respectively.

3.69 Collaboration Roles

3.69.1 Semantics

A ClassifierRole defines a role to be played by an Instance within a Collaboration. The role describes the kind of Instance that may play the role, such as required Operations and Attributes, and describes its relationships to Instances playing other roles. The relationships to other roles are defined by AssociationRoles. These describe the required Links between the Instances; that is, a subset of the existing Links.

3.69.2 Notation

A ClassifierRole is shown using a class rectangle symbol. Normally, only the name compartment is shown, but the attribute and operation compartments may also be shown when needed. The name compartment contains the string:

‘/’ ClassifierRoleName ‘:’ ClassifierName [‘,’ ClassifierName]*

The name of the Classifier (or Classifiers if multiple classification is used) can include a full pathname of enclosing Packages, if necessary. A tool will normally permit shortened pathnames to be used when they are unambiguous. The Package names precede the Classifier name and are separated by double colons. For example:

```
display_window: WindowingSystem::GraphicWindows::Window
```

A stereotype may be shown textually (in guillemets above the name string) or as an icon in the upper right corner. A ClassifierRole representing a set of Instances can include a multiplicity indicator (such as “*”) in the upper right corner of the class box.

An AssociationRole is shown with the usual association line. The name string of the AssociationRole follows the same syntax as for the ClassifierRole. If the name is omitted, a line connected to ClassifierRole symbols denotes an AssociationRole. The information attached to the ends of the AssociationRole; that is, to the AssociationEndRoles, are shown using the same notation as for AssociationEnds.

An Instance playing the role defined by a ClassifierRole is depicted by an object box, normally without an attribute compartment. The name of the Instance is shown as a string:

ObjectName '/' ClassifierRoleName ':' ClassifierName [',' ClassifierName]*

That is it starts with the name of the Instance, followed by the complete name of the ClassifierRole, all underlined. If the attribute compartment is shown, it contains the names of the Attributes required by an Instance playing the role. If some Attributes are required to have certain values, this is shown in the same way as in object diagrams; that is, the name of the attribute followed by an equal sign and the relevant values.

A Link is shown by a line between object boxes. Its name string follows the syntax of an Object playing a specific role.

3.69.3 Presentation options

The name of a ClassifierRole may be omitted. In this case, the colon is kept together with the Classifier name. The role name may be omitted only if there is only *one* role to be played by Instances of the base Classifier in the Collaboration.

The name of the Classifier may be omitted together with the colon.

At least one of the Classifier name (together with the colon) or the ClassifierRole name (together with the slash) must be present to denote a ClassifierRole. Otherwise, the rectangle denotes an ordinary Classifier or Instance depending on whether the name is underlined or not.

If the role is to be played by an Instance originating from multiple Classifiers, the names of the Classifiers are shown in a comma separated list after the colon.

In an object box the Instance name, the role name and / or the classifier name may be omitted. However, the colon should be kept in front of the classifier name, and the slash should be kept in front of the role name. The notation used is the same for Instances in general, with the possible addition of the name of the ClassifierRole that the Instance conforms to.

Note, the name of an Instance is always underlined, whereas the name of a Classifier (including ClassifierRole) is never underlined. Furthermore, an un-named line between icons representing Instances is always a Link, and between icons representing Classifiers (except ClassifierRoles) it is always an Association.

These tables summarize the different combinations of names:

syntax	explanation
: C	un-named Instance originating from the Classifier C
/ R	un-named Instance playing the role R
/ R : C	un-named Instance originating from the Classifier C playing the role R
O / R	an Instance named O playing the role R
O : C	an Instance named O originating from the Classifier C
O / R : C	an Instance named O originating from the Classifier C playing the role R
O	an Instance named O
/ R	a role named R
: C	an un-named role with the *base* Classifier C
/ R : C	a role named R with the *base* Classifier C

3.69.4 Example

See figures in Section 3.65, “Collaboration Diagram,” on page 3-114.

3.69.5 Mapping

A classifier role rectangle maps onto one ClassifierRole. The role name is the name of the ClassifierRole and the sequence of classifier names are the names of the *base* Classifiers. An association role line maps onto an AssociationRole attached to the ClassifierRoles corresponding to the rectangles at the end points of the line.

An object symbol maps onto an Instance whose name is the *object* part of the name string. The Classifiers of the Instance are those named according to the sequence of names in the *class* part of the string (or children of these Classifiers). The Instance conforms to the ClassifierRole, whose name is the *role* part of the string.

A Collaboration can also be used for describing the internal structure of a Classifier. In such case, the names of the roles are the same as the names of the attributes of the Classifier. In this way, the connection between the roles and the Attributes they represent are established. (The base of the roles are not enough for uniquely identifying this mapping, since several Attributes may have the same type.)

3.70 Multiobject

3.70.1 Semantics

A multiobject represents a set of Instances on the “many” end of an Association. This is used to show Operations and Signals that address the entire set, rather than a single Instance in it. The underlying static model is unaffected by this grouping. This corresponds to an Association with multiplicity “many” used to access a set of associated Instances.

3.70.2 Notation

A multiobject is shown as two rectangles in which the top rectangle is shifted slightly vertically and horizontally to suggest a stack of rectangles. A message arrow to the multiobject symbol indicates a Stimulus to the set of Instances (for example, a selection Operation to find an individual Object).

To perform an Operation on each Instance in a set of associated Instances requires two Stimuli: an iteration to the multiobject to extract Links to the individual Instances and then a Stimulus sent to each individual Instance using the (temporary) Link. This may be elided on a diagram by combining the arrows into a single arrow that includes an iteration and an application to each individual Instance. The target rolename takes a “many” indicator (*) to show that many individual Links are implied. Although this may be written as a single Stimulus, in the underlying model (and in any actual code) it requires the two layers of structure (iteration to find Links, communication using each Link) mentioned previously.

An Instance from the set is shown as a normal object symbol, but it may be attached to the multiobject symbol using a composition Link to indicate that it is part of the set. A communication arrow to the simple object symbol indicates a Stimulus to an individual Instance.

Typically a selection Stimulus to a multiobject returns a reference to an individual Instance, to which the original sender then sends a Stimulus.

3.70.3 Example

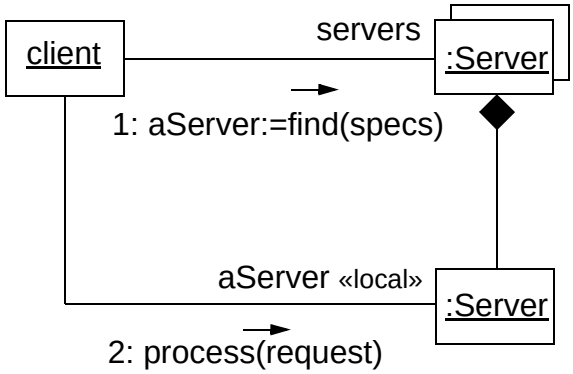


Figure 3-69 Multiobject

3.70.4 Mapping

A multiobject symbol maps to a collection of Instances in which each Instance conforms to the ClassifierRole and this role has the multiplicity “many” (or whatever is explicitly specified). In other respects, it maps the same as an object symbol. (The stereotype is explained in Section 3.49, “Link,” on page 3-84.)

3.71 Active object

An *active object* is one that owns a thread of control and may initiate control activity. A passive object is one that holds data, but does not initiate control. However, a passive object may send Stimuli in the process of processing a request that it has received. In a collaboration diagram, a ClassifierRole that is an active class represents the active objects that occur during execution.

3.71.1 Semantics

An active object is an Instance that owns a thread of control. Processes and tasks are traditional kinds of active objects.

3.71.2 Notation

A role for an active object is shown as a rectangle with a heavy border. Frequently, active object roles are shown as composites with embedded parts.

The property keyword *{active}* may also be used to indicate an active object.

3.71.3 Example

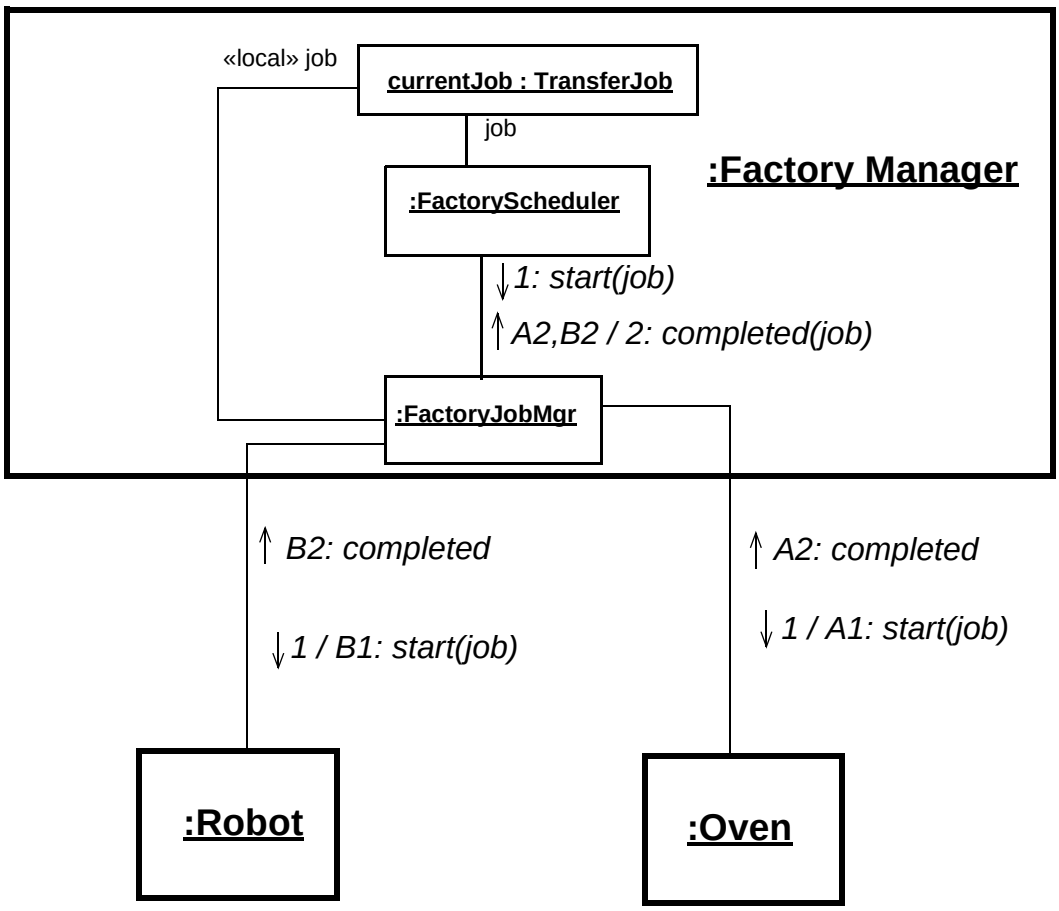


Figure 3-70 Composite Active Object

3.71.4 Mapping

An active object symbol maps as an object symbol does, with the addition that the class of the object has the *active* property set.

3.72 Message and Stimulus

3.72.1 Semantics

In a collaboration diagram a Stimulus is a communication between two Instances that conveys information with the expectation that action will ensue. A Stimulus will cause an Operation to be invoked, raise a Signal, or an Instance to be created or destroyed.

A Message is a specification of Stimulus; i.e., it specifies the roles that the sender and the receiver Instances should conform to, as well as the Procedure which will, when executed, dispatch a Stimulus that conforms to the Message.

3.72.2 Notation

Messages and Stimuli are shown as labeled arrows placed near an AssociationRole or a Link, respectively. The meaning is that the Link is used for transportation of the Stimulus to the target Instance. The arrow points along the line in the direction of the receiving Instance.

3.72.2.1 Control flow type

The following arrowhead variations may be used to show different kinds of communications.

filled solid arrowhead

Operation call or other nested flow of control. The entire nested sequence is completed before the outer level sequence resumes. The arrowhead may be used to denote ordinary operation calls, but it may also be used to denote concurrently active instances when one of them sends a Signal and waits for a nested sequence of behavior to complete before it continues.

stick arrowhead

Asynchronous communication; that is, no nesting of control. The sender dispatches the Stimulus and immediately continues with the next step in the execution.

dashed arrow with stick arrowhead

Return from an operation call. The return arrow may be suppressed as it is implicit at the end of an activation.

other variations

Other kinds of control may be shown, such as “balking” or “time-out;” however, these are treated as extensions to the UML core.

A half stick arrowhead can be used to show asynchronous communication. This alternative is included for backwards compatibility. UML 1.3 and previous versions, included both half stick arrowhead and stick arrowhead with a very small (and not well-understood) distinction.

3.72.2.2 Arrow label

In the following the term *Message* is used, but the text applies to *Stimulus*, as well.

The label has the following syntax:

predecessor sequence-expression return-value := *message-name argument-list*

The label indicates the Message being sent, its arguments and return values, and the sequencing of the Message within the larger interaction, including call nesting, iteration, branching, concurrency, and synchronization.

3.72.2.3 Predecessor

The predecessor is a comma-separated list of sequence numbers followed by a slash ('/'):

sequence-number ',' . . . '/'

The clause is omitted if the list is empty.

Each sequence-number is a sequence-expression without any recurrence terms. It must match the sequence number of another Message.

The meaning is that the Message is not enabled until all of the communications whose sequence numbers appear in the list have occurred. Therefore, the list of predecessors represents a synchronization of threads.

Note that the Message corresponding to the numerically preceding sequence number is an implicit predecessor and need not be explicitly listed. All of the sequence numbers with the same prefix form a sequence. The numerical predecessor is the one in which the final term is one less. That is, number 3.1.4.5 is the predecessor of 3.1.4.6.

3.72.2.4 Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (':').

sequence-term '.' . . . ':'

Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[*integer* | *name*] [*recurrence*]

The *integer* represents the sequential order of the Message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows Message 3.1.3 within activation 3.1. The *name* represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: Message 3.1a and Message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more Messages that are executed depending on the conditions involved. The choices are:

‘*’ ‘[’ iteration-clause ‘]’an iteration

‘[’ condition-clause ‘]’a branch

An iteration represents a sequence of Messages at the given nesting depth. The iteration clause may be omitted (in which case the iteration conditions are unspecified). The iteration-clause is meant to be expressed in pseudocode or an actual programming language, UML does not prescribe its format. An example would be: **[i := 1..n]*.

A condition represents a Message whose execution is contingent on the truth of the condition clause. The condition-clause is meant to be expressed in pseudocode or an actual programming language; UML does not prescribe its format. An example would be: *[x > y]*.

Note that a branch is notated the same as an iteration without a star. One might think of it as an iteration restricted to a single occurrence.

The iteration notation assumes that the Messages in the iteration will be executed sequentially. There is also the possibility of executing them concurrently. The notation for this is to follow the star by a double vertical line (for parallelism): *||.

Note that in a nested control structure, the recurrence is not repeated at inner levels. Each level of structure specifies its own iteration within the enclosing context.

3.72.2.5 Signature

A signature is a string that indicates the name, the arguments, and the return value of an Operation or a Reception. The signature of a Message is derived from (is the same as) the signature of the Operation invoked by the Message’s dispatching Procedure, or the Reception for the Signal sent by the Procedure. These have the following properties.

Return-value

This is a list of names that designates the values returned at the end of the communication within the subsequent execution of the overall interaction. These identifiers can be used as arguments to subsequent Messages. If the Message does not return a value, then the return value and the assignment operator are omitted.

Message-name

This is the name of the Operation to be applied on the receiver, or the Signal that is sent to the receiver.

Argument list

This is a comma-separated list of arguments (actual parameters) enclosed in parentheses. The parentheses can be used even if the list is empty. Each argument is either a reference to an Instance, or an expression in pseudocode or an appropriate programming language (UML does not prescribe). The expressions may use return values of previous messages (in the same scope) and navigation expressions starting from the source Instance; that is, Attributes of it and Links from it and paths reachable from them.

3.72.3 Presentation Options

Instead of text expressions for arguments and return values, data tokens may be shown near a message label. A token is a small circle labeled with the argument expression or return value name. It has a small arrow on it that points along the Message (for an argument) or opposite the Message (for a return value). Tokens represent arguments and return values. The choice of text syntax or tokens is a presentation option.

The syntax of Messages may instead be expressed in the syntax of a programming language, such as C++ or Smalltalk. All of the expressions on a single diagram should use the same syntax, however.

A return flow may be explicitly shown with a dashed arrow.

3.72.4 Example

See Figure 3-59 on page 3-116 for examples within a diagram.

Samples of control message label syntax:

2: display (x, y)simple Message

1.3.1: p:= find(specs)nested call with return value

4 [x < 0] : invert (x, color)conditional Message

A3,B4/ C3.1*: update ()synchronization with other threads, iteration

3.72.5 Mapping

An arrow symbol maps either onto a Message or a Stimulus. If the arrow is attached to a line corresponding to an AssociationRole, it maps onto a Message, with the ClassifierRoles corresponding to the end-points of the line as the sender and the receiver roles. If the line corresponds to a Link, the arrow maps onto a Stimulus, with

the Instances corresponding to the end-points of the line as the sender and the receiver Instances. The line is the *communication connection* or the *communication link* of the Message or the Stimulus, respectively.

The control flow type sets the corresponding properties:

- *solid arrowhead*: a synchronous operation invocation
- *stick arrowhead*: an asynchronous operation invocation
- *dashed arrow with stick arrowhead:* return from an synchronous operation invocation

The predecessor expression, together with the sequence expression, determines the *predecessor* and *activation* (caller) relationships of a Message or a Stimulus. The predecessors of a Message (Stimulus) are those Messages (Stimuli) corresponding to the sequence numbers in the predecessor list as well as the Message (Stimulus) corresponding to the immediate preceding sequence number as the Message (Stimulus); that is, 1.2.2 is the one preceding 1.2.3. The caller is the ClassifierRole (Instance) receiving the Message (Stimulus) whose sequence number is truncated by one position; that is, 1.2 is the caller of 1.2.3. The thread-of-control name maps onto a Classifier stereotyped *thread*; that is, an active class.

The label of the arrow is mapped into either the body attribute of the Procedure, or into a detailed action model starting with recurrence. The return of a value maps into a Message from the called Instance to the caller with the dispatching Procedure that outputs the return value. Its *predecessor* is the final Message within the procedure. Its *activation* is the Message that called the procedure.

The recurrence expression, the iteration clause, and the condition clause determine if a ConditionalAction or LoopAction is used in the Procedure attached to the Message.

The operation name and the form of the signature determine the Operation attached to the CallOperationAction in the Procedure of the Message. Similarly for a Signal and SendSignalAction. The arguments of the signature determine the arguments associated with the CallOperationAction and SendSignalAction, respectively

In a procedural interaction, each arrow symbol also maps into a second Message representing the return flow, unless the return flow is explicitly shown. This Message has an *activation* Association to the original call Message. Its associated Procedure outputs the return values as arguments (if any).

3.73 Creation/Destruction Markers

3.73.1 Semantics

During the execution of an interaction some Instances and Links are created and some are destroyed. The creation and destruction of elements can be marked.

3.73.2 Notation

An Instance or a Link that is created during an interaction has the standard constraint *new* attached to it. An Instance or a Link that is destroyed during an interaction has the standard constraint *destroyed* attached. These constraints may be used even if the element has no name. Both constraints may be used together, but the standard constraint *transient* may be used in place of *new destroyed*.

3.73.3 Presentation options

Tools may use other graphic markers in addition to or in place of the keywords. For example, each kind of lifetime might be shown in a different color. A tool may also use animation to show the creation and destruction of elements and the state of the system at various times.

3.73.4 Example

See Figure 3-59 on page 3-116.

3.73.5 Mapping

Creation or destruction indicators map either into procedures containing CreateObjectActions or DestroyObjectActions in the corresponding ClassifierRoles. The former two Actions dispatch the Stimuli that cause the changes. These status indicators are merely summaries of the total actions.

Part 9 - Statechart Diagrams

A statechart diagram can be used to describe the behavior of instances of a model element such as an object or an interaction. Specifically, it describes possible sequences of states and actions through which the element instances can proceed during its lifetime as a result of reacting to discrete events (for example, signals, operation invocations).

The semantics and notation described in this chapter are substantially those of David Harel's statecharts with modifications to make them object-oriented. His work was a major advance on the traditional flat state machines. Statechart notation also implements aspects of both Moore machines and Mealy machines, traditional state machine models.

3.74 Statechart Diagram

3.74.1 Semantics

Statechart diagrams represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances. Typically, it is used for describing the behavior of class instances, but statecharts may also describe the behavior of other entities such as use-cases, actors, subsystems, operations, or methods.

3.74.2 Notation

A statechart diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that interconnect them. States may also contain subdiagrams by physical containment or tiling. Note that every state machine has a top state that contains all the other elements of the entire state machine. The graphical rendering of this top state is optional.

The association between a state machine and its context does not have a special notation.

An example statechart diagram for a simple telephone object is depicted in Figure 3-71 on page 3-137.

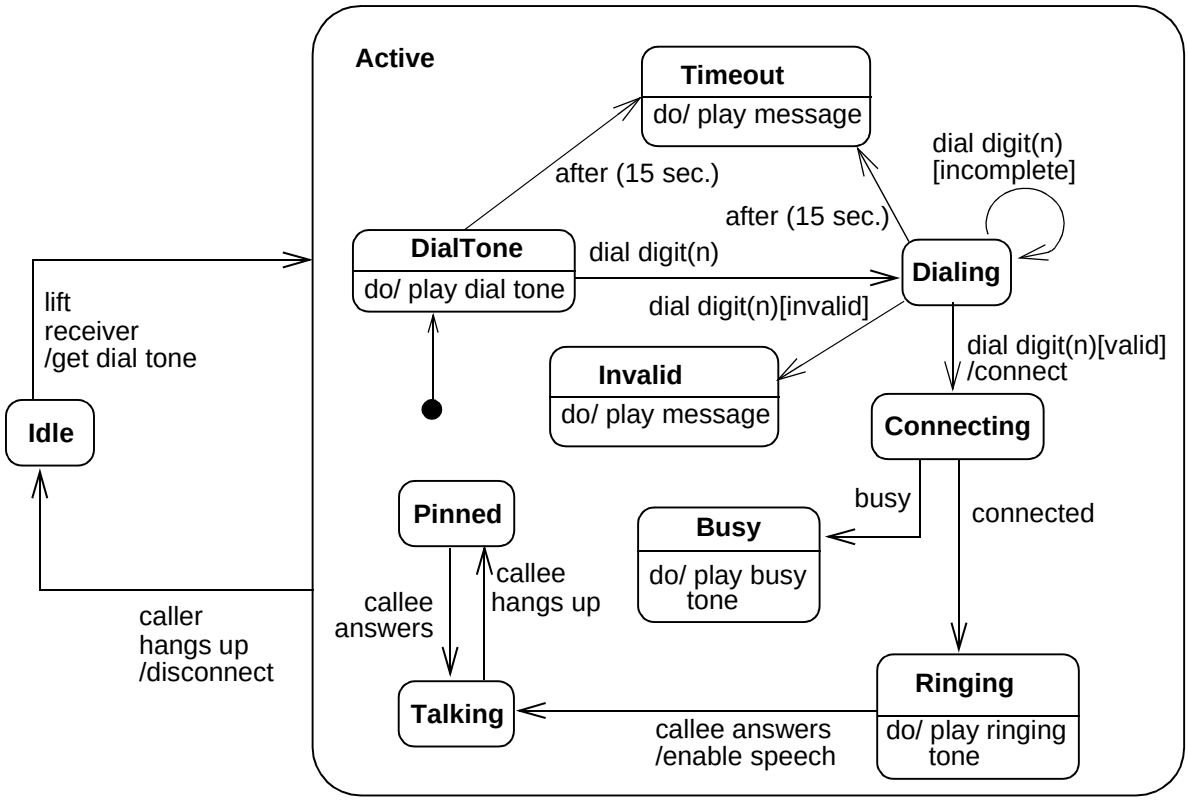


Figure 3-71 State Diagram

3.74.3 Mapping

A statechart diagram maps into a StateMachine. That StateMachine may be owned by an instance of a model element capable of dynamic behavior, such as classifier or a behavioral feature, which provides the context for that state machine. Different contexts may apply different semantic constraints on the state machine.

3.75 State

3.75.1 Semantics

A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. A *composite* state is a state that, in contrast to a *simple* state, has a graphical decomposition. (Composite states and their notation are described in more detail in Section 3.76, “Composite States,” on page 3-140.) Conceptually, an object remains in a state for an interval of time. However, the semantics allow for modeling “flow-through” states that are instantaneous, as well as transitions that are not instantaneous.

A state may be used to model an ongoing activity. Such an activity is specified either by a nested state machine or by a computational expression.

3.75.2 Notation

A state is shown as a rectangle with rounded corners (Figure 3-72 on page 3-139). Optionally, it may have an attached name tab. The name tab is a rectangle, usually resting on the outside of the top side of a state and it contains the name of that state. It is normally used to keep the name of a composite state that has concurrent regions, but may be used in other cases as well (the Process state in Figure 3-77 on page 3-147 illustrates the use of the name tab).

A state may be optionally subdivided into multiple compartments separated from each other by a horizontal line. They are as follows:

- Name compartment

  This compartment holds the (optional) name of the state as a string. States without names are anonymous and are all distinct. It is undesirable to show the same named state twice in the same diagram, as confusion may ensue. Name compartments should not be used if a name tab is used and vice versa.

- Internal transitions compartment

  This compartment holds a list of internal actions or activities that are performed while the element is in the state.

The action label identifies the circumstances under which the action specified by the action expression will be invoked. The action expression may use any attributes and links that are in the scope of the owning entity. For list items where the action expression is empty, the backslash separator is optional.

A number of action labels are reserved for various special purposes and, therefore, cannot be used as event names. The following are the reserved action labels and their meaning.

entry	This label identifies an action, specified by the corresponding action expression, which is performed upon entry to the state (entry action).
exit	This label identifies an action, specified by the corresponding action expression, that is performed upon exit from the state (exit action).
do	This label identifies an ongoing activity (“do activity”) that is performed as long as the modeled element is in the state or until the computation specified by the action expression is completed (the latter may result in a completion event being generated).
include	This label is used to identify a submachine invocation. The action expression contains the name of the submachine that is to be invoked. Submachine states and the corresponding notation are described in Section 3.82, “Submachine States,” on page3-152.

In all other cases, the action label identifies the event that triggers the corresponding action expression. These events are called internal transitions and are semantically equivalent to self transitions *except that the state is not exited or re-entered*. This means that the corresponding exit and entry actions are not performed. The general format for the list item of an internal transition is:

event-name ‘(’ *comma-separated-parameter-list* ‘)’ ‘[’ *guard-condition*‘]’ ‘/’ *action-expression*

Each event name may appear more than once per state if the guard conditions are different. The event parameters and the guard conditions are optional. If the event has parameters, they can be used in the action expression through the current event variable.

3.75.3 Example

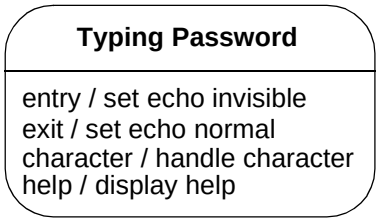


Figure 3-72 State

3.75.4 Mapping

A state symbol maps into a State. See Section 3.76, “Composite States,” on page 3-140 for further details on which kind of state.

The name string in the symbol maps to the name of the state. Two symbols with the same name map into the same state. However, each state symbol with no name (or an empty name string) maps into a distinct anonymous State.

A list item in the internal transition compartment maps into a corresponding Action associated with a state. An “entry” list item; that is, an item with the “entry” label maps to the “entry” role, an “exit” list item maps to the “exit” role, and a “do” item maps to the “doActivity” role. (The mapping of “include” items is discussed in Section 3.82, “Submachine States,” on page 3-152.)

A list item with an event name maps to a Transition associated with the “internal” role relative to the state. The action expression maps into the ActionSequence and Guard for the Transition. The event name and arguments map into an Event corresponding to the event name and arguments. The Event plays the role of a *trigger* to the Transition.

3.76 Composite States

3.76.1 Semantics

A composite state is decomposed into two or more concurrent substates (called *regions*) or into mutually exclusive disjoint substates. A given state may only be refined in one of these two ways. Naturally, any substate of a composite state can also be a composite state of either type.

A newly-created object takes its topmost default transition, originating from the topmost initial pseudostate. An object that transitions to its outermost final state is terminated.

Each region of a state may have initial pseudostates and final states. A transition to the enclosing state represents a transition to the initial pseudostate. A transition to a final state represents the completion of activity in the enclosing region. Completion of activity in all concurrent regions represents completion of activity by the enclosing state and triggers a completion event on the enclosing state. Completion of the top state of an object corresponds to its termination.

3.76.2 Notation

An expansion of a state shows its internal state machine structure. In addition to the (optional) name and internal transition compartments, the state may have an additional compartment that contains a region holding a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic region.

An expansion of a state into concurrent substates is shown by tiling the graphic region of the state using dashed lines to divide it into regions. Each region is a concurrent substate. Each region may have an optional name and must contain a nested state diagram with disjoint states. The text compartments of the entire state are separated from the concurrent substates by a solid line. It is also possible to use a tab notation to place the name of a concurrent state. The tab notation is more space efficient.

An expansion of a state into disjoint substates is shown by showing a nested state diagram within the graphic region.

An initial pseudostate is shown as a small solid filled circle. In a top-level state machine, the transition from an initial pseudostate may be labeled with the event that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any transition to the enclosing state. The initial transition may have an action.

A final state is shown as a circle surrounding a small solid filled circle (a bull’s eye). It represents the completion of activity in the enclosing state and it triggers a transition on the enclosing state labeled by the implicit activity completion event (usually displayed as an unlabeled transition), if such a transition is defined.

In some cases, it is convenient to hide the decomposition of a composite state. For example, the state machine inside a composite state may be very large and may simply not fit in the graphical space available for the diagram. In that case, the composite state may be represented by a simple state graphic with a special “composite” icon, usually in the lower right-hand corner. This icon, consisting of two horizontally placed and connected states, is an *optional* visual cue that the state has a decomposition that is not shown in this particular statechart diagram (Figure 3-74 on page 3-141). Instead, the contents of the composite state are shown in a separate diagram. Note that the “hiding” here is purely a matter of graphical convenience and has no semantic significance in terms of access restrictions.

3.76.3 Examples

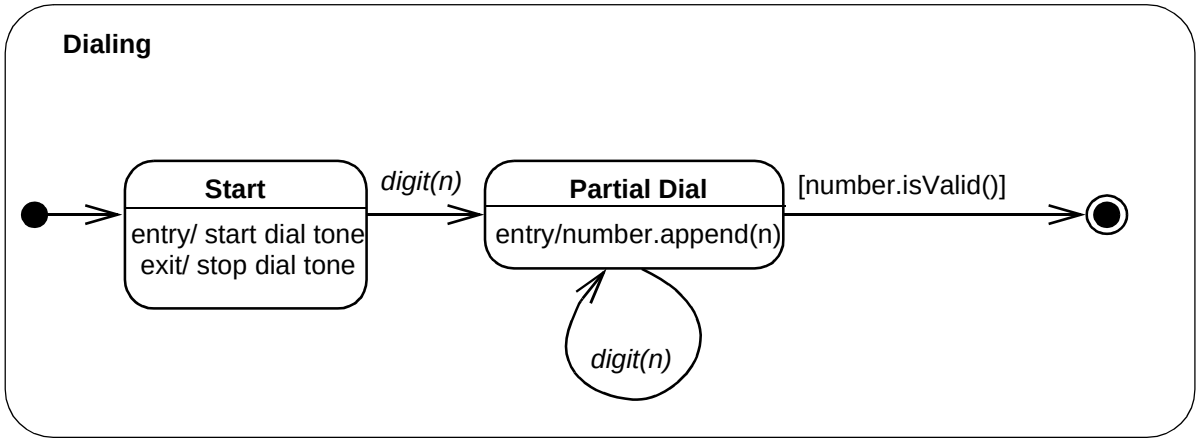


Figure 3-73 Sequential Substates

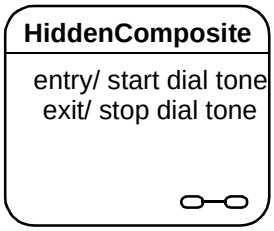


Figure 3-74 Composite State with hidden decomposition indicator icon

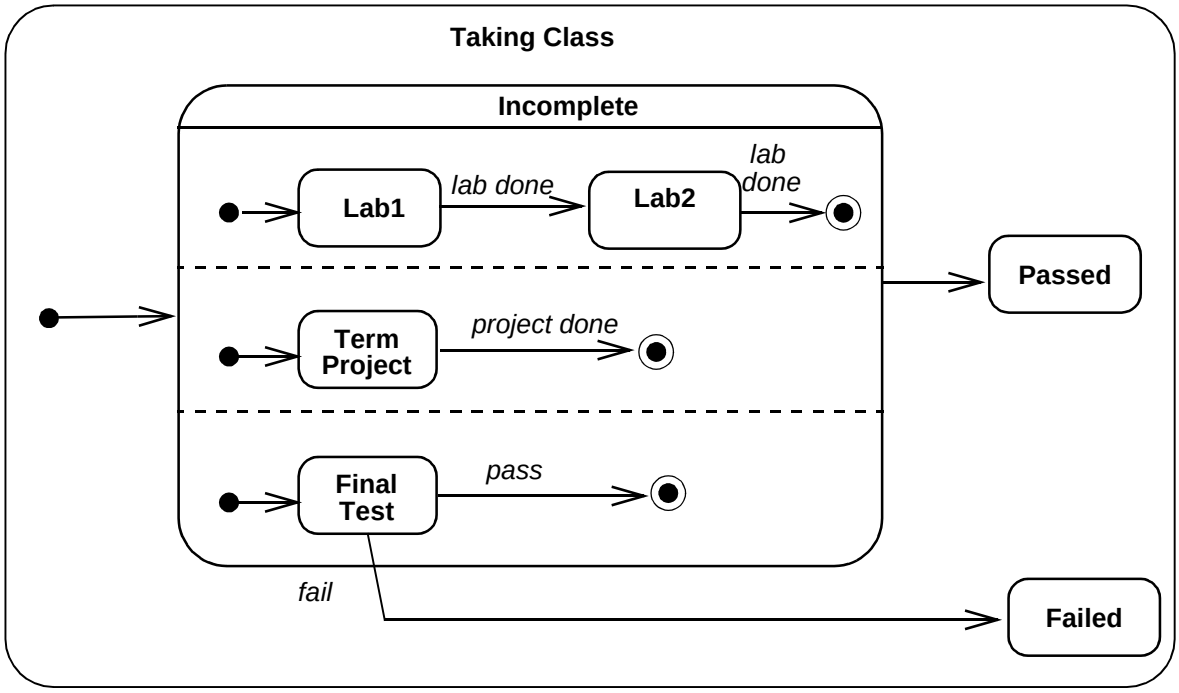


Figure 3-75 Concurrent Substates

3.76.4 Mapping

A state symbol maps into a State. If the symbol has no subdiagrams in it, it maps into a SimpleState. If it is tiled by dashed lines into regions, then it maps into a CompositeState with the *isConcurrent* value true; otherwise, it maps into a CompositeState with the *isConcurrent* value false. A region maps into a CompositeState with the *isRegion* value true and the *isConcurrent* value false.

An initial pseudostate symbol maps into a Pseudostate of kind *initial*. A final state symbol maps to a *final* state.

3.77 Events

3.77.1 Semantics

An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. Events may be of several kinds (not necessarily mutually exclusive).

- A designated condition becoming true (described by a Boolean expression) results in a change event instance. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition. A guard condition is evaluated *once* whenever its event fires. If it is false, then the transition does not occur and the event is lost.
- The receipt of an explicit signal from one object to another results in a signal event instance. It is denoted by the signature of the event as a trigger on a transition.
- The receipt of a call for an operation implemented as a transition by an object represents a call event instance.
- The passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time is a TimeEvent.

The event declaration has scope within the package it appears in and may be used in state diagrams for classes that have visibility inside the package. An event is *not* local to a single class.

3.77.2 Notation

A signal or call event can be defined using the following format:

event-name '(' *comma-separated-parameter-list* ')'

A parameter has the format:

parameter-name ':' *type-expression*

A signal can be declared using the «signal» keyword on a class symbol in a class diagram. The parameters are specified as attributes. A signal can be specified as a subclass of another signal. This indicates that an occurrence of the subevent triggers any transition that depends on the event or any of its ancestors.

An elapsed-time event can be specified with the keyword **after** followed by an expression that evaluates (at modeling time) to an amount of time, such as "**after** (5 seconds)" or **after** (10 seconds since exit from state A)." If no starting point is indicated, then it is the time since the entry to the current state. Other time events can be specified as conditions, such as **when** (date = Jan. 1, 2000).

A condition becoming true is shown with the keyword **when** followed by a Boolean expression. This may be regarded as a continuous test for the condition until it is true, although in practice it would only be checked on a change of values.

Signals can be declared on a class diagram with the keyword «signal» on a rectangle symbol. These define signal names that may be used to trigger transitions. Their parameters are shown in the attribute compartment. They have no operations. They may appear in a generalization hierarchy.

3.77.3 Example

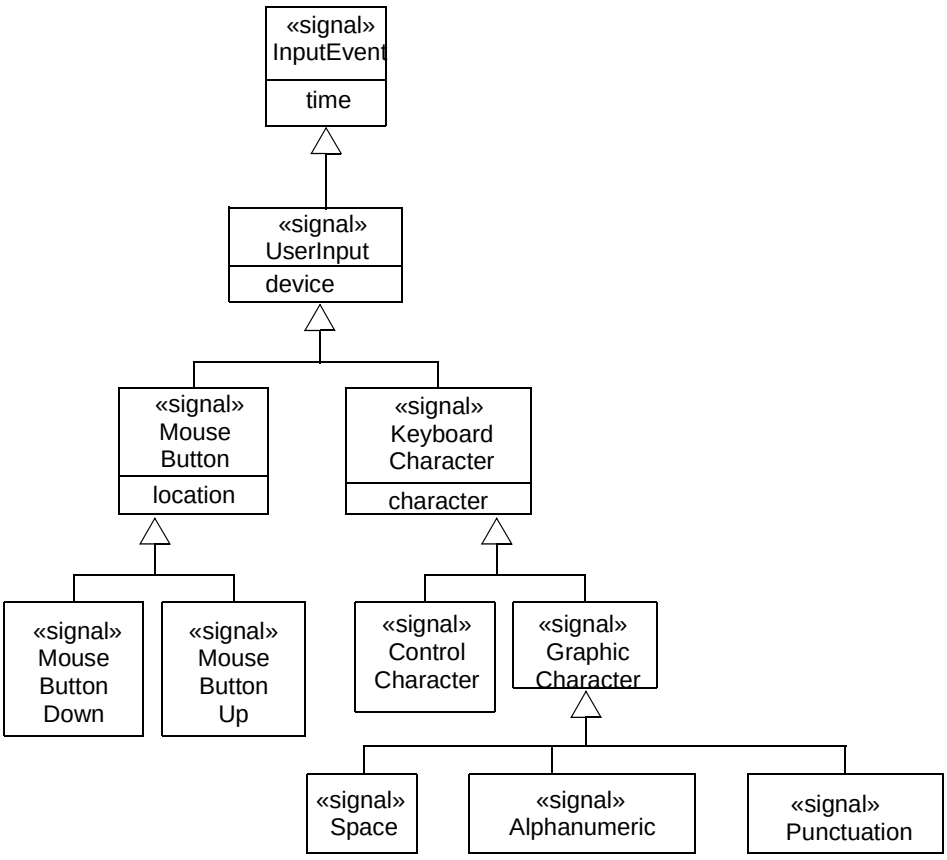


Figure 3-76 Signal Declaration

3.77.4 Mapping

A class box with stereotype «signal» maps into a Signal. The name and parameters are given by the name string and the attribute list of the box. Generalization arrows between signal class boxes map into Generalization relationships between the Signal.

The usage of an event string expression in a context requiring an event maps into an implicit reference of the Event with the given name. It is an error if various uses of the same name (including any explicit declarations) do not match.

3.78 Simple Transitions

3.78.1 Semantics

A simple transition is a relationship between two states indicating that an instance in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain specified conditions are satisfied. On such a change of state, the transition is said to “fire.” The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. Events are processed one at a time. If an event does not trigger any transition, it is discarded. If it can trigger more than one transition within the same sequential region; that is, not in different concurrent regions, only one will fire. If these conflicting transitions are of the same priority, an arbitrary one is selected and triggered.

3.78.2 Notation

A transition is shown as a solid line originating from the *source* state and terminated by an arrow on the *target* state. It may be labeled by a *transition string* that has the following general format:

event-signature ‘[’ guard-condition ‘]’ ‘/’ *action-expression*

The *event-signature* describes an event with its arguments:

event-name ‘(’ *comma-separated-parameter-list* ‘)’

The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. The guard condition may also involve tests of concurrent states of the current machine, or explicitly designated states of some reachable object (for example, “**in** State1” or “**not in** State2”). State names may be fully qualified by the nested states that contain them, yielding pathnames of the form “State1::State2::State3.” This may be used in case same state name occurs in different composite state regions of the overall machine.

The *action-expression* is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the owning object and the parameters of the triggering event, or any other features visible in its scope. The corresponding action must be executed entirely before any other actions are considered. This model of execution is referred to as *run-to-completion* semantics. The action expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending signals or invoking operations. The details of this expression are dependent on the action language chosen for the model.

3.78.2.1 Transition times

Names may be placed on transitions to designate the times at which they fire. See Section 3.64, “Transition Times,” on page 3-113.

3.78.3 Example

```
right-mouse-down (location) [location in window] / object := pick-object (location);
object.highlight ()
```

The event may be any of the standard event types. Selecting the type depends on the syntax of the name (for time events, for example); however, SignalEvents and CallEvents are not distinguishable by syntax and must be discriminated by their declaration elsewhere.

3.78.4 Mapping

A transition string and the transition arrow that it labels together map into a Transition and its attachments. The arrow connects two state symbols. The Transition has the corresponding States as its source (the state at the tail) and destination (the state at the head) States in associations to the Transition.

The event name and parameters map into an Event element, which may be a SignalEvent, a CallEvent, a TimeExpression (if it has the proper syntax), or a ChangeEvent (if it is expressed as a Boolean expression). The event is attached as a "trigger" role in the association to the transition.

The guard condition maps into a Guard element attached to the Transition. Note that a guard condition is distinguished graphically from a change event specification by being enclosed in brackets.

An action expression maps into an Action attached as an "effect" role relative to the Transition.

3.79 Transitions to and from Concurrent States

A concurrent transition may have multiple source states and target states. It represents a synchronization and/or a splitting of control into concurrent threads without concurrent substates.

3.79.1 Semantics

A concurrent transition is enabled when all the source states are occupied. After a compound transition fires, all its destination states are occupied.

3.79.2 Notation

A concurrent transition includes a short heavy bar (a *synchronization* bar, which can represent synchronization, forking, or both). The bar may have one or more arrows from states to the bar (these are the *source states*). The bar may have one or more arrows from the bar to states (these are the *destination states*). A transition string may be shown near the bar. Individual arrows do not have their own transition strings.

3.79.3 Example

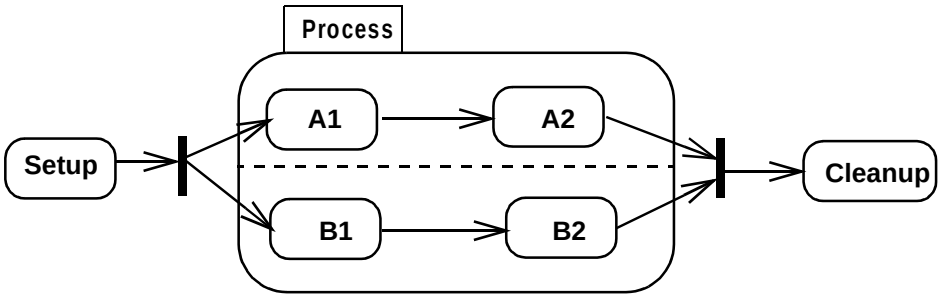


Figure 3-77 Concurrent Transitions

3.79.4 Mapping

A bar with multiple transition arrows leaving it maps into a fork pseudostate. A bar with multiple transition arrows entering it maps into a join pseudostate. The transitions corresponding to the incoming and outgoing arrows attach to the pseudostate as if it were a regular state. If a bar has multiple incoming and multiple outgoing arrows, then it maps into a join connected to a fork pseudostate by a single transition with no attachments.

3.80 Transitions to and from Composite States

3.80.1 Semantics

A transition drawn to the boundary of a composite state is equivalent to a transition to its initial point (or to a complex transition to the initial point of each of its concurrent regions, if it is concurrent). The entry action is always performed when a state is entered from outside.

A transition from a composite state indicates a transition that applies to each of the states within the state region (at any depth). It is “inherited” by the nested states. Inherited transitions can be masked by the presence of nested transitions with the same trigger.

3.80.2 Notation

A transition drawn to a composite state boundary indicates a transition to the composite state. This is equivalent to a transition to the initial pseudostate within the composite state region. The initial pseudostate must be present. If the state is a concurrent composite state, then the transition indicates a transition to the initial pseudostate of each of its concurrent substates.

Transitions may be drawn directly to states within a composite state region at any nesting depth. All entry actions are performed for any states that are entered on any transition. On a transition within a concurrent composite state, transition arrows from the synchronization bar may be drawn to one or more concurrent states. Any other concurrent regions start with their default initial pseudostate.

A transition drawn from a composite state boundary indicates a transition of the composite state. If such a transition fires, any nested states are forcibly terminated and perform their exit actions, then the transition actions occur and the new state is established.

Transitions may be drawn directly from states within a composite state region at any nesting depth to outside states. All exit actions are performed for any states that are exited on any transition. On a transition from within a concurrent composite state, transition arrows may be specified from one or more concurrent states to a synchronization bar; therefore, specific states in the other regions are irrelevant to triggering the transition.

A state region may contain a *history state indicator* shown as a small circle containing an 'H.' The history indicator applies to the state region that directly contains it. A history indicator may have any number of incoming transitions from outside states. It may have at most one outgoing unlabeled transition. This identifies the default "previous state" if the region has never been entered. If a transition to the history indicator fires, it indicates that the object resumes the state it last had within the composite region. Any necessary entry actions are performed. The history indicator may also be 'H*' for *deep history.* This indicates that the object resumes the state it last had at any depth within the composite region, rather than being restricted to the state at the same level as the history indicator. A region may have both shallow and deep history indicators.

3.80.3 Presentation Options

3.80.3.1 Stubbed transitions

Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state. Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial pseudostate may (but need not) be shown as coming from or going to *stubs*. A *stub* is shown as a small vertical line (bar) drawn inside the boundary of the enclosing state. It indicates a transition connected to a suppressed internal state. Stubs are not used for transitions to initial or from final states.

Note that events should be shown on transitions leading into a state, either to the state boundary or to an internal substate, including a transition to a stubbed state. Normally events should not be shown on transitions leading from a stubbed state to an external state. Think of a transition as belonging to its source state. If the source state is suppressed, then so are the details of the transition. Note also that a transition from a final state is summarized by an unlabeled transition from the composite state contour (denoting the implicit event "action complete" for the corresponding state).

3.80.4 Example

See Figure 3-76 on page 3-144 and Figure 3-77 on page 3-147 for examples of composite transitions. The following are examples of stubbed transitions and the history indicator.

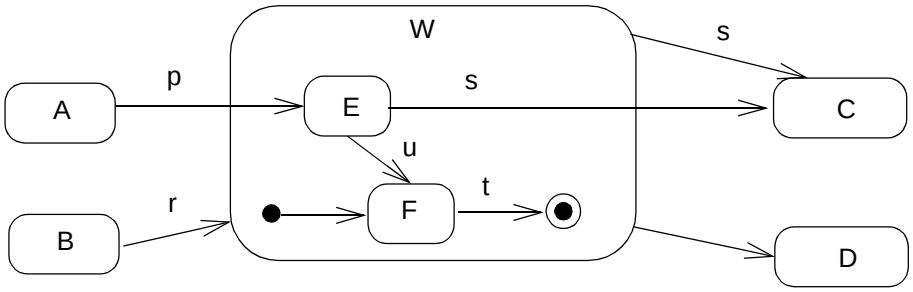


Figure 3-78 Stubbed Transitions

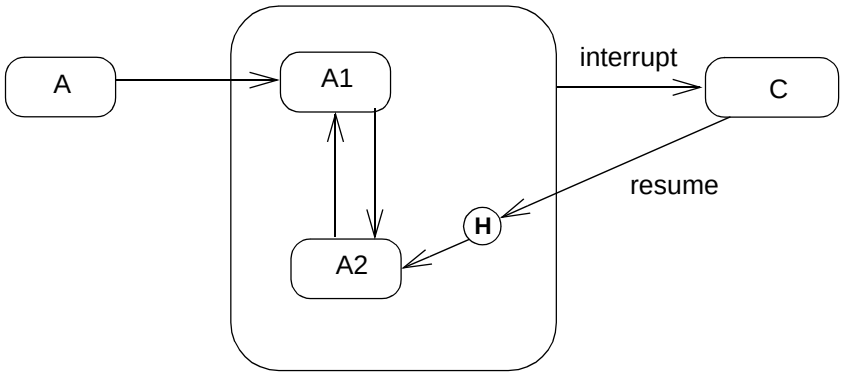


Figure 3-79 History Indicator

3.80.5 Mapping

An arrow to any state boundary, nested or not, maps into a Transition between the corresponding States and similarly for transitions directly to history states.

A history indicator maps into a Pseudostate of kind *shallowHistory* or *deepHistory.*

A stubbed transition does not map into anything in the model. It is a notational elision that indicates the presence of transitions to additional states in the model that are not visible in the diagram.

3.81 Factored Transition Paths

3.81.1 Semantics

By definition, a transition connects exactly two vertices in the state machine graph. However, since some of these vertices may be pseudostates—which are transient in nature—there is a need for describing chains of transitions that may be executed in the context of a single run-to-completion step. Such a transition is known as a *compound transition*.

As a practical measure, it is often useful to share segments of a compound transition. For example, two or more distinct compound transitions may come together and continue via a common path, sharing its action, and possibly terminating on the same target state. In other cases, it may be useful to split a transition into separate mutually exclusive; that is, non-concurrent paths.

Both of these examples of graphical factoring in which some transitions are shared result in simplified diagrams. However, factoring is also useful for modeling dynamically adaptive behavior. An example of this occurs when a single event may lead to any of a set of possible target states, but where the final target state is only determined as the result of an action (calculation) performed after the triggering of the compound transition.

Note that the splitting and joining of paths due to factoring is different from the splitting and joining of concurrent transitions described in Sect i on3.79, “Transitions to and from Concurrent States,” on page 3-146. The sources and targets of these factored transitions are not concurrent.

3.81.2 Notation

Two or more transitions emanating from different non-concurrent states or pseudostates can terminate on a common junction point. This allows their respective compound transitions to share the path that emanates from that junction point. A junction point is represented by a small black circle. Alternatively, it may be represented by a diamond shape (see Section 3.87, “Decisions,” on page 3-159).

Two or more guarded transitions emanating from the same junction point represent a *static branch point*. Normally, the guards are mutually exclusive. This is equivalent to a set of individual transitions, one for each path through the tree, whose guard

condition is the “and” of all of the conditions along the path. Note that the semantics of static branches is that all the outgoing guards are evaluated *before* any transition is taken.

Two or more guarded transitions emanating from a common *dynamic choice point* are used to model dynamic choices. In this case, the guards of the outgoing transitions are evaluated at the time the choice point has been reached. The value of these guards may be a function of some calculations performed in the actions of the incoming transition (s). A dynamic choice point is represented by a small white circle (reminiscent of a small state icon).

3.81.3 Examples

In Figure 3-80 a single junction point is used to merge and split transitions. Regardless of whether the junction point was reached from state State0 or from state State1, the outgoing paths are the same for both cases.

If the state machine in this example is in state State1 and b is less than 0 when event e1 occurs, the outgoing transition will be taken only if one of the three downstream guards is true. Thus, if a is equal to 6 at that point, no transition will be triggered.

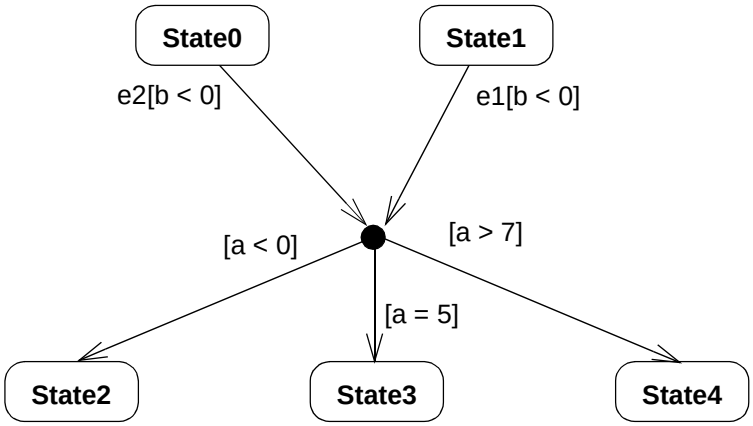


Figure 3-80 Junction points

In the dynamic choice point example in Figure 3-81 on page 3-152, the decision on which branch to take is only made after the transition from State1 is taken and the choice point is reached. Note that the action associated with that incoming transition computes a new value for a. This new value can then be used to determine the outgoing transition to be taken. The use of the predefined condition[else] is recommended to avoid run-time errors.

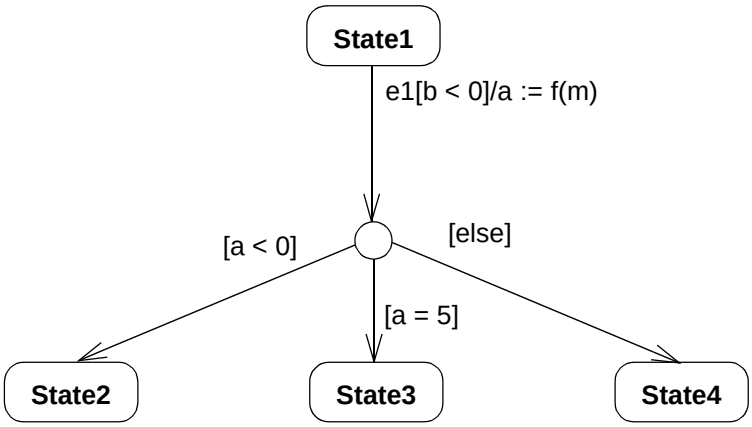


Figure 3-81 Dynamic choice points

3.82 Submachine States

3.82.1 Semantics

A submachine state represents the *invocation* of a state machine defined elsewhere. It is similar to a macro call in the sense that it represents a (graphical) shorthand that implies embedding of a complex specification within another specification. The submachine must be contained in the same context as the invoking state machine.

In the general case, an invoked state machine can be entered at any of its substates or through its default (initial) pseudostate. Similarly, it can be exited from any substate or as a result of the invoked state machine reaching its final state or by an “inherited” or “group” transition that applies to all substates in the submachine.

The non-default entry and exits are specified through special stub states.

3.82.2 Notation

The submachine state is depicted as a normal state with the appropriate “include” declaration within its internal transitions compartment (see Section 3.75, “State,” on page 3-137). The expression following the include reserved word is the name of the invoked submachine.

Optionally, the submachine state may contain one or more entry stub states and one or more exit stub states. The notation for these is similar to that used for stub ends of stubbed transitions, except that the ends are labeled. The labels represent the names of the corresponding substates within the invoked submachine. A pathname may be used if the substate is not defined at the top level of the invoked submachine. Naturally, this name must be a valid name of a state in the invoked state machine.

If the submachine is entered through its default pseudostate or if it is exited as a result of the completion of the submachine, it is not necessary to use the stub state notation for these cases. Similarly, a stub state is not required if the exit occurs through an explicit “group” transition that emanates from the boundary of the submachine state (implying that it applies to all the substates of the submachine).

Submachine states invoking the same submachine may occur multiple times in the same state diagram with different entry and exit configurations and with different internal transitions and exit and entry action specifications in each case.

3.82.3 Example

The following diagram shows a fragment from a statechart diagram in which a submachine (the FailureSubmachine) is invoked in a particular way. The actual submachine is presumably defined elsewhere and is not shown in this diagram. Note that the same submachine could be invoked elsewhere in the same statechart diagram with different entry and exit configurations.

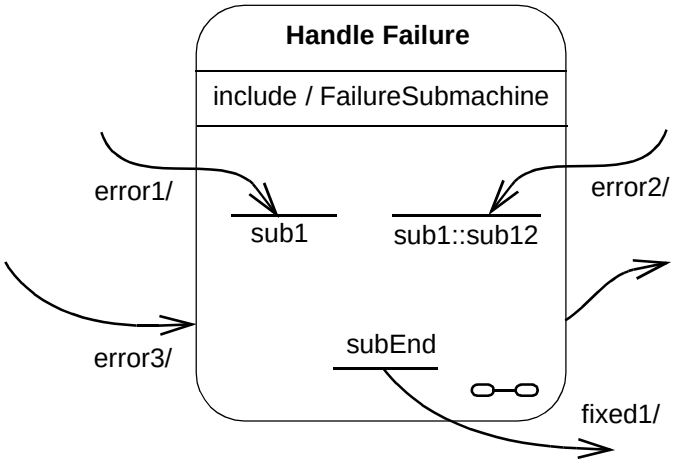


Figure 3-82 Submachine State

In the above example, the transition triggered by event “error1” will terminate on state “sub1” of the FailureSubmachine state machine. Since the entry point does not contain a path name, this means that “sub1” is defined at the top level of that submachine. In contrast, the transition triggered by “error2” will terminate on the “sub12” substate of the “sub1”substate (as indicated by the path name), while the “error3” transition implies taking of the default transition of the FailureSubmachine.

The transition triggered by the event “fixed1” emanates from the “subEnd” substate of the submachine. Finally, the transition emanating from the edge of the submachine state is taken as a result of the completion event generated when the FailureSubmachine reaches its final state.

3.82.4 Mapping

A submachine state in a statechart diagram maps directly to a SubmachineState in the metamodel. The name following the “include” reserved action label represents the state machine indicated by the “submachine” attribute. Stub states map to the Stub State concept in the metamodel. The label on the diagram corresponds to the pathname represented by the “referenceState” attribute of the stub state.

3.83 Synch States

3.83.1 Semantics

A synch state is for synchronizing concurrent regions of a state machine. It is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states. The firing of outgoing transitions from a synch state can be limited by specifying a bound on the difference between the number of times outgoing and incoming transitions have fired.

3.83.2 Notation

A synch state is shown as a small circle with the upper bound inside it. The bound is either a positive integer or an asterisk ('*') for unlimited. Synch states are drawn on the boundary between two regions when possible.

3.83.3 Example

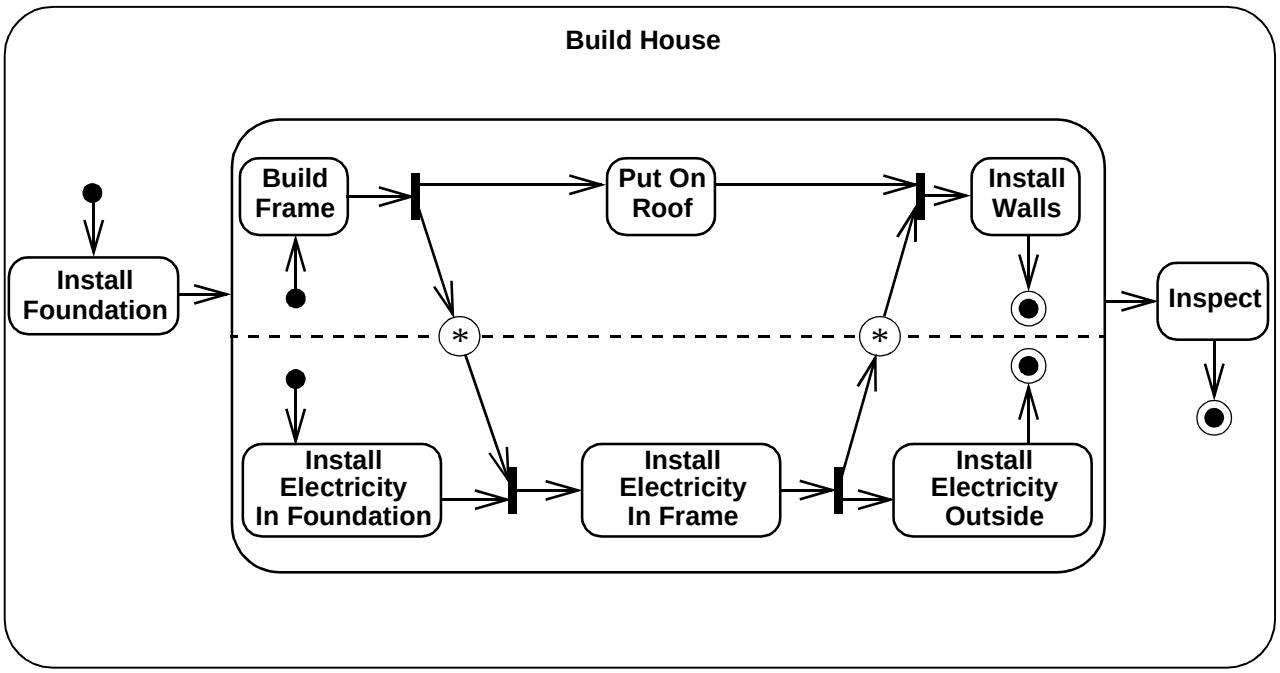


Figure 3-83 Synch states

3.83.4 Mapping

A synch state circle maps into a SynchState, contained by the least common containing state of the regions it is synchronizing. The number inside it maps onto the bound attribute of the synch state. A star ('*') inside the synch state circle maps to a value of Unlimited for the bound attribute.

Part 10 - Activity Diagrams

3.84 Activity Diagram

3.84.1 Semantics

An activity graph is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. It represents a state machine of a computation itself.

3.84.2 Notation

An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states. The entire activity diagram is attached (through the model) to a classifier, such as a use case, or to a package, or to the implementation of an operation. The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions (that is, procedural flow of control). Use ordinary state diagrams in situations where asynchronous events occur.

3.84.3 Example

Person::Prepare Beverage

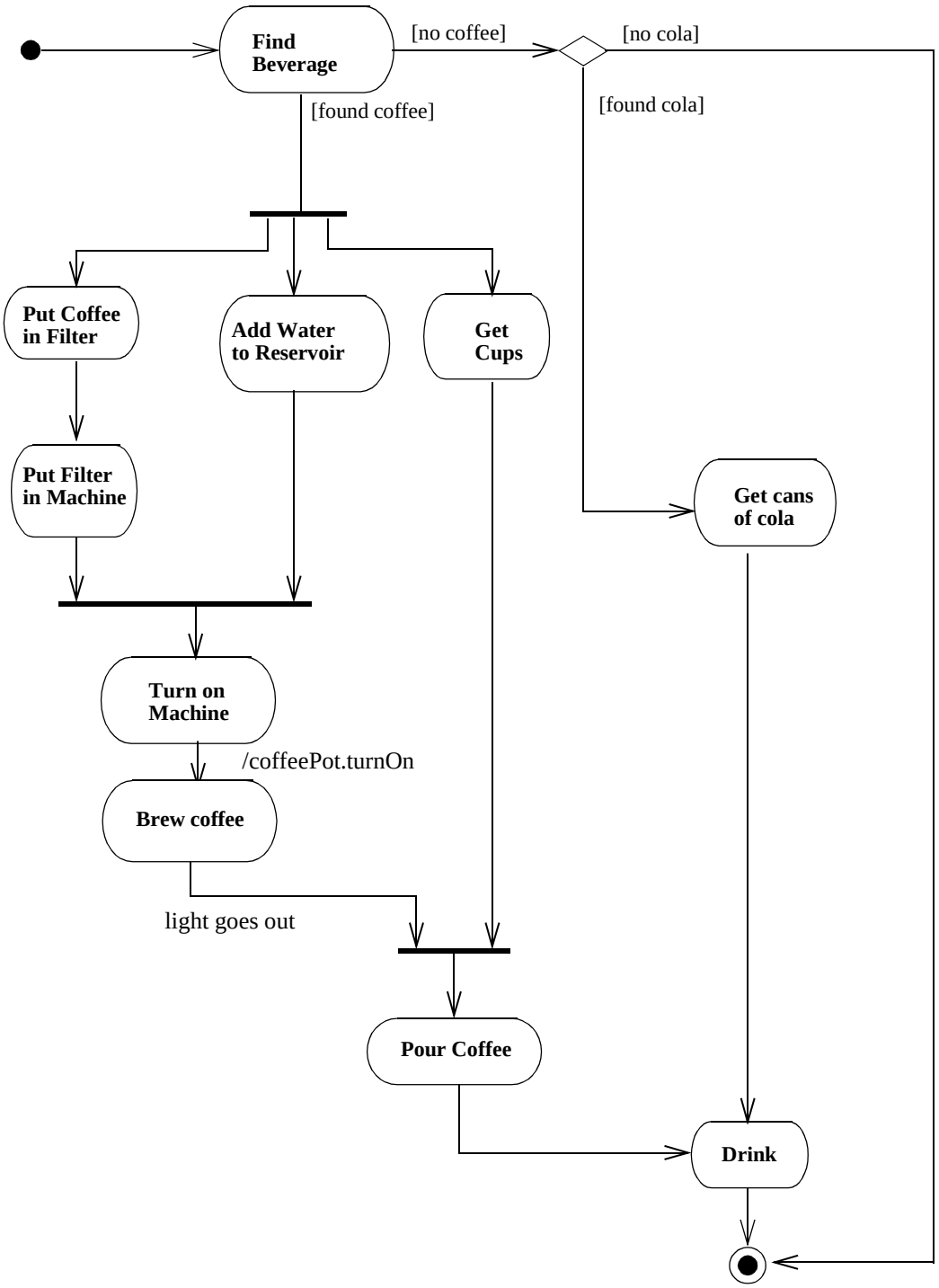


Figure 3-84 Activity Diagram

3.84.4 Mapping

An activity diagram maps into an ActivityGraph.

3.85 Action state

3.85.1 Semantics

An *action state* is a shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action (there may be several such transitions if they have guard conditions). Action states should not have internal transitions, outgoing transitions based on explicit events, or exit actions, use normal states for this situation. Transitions leaving an action state should not include an event signature. Such transitions are implicitly triggered by the completion of the action in the state. The transitions may include guard conditions and actions. A common use of an action state is to model a step in the execution of a workflow process.

3.85.2 Notation

An action state is shown as a shape with straight top and bottom and with convex arcs on the two sides. The *action-expression* is placed in the symbol. The action expression need not be unique within the diagram.

3.85.3 Presentation options

The action may be described by natural language, pseudocode, action language, or programming language code. It may use only attributes and links of the owning object.

Note that action state notation may be used within ordinary state diagrams; however, they are more commonly used with activity diagrams, which are special cases of state diagrams.

3.85.4 Example

Figure 3-85 Action States

3.85.5 Mapping

An action state symbol maps into an ActionState with the action-expression mapped to either the body of the entry action procedure of the State, or to a detailed action model within the procedure. The State is normally anonymous.

3.86 Subactivity state

3.86.1 Semantics

A *subactivity state* invokes an activity graph. When a subactivity state is entered, the activity graph "nested" in it is executed as any activity graph would be. The subactivity state is not exited until the final state of the nested graph is reached, or when trigger events occur on transitions coming out of the subactivity state. Since states in activity graphs do not normally have trigger events, subactivity states are normally exited when their nested graph is finished. A single activity graph may be invoked by many subactivity states.

3.86.2 Notation

A subactivity state is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol. The subactivity need not be unique within the diagram.

This notation is applicable to any UML construct that supports "nested" structure. The icon must suggest the type of nested structure.

3.86.3 Example

Figure 3-86 Subactivity States

3.86.4 Mapping

A subactivity state symbol maps into a SubactivityState. The name of the subactivity maps to a submachine link between the SubactivityState and an ActivityGraph of that name. The SubactivityState is normally anonymous.

3.87 Decisions

3.87.1 Semantics

A state diagram (and by derivation an activity diagram) expresses a decision when guard conditions are used to indicate different possible transitions that depend on Boolean conditions of the owning object. UML provides a shorthand for showing decisions and merging their separate paths back together. Each possible outcome

should appear on one of the outgoing transitions. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.

3.87.2 Notation

A decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger.

The same icon can be used to merge decision branches back together, in which case it is called a merge. A merge has two or more incoming arrows and one outgoing arrow.

Note that a chain of decisions may be part of a complex transition, but only the first segment in such a chain may contain an event trigger label. All segments may have guard expressions. The transition coming from a merge may not have a trigger label or guard expressions.

3.87.3 Example

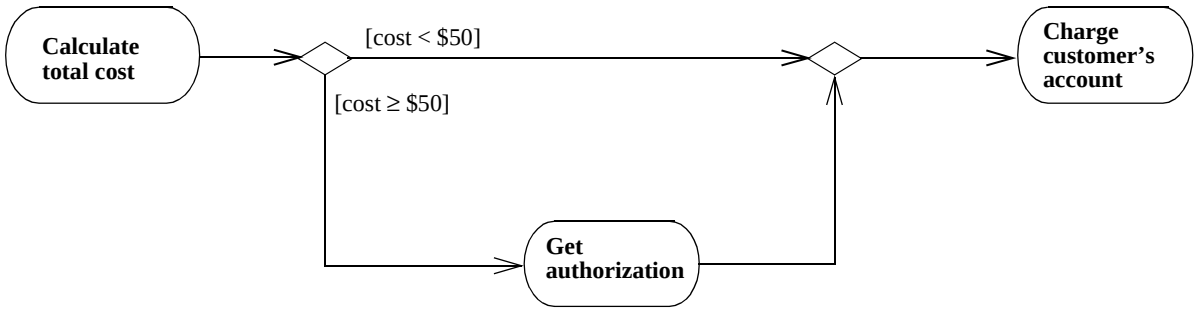


Figure 3-87 Decision and merge

3.87.4 Mapping

A decision symbol maps into a Pseudostate of kind *junction*. Each label on an outgoing arrow maps into a Guard on the corresponding Transition leaving the Pseudostate. A merge symbol maps also maps into a Pseudostate of kind *junction*.

3.88 Call States

3.88.1 Semantics

A call state is an action state that calls a single operation. It is useful in object flow modeling to reduce notational ambiguity over which action is taking input or providing output.

3.88.2 Notation

A call state is shown in the same way as an action state, except that the name of the operation being called is put in the symbol, along with the name of the classifier that hosts the operation in parentheses under it.

3.88.3 Example

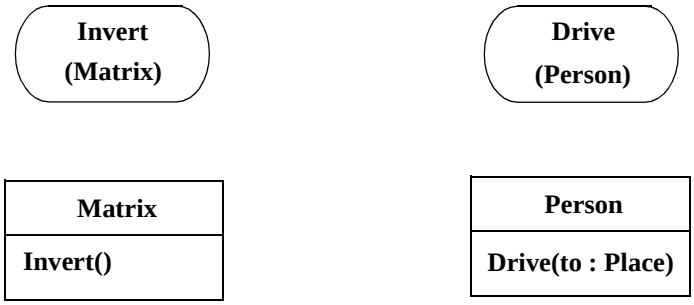


Figure 3-88 Call states and the operations they invoke

3.88.4 Mapping

The top name maps into the operation being called in the entry action of the call state. The name in parentheses maps into the classifier hosting the operation.

3.89 Swimlanes

3.89.1 Semantics

Actions and subactivities may be organized into *swimlanes.* Swimlanes are used to organize responsibility for actions and subactivities. They often correspond to organizational units in a business model.

3.89.2 Notation

An activity diagram may be divided visually into “swimlanes,” each separated from neighboring swimlanes by vertical solid lines on both sides. The relative ordering of the swimlanes has no semantic significance. Each action is assigned to one swimlane. Transitions may cross lanes. There is no significance to the routing of a transition path.

3.89.3 Example

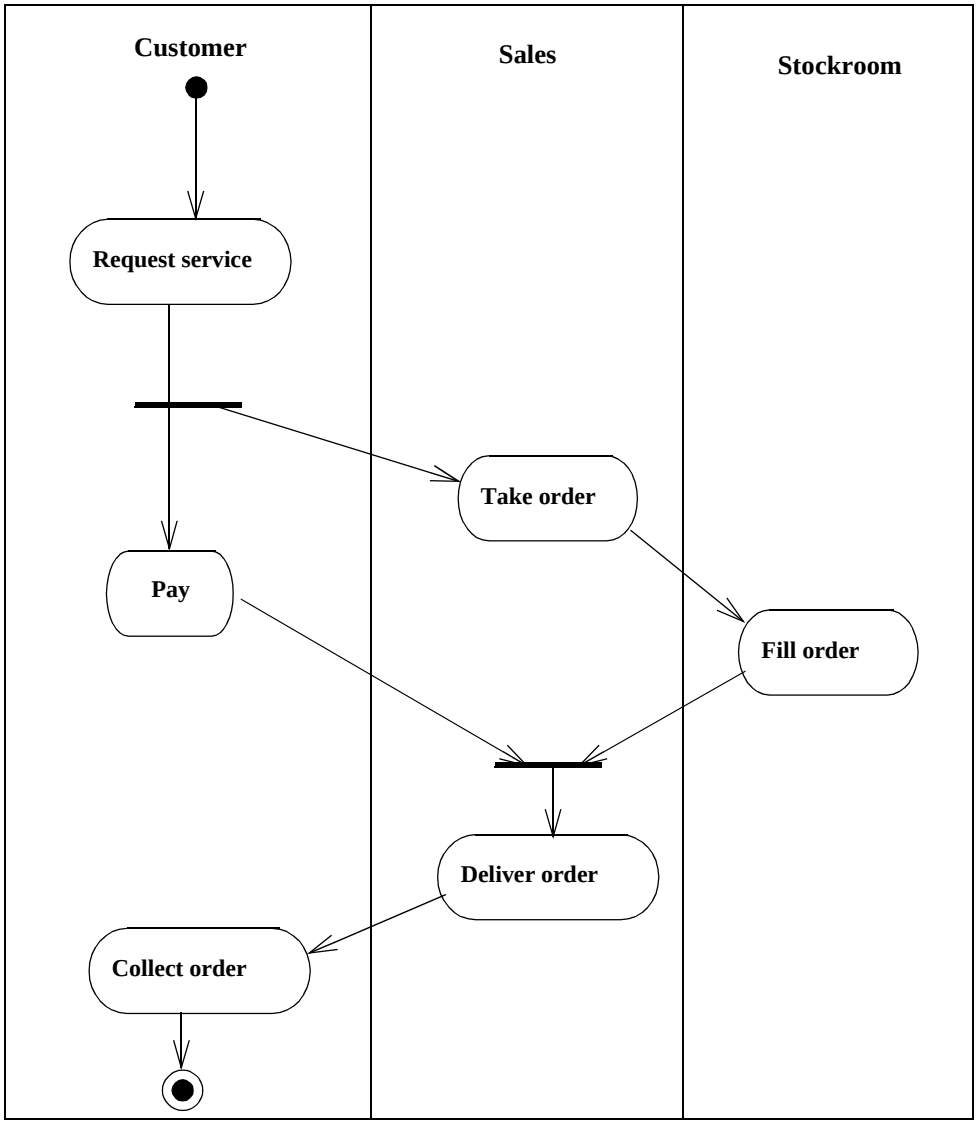


Figure 3-89 Swimlanes in Activity Diagram

3.89.4 Mapping

A swimlane maps into a Partition of the States in the ActivityGraph. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

3.90 Action-Object Flow Relationships

3.90.1 Semantics

Actions operate by and on objects. These objects either have primary responsibility for initiating an action, or are used or determined by the action. Actions usually specify calls sent between the object owning the activity graph, which initiates actions, and the objects that are the targets of the actions.

3.90.2 Notation

3.90.2.1 Object responsible for an action

In sequence diagrams, the object responsible for performing an action is shown by drawing a lifeline and placing actions on lifelines. See “Sequence Diagram” on page 3-102. Activity diagrams do not show the lifeline, but each action specifies which object performs its operation. These objects may also be related to the swimlane in some way. The actions within a swimlane can all be handled by the same object or by multiple objects.

3.90.2.2 Object flow

Objects that are input to or output from an action may be shown as object symbols. A dashed arrow is drawn from an action state to an output object, and a dashed arrow is drawn from an input object to an action state. The same object may be (and usually is) the output of one action and the input of one or more subsequent actions.

The control flow (solid) arrows must be omitted when the object flow (dashed) arrows supply a redundant constraint. In other words, when a state produces an output that is input to a subsequent state, that object flow relationship implies a control constraint.

3.90.2.3 Object in state

Frequently the same object is manipulated by a number of successive actions or subactivities. It is possible to show one object with arrows to and from all of the relevant actions and subactivities, but for greater clarity, the object may be displayed multiple times on a diagram. Each appearance denotes a different point during the object’s life. To distinguish the various appearances of the same object, the state of the object at each point may be placed in brackets and appended to the name of the object (for example, PurchaseOrder[approved]). This notation may also be used in collaboration and sequence diagrams.

3.90.3 Example

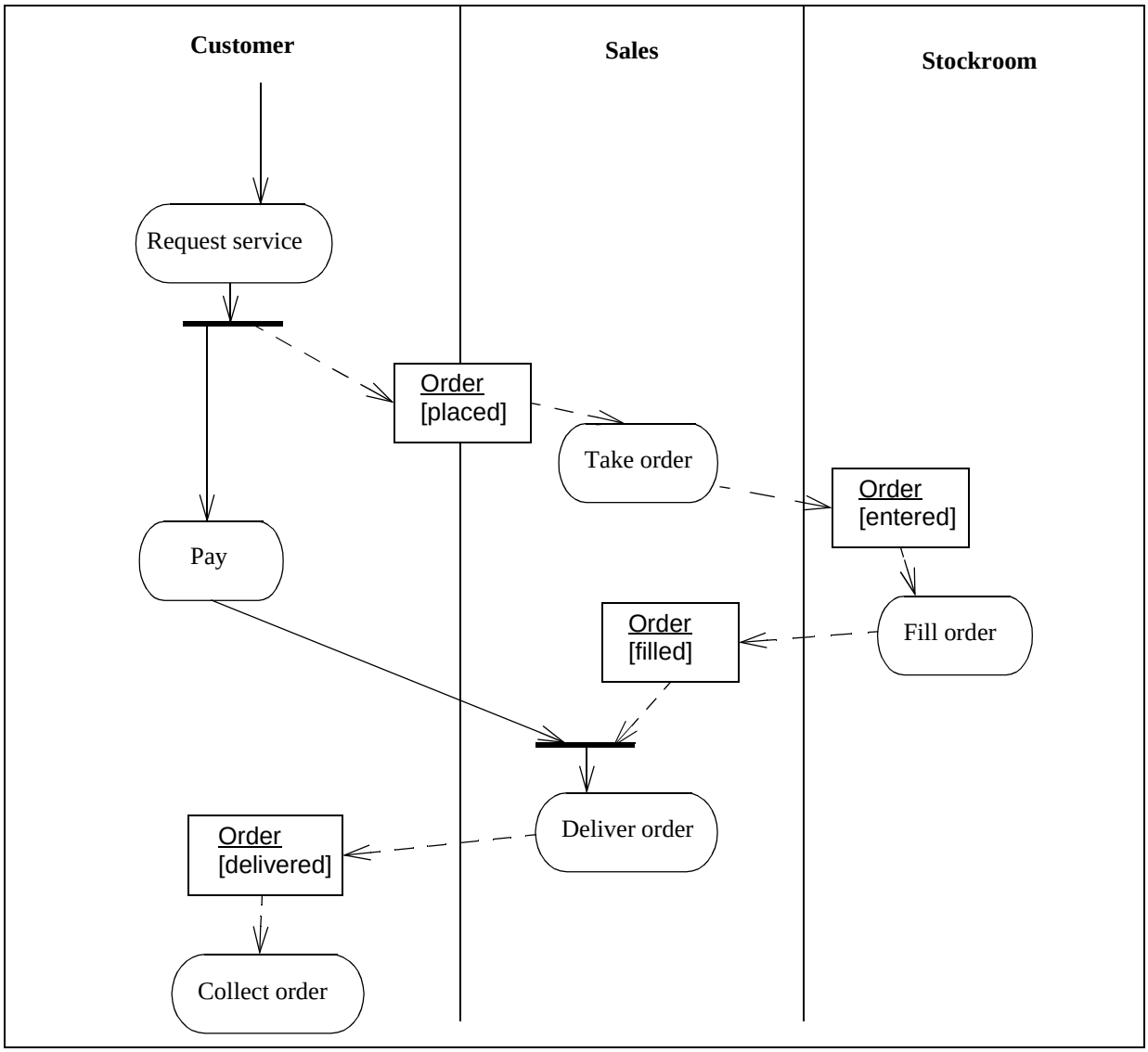


Figure 3-90 Actions and Object Flow

3.90.4 Mapping

An object flow symbol maps into an ObjectFlowState whose incoming and outgoing Transitions correspond to the incoming and outgoing arrows. The Transitions have no attachments. The classifier name and (optional) state name of the object flow symbol map into a Class or a ClassifierInState corresponding to the name(s). Solid and dashed arrows both map to transitions.

3.91 Control Icons

The following icons provide explicit symbols for certain kinds of information that can be specified on transitions. These icons are not necessary for constructing activity diagrams, but many users prefer the added impact that they provide.

3.91.1 Notation

3.91.1.1 Signal receipt

The receipt of a signal may be shown as a concave pentagon that looks like a rectangle with a triangular notch in its side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. A dashed arrow may be drawn from an object symbol to the notch on the pentagon to show the sender of the signal; this is optional.

3.91.1.2 Signal sending

The sending of a signal may be shown as a convex pentagon that looks like a rectangle with a triangular point on one side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. A dashed arrow may be drawn from the point on the pentagon to an object symbol to show the receiver of the signal, this is optional.

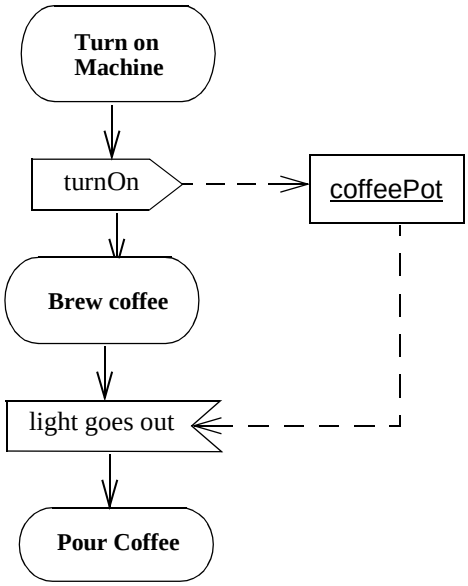


Figure 3-91 Symbols for Signal Receipt and Sending

3.91.1.3 *Deferred events*

A frequent situation is when an event that occurs must be “deferred” for later use while some other action or subactivity is underway. (Normally an event that is not handled immediately is lost.) This may be thought of as having an internal transition that handles the event and places it on an internal queue until it is needed or until it is discarded. Each state specifies a set of events that are deferred if they occur during the state and are not used to trigger a transition. If an event is not included in the set of deferrable events for a state, and it does not trigger a transition, then it is discarded from the queue even if it has already occurred. If a transition depends on an event, the transition fires immediately if the event is already on the internal queue. If several transitions are possible, the leading event in the queue takes precedence.

A deferrable event is shown by listing it within the state followed by a slash and the special operation *defer*. If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again. When the object reaches a state in which the event is not deferred, it must be accepted or lost. The indication may be placed on a composite state or its equivalents, submachine and subactivity states, in which case it remains deferrable throughout the composite state. A contained transition may still be triggered by a deferrable event, whereupon it is removed from the queue.

It is not necessary to defer events on action states, because these states are not interruptible for event processing. In this case, both deferred and undeferred events that occur during the state are deferred until the state is completed. This means that the timing of the transition will be the same regardless of the relative order of the event and the state completion, and regardless of whether events are deferred.

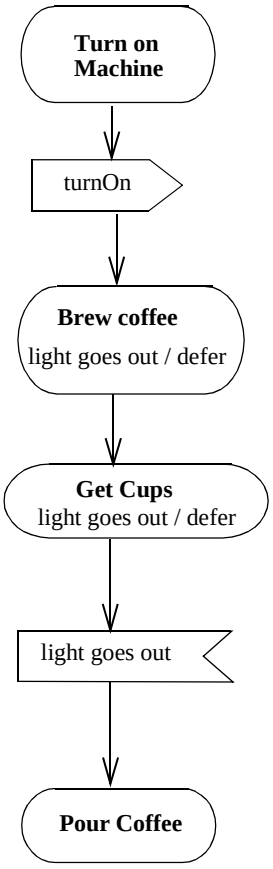


Figure 3-92 Deferred Event

3.91.2 Mapping

A signal receipt symbol maps into a state with no actions or internal transitions. Its specified event maps to a trigger event on the outgoing transition between it and the following state.

A signal send symbol maps into a procedure containing a SendSignalAction on the incoming transition between it and the previous state.

A deferred event attached to a state maps into a *deferrableEvent* association from the State to the Event.

3.92 Synch States

The SynchState notation may be omitted in Activity Diagrams when a SynchState has one incoming and one outgoing transition, and an unlimited bound. The semantics and mapping are the same as if the synch state circles were included, as defined for state machine notation.

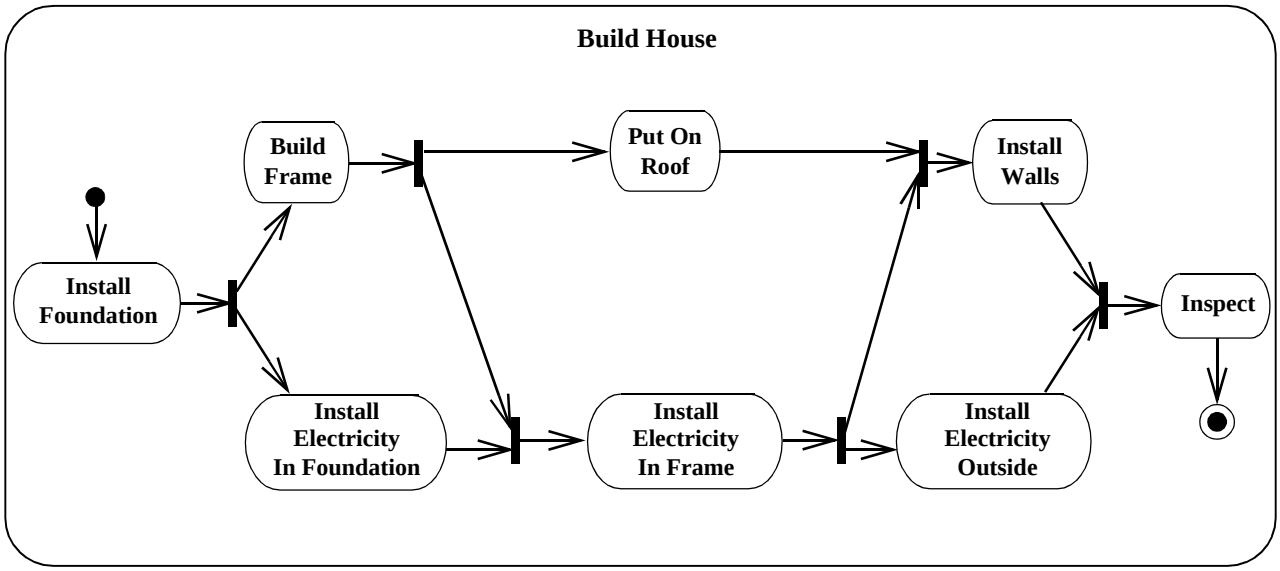


Figure 3-93 Synchronizing parallel activities

3.93 Dynamic Invocation

3.93.1 Semantics

The actions of an action state or the activity graph of a subactivity state may be executed more than once concurrently. The number of concurrent invocations is determined at runtime by a concurrency expression, which evaluates to a set of argument lists, one argument list for each invocation.

3.93.2 Notation

If the dynamic concurrency of an action or subactivity state is not always exactly one, its multiplicity is shown in the upper right corner of the state. Otherwise, nothing is shown.

3.93.3 Mapping

A multiplicity string in the upper right corner of an action or subactivity state maps to the same value in the dynamicMultiplicity attribute of the state. The presence of a multiplicity string also maps to a value of true for the isDynamic attribute of the state. If no multiplicity is present, the value of the isDynamic attribute is false.

3.94 Conditional Forks

In Activity Diagrams, transitions outgoing from forks may have guards. This means the region initiated by a fork transition might not start, and therefore is not required to complete at the corresponding join. The usual notation and mapping for guards may be used on the transition outgoing from a fork.

Part 11 - Implementation Diagrams

Implementation diagrams show aspects of physical implementation, including the structure of components and the run-time deployment system. They come in two forms: 1) component diagrams show the structure of components, including the classifiers that specify them and the artifacts that implement them; and 2) deployment diagrams show the structure of the nodes on which the components are deployed. These diagrams can also be applied in a broader way to business modeling where the components represent business procedures and artifacts, and the deployment nodes represent the organization units and resources (human and otherwise) of the business.

3.95 Component Diagram

3.95.1 Semantics

A component diagram shows the dependencies among software components, including the classifiers that specify them (for example, implementation classes) and the artifacts that implement them; such as, source code files, binary code files, executable files, scripts.

A component diagram has only a type form, not an instance form. To show component instances, use a deployment diagram (possibly a degenerate one without nodes).

3.95.2 Notation

A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.

Classifiers that specify components can be connected to them by physical containment or by a «reside» relationship, which is an instance of the metaassociation between Component and ModelElement. Likewise, artifacts that specify components can be connected to them by physical containment or by an «implement» relationship, which is an instance of the metaassociation between Component and Artifact.

A diagram containing component types may be used to show static dependencies, such as compiler dependencies between programs, which are shown as dashed arrows (dependencies) from a client component to a supplier component that it depends on in some way. The kinds of dependencies are implementation-specific and may be shown as stereotypes of the dependencies.

Although a component does not have its own features (for example, attributes, operations), it acts as a container for other classifiers that are defined with features. Components typically expose a set of interfaces, which represent the services provided by the elements that reside on the component. The diagram may show these interfaces and calling dependencies among components, using dashed arrows from components to interfaces on other components.

3.95.3 Example

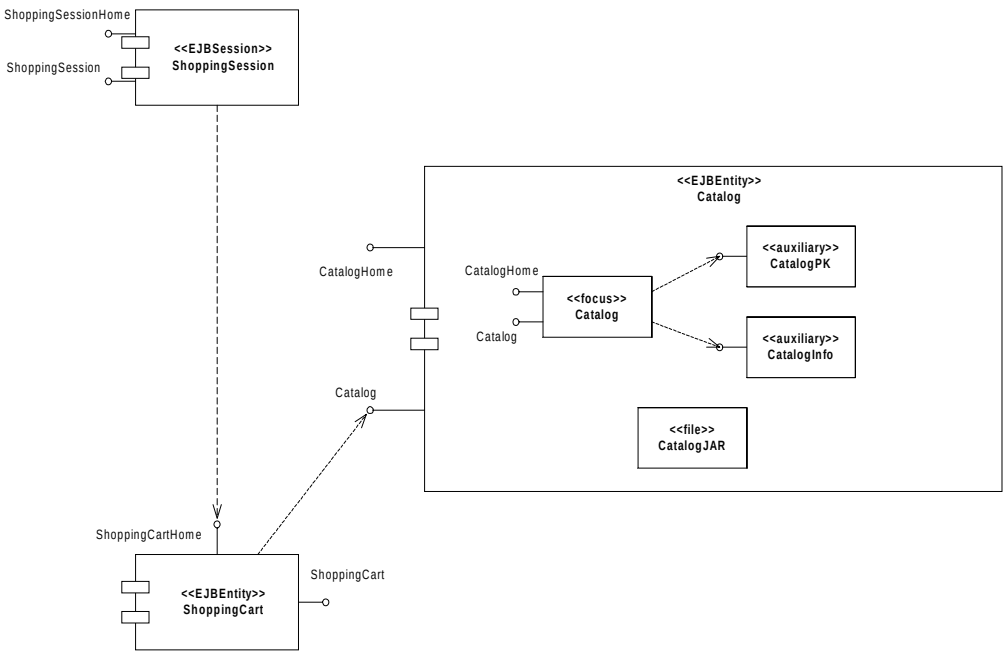


Figure 3-94 Component Diagram

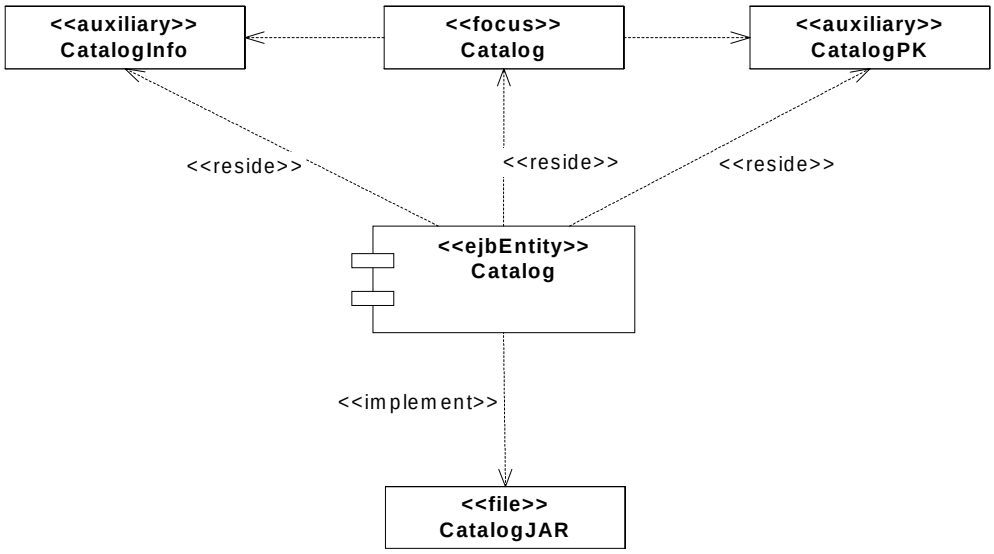


Figure 3-95 Component Diagram Showing Relationships with Classifiers and Artifacts

3.95.4 Mapping

A component diagram maps to a static model whose elements include Components. The physical containment of a Classifier by a Component represents a «reside» relationship, which is an instance of the metaassociation between Component and ModelElement. The physical containment of an Artifact by a Component represents an «implement» relationship, which is an instance of the metaassociation between Component and Artifact.

3.96 Deployment Diagram

3.96.1 Semantics

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that execute on them. Software component instances represent run-time manifestations of software code units. Components that do not exist as run-time entities (because they have been compiled away) do not appear on these diagrams, they should be shown on component diagrams.

For business modeling, the run-time processing elements include workers and organizational units, and the software components include procedures and documents used by the workers and organizational units.

3.96.2 Notation

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances. This indicates that the component runs or executes on the node. Components may contain instances of classifiers, which indicates that the instance resides on the component. Components are connected to other components by dashed-arrow dependencies (possibly through interfaces). This indicates that one component uses the services of another component. A stereotype may be used to indicate the precise dependency, if needed.

The deployment type diagram may also be used to show which components may reside on which nodes, by using dashed arrows with the stereotype «deploy» from the component symbol to the node symbol or by graphically nesting the component symbol within the node symbol.

Migration of component instances from node instance to node instance or objects from component instance to component instance may be shown using the «become» stereotype of the dependency relationship. In this case the component instance or object is resident on its node instance or component instance only part of the entire time.

Note that a process is just a special kind of object (see Section 3.71, “Active object,” on page 3-128).

3.96.3 Example

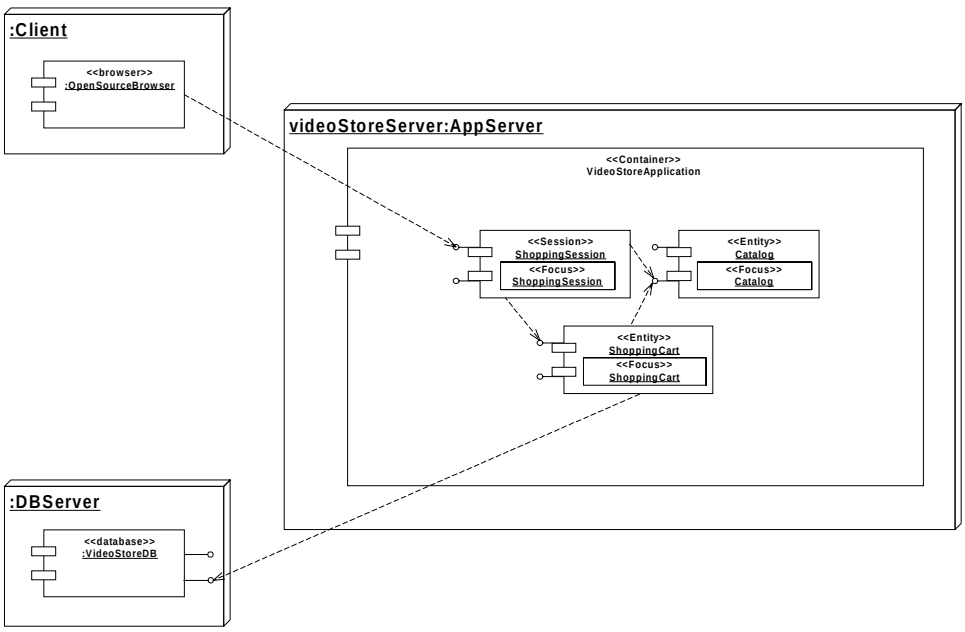


Figure 3-96 Deployment Diagram

3.96.4 Mapping

A deployment diagram maps to a static model whose elements include Nodes. It is not particularly distinguished in the model.

3.97 Node

3.97.1 Semantics

A node is a physical object that represents a processing resource, generally, having at least a memory and often processing capability as well. Nodes include computing devices but also human resources or mechanical processing resources. Nodes may be represented as types and as instances. Run time computational instances, both objects and component instances, may reside on node instances.

3.97.2 Notation

A node is shown as a figure that looks like a 3-dimensional view of a cube. A node type has a type name:

node-type

A node instance has a name and a type name. The node may have an underlined name string in it or below it. The name string has the syntax:

name ':' *node-type*

The name is the name of the individual node (if any). The node-type says what kind of a node it is. Either or both elements are optional; if the node-type is omitted, then so is the colon.

Dashed arrows with the keyword «deploy» show the capability of a node type to support a component type. Alternatively, this may be shown by nesting component symbols inside the node symbol.

Component instances and objects may be contained within node instance symbols. This indicates that the items reside on the node instances.

Nodes may be connected by associations to other nodes. An association between nodes indicates a communication path between the nodes. The association may have a stereotype to indicate the nature of the communication path (for example, the kind of channel or network).

3.97.3 Example

This example shows two nodes containing components, where a «become» flow shows the *backupBroker* migrating from the *backupServer* to the *primaryServer* while the other components remain in place.

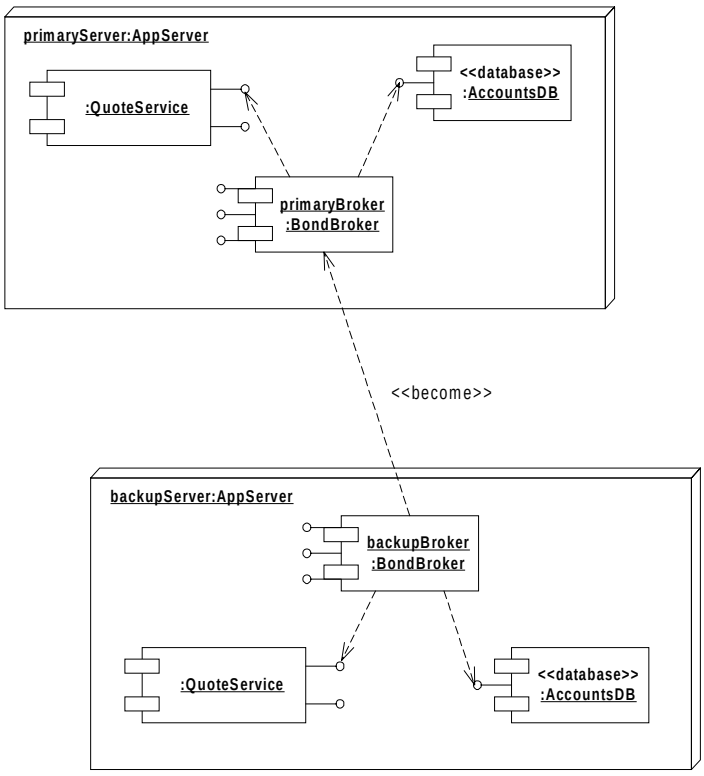


Figure 3-97 Node and Component Instances

3.97.4 Mapping

A node maps to a Node.

A «deploy» arrow or the nesting of a component symbol within a node symbol maps into a residence metassociation between Component and Node. The nesting of a component-instance symbol within a node-instance symbol maps to a residence metaassociation between the ComponentInstance and the NodeInstance.

3.98 Component

3.98.1 Semantics

A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.

A component is typically specified by one or more classifiers that reside on the component. A subset of these classifiers explicitly define the component's external interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component. A component may be implemented by one or more artifacts, such as binary, executable, or script files. A component may be deployed on a node.

3.98.2 Notation

A component is shown as a rectangle with two small rectangles protruding from its side. A component type has a type name:

component-type

A component instance has a name and a type. The name of the component and its type may be shown as an underlined string either within the component symbol or above or below it, with the syntax:

component-name ':' *component-type*

Either or both elements are optional. If the component-type is omitted, then so is the colon.

Objects that reside on a component instance are shown as nested inside the component instance symbol. By analogy, classes that are implemented by a component may be shown as nested within it; this indicates residence and not ownership.

Elements that reside on a component are shown nested inside the component symbol. The visibility of a resident element to other components may be shown using the same notation as for the visibility of the contents of a package (prepending a visibility symbol to the name of the package). The meaning of the visibility depends on the nature of the component. For a source-language component (such as program text), it would control the accessibility of source-language constructs. For a run-time code component (such as executable code), it would control the ability of code in other components to call or otherwise access code in the component.

3.98.3 Example

The example shows a component with interfaces and also a component that contains objects at run time.

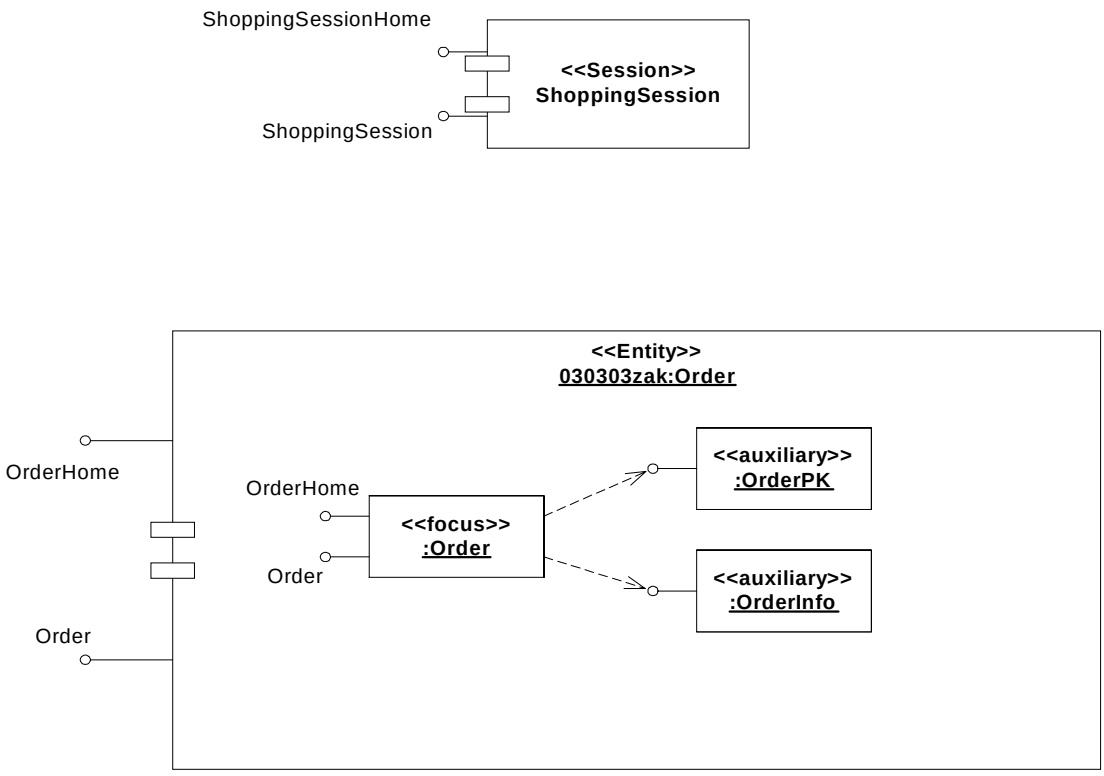


Figure 3-98 Components

3.98.4 Mapping

A component symbol maps to a Component.

The graphical nesting of an element (other than a component symbol) in a component symbol maps to an ElementResidence metaassociation class between ModelElement and the Component. Graphical nesting of a component symbol in another component symbol maps to a composition association. The graphical nesting of an instance symbol in a component instance symbol maps to a residence metaassociation between Instance and ComponentInstance.

UML Example Profiles 4

Contents

This chapter contains the following topics.

Topic	Page
"Example 1 - UML Profile for Software Development Processes"	
"Introduction"	4-1
"Summary of Profile"	4-2
"Stereotypes and Notation"	4-2
"Well-formedness Rules"	4-9
"Example 2 - UML Profile for Business Modeling"	
"Introduction"	4-9
"Summary of Profile"	4-10
"Stereotypes and Notation"	4-10
"Well-formedness Rules"	4-15

Example 1 - UML Profile for Software Development Processes

4.1 Introduction

The UML Profile for Software Development Processes is an example profile that is based on the Unified Process for software engineering. The profile is defined using the extensibility mechanisms of UML, which allow modelers to customize UML for specific domains, such as software development processes.

Note that this profile is not a complete definition of the Unified Process or how to apply it, but rather an example that shows how some of the profile terminology and notation is used. This example is defined only through stereotypes and constraints; profiles also commonly include tagged values.

4.2 Summary of Profile

The stereotypes that are defined by this profile are summarized below.

Name	Base Class
UseCaseModel	Model
AnalysisModel	Model
DesignModel	Model
ImplementationModel	Model
UseCaseSystem	Package
AnalysisSystem	Package
DesignSystem	Subsystem
ImplementationSystem	Subsystem
AnalysisPackage	Package
DesignSubsystem	Subsystem
ImplementationSubsystem	Subsystem
UseCasePackage	Package
AnalysisServicePackage	Package
DesignServiceSubsystem	Subsystem
Boundary	Class
Entity	Class
Control	Class
Communicate	Association
Subscribe	Association

4.3 Stereotypes and Notation

A system modeled by the Unified Process consists of several different, but related models. These models are characterized by the lifecycle stage that they represent, and each model makes use of one specific stereotype. Many of the stereotypes are used particularly to give the ability to structure and categorize models and systems during different stages of the development process.

In addition, there are stereotypes describing different kinds of commonly occurring analysis classes (such as boundary, entity, and control) and their relationships, whereas design classes are by default not stereotyped in the Unified Process.

4.3.1 Use Case Stereotypes

4.3.1.1 UseCaseModel

Stereotype	Base Class	Parent	Description	Constraints
UseCaseModel «useCaseModel»	Model	NA	A use case model specifies the services a system provides to its users; that is, the different ways of using the system, and whose top-level package is a use case system.	None.

The notation used for a UseCaseModel is a package stereotyped as «useCaseModel». Though superfluous, it is optionally possible to in addition use the model icon in the upper right corner of the package symbol.

The explicit modeling of the stereotype is shown in Figure 4-1.

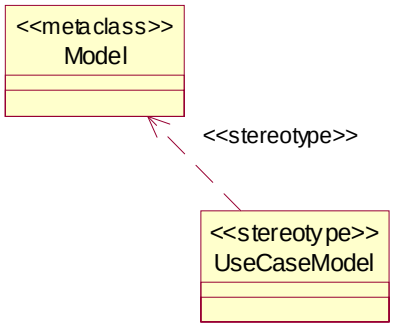


Figure 4-1 Explicit Modeling of a Stereotype

4.3.1.2 UseCaseSystem

Stereotype	Base Class	Parent	Description	Constraints
UseCaseSystem «useCaseSystem»	Package	NA	A use case system is a top-level package that may contain use case packages, use cases, and relationships.	None.

The notation used for a UseCaseSystem is a package stereotyped as «useCaseSystem».

4.3.1.3 UseCasePackage

Stereotype	Base Class	Parent	Description	Constraints
UseCasePackage «useCasePackage»	Package	NA	A use case package contains use cases and relationships.	A use case is not partitioned over several use case packages.

The notation used for a UseCasePackage is a package stereotyped as «useCasePackage».

4.3.2 Analysis Stereotypes

4.3.2.1 AnalysisModel

Stereotype	Base Class	Parent	Description	Constraints
AnalysisModel «analysisModel»	Model	NA	An analysis model is a model whose top-level package is an analysis system.	None.

The notation used for an AnalysisModel is a package stereotyped as «analysisModel».

4.3.2.2 AnalysisSystem

Stereotype	Base Class	Parent	Description	Constraints
AnalysisSystem «analysisSystem»	Package	NA	An analysis system is a top-level package that may contain analysis packages, analysis service packages, analysis classes, and relationships.	None.

The notation used for an AnalysisSystem is a package stereotyped as «analysisSystem».

4.3.2.3 AnalysisPackage

Stereotype	Base Class	Parent	Description	Constraints
AnalysisPackage «analysisPackage»	Package	NA	An analysis package is a package that may contain other analysis packages, analysis service packages, analysis classes, and relationships.	None.

The notation used for an AnalysisPackage is a package stereotyped as «analysisPackage».

4.3.2.4 AnalysisServicePackage

Stereotype	Base Class	Parent	Description	Constraints
AnalysisServicePackage «analysisServicePackage»	Package	NA	An analysis service package is a package that may contain analysis classes and relationships.	None.

The notation used for an AnalysisServicePackage is a package stereotyped as «analysisServicePackage».

4.3.3 Design Stereotypes

4.3.3.1 DesignModel

Stereotype	Base Class	Parent	Description	Constraints
DesignModel «designsModel»	Model	NA	A design model is a model whose top-level package is a design system.	None.

The notation used for a DesignModel is a package stereotyped as «designModel».

4.3.3.2 DesignSystem

Stereotype	Base Class	Parent	Description	Constraints
DesignSystem «designSystem»	Subsystem	NA	A design system is a top-level subsystem that may contain design subsystems, design service subsystems, design classes, and relationships.	None.

The notation used for a DesignSystem is a package stereotyped as «designSystem». Though superfluous, it is optionally possible to in addition use the subsystem icon in the upper right corner of the package symbol.

4.3.3.3 DesignSubsystem

Stereotype	Base Class	Parent	Description	Constraints
DesignSubsystem «designSubsystem»	Subsystem	NA	A design subsystem is a subsystem that may contain other design subsystems, design classes, and relationships.	None.

The notation used for a DesignSubsystem is a package stereotyped as «designSubsystem».

4.3.3.4 DesignServiceSubsystem

Stereotype	Base Class	Parent	Description	Constraints
DesignServiceSubsystem «designServiceSubsystem»	Subsystem	NA	A design service subsystem is a subsystem that may contain design classes and relationships.	None.

The notation used for a DesignServiceSubsystem is a package stereotyped as «designServiceSubsystem».

4.3.4 Implementation Stereotypes

4.3.4.1 ImplementationModel

Stereotype	Base Class	Parent	Description	Constraints
ImplementationModel «implementationModel»	Model	NA	An implementation model is a model whose top-level package is an implementation system.	None.

The notation used for an ImplementationModel is a package stereotyped as «implementationModel».

4.3.4.2 ImplementationSystem

Stereotype	Base Class	Parent	Description	Constraints
ImplementationSystem «implementationSystem»	Subsystem	NA	An implementation model is a subsystem that may contain implementation subsystems, components, and relationships.	None.

The notation used for an ImplementationSystem is a package stereotyped as «implementationSystem».

4.3.4.3 ImplementationSubsystem

Stereotype	Base Class	Parent	Description	Constraints
ImplementationModel «implementationModel»	Model	NA	An implementation model is a model whose top-level package is an implementation system.	None.

The notation used for an ImplementationModel is a package stereotyped as «implementationModel».

4.3.5 Class Stereotypes

4.3.5.1 Entity

Stereotype	Base Class	Parent	Description	Constraints
Entity «entity»	Class	NA	An entity is a passive class; that is, its objects do not initiate interactions on their own. An entity object may participate in many different use case realizations and usually outlives any single interaction.	None.

The notation for Entity is shown below.

4.3.5.2 Control

Stereotype	Base Class	Parent	Description	Constraints
Control «control»	Class	NA	A control is a class whose objects manage interactions between collections of objects. A control class usually has behavior that is specific for one use case, and a control object usually does not outlive the use case realizations in which it participates.	None.

The notation for Control is shown below.

4.3.5.3 Boundary

Stereotype	Base Class	Parent	Description	Constraints
Boundary «boundary»	Class	NA	A boundary is a class that lies on the periphery of a system, but within it. It interacts with actors outside the system as well as with entity, control, and other boundary classes within the system.	None.

The notation for Boundary is shown below.

4.3.5.4 Notation

The notation given as part of the UML specification for stereotyped classes can be used for entity, control, and boundary, but it is also possible to substitute that notation with the icons shown below.

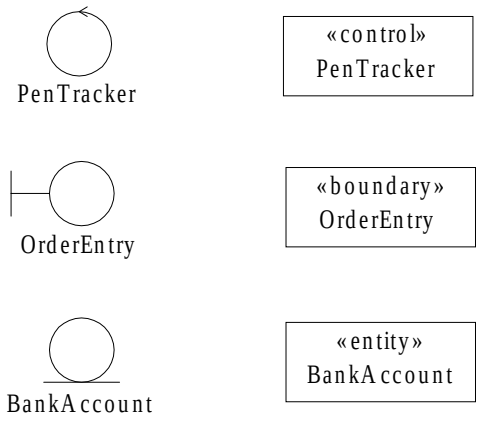


Figure 4-2 Class Stereotypes

4.3.6 Association Stereotypes

4.3.6.1 Communicate

Stereotype	Base Class	Parent	Description	Constraints
Communicate «communicate»	Association	NA	Communicate is an association between actors and use cases that is used to denote messages that may be sent between them. It may also be used between boundary, control, and entity, and between actor and boundary.	None.

The notation used for Communicate is an association that is marked with the stereotype «communicate».

4.3.6.2 Subscribe

Stereotype	Base Class	Parent	Description	Constraints
Subscribe «subscribe»	Association	NA	A subscribe association between two classes states that objects of the source class (called the subscriber) will be notified when a particular event has occurred in objects of the target class (called the publisher). The association includes a specification of a set of events defining the events that causes the subscriber to be notified.	None.

The notation used for Subscribe is an association that is marked with the stereotype «subscribe».

4.4 Well-formedness Rules

The UML Specification relies on the use of well-formedness rules to express constraints on model elements, and this profile uses the same approach. The constraints applicable to the profile are added to the ones of the stereotyped base model elements, which cannot be changed.

4.4.1 Generalization

All the modeling elements in a generalization must be of the same stereotype; for example, a boundary class may only inherit from other boundary classes.

```
context Generalization inv:
    (self.parent.stereotype->size>0) implies
        (if (self.parent.stereotype->name->includes(“boundary”)then
            ((self.child.stereotype->name->includes(“boundary”) and
            (self.child.stereotype->name->excludes(“control”) and
            (self.child.stereotype->name->excludes(“entity”))
        else
        (if (self.parent.stereotype->name->includes(“control”)then
            ((self.child.stereotype->name->includes(“control”) and
            (self.child.stereotype->name->excludes(“boundary”) and
            (self.child.stereotype->name->excludes(“entity”))
        else
        (if (self.parent.stereotype->name->includes(“entity”)then
            ((self.child.stereotype->name->includes(“entity”) and
            (self.child.stereotype->name->excludes(“boundary”) and
            (self.child.stereotype->name->excludes(“control”))))
```

4.4.2 Containment

Something that has been stereotyped using a stereotype of kind use case, analysis, design, or implementation may not contain elements that are stereotyped with one of the other kinds. For example, a use case model may not contain analysis systems.

Example 2 - UML Profile for Business Modeling

4.5 Introduction

The UML Profile for Business Modeling is an example profile that describes how UML can be customized for business modeling. Although all UML concepts can be brought to bear on this domain, but example emphasizes common stereotypes and some useful terminology. Note that UML can be used to model different kinds of systems (such as software systems, hardware systems, and real-world organizations).

This example is defined only through stereotypes and constraints; profiles also commonly include tagged values.

4.6 Summary of Profile

The stereotypes that are defined by this profile are summarized below.

Stereotype	Base Class
UseCaseModel	Model
UseCaseSystem	Package
UseCasePackage	Package
ObjectModel	Model
ObjectSystem	Subsystem
OrganizationUnit	Subsystem
WorkUnit	Subsystem
Worker	Class
CaseWorker	Class
InternalWorker	Class
Entity	Class
Communicate	Association
Subscribe	Association

4.7 Stereotypes and Notation

A business system comprises several different, but related, models. The models are characterized by being exterior or interior to the business system they represent. Exterior models are use case models and interior models are object models. A large business system may be partitioned into subordinate business systems.

4.7.1 Use Case Stereotypes

4.7.1.1 Use Case Model

Stereotype	Base Class	Parent	Description	Constraints
UseCaseModel «useCaseModel»	Model	NA	A use case model is a model that describes the business processes of a business and their interactions with external parties such as customers and partners. A use case model describes: • The business modeled as use cases • Parties exterior to the business modeled as actors • The relationships between the external parties and the business process	None.

The notation used for a UseCaseModel is a package stereotyped as «useCaseModel».

4.7.1.2 UseCaseSystem

Stereotype	Base Class	Parent	Description	Constraints
UseCaseSystem «useCaseSystem»	Package	NA	A use case system is the top-level package in a use case model, and may contain use case packages, use cases, and relationships.	None.

The notation used for a UseCaseSystem is a package stereotyped as «useCaseSystem».

4.7.1.3 UseCasePackage

Stereotype	Base Class	Parent	Description	Constraints
UseCasePackage «useCasePackage»	Package	NA	A use case package is a package that may contain use cases and relationships.	A use case is not partitioned over several use case packages.

The notation used for a UseCasePackage is a package stereotyped as «useCasePackage».

4.7.2 Organization Stereotypes

4.7.2.1 ObjectModel

Stereotype	Base Class	Parent	Description	Constraints
ObjectModel «objectModel»	Model	NA	An object model is a model whose top-level package is an object system that describe the things interior to the business system itself.	None.

The notation used for an ObjectModel is a package stereotyped as «objectModel».

4.7.2.2 ObjectSystem

Stereotype	Base Class	Parent	Description	Constraints
ObjectSystem «objectSystem»	Subsystem	NA	An object system is the top-level subsystem in an object model, and may contain organization units, work units, classes, and relationships.	None.

The notation used for an ObjectSystem is a package stereotyped as «objectSystem».

4.7.2.3 OrganizationUnit

Stereotype	Base Class	Parent	Description	Constraints
OrganizationUnit «organizationUnit»	Subsystem	NA	An organization unit is a subsystem that may contain other organization units, work units, classes, and relationships.	None.

The notation used for an OrganizationUnit is a package stereotyped as «organizationUnit».

4.7.2.4 WorkUnit

Stereotype	Base Class	Parent	Description	Constraints
WorkUnit «workUnit»	Subsystem	NA	A work unit is a subsystem that may contain one or more entities. It is a task-oriented set of objects that forms a recognizable whole to the end user, and may have a facade defining the view of the work unit's entities relevant to the task.	None.

The notation used for a WorkUnit is a package stereotyped as «workUnit».

4.7.3 Class Stereotypes

4.7.3.1 Worker

Stereotype	Base Class	Parent	Description	Constraints
Worker «worker»	Class	NA	A worker is a class that represents an abstraction of a human that acts within the system. A worker interacts with other workers and manipulates entities while participating in use case realizations.	None.

The notation for Worker is shown below.

4.7.3.2 CaseWorker

Stereotype	Base Class	Parent	Description	Constraints
CaseWorker «caseWorker»	Class	Worker	A case worker is a special case of worker that interacts directly with actors outside the system.	None.

The notation for CaseWorker is shown below. Note that CaseWorker is not stereotyped of a UML metaclass, but rather inherits its properties from the stereotype Worker that was previously defined.

The explicit subtyping of a stereotype is shown in Figure 4-3.

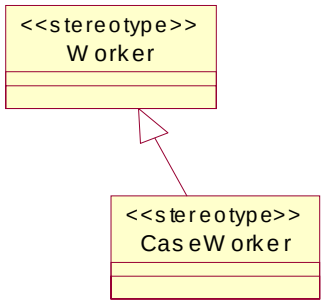


Figure 4-3 Subtyping a Stereotype

4.7.3.3 InternalWorker

Stereotype	Base Class	Parent	Description	Constraints
InternalWorker «internalWorker»	Class	Worker	An internal worker is a special case of worker that interacts with other workers and entities inside the system.	None.

The notation for InternalWorker is shown below. Note that InternalWorker, like CaseWorker above, is subtyped from the previously defined stereotype Worker.

4.7.3.4 Entity

Stereotype	Base Class	Parent	Description	Constraints
Entity «entity»	Class	NA	An entity is a passive class; that is, its objects do not initiate interactions on their own. An entity object may participate in many different use case realizations and usually outlives any single interaction.	None.

The notation for Entity is shown below.

4.7.3.5 Notation

The notation given as part of the UML specification for stereotyped classes can be used for entity, control, and boundary, but it is also possible to substitute that notation with the icons shown below.

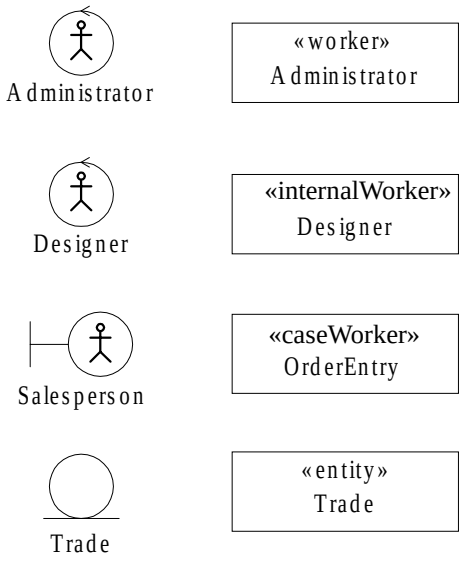


Figure 4-4 Class Stereotypes

4.7.4 Association Stereotypes

4.7.4.1 Communicate

Stereotype	Base Class	Parent	Description	Constraints
Communicate «communicate»	Association	NA	Communicate is an association used for defining that instances of the associated classifiers interact.	None.

The notation used for Communicate is an association that is marked with the stereotype «communicate».

4.7.4.2 Subscribe

Stereotype	Base Class	Parent	Description	Constraints
Subscribe «subscribe»	Association	NA	A subscribe association between two classes states that objects of the source class (called the subscriber) will be notified when a particular event has occurred in objects of the target class (called the publisher). The association includes a specification of a set of events defining the event that causes the subscriber to be notified.	None.

The notation used for Subscribe is an association that is marked with the stereotype «subscribe».

4.8 Well-formedness Rules

The UML Specification relies on the use of well-formedness rules to express constraints on model elements, and this profile uses the same approach. The constraints applicable to the profile are added to the ones of the stereotyped base model elements, which cannot be changed.

4.8.1 Generalization

All the modeling elements in a generalization must be of the same stereotype; for example, a worker class may only inherit from other worker classes.

```
context Generalization inv:
let stNames : Set(Name) = self.child.stereotype->name
self.parent.stereotype->size>0) implies
    (if (self.parent.stereotype->name->includes(“worker”) then
        ((stNames->includes(“worker”) and
        (selfstNames->excludes(“case worker”) and
        (stNames->excludes(“internal worker”) and
        (stNames->excludes(“entity”))
    else
    (if (self.parent.stereotype->name->includes(“case worker”) then
        ((stNames->includes(“case worker”) and
```

```
            (selfstNames->excludes("worker") and
            (stNames->excludes("internal worker") and
            (stNames->excludes("entity"))
        else
        (if (self.parent.stereotype->name->includes("internal worker")
    then
            ((stNames->includes("internal worker") and
            (selfstNames->excludes("case worker") and
            (stNames->excludes("worker") and
            (stNames->excludes("entity"))
        else
        (if (self.parent.stereotype->name->includes("entity") then
            ((stNames->includes("entity") and
            (selfstNames->excludes("case worker") and
            (stNames->excludes("internal worker") and
            (self.child.stereotype->name->excludes("worker"))))))
```

UML Model Interchange 5

Note – Change bars mark the differences between UML 1.4 and UML 1.5.

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	5-1
“Model Interchange Using XMI”	5-23
“Model Interchange Using CORBA IDL”	5-24

5.1 Overview

UML model interchange is based on the Metaobject Facility (MOF) 1.3 Specification. The UML Semantics abstract syntax is mapped to a set of MOF packages called the UML Interchange Metamodel. The packages are available as an XML document called UML_1.4_Interchange_Metamodel.xml (OMG document ad/01-02-15) whose document type is based on the MOF 1.3 Model and the XML Metadata Interchange (XMI) 1.1 Specification.

Except for the Data_Types package, each package of the UML Interchange Metamodel defines a separate unit of compliance. The Core package defines the most basic level of compliance. The UML package, which is a cluster of all of the others, defines complete compliance.

Dependencies between the packages are shown in Figure 5-1. Each package imports whatever other packages it requires such that it can be directly deployed within a MOF facility. The packages can also be incorporated into other clusters in order to create other package groupings or to define extensions.

The UML Interchange Metamodel closely follows the UML Semantics Metamodel as expressed in its abstract syntax. Changes are introduced as needed to conform to MOF requirements. Details are added to support XML and IDL generation. The following changes are made.

- Spaces in package names are changed to "_".
- Each unnamed association end is given its type's name with the first letter downcased.
- Associations in the UML Semantics Metamodel are unnamed, so names are generated by this pattern: "A_" followed by the first end's name followed by "_" and the second end's name.
- MOF references are added for most association ends in order to facilitate easy navigation. References are not added were they would create new package dependencies or where they would prevent linking to external models.
- MOF does not support association classes, so the ElementOwnership association class is removed and its attributes moved to ModelElement. Each other association class is changed into a class with each connection made into a separate association.
- Prefixes are added to enumeration literals to make them unique for IDL generation.

The Interchange Metamodel addresses semantic content of UML models and does not address diagram layout details. The metamodel can be extended to handle diagrams by subclassing the abstract class PresentationElement of the Core package. There is currently no standard extension for diagram interchange.

The Interchange Metamodel is shown using UML notation below. Figure 5-1 shows the separate packages and their dependencies. Figure 5-2 through Figure 5-21 show the classes, features, and associations of the metamodel.

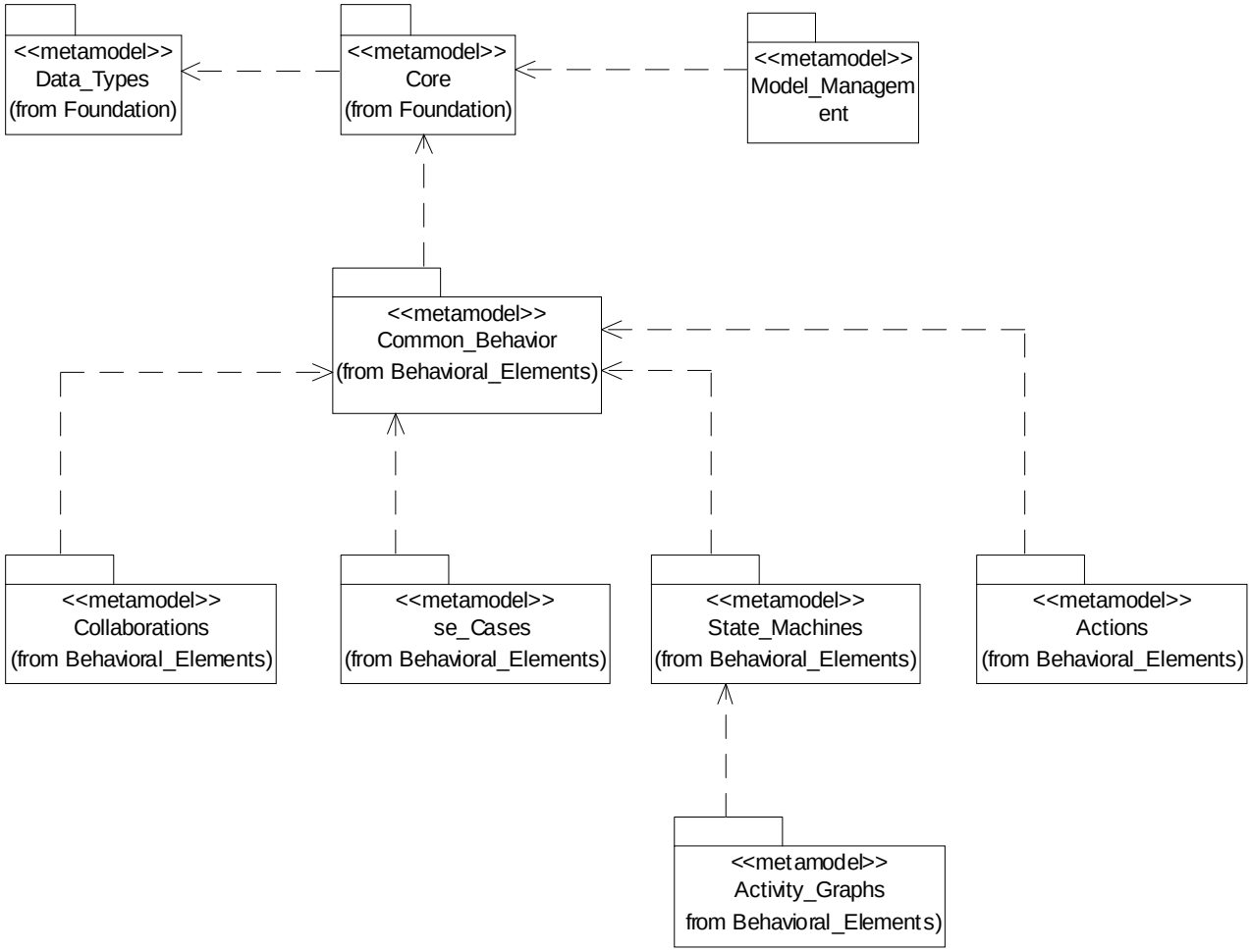


Figure 5-1 UML Package Dependencies

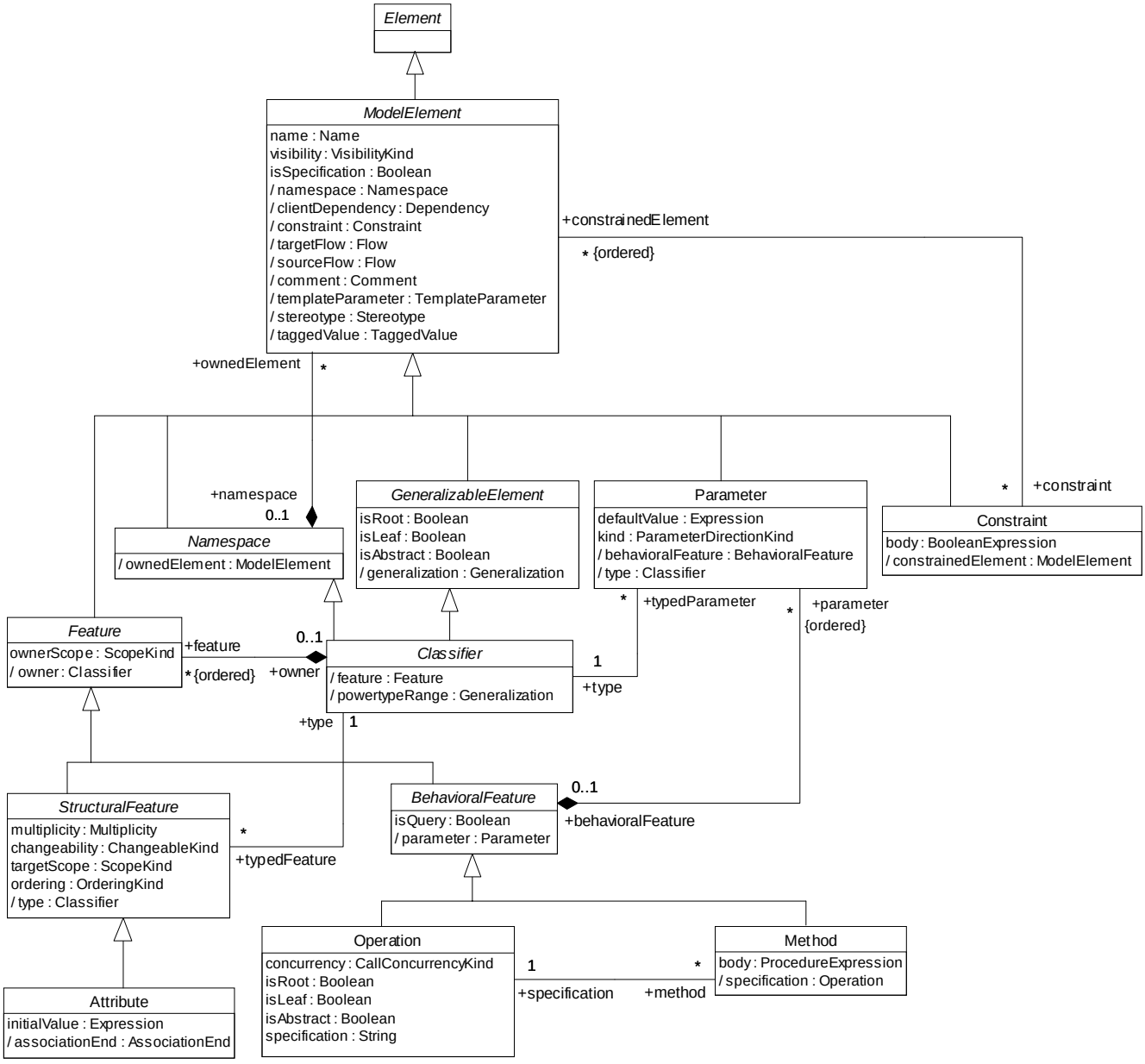


Figure 5-2 Core Package - Backbone

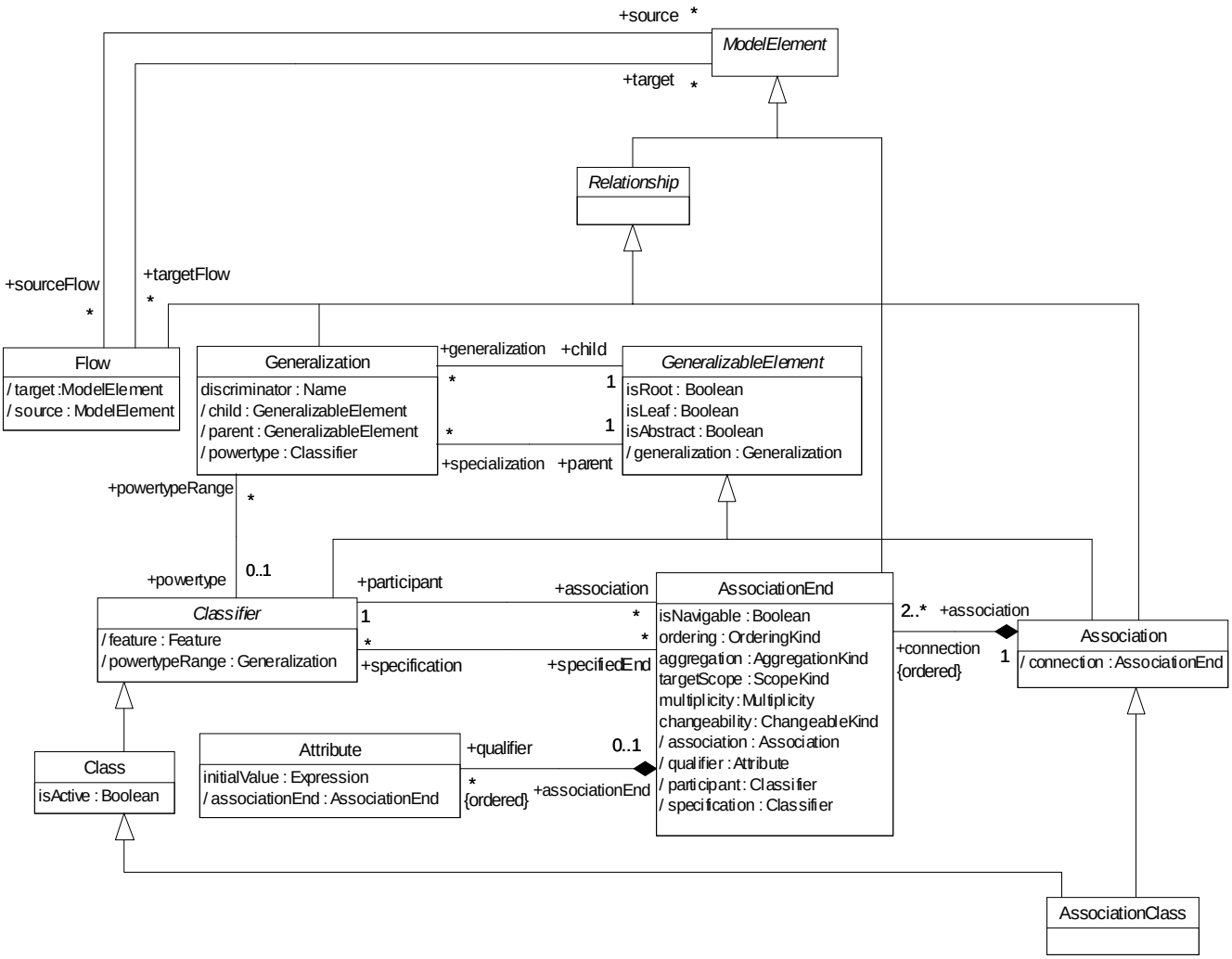


Figure 5-3 Core Package - Relationships

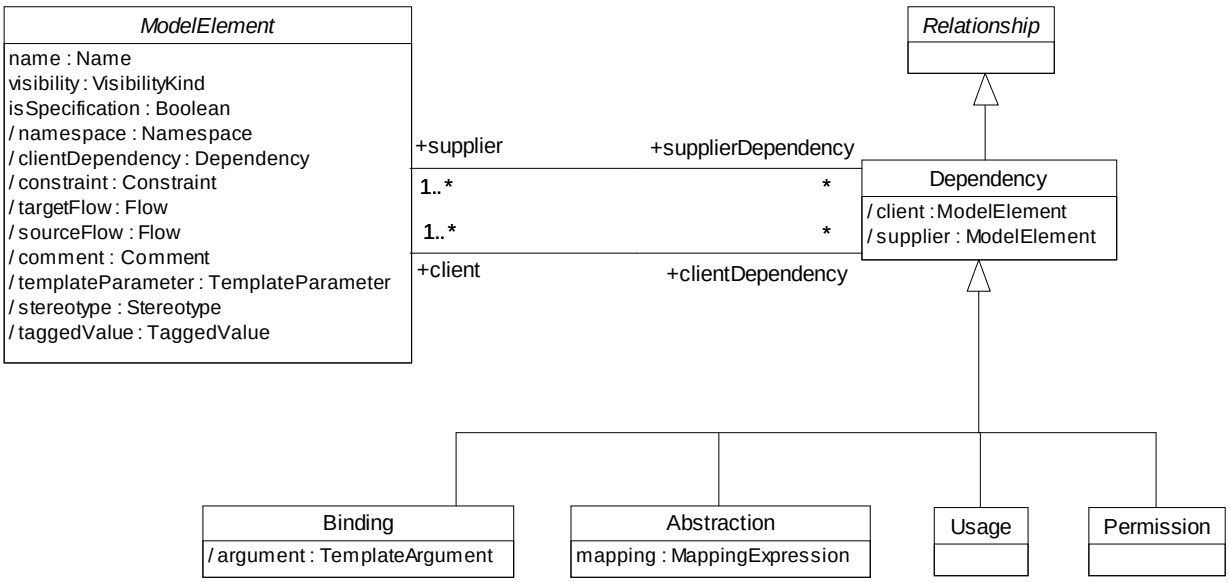


Figure 5-4 Core Package - Dependencies

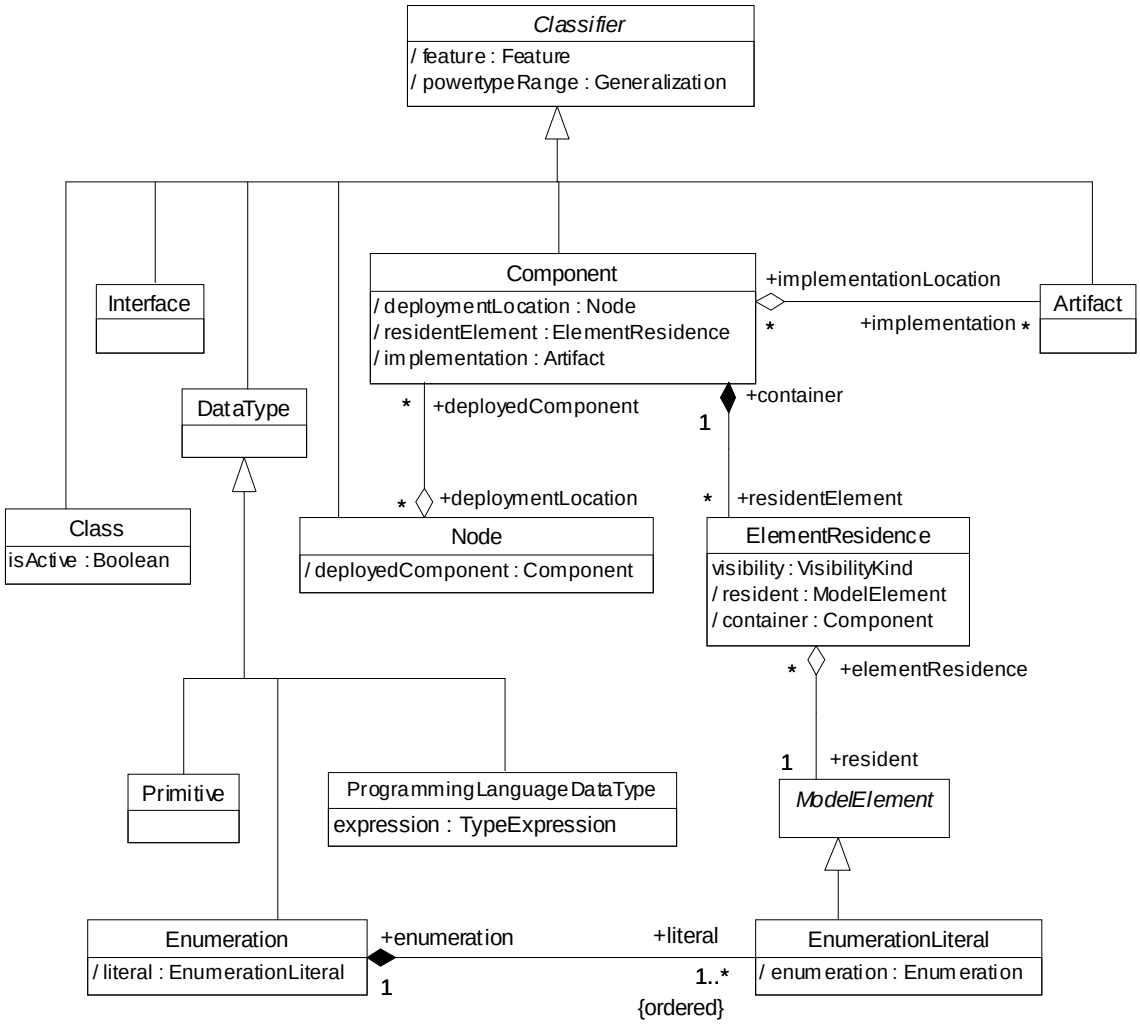


Figure 5-5 Core Package - Classifiers

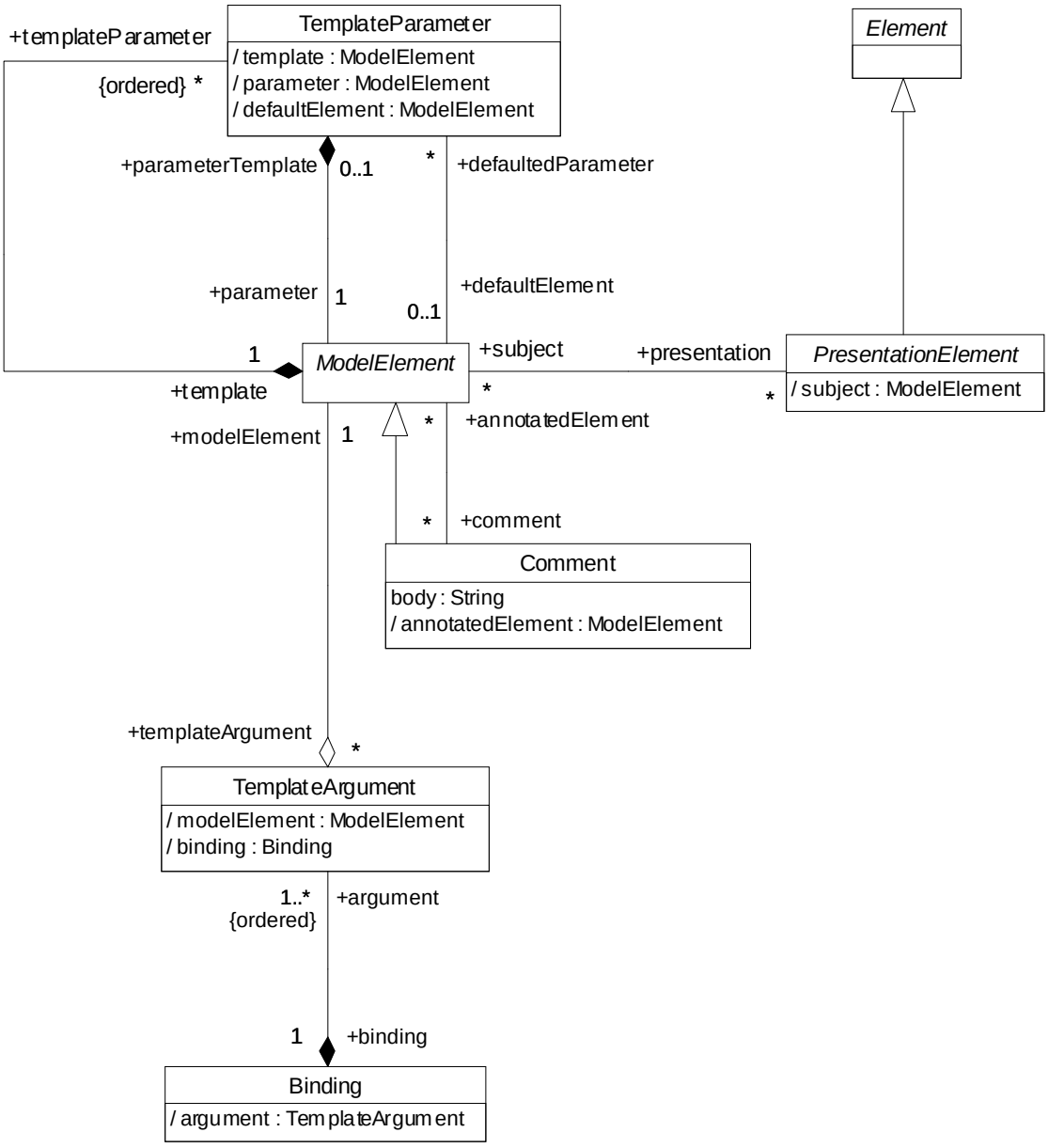


Figure 5-6 Core Package - Auxiliary Elements

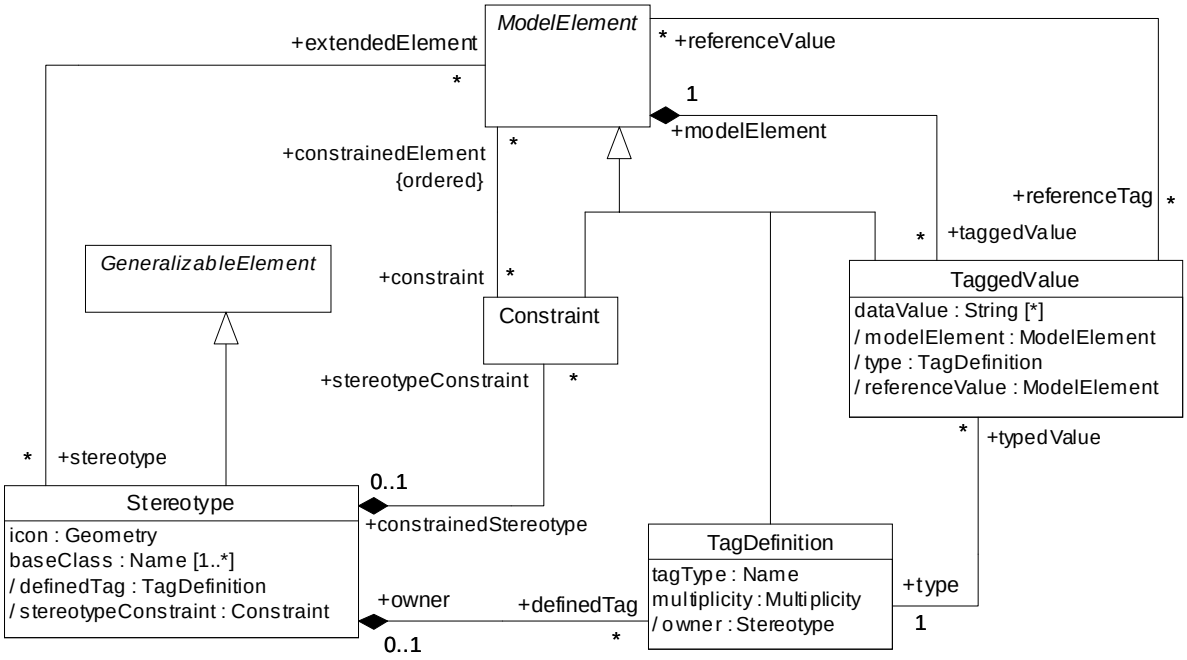


Figure 5-7 Extension Mechanisms

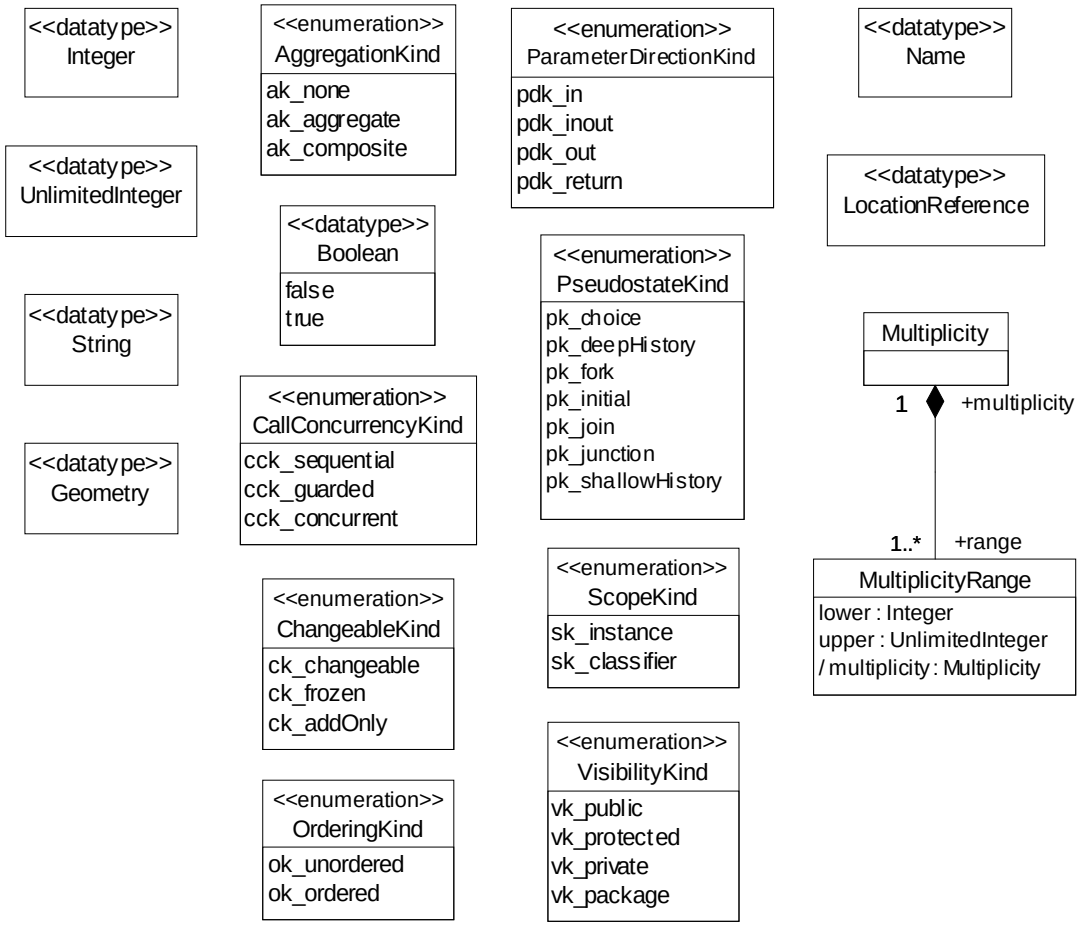


Figure 5-8 Data Types

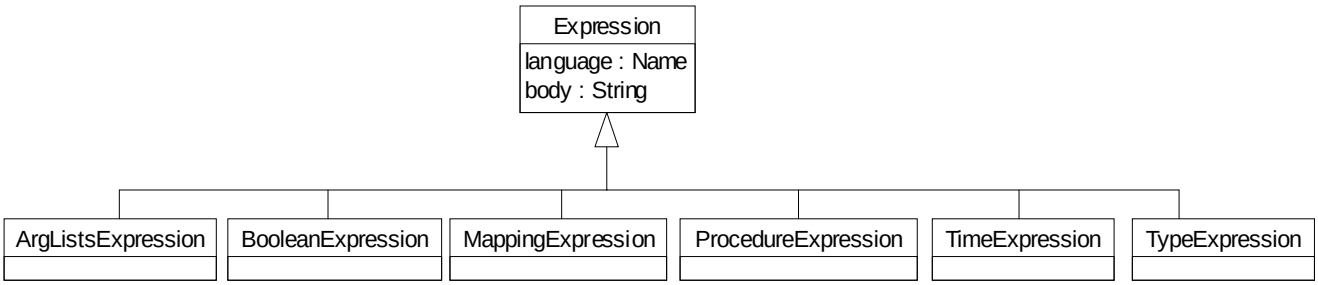


Figure 5-9 Datatypes - Expressions

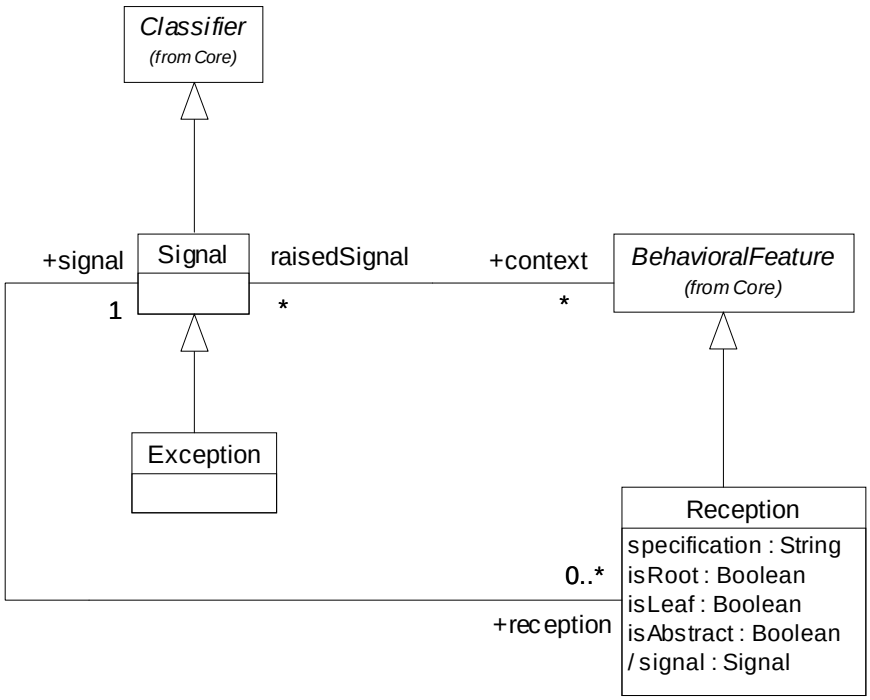


Figure 5-10 Common Behavior - Signals

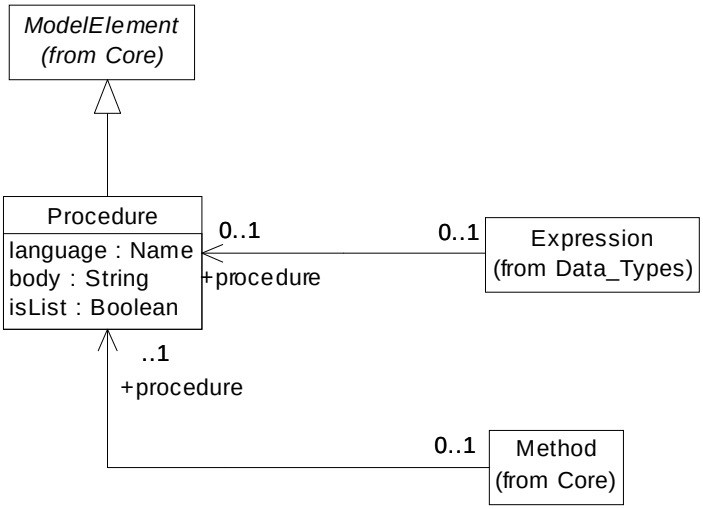


Figure 5-11 Common Behavior - Procedure

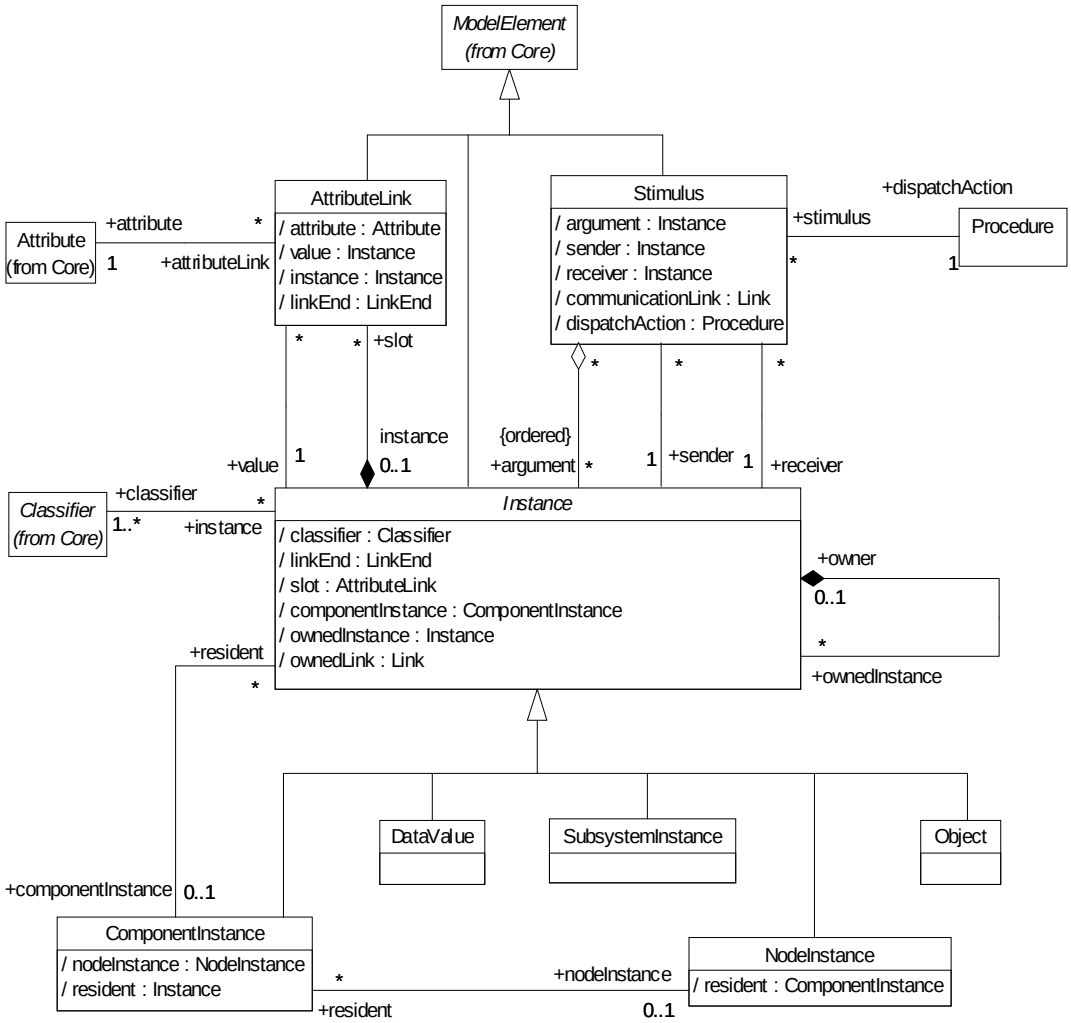


Figure 5-12 Common Behavior - Instances

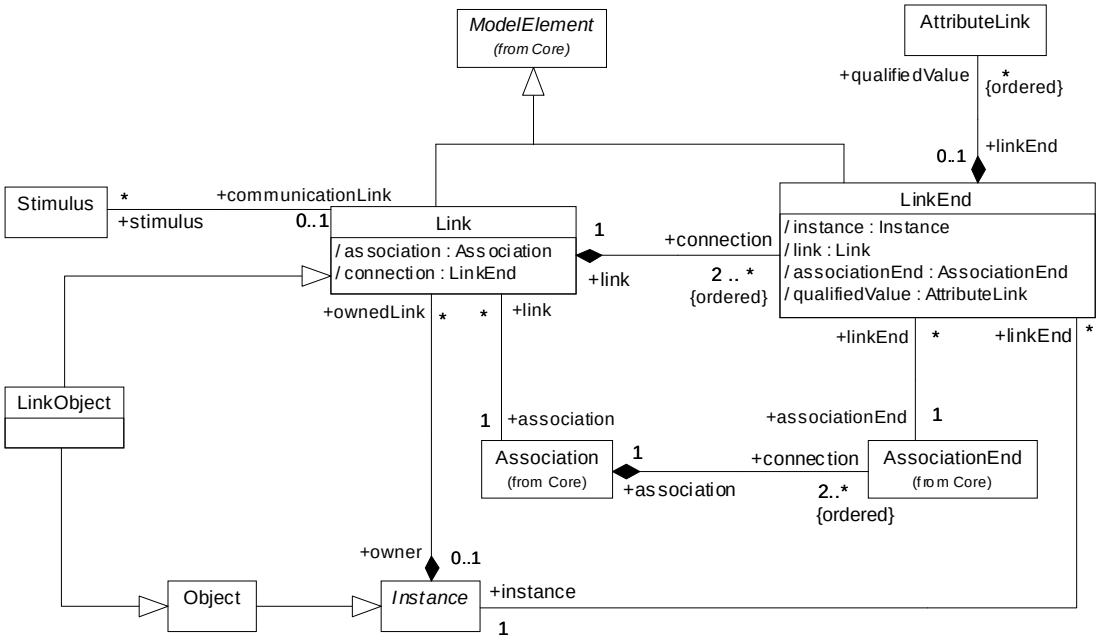


Figure 5-13 Common Behavior - Links

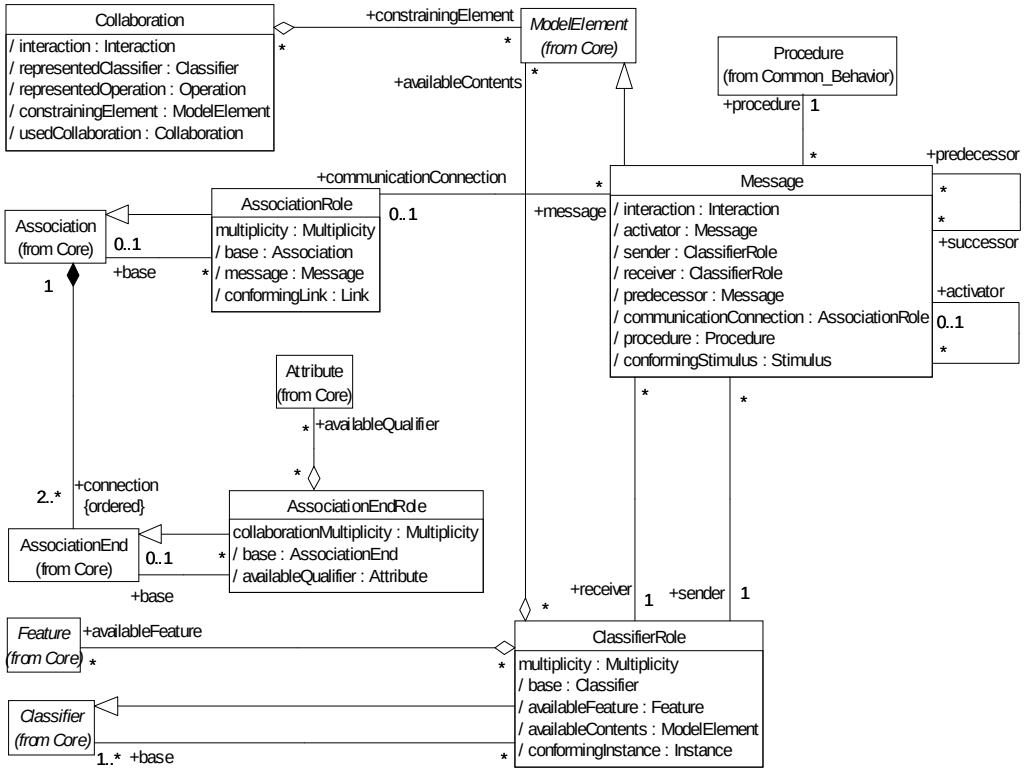


Figure 5-14 Collaborations - Roles

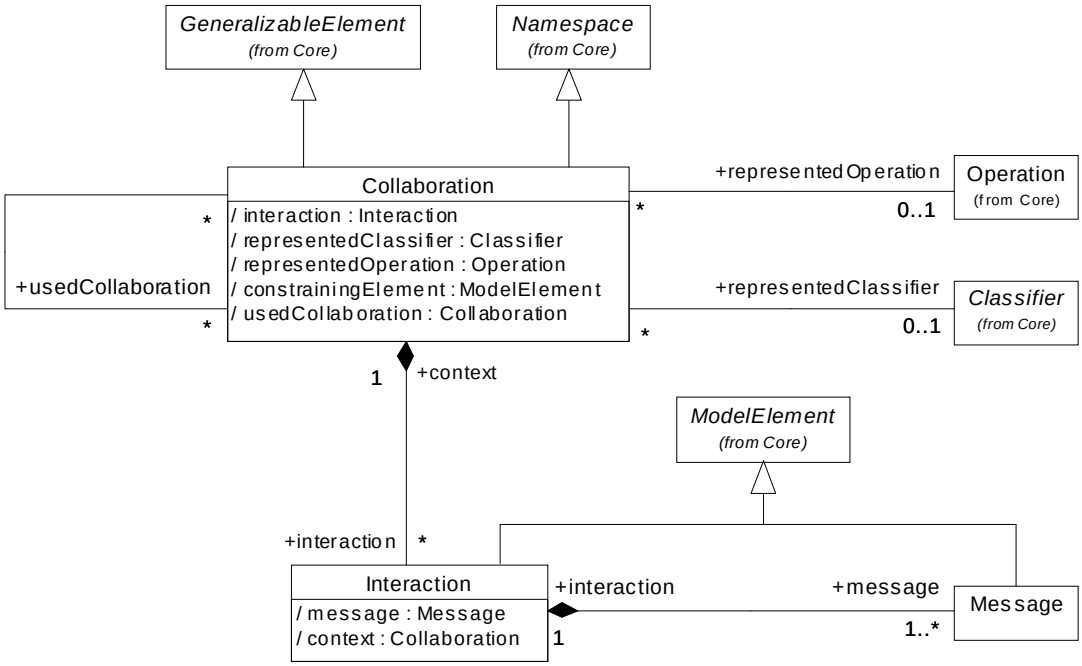


Figure 5-15 Collaborations - Interactions

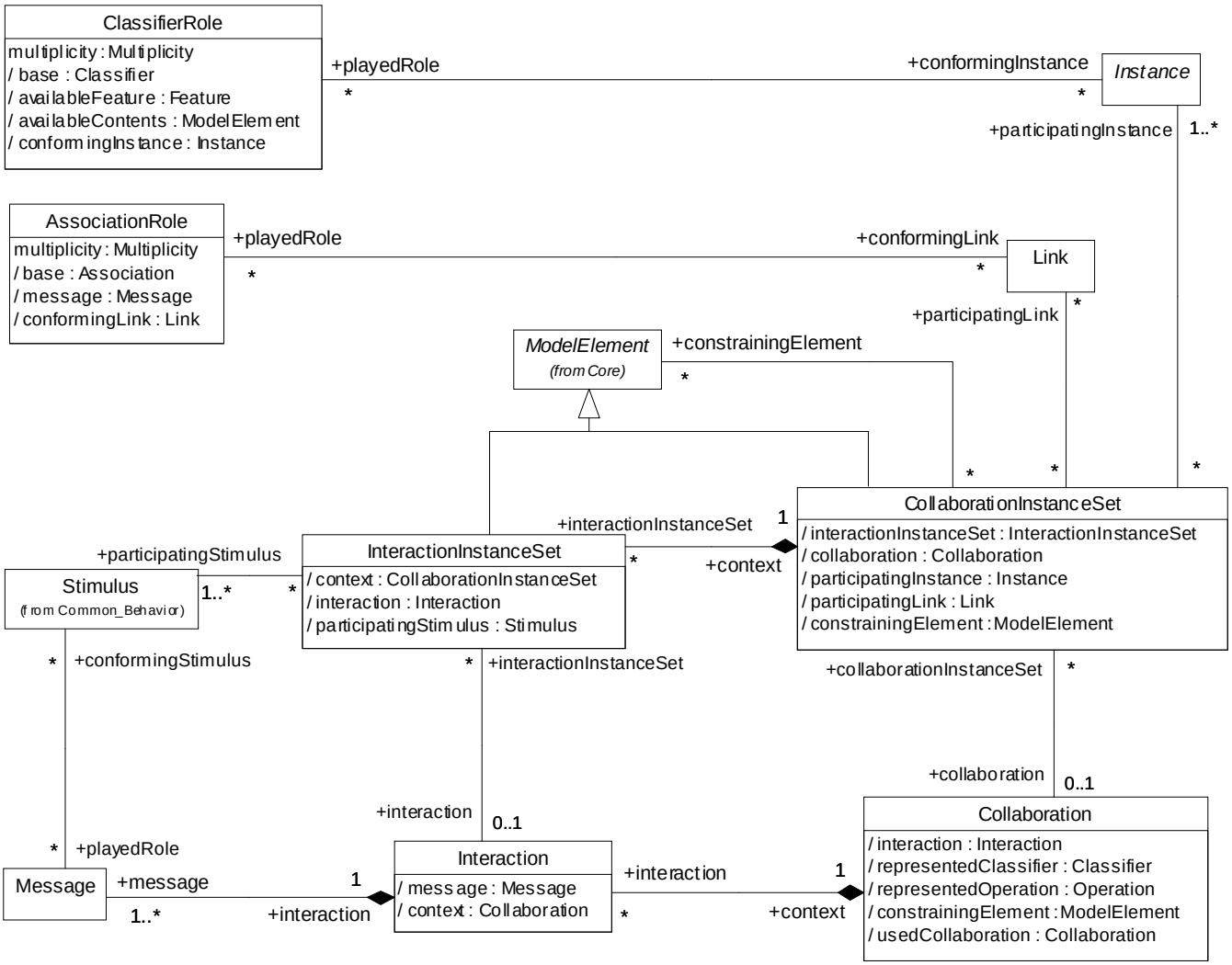


Figure 5-16 Collaborations - Instances

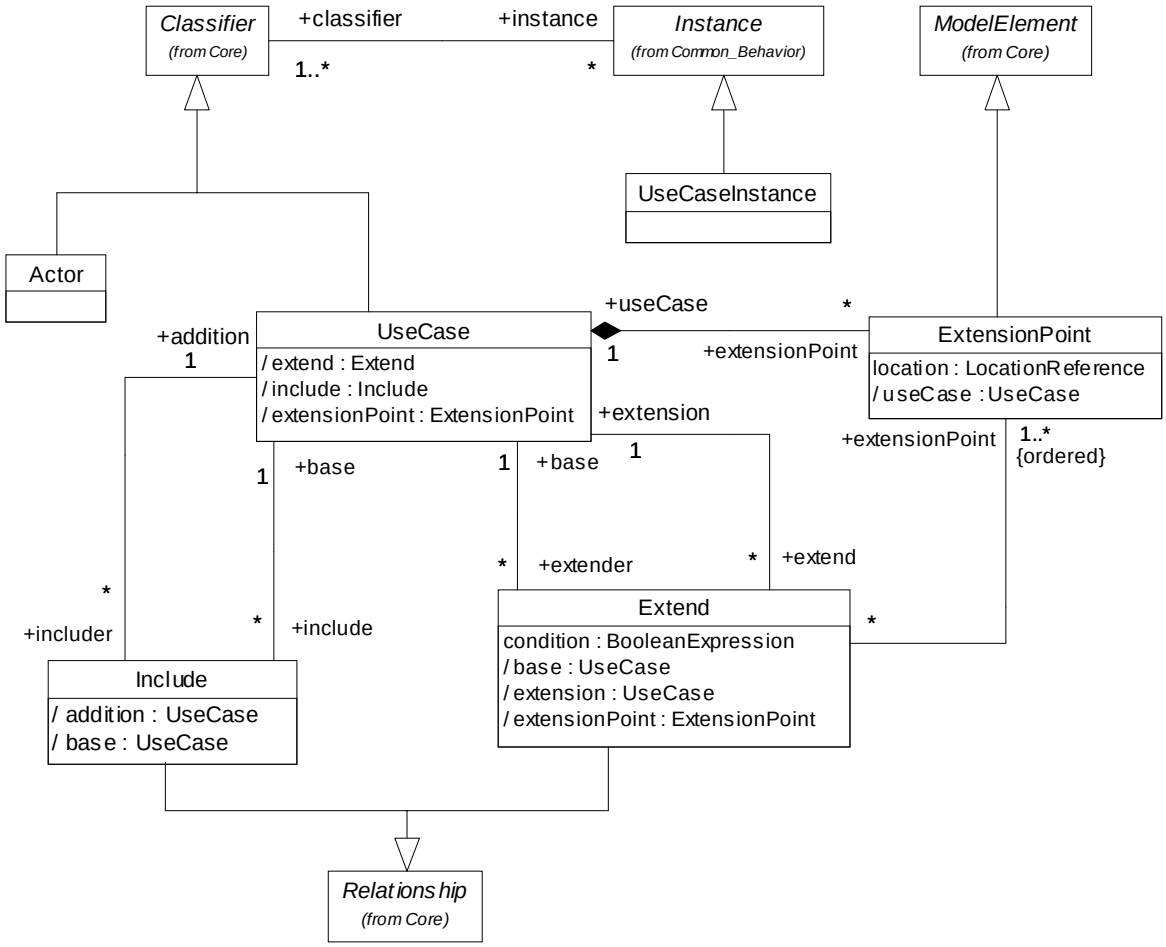


Figure 5-17 Use Cases

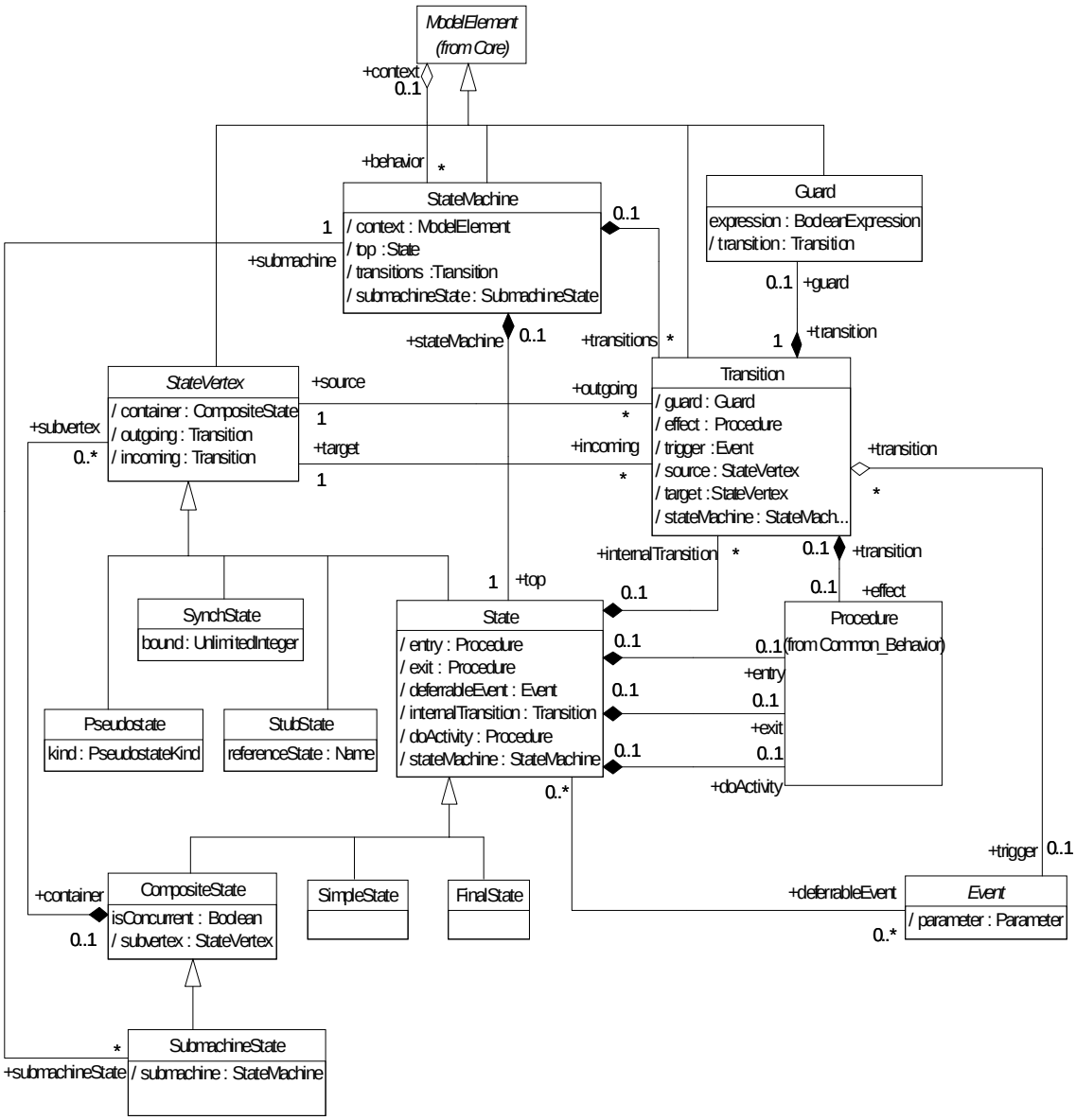


Figure 5-18 State Machines

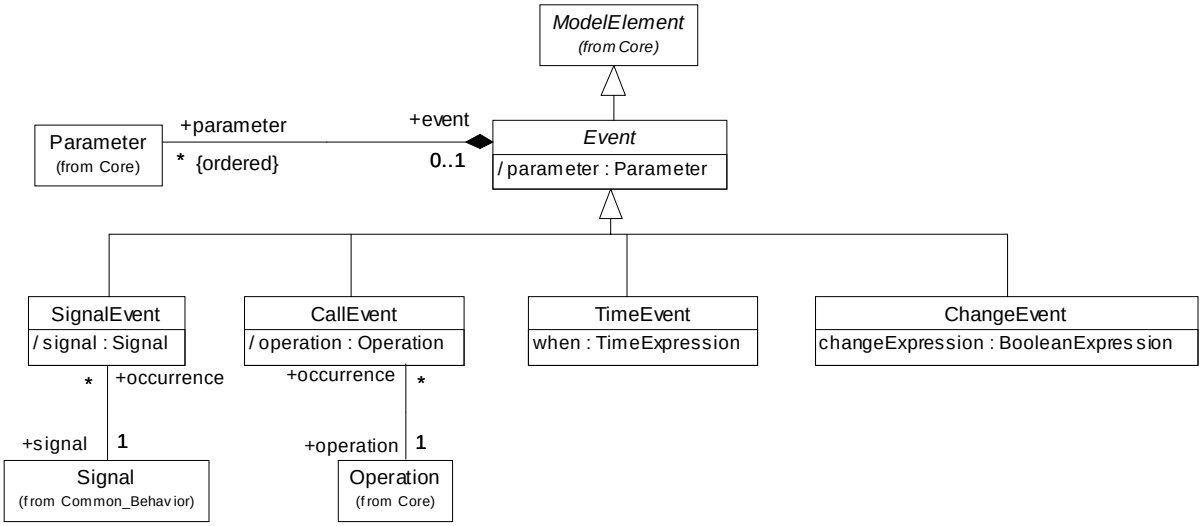


Figure 5-19 State Machines - Events

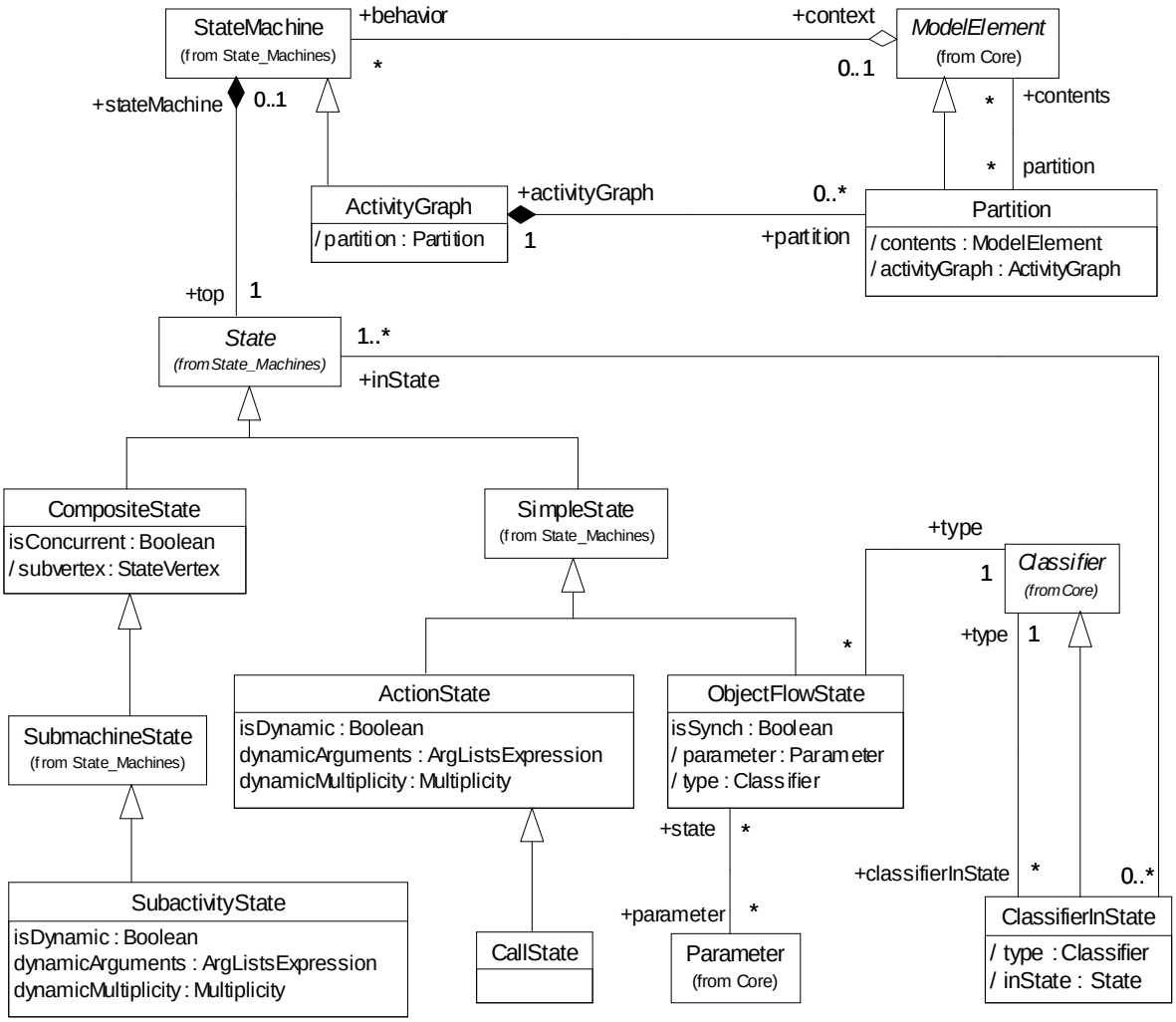


Figure 5-20 Activity Graphs

The interchange metamodels of Actions package are the same as the logical metamodels of Actions package with the following exception:

- The MOF IDL name for the attribute end of the association between AttributeAction and Attribute is “umlAttribute”.
- The MOF IDL name for the object end of the association between AttributeAction and InputPin is actionObject. Likewise for ReadLinkObjectEndAction, ReadLinkObjectQualifierAction, ClearAssociationAction.

- The Action Foundation diagram and its contents (except for Procedure) are in a separate package with explicit references. Associations from Procedure made bidirectional with references from Action.
- The associations from Procedure to Expression and Method are bidirectional with references on Procedure end.
- The association end between Action and Procedure on the Procedure end is named procedure.
- All slashes are removed from association end names and these associations are marked as derived with MOF changeability set to false, except for the association between CreateLinkAction and LinkEndCreationData.

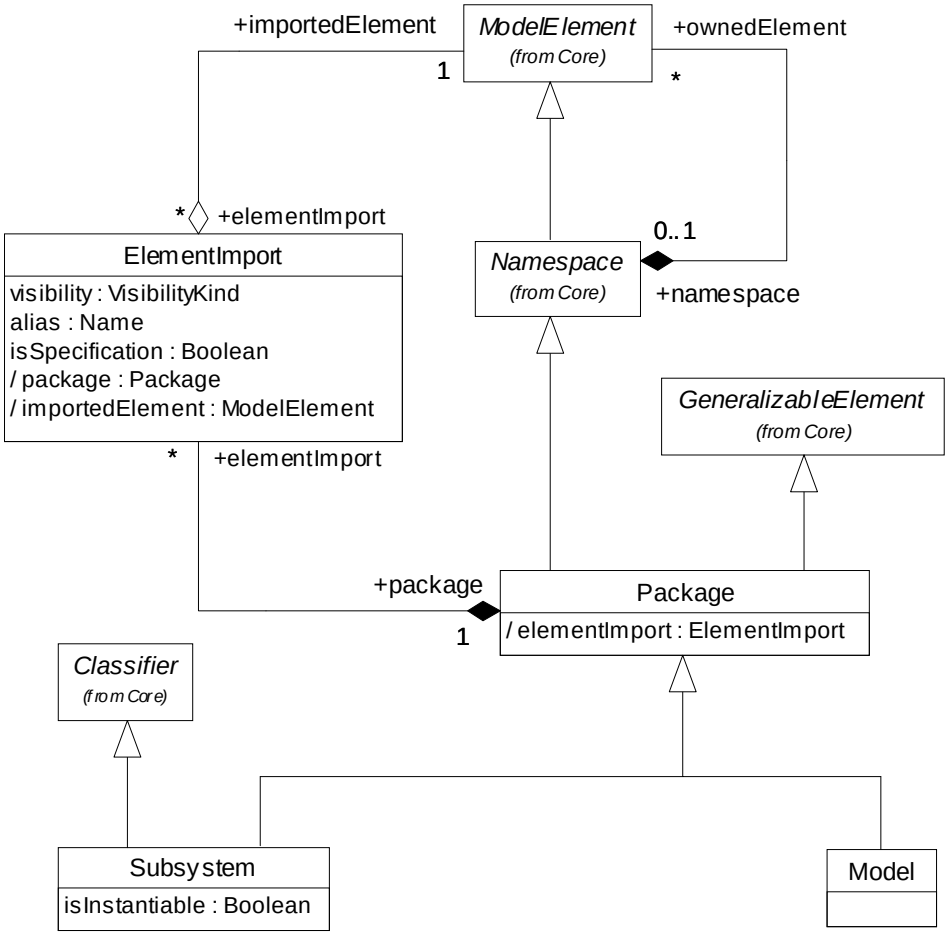


Figure 5-21 Model Management

5.2 Model Interchange Using XMI

UML models can be exchanged between software tools as streams or files with a standard XML format. An XML document type file named UML_1.4_XMI_1.1.dtd (OMG document ad/01-02-16) is generated from the UML Interchange Metamodel following the rules of the XML Metadata Interchange (XMI) 1.1 Specification. The single document type supports all packages of the UML Interchange Metamodel, but a tool that exchanges models using XML might support some packages and not others.

To illustrate use of XML to represent a UML model, Figure 5-22 shows an example model.

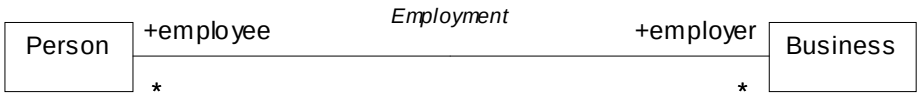


Figure 5-22 Example: Employment Model

The model shown above is expressed in XML below.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE XMI SYSTEM 'UML_1.4_XMI_1.1.dtd'>
<XMI xmi.version='1.2' xmlns:UML='omg.org/UML/1.4'>
 <XMI.header>
  <XMI.metamodel xmi.name='UML' xmi.version='1.4'/>
 </XMI.header>
 <XMI.content>
  <UML:Model xmi.id='S.1' name='Employment Model' visibility='public'
     isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
    <UML:Namespace.ownedElement>
     <UML:Class xmi.id='S.2' name='Person' visibility='public' isSpecification='false'
        namespace='S.1' isRoot='true' isLeaf='true' isAbstract='false' isActive='false'/>
     <UML:Class xmi.id='S.3' name='Business' visibility='public' isSpecification='false'
        namespace='S.1' isRoot='true' isLeaf='true' isAbstract='false' isActive='false'/>
     <UML:Association xmi.id='G.1' name='Employment' visibility='public'
        isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
      <UML:Association.connection>
       <UML:AssociationEnd name='employer' visibility='public' isSpecification='false'
          isNavigable='true' ordering='unordered' aggregation='none' targetScope='instance'
          changeability='changeable' participant='S.3' association='G.1'>
        <UML:AssociationEnd.multiplicity>
         <UML:Multiplicity>
          <UML:Multiplicity.range>
           <UML:MultiplicityRange lower='0' upper='-1'/>
          </UML:Multiplicity.range>
         </UML:Multiplicity>
        </UML:AssociationEnd.multiplicity>
       </UML:AssociationEnd>
       <UML:AssociationEnd name='employee' visibility='public' isSpecification='false'
          isNavigable='true' ordering='unordered' aggregation='none' targetScope='instance'
          changeability='changeable' participant='S.2' association='G.1'>
        <UML:AssociationEnd.multiplicity>
```

```
            <UML:Multiplicity>
              <UML:Multiplicity.range>
                <UML:MultiplicityRange lower='0' upper='-1'/>
              </UML:Multiplicity.range>
            </UML:Multiplicity>
          </UML:AssociationEnd.multiplicity>
        </UML:AssociationEnd>
      </UML:Association.connection>
    </UML:Association>
  </UML:Namespace.ownedElement>
 </UML:Model>
</XMI.content>
</XMI>
```

5.3 Model Interchange Using CORBA IDL

CORBA interfaces can be used for creating, accessing, and manipulating UML models. The MOF Specification's Reflective module provides generic interfaces for accessing all objects of a model. Tailored interfaces extend the generic interfaces. One tailored IDL module is generated from each UML Interchange Metamodel package following rules defined in the MOF Specification. The tailored interfaces support fine-grained creation, access, and modification of model elements with type safety in terms of the UML Interchange Metamodel. Support of tailored interfaces is optional. A facility might support some packages and not others.

The module files are combined in a file named UML_1.4_CORBA_IDL.zip (OMG document ad/01-02-17).

The behavior of a CORBA Facility is defined by the MOF Specification for both reflective and tailored interfaces. Additionally, a UML CORBA Facility must provide access to UML Standard Elements (stereotypes, constraints, and tags) documented in chapter 2, UML Semantics.

Object Constraint Language Specification 6

Note – Changes based on the ISO version of UML 1.4.1 (formal/03-02-04) are in this font.

This chapter introduces and defines the Object Constraint Language (OCL), a formal language to express side-effect-free constraints.

Contents

This chapter contains the following topics.

Topic	Page
“Overview”	6-2
“Introduction”	6-3
“Relation to the UML Metamodel”	6-4
“Basic Values and Types”	6-7
“Objects and Properties”	6-12
“Collection Operations”	6-22
“The Standard OCL Package”	6-28
“Predefined OCL Types”	6-29
“Grammar”	6-45

6 Object Constraint Language Specification

6.1 Overview

This chapter introduces and defines the Object Constraint Language (OCL), a formal language used to express constraints. These typically specify invariant conditions that must hold for the system being modeled. Note that when the OCL expressions are evaluated, they do not have side effects; that is, their evaluation cannot alter the state of the corresponding executing system. In addition, to specifying invariants of the UML metamodel, UML modelers can use OCL to specify application-specific constraints in their models.

OCL is used in the UML Semantics chapter to specify the well-formedness rules of the metaclasses comprising the UML metamodel. A well-formedness rule in the static semantics chapters in the UML Semantics section normally contains an OCL expression, specifying an invariant for the associated metaclass. The grammar for OCL is specified at the end of this chapter. A parser generated from this grammar has correctly parsed all the constraints in the UML Semantics section, a process which improved the correctness of the specifications for OCL and UML.

6.1.1 Why OCL?

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division, and has its roots in the Syntropy method.

OCL is a pure expression language; therefore, an OCL expression is guaranteed to be without side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to *specify* a state change (for example, in a post-condition).

OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, not everything in it is promised to be directly executable.

OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type. In addition, OCL includes a set of supplementary predefined types (these are described in Section 6.8, “Predefined OCL Types,” on page 6-29).

As a specification language, all implementation issues are out of scope and cannot be expressed in OCL.

The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation.

6.1.2 Where to Use OCL

OCL can be used for a number of different purposes:

- To specify invariants on classes and types in the class model
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on Operations and Methods
- To describe Guards
- As a navigation language
- To specify constraints on operations

Within the UML Semantics chapter, OCL is used in the well-formedness rules as invariants on the metaclasses in the abstract syntax. In several places, it is also used to define ‘additional’ operations which are used in the well-formedness rules. Starting with UML 1.4, these additional operations can be formally defined using «definition» constraints and let-expressions.

6.2 Introduction

6.2.1 Legend

Text written in the courier typeface as shown below is an OCL expression.

```
'This is an OCL expression'
```

The *context* keyword introduces the context for the expression. The keyword *inv*, *pre* and *post* denote the stereotypes, respectively «invariant», «precondition», and «postcondition», of the constraint. The actual OCL expression comes after the colon.

```
context TypeName inv:
'this is an OCL expression with stereotype <<invariant>> in the
context of TypeName' = 'another string'
```

In the example, the keywords of OCL are written in boldface in this document. The boldface has no formal meaning, but is used to make the expressions more readable in this document. OCL expressions in this document are written using ASCII characters only.

Words in *Italics* within the main text of the paragraphs refer to parts of OCL expressions.

6.2.2 Example Class Diagram

Figure 6-1 on page 6-4 is used in the examples in this document.

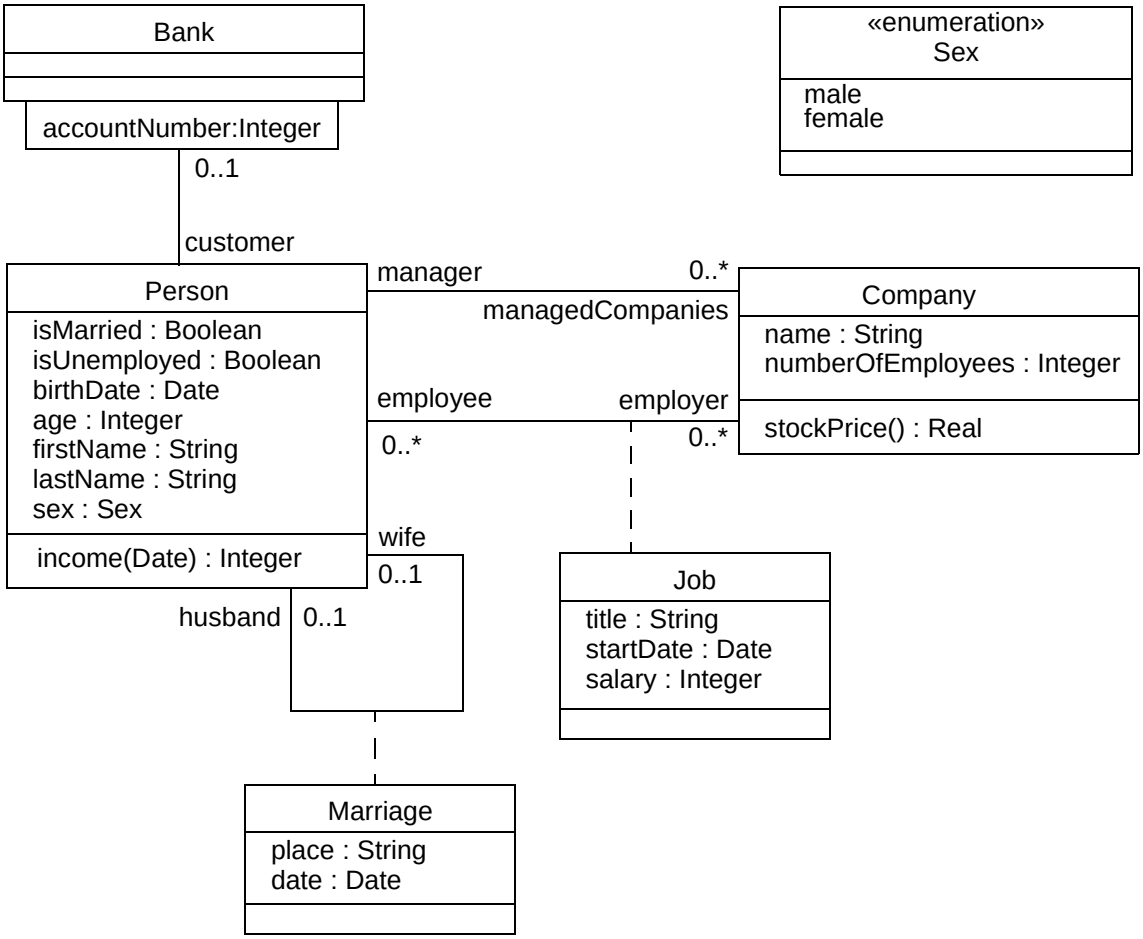


Figure 6-1 Class Diagram Example

6.3 Relation to the UML Metamodel

6.3.1 Self

Each OCL expression is written in the context of an instance of a specific type. In an OCL expression, the reserved word *self* is used to refer to the contextual instance. For instance, if the context is Company, then *self* refers to an instance of Company.

6.3.2 Specifying the UML context

The context of an OCL expression within a UML model can be specified through a so-called context declaration at the beginning of an OCL expression. The context declaration of the constraints in the following sections is shown.

If the constraint is shown in a diagram with the proper stereotype and the dashed lines to connect it to its contextual element, there is no need for an explicit context declaration in the test of the constraint. The context declaration is optional.

6.3.3 Invariants

The OCL expression can be part of an Invariant which is a Constraint stereotyped as an «invariant». When the invariant is associated with a Classifier, the latter is referred to as a “type” in this chapter. An OCL expression is an invariant of the type and must be true for all instances of that type at any time. (Note that all OCL expressions that express invariants are of the type Boolean.)

For example, if in the context of the Company type in Figure 6-1 on page 6-4, the following expression would specify an invariant that the number of employees must always exceed 50:

```
self.numberOfEmployees > 50
```

where *self* is an instance of type Company. (We can view *self* as the object from where we start the expression.) This invariant holds for every instance of the Company type.

The type of the contextual instance of an OCL expression, which is part of an invariant, is written with the *context* keyword, followed by the name of the type as follows. The label *inv:* declares the constraint to be an «invariant» constraint.

```
context Company inv:
    self.numberOfEmployees > 50
```

In most cases, the keyword *self* can be dropped because the context is clear, as in the above examples. As an alternative for self, a different name can be defined playing the part of self:

```
context c : Company inv:
    c.numberOfEmployees > 50
```

This invariant is equivalent to the previous one.

Optionally, the name of the constraint may be written after the *inv* keyword, allowing the constraint to be referenced by name. In the following example the name of the constraint is *enoughEmployees*. In the UML metamodel, this name is an attribute of the metaclass Constraint that is inherited from ModelElement.

```
context c : Company inv enoughEmployees:
    c.numberOfEmployees > 50
```

6.3.4 Pre- and Postconditions

The OCL expression can be part of a Precondition or Postcondition, corresponding to «precondition» and «postcondition» stereotypes of Constraint associated with an Operation or Method. The contextual instance *self* then is an instance of the type that owns the operation or method as a feature. The context declaration in OCL uses the *context* keyword, followed by the type and operation declaration.

The labels *pre:* and *post:* declare the constraints to be a «precondition» constraint and a «postcondition» constraint respectively.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
    pre :  param1 > ...
    post:  result = ...
```

The name *self* can be used in the expression referring to the object on which the operation was called. The reserved word *result* denotes the result of the operation, if there is one. The names of the parameters (*param1*) can also be used in the OCL expression. In the example diagram, we can write:

```
context Person::income(d : Date) : Integer
    post:  result = 5000
```

Optionally, the name of the precondition or postcondition may be written after the *pre* or *post* keyword, allowing the constraint to be referenced by name. In the following example the name of the precondition is *parameterOk* and the name of the postcondition is *resultOk*. In the UML metamodel, these names are attributes of the metaclass Constraint that is inherited from ModelElement.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
    pre parameterOk:  param1 > ...
    post   resultOk:  result = ...
```

6.3.5 Package context

The above context declaration is precise enough when the package in which the Classifier belongs is clear from the environment. To specify explicitly in which package invariant, pre or postcondition Constraints belong, these constraints can be enclosed between 'package' and 'endpackage' statements. The package statements have the syntax:

```
package Package::SubPackage

context X inv:
    ... some invariant ...
context X::operationName(..)
    pre: ... some precondition ...

endpackage
```

An OCL file (or stream) may contain any number package statements, thus allowing all invariant, preconditions, and postconditions to be written down and stored in one file. This file may co-exist with a UML model as a separate entity.

6.3.6 General Expressions

Any OCL expression can be used as the value for an attribute of the UML metaclass Expression or one of its subtypes. In that case, the semantics section describes the meaning of the expression.

6.4 Basic Values and Types

In OCL, a number of basic types are predefined and available to the modeler at all times. These predefined value types are independent of any object model and part of the definition of OCL.

The most basic value in OCL is a value of one of the basic types. Some basic types used in the examples in this document, with corresponding examples of their values, are shown in Table 6-1.

Table 6-1 Basic Types

type	values
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

OCL defines a number of operations on the predefined types. Table 6-2 gives some examples of the operations on the predefined types. See Section 6.8, “Predefined OCL Types,” on page 6-29 for a complete list of all operations.

Table 6-2 Operations on predefined types

type	operations
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	toUpper(), concat()

The complete list of operations provided for each type is described at the end of this chapter. Collection, Set, Bag, and Sequence are basic types as well. Their specifics will be described in the upcoming sections.

6.4.1 Types from the UML Model

Each OCL expression is written in the context of a UML model, a number of classifiers (types/classes, ...), their features and associations, and their generalizations. All classifiers from the UML model are types in the OCL expressions that are attached to the model.

6.4.2 Enumeration Types

Enumerations are Datatypes in UML and have a name, just like any other Classifier. An enumeration defines a number of enumeration literals, that are the possible values of the enumeration. Within OCL one can refer to the value of an enumeration. When we have Datatype named Sex with values ‘female’ or ‘male’ they can be used as follows:

```
context Person inv:
       sex = Sex::male
```

6.4.3 Let Expressions and «definition» Constraints

Sometimes a sub-expression is used more than once in a constraint. The *let* expression allows one to define an attribute or operation that can be used in the constraint.

```
context Person inv:
    let income : Integer = self.job.salary->sum()
    let hasTitle(t : String) : Boolean =
                    self.job->exists(title = t) in
    if isUnemployed then
        self.income < 100
    else
        self.income >= 100 and self.hasTitle('manager')
    endif
```

A let expression may be included in an invariant or pre- or postcondition. It is then only known within this specific constraint. To enable reuse of let variables/operations one can use a Constraint with the stereotype «definition», in which let variables/operations are defined. This «definition» Constraint must be attached to a Classifier and may only contain let definitions. All variables and operations defined in the «definition» constraint are known in the same context as where any property of the Classifier can be used. In essence, such variables and operations are psuedo-attributes and psuedo-operations of the classifier. They are used in an OCL expression in exactly the same way as attributes or operations are used. The textual notation for a «definition» Constraint uses the keyword ‘def’ as shown below:

```
context Person def:
    let income : Integer = self.job.salary->sum()
    let hasTitle(t : String) : Boolean =
                    self.job->exists(title = t)
```

The names of the attributes / operations in a let expression may not conflict with the names of respective attributes/associationEnds and operations of the Classifier. Also, the names of all let variables and operations connected with a Classifier must be unique.

6.4.4 Type Conformance

OCL is a typed language and the basic value types are organized in a type hierarchy. This hierarchy determines conformance of the different types to each other. You cannot, for example, compare an Integer with a Boolean or a String.

An OCL expression in which all the types conform is a valid expression. An OCL expression in which the types don't conform is an invalid expression. It contains a type *conformance error.* A type *type1* conforms to a type *type2* when an instance of *type1* can be substituted at each place where an instance of *type2* is expected. The type conformance rules for types in the class diagrams are simple.

- Each type conforms to each of its supertypes.
- Type conformance is transitive: if *type1* conforms to *type2*, and *type2* conforms to *type3*, then *type1* conforms to *type3*.

The effect of this is that a type conforms to its supertype, and all the supertypes above. The type conformance rules for the value types are listed in Table 6-3.

Table 6-3 Type conformance rules

Type	Conforms to/Is a subtype of
Set(T)	Collection(T)
Sequence(T)	Collection(T)
Bag(T)	Collection(T)
Integer	Real

The conformance relation between the collection types only holds if they are collections of element types that conform to each other. See Section6.5.14, "Collection Type Hierarchy and Type Conformance Rules," on page 6-21 for the complete conformance rules for collections.

Table 6-4 provides examples of valid and invalid expressions.

Table 6-4 Valid expressions

OCL expression	valid	explanation
1 + 2 * 34	yes	
1 + 'motorcycle'	no	type String does not conform to type Integer
23 * false	no	type Boolean does not conform to Integer
12 + 13.5	yes	

6.4.5 Re-typing or Casting

In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error.

When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation *oclAsType(OclType)*. This operation results in the same object, but the known type is the argument *OclType*. When there is an object *object* of type *Type1* and *Type2* is another type, it is allowed to write:

```
object.oclAsType(Type2) --- evaluates to object with type Type2
```

An object can only be re-typed to one of its subtypes; therefore, in the example, *Type2* must be a subtype of *Type1*.

If the actual type of the object is not a subtype of the type to which it is re-typed, the expression is undefined (see Section 6.4.10, “Undefined Values,” on page 6-11).

6.4.6 Precedence Rules

The precedence order for the operations, starting with highest precedence, in OCL is:

- @pre
- dot and arrow operations: ‘.’ and ‘->’
- unary ‘not’ and unary minus ‘-’
- ‘*’ and ‘/’
- ‘+’ and binary ‘-’
- ‘if-then-else-endif’
- ‘<’, ‘>’, ‘<=’, ‘>=’
- ‘=’, ‘<>’
- ‘and’, ‘or’ and ‘xor’
- ‘implies’

Parentheses ‘(’ and ‘)’ can be used to change precedence.

6.4.7 Use of Infix Operators

The use of infix operators is allowed in OCL. The operators ‘+’, ‘-’, ‘*’, ‘/’, ‘<‘, ‘>’, ‘<>’ ‘<=’ ‘>=’, ‘and’, ‘or’, and ‘xor’ are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the '+' operation on `a` with `b` as the parameter to the operation.

The infix operators defined for a type must have exactly one parameter. For the infix operators '<', '>', '<=', '>=', '<>', 'and', 'or', and 'xor' the return type must be Boolean.

6.4.8 Keywords

Keywords in OCL are reserved words. That means that the keywords cannot occur anywhere in an OCL expression as the name of a package, a type or a property. The list of keywords is shown below:

`if`	`implies`
`then`	`endpackage`
`else`	`package`
`endif`	`context`
`not`	`def`
`let`	`inv`
`or`	`pre`
`and`	`post`
`xor`	`in`

6.4.9 Comment

Comments in OCL are written following two successive dashes (minus signs). Everything immediately following the two dashes up to and including the end of line is part of the comment. For example:

```
-- this is a comment
```

6.4.10 Undefined Values

Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

There are two exceptions to this for the Boolean operators:

- True OR-ed with anything is True
- False AND-ed with anything is False

The above two rules are valid irrespective of the order of the arguments and the above rules are valid whether or not the value of the other sub-expression is known.

6.5 Objects and Properties

OCL expressions can refer to Classifiers; for example, types, classes, interfaces, associations (acting as types), and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types, etc. can be used. In a class model, an operation or method is defined to be side-effect-free if the isQuery attribute of the operations is true. For the purpose of this document, we will refer to attributes, association-ends, and side-effect-free methods and operations as being *properties*. A property is one of:

- an Attribute
- an AssociationEnd
- an Operation with *isQuery* being true
- a Method with *isQuery* being true

6.5.1 Properties

The value of a property on an object that is defined in a class diagram is specified by a dot followed by the name of the property.

```
context AType inv:
    self.property
```

If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self*.

6.5.2 Properties: Attributes

For example, the age of a Person is written as *self.age*:

```
context Person inv:
    self.age > 0
```

The value of the subexpression *self.age* is the value of the *age* attribute on the particular instance of Person identified by *self*. The type of this subexpression is the type of the attribute *age*, which is the basic type Integer.

Using attributes, and operations defined on the basic value types, we can express calculations etc. over the class model. For example, a business rule might be “the age of a Person is always greater than zero.” This can be stated as shown in the invariant above.

6.5.3 Properties: Operations

Operations may have parameters. For example, as shown earlier, a Person object has an income expressed as a function of the date. This operation would be accessed as follows, for a Person *aPerson* and a date *aDate*:

```
aPerson.income(aDate)
```

The operation itself could be defined by a postcondition constraint. This is a constraint that is stereotyped as «postcondition». The object that is returned by the operation can be referred to by *result*. It takes the following form:

```
context Person::income (d: Date) : Integer
    post: result = age * 1000
```

The right-hand-side of this definition may refer to the operation being defined; that is, the definition may be recursive as long as the recursion is not infinite. The type of *result* is the return type of the operation, which is Integer in the above example.

To refer to an operation or a method that doesn't take a parameter, parentheses with an empty argument list are mandatory:

```
context Company inv:
    self.stockPrice() > 0
```

6.5.4 Properties: Association Ends and Navigation

Starting from a specific object, we can navigate an association on the class diagram to refer to other objects and their properties. To do so, we navigate the association by using the opposite association-end:

```
object.rolename
```

The value of this expression is the set of objects on the other side of the *rolename* association. If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of this expression is an object. In the example class diagram, when we start in the context of a Company; that is, *self* is an instance of Company, we can write:

```
context Company
    inv: self.manager.isUnemployed = false
    inv: self.employee->notEmpty()
```

In the first invariant *self.manager* is a Person, because the multiplicity of the association is one. In the second invariant *self.employee* will evaluate in a Set of Persons. By default, navigation will result in a Set. When the association on the Class Diagram is adorned with {ordered}, the navigation results in a Sequence.

Collections, like Sets, Bags, and Sequences are predefined types in OCL. They have a large number of predefined operations on them. A property of the collection itself is accessed by using an arrow '->' followed by the name of the property. The following example is in the context of a person:

```
context Person inv:
    self.employer->size() < 3
```

This applies the *size* property on the Set *self.employer*, which results in the number of employers of the Person *self*.

```
context Person inv:
    self.employer->isEmpty()
```

This applies the *isEmpty* property on the Set *self.employer*. This evaluates to true if the set of employers is empty and false otherwise.

6.5.4.1 Missing Rolenames

When a rolename is missing at one of the ends of an association, the name of the type at the association end, starting with a lowercase character, is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

6.5.4.2 Navigation over Associations with Multiplicity Zero or One

Because the multiplicity of the role manager is one, *self.manager* is an object of type Person. Such a single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set. This is shown in the following example:

```
context Company inv:
    self.manager->size() = 1
```

The sub-expression *self.manager* is used as a Set, because the arrow is used to access the *size* property on Set. This expression evaluates to true.

The following example shows how a property of a collection can be used.

```
context Company inv:
    self.manager->foo
```

The sub-expression *self.manager* is used as Set, because the arrow is used to access the *foo* property on the Set. This expression is incorrect, because *foo* is not a defined property of Set.

```
context Company inv:
    self.manager.age> 40
```

The sub-expression *self.manager* is used as a Person, because the dot is used to access the *age* property of Person.

In the case of an optional (0..1 multiplicity) association, this is especially useful to check whether there is an object or not when navigating the association. In the example we can write:

```
context Person inv:
    self.wife->notEmpty() implies self.wife.sex = Sex::female
```

6.5.4.3 Combining Properties

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to a specific object of a specific type. After obtaining a result, one can always apply another property to the result to get a new result value. Therefore, each OCL expression can be read and evaluated left-to-right.

Following are some invariants that use combined properties on the example class diagram:

[1] Married people are of age >= 18

```
context Person inv:
    self.wife->notEmpty() implies self.wife.age >= 18 and
    self.husband->notEmpty() implies self.husband.age >= 18
```

[2] a company has at most 50 employees

```
context Company inv:
    self.employee->size() <= 50
```

6.5.5 Navigation to Association Classes

To specify navigation to association classes (Job and Marriage in the example), OCL uses a dot and the name of the association class starting with a lowercase character:

```
context Person inv:
    self.job
```

The sub-expression *self.job* evaluates to a Set of all the jobs a person has with the companies that are his/her employer. In the case of an association class, there is no explicit rolename in the class diagram. The name *job* used in this navigation is the name of the association class starting with a lowercase character, similar to the way described in the section “Missing Rolenames” above.

In case of a recursive association, that is an association of a class with itself, the name of the association class alone is not enough. We need to distinguish the direction in which the association is navigated as well as the name of the association class. Take the following model as an example.

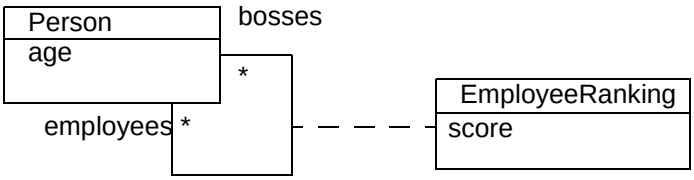


Figure 6-2 Navigating recursive association classes

When navigating to an association class such as *employeeRanking* there are two possibilities depending on the direction. For instance, in the above example, we may navigate towards the *employees* end, or the *bosses* end. By using the name of the association class alone, these two options cannot be distinguished. To make the distinction, the rolename of the direction in which we want to navigate is added to the association class name, enclosed in square brackets.

In the expression

```
context Person inv:
    self.employeeRanking[bosses]->sum() > 0
```

the *self.employeeRanking[bosses]* evaluates to the set of *EmployeeRankings* belonging to the collection of *bosses*. And in the expression

```
context Person inv:
    self.employeeRanking[employees]->sum() > 0
```

the *self.employeeRanking[employees]* evaluates to the set of *EmployeeRankings* belonging to the collection of *employees*. The unqualified use of the association class name is not allowed in such a recursive situation. Thus, the following example is invalid:

```
context Person inv:
    self.employeeRanking->sum() > 0 -- INVALID!
```

In a non-recursive situation, the association class name alone is enough, although the qualified version is allowed as well. Therefore, the examples at the start of this section could also be written as:

```
context Person inv:
    self.job[employer]
```

6.5.6 Navigation from Association Classes

We can navigate from the association class itself to the objects that participate in the association. This is done using the dot-notation and the role-names at the association-ends.

```
context Job
    inv: self.employer.numberOfEmployees >= 1
    inv: self.employee.age > 21
```

Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using the arrow (->).

6.5.7 Navigation through Qualified Associations

Qualified associations use one or more qualifier attributes to select the objects at the other end of the association. To navigate them, we can add the values for the qualifiers to the navigation. This is done using square brackets, following the role-name. It is permissible to leave out the qualifier values, in which case the result will be all objects at the other end of the association.

```
context Bank inv:
    self.customer
```

This results in a Set(Person) containing all customers of the Bank.

```
context Bank inv:
    self.customer[8764423]
```

This results in one Person, having accountnumber 8764423.

If there is more than one qualifier attribute, the values are separated by commas, in the order which is specified in the UML class model. It is not permissible to partially specify the qualifier attribute values.

6.5.8 Using Pathnames for Packages

Within UML, different types are organized in packages. OCL provides a way of explicitly referring to types in other packages by using a package-pathname prefix. The syntax is a package name, followed by a double colon:

```
Packagename::Typename
```

This usage of pathnames is transitive and can also be used for packages within packages:

```
Packagename1::Packagename2::Typename
```

6.5.9 Accessing overridden properties of supertypes

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the *oclAsType()* operation. Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

```
context B inv:
    self.oclAsType(A).p1  -- accesses the p1 property defined in A
    self.p1               -- accesses the p1 property defined in B
```

Figure 6-3 shows an example where such a construct is needed.

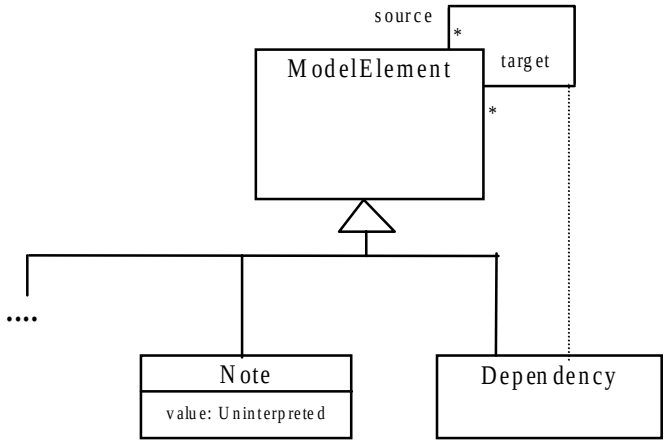


Figure 6-3 Accessing Overridden Properties Example

In this model fragment there is an ambiguity with the OCL expression on Dependency:

```
context Dependency inv:
    self.source <> self
```

This can either mean normal association navigation, which is inherited from ModelElement, or it might also mean navigation through the dotted line as an association class. Both possible navigations use the same role-name, so this is always ambiguous. Using *oclAsType()* we can distinguish between them with:

```
context Dependency
    inv: self.oclAsType(Dependency).source
    inv: self.oclAsType(ModelElement).source
```

6.5.10 Predefined properties on All Objects

There are several properties that apply to all objects, and are predefined in OCL. These are:

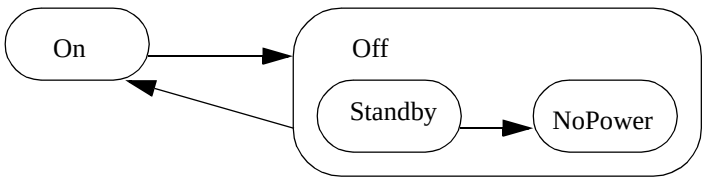
```
oclIsTypeOf(t : OclType)  : Boolean
oclIsKindOf(t : OclType)  : Boolean
oclInState(s : OclState)  : Boolean
oclIsNew()                : Boolean
oclAsType(t : OclType) : instance of OclType
```

The operation is *oclTypeOf* results in true if the *type* of self and *t* are the same. For example:

```
context Person
    inv: self.oclIsTypeOf( Person )      -- is true
    inv: self.oclIsTypeOf( Company)      -- is false
```

The above property deals with the direct type of an object. The *oclIsKindOf* property determines whether *t* is either the direct type or one of the supertypes of an object.

The operation *oclInState(s)* results in true if the object is in the state *s*. Values for *s* are the names of the states in the statemachine(s) attached to the Classifier of *object*. For nested states the statenames can be combined using the double colon '::' .



In the example statemachine above, values for *s* can be *On, Off, Off::Standby, Off::NoPower*. If the classifier of *object* has the above associated statemachine valid OCL expressions are:

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off:NoPower)
```

If there are multiple statemachines attached to the object's classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double semicolon ::, as with nested states.

The operation *oclIsNew* evaluates to true if, used in a postcondition, the object is created during performing the operation; that is, it didn't exist at precondition time.

6.5.11 Features on Classes Themselves

All properties discussed until now in OCL are properties on instances of classes. The types are either predefined in OCL or defined in the class model. In OCL, it is also possible to use features defined on the types/classes themselves. These are, for example, the *class*-scoped features defined in the class model. Furthermore, several features are predefined on each type.

A predefined feature on each type is *allInstances*, which results in the Set of all instances of the type in existence at the specific time when the expression is evaluated. If we want to make sure that all instances of Person have unique names, we can write:

```
context Person inv:
    Person.allInstances->forAll(p1, p2 |
                                p1 <> p2 implies p1.name <> p2.name)
```

The *Person.allInstances* is the set of all persons and is of type Set(Person). It is the set of all persons that exist at the snapshot in time that the expression is evaluated.

Note – The use of *allInstances* has some problems and its use is discouraged in most cases. The first problem is best explained by looking at the types like Integer, Real and String. For these types the meaning of *allInstances* is undefined. What does it mean for an Integer to exist? The evaluation of the expression *Integer.allInstances* results in an infinite set and is therefore undefined within OCL. The second problem with *allInstances* is that the existence of objects must be considered within some overall context, like a system or a model. This overall context must be defined, which is not done within OCL. A recommended style is to model the overall contextual system explicitly as an object within the system and navigate from that object to its containing instances without using *allInstances*.

6.5.12 Collections

Single navigation results in a Set, combined navigations in a Bag, and navigation over associations adorned with {ordered} results in a Sequence. Therefore, the collection types play an important role in OCL expressions.

The type Collection is predefined in OCL. The Collection type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections; *isQuery* is always true. They may result in a collection, but rather than changing the original collection they project the result into a new one.

Collection is an abstract type, with the concrete collection types as its subtypes. OCL distinguishes three different collection types: Set, Sequence, and Bag. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, which may contain duplicates; that is, the same element may be in a bag twice or more. A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them. Sets, Sequences, and Bags can be specified by a literal in OCL. Curly brackets surround the elements of the collection, elements in the collection are written within, separated by commas. The type of the collection is written before the curly brackets:

```
Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }
```

A Sequence:

```
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
```

A bag:

```
Bag {1 , 3 , 4, 3, 5 }
```

Because of the usefulness of a Sequence of consecutive Integers, there is a separate literal to create them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer, *Int-expr1* and *Int-expr2*, separated by '..'. This denotes all the Integers between the values of *Int-expr1* and *Int-expr2*, including the values of *Int-expr1* and *Int-expr2* themselves:

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

The complete list of Collection operations is described at the end of this chapter.

Collections can be specified by a literal, as described above. The only other way to get a collection is by navigation. To be more precise, the only way to get a Set, Sequence, or Bag is:

1. a literal, this will result in a Set, Sequence, or Bag:

```
Set      {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
Sequence {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
Bag      {1, 2, 3, 2, 1}
```

2. a navigation starting from a single object can result in a collection:

```
context Company inv:
```

```
self.employee
```

3. operations on collections may result in new collections:

```
collection1->union(collection2)
```

6.5.13 Collections of Collections

Within OCL, all Collections of Collections are flattened automatically; therefore, the following two expressions have the same value:

```
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
Set{ 1, 2, 3, 4, 5, 6 }
```

6.5.14 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in Section 6.4.4, “Type Conformance,” on page 6-9, the following rules hold for all types, including the collection types:

- The types Set (X), Bag (X) and Sequence (X) are all subtypes of Collection (X).

Type conformance rules are as follows for the collection types:

- *Type1* conforms to *Type2* when they are identical (standard rule for all types).
- *Type1* conforms to *Type2* when it is a subtype of Type2 (standard rule for all types).
- *Collection(Type1)* conforms to *Collection(Type2)*, when *Type1* conforms to *Type2*.
- Type conformance is transitive: if *Type1* conforms to *Type2*, and *Type2* conforms to *Type3*, then *Type1* conforms to *Type3* (standard rule for all types).

For example, if *Bicycle* and *Car* are two separate subtypes of *Transport:*

`Set(Bicycle)` conforms to `Set(Transport)`
`Set(Bicycle)` conforms to `Collection(Bicycle)`
`Set(Bicycle)` conforms to `Collection(Transport)`

Note that Set(Bicycle) does not conform to Bag(Bicycle), nor the other way around. They are both subtypes of Collection(Bicycle) at the same level in the hierarchy.

6.5.15 Previous Values in Postconditions

As stated in Section 6.3.4, “Pre- and Postconditions,” on page 6-6, OCL can be used to specify pre- and post-conditions on operations and methods in UML. In a postcondition, the expression can refer to two sets of values for each property of an object:

- the value of a property at the start of the operation or method
- the value of a property upon completion of the operation or method

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword ‘*@pre*’:

```
context Person::birthdayHappens()
    post: age = age@pre + 1
```

The property *age* refers to the property of the instance of Person on which executes the operation. The property *age@pre* refers to the value of the property *age* of the Person that executes the operation, at the start of the operation.

If the property has parameters, the '@pre' is postfixed to the propertyname, before the parameters.

```
context Company::hireEmployee(p : Person)
    post: employees = employees@pre->including(p) and
        stockprice() = stockprice@pre() + 10
```

The above operation can also be specified by a postcondition and a precondition together:

```
context Company::hireEmployee(p : Person)
    pre : not employee->includes(p)
    post: employees->includes(p) and
         stockprice() = stockprice@pre() + 10
```

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c          -- takes the old value of property b of a, say x
                   -- and then the new value of c of x.
a.b@pre.c@pre      -- takes the old value of property b of a, say x
                   -- and then the old value of c of x.
```

The '@pre' postfix is allowed only in OCL expressions that are part of a Postcondition. Asking for a current property of an object that has been destroyed during execution of the operation results in Undefined. Also, referring to the previous value of an object that has been created during execution of the operation results in Undefined.

6.6 Collection Operations

OCL defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of projecting new collections from existing ones. The different constructs are described in the following sections.

6.6.1 Select and Reject Operations

Sometimes an expression using operations and navigations delivers a collection, while we are interested only in a special subset of the collection. OCL has special constructs to specify a selection from a specific collection. These are the *select* and *reject* operations. The select specifies a subset of a collection. A select is an operation on a collection and is specified using the arrow-syntax:

```
collection->select( ... )
```

The parameter of select has a special syntax that enables one to specify which elements of the collection we want to select. There are three different forms, of which the simplest one is:

```
collection->select( boolean-expression )
```

This results in a collection that contains all the elements from *collection* for which the *boolean-expression* evaluates to true. To find the result of this expression, for each element in *collection* the expression *boolean-expression* is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not. As an example, the following OCL expression specifies that the collection of all the employees older than 50 years is not empty:

```
context Company inv:
    self.employee->select(age > 50)->notEmpty()
```

The *self.employee* is of type Set(Person). The *select* takes each person from *self.employee* and evaluates *age > 50* for this person. If this results in *true*, then the person is in the result Set.

As shown in the previous example, the context for the expression in the select argument is the element of the collection on which the select is invoked. Thus the *age* property is taken in the context of a person.

In the above example, it is impossible to refer explicitly to the persons themselves; you can only refer to properties of them. To enable to refer to the persons themselves, there is a more general syntax for the select expression:

```
collection->select( v | boolean-expression-with-v )
```

The variable *v* is called the iterator. When the select is evaluated, *v* iterates over the *collection* and the *boolean-expression-with-v* is evaluated for each *v*. The *v* is a reference to the object from the collection and can be used to refer to the objects themselves from the *collection*. The two examples below are identical:

```
context Company inv:
    self.employee->select(age > 50)->notEmpty()
context Company inv:
    self.employee->select(p | p.age > 50)->notEmpty()
```

The result of the complete select is the collection of persons *p* for which the *p.age > 50* evaluates to True. This amounts to a subset of *self.employee*.

As a final extension to the select syntax, the expected type of the variable v can be given. The select now is written as:

```
collection->select( v : Type | boolean-expression-with-v )
```

The meaning of this is that the objects in *collection* must be of type *Type*. The next example is identical to the previous examples:

```
context Company inv:
    self.employee.select(p : Person | p.age > 50)->notEmpty()
```

The compete select syntax now looks like one of:

```
collection->select( v : Type | boolean-expression-with-v )
```

```
collection->select( v | boolean-expression-with-v )
collection->select( boolean-expression )
```

The *reject* operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax:

```
collection->reject( v : Type | boolean-expression-with-v )
collection->reject( v | boolean-expression-with-v )
collection->reject( boolean-expression )
```

As an example, specify that the collection of all the employees who are **not** married is empty:

```
context Company inv:
    self.employee->reject( isMarried )->isEmpty()
```

The reject operation is available in OCL for convenience, because each reject can be restated as a select with the negated expression. Therefore, the following two expressions are identical:

```
collection->reject( v : Type | boolean-expression-with-v )
collection->select( v : Type  | not (boolean-expression-with-v) )
```

6.6.2 Collect Operation

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection that is derived from some other collection, but which contains different objects from the original collection; that is, it is not a sub-collection, we can use a *collect* operation. The collect operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

The value of the reject operation is the collection of the results of all the evaluations of *expression-with-v.*

An example: specify the collection of *birthDates* for all employees in the context of a company. This can be written in the context of a Company object as one of:

```
self.employee->collect( birthDate )
self.employee->collect( person | person.birthDate )
self.employee->collect( person : Person | person.birthDate )
```

An important issue here is that the resulting collection is not a Set, but a Bag. When more than one employee has the same value for *birthDate*, this value will be an element of the resulting Bag more than once. The Bag resulting from the *collect* operation always has the same size as the original collection.

It is possible to make a Set from the Bag, by using the asSet property on the Bag. The following expression results in the Set of different *birthDates* from all employees of a Company:

```
self.employee->collect( birthDate )->asSet()
```

6.6.2.1 Shorthand for Collect

Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable. Instead of

```
self.employee->collect(birthdate)
```

we can also write:

```
self.employee.birthdate
```

In general, when we apply a property to a collection of Objects, then it will automatically be interpreted as a *collect* over the members of the collection with the specified property.

For any *propertyname* that is defined as a property on the objects in a collection, the following two expressions are identical:

```
collection.propertyname
collection->collect(propertyname)
```

and so are these if the property is parameterized:

```
collection.propertyname(par1, par2, ...)
collection->collect(propertyname(par1, par2, ...)
```

6.6.3 ForAll Operation

Many times a constraint is needed on all elements of a collection. The forAll operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll( v : Type | boolean-expression-with-v )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( boolean-expression )
```

This forAll expression results in a Boolean. The result is true if the *boolean-expression-with-v* is true for all elements of *collection*. If the *boolean-expression-with-v* is false for one or more *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
context Company
    inv:    self.employee->forAll( forename = 'Jack' )
    inv:    self.employee->forAll( p | p.forename = 'Jack' )
    inv:    self.employee->forAll( p : Person | p.forename = 'Jack' )
```

These invariants evaluate to true if the forename feature of each employee is equal to ‘Jack.’

The forAll operation has an extended variant in which more then one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a forAll on the Cartesian product of the collection with itself.

```
context Company inv:
    self.employee->forAll( e1, e2 |
                    e1 <> e2 implies e1.forename <> e2.forename)
context Company inv:
    self.employee->forAll( e1, e2 : Person |
                    e1 <> e2 implies e1.forename <> e2.forename)
```

This expression evaluates to true if the forenames of all employees are different. It is semantically equivalent to:

```
context Company inv:
    self.employee->forAll(e1 | self.employee->forAll (e2 |
                      e1 <> e2 implies e1.forename <> e2.forename)))
```

6.6.4 Exists Operation

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The *exists* operation in OCL allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists( v : Type | boolean-expression-with-v )
collection->exists( v | boolean-expression-with-v )
collection->exists( boolean-expression )
```

This exists operation results in a Boolean. The result is true if the *boolean-expression-with-v* is true for at least one element of *collection*. If the *boolean-expression-with-v* is false for all *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
context Company inv:
    self.employee->exists( forename = 'Jack' )
context Company inv:
    self.employee->exists( p | p.forename = 'Jack' )
context Company inv:
    self.employee->exists( p : Person | p.forename = 'Jack' )
```

These expressions evaluate to true if the forename feature of at least one employee is equal to ‘Jack.’

6.6.5 Iterate Operation

The *iterate* operation is slightly more complicated, but is very generic. The operations *reject, select, forAll, exists, collect* can all be described in terms of *iterate*.

An accumulation builds one value by iterating over a collection.

```
collection->iterate( elem : Type; acc : Type = <expression> |
```

```
                                    expression-with-elem-and-acc )
```

The variable *elem* is the iterator, as in the definition of *select*, *forAll*, etc. The variable *acc* is the accumulator. The accumulator gets an initial value <*expression*>.

When the iterate is evaluated, *elem* iterates over the *collection* and the *expression-with-elem-and-acc* is evaluated for each *elem*. After each evaluation of *expression-with-elem-and-acc*, its value is assigned to *acc*. In this way, the value of *acc* is built up during the iteration of the collection. The collect operation described in terms of iterate will look like:

```
    collection->collect(x : T | x.property)
-- is identical to:
    collection->iterate(x : T; acc : T2 = Bag{} |
                                    acc->including(x.property))
```

Or written in Java-like pseudocode the result of the iterate can be calculated as:

```
iterate(elem : T; acc : T2 = value)
{
   acc = value;
  for(Enumeration e = collection.elements() ; e.hasMoreElements();
){
       elem = e.nextElement();
       acc  = <expression-with-elem-and-acc>
   }
}
```

Although the Java pseudo code uses a ‘next element,’ the *iterate* operation is defined for each collection type and the order of the iteration through the elements in the collection is not defined for Set and Bag. For a Sequence the order is the order of the elements in the sequence.

6.6.6 Iterators in Collection Operations

The collection operations that take an OclExpression as parameter may all have an optional iterator declaration. For any operation name *op*, the syntax options are:

```
collection->op( iter : Type | OclExpression )
collection->op( iter | OclExpression )
collection->op( OclExpression )
```

6.6.7 Resolving Properties

For any property (attribute, operation, or navigation), the full notation includes the object of which the property is taken. As seen in Section 6.3.3, “Invariants,” on page 6-5, *self* can be left implicit, and so can the iterator variables in collection operations. At any place in an expression, when an iterator is left out, an implicit iterator-variable is introduced. For example in:

```
context Person inv:
    employer->forAll( employee->exists( lastName = name) )
```

three implicit variables are introduced. The first is *self*, which is always the instance from which the constraint starts. Secondly an implicit iterator is introduced by the *forAll* and third by the *exists*. The implicit iterator variables are unnamed. The properties *employer*, *employee*, *lastName* and *name* all have the object on which they are applied left out. Resolving these goes as follows:

- At the place of *employer* there is one implicit variable: *self : Person.* Therefore *employer* must be a property of *self.*
- At the place of *employee* there are two implicit variables: *self : Person* and *iter1 : Company.* Therefore *employer* must be a property of either *self* or *iter1*. If *employee* is a property of both *self* and *iter1*, then this is unambiguous and the instance on which *employee* is applied must be stated explicitly. In this case only *iter1.employee* is possible.
- At the place of *lastName* and *name* there are three implicit variables: *self : Person* , *iter1 : Company* and *iter2 : Person.* Therefore *lastName* and *name* must both be a property of either *self* or *iter1* or *iter2*. Property *name* is a property of *iter1*. However, *lastName* is a property of both *self* and *iter2*. This is ambiguous and therefore the OCL expression is incorrect. The expression must state either *self.lastName* or define the *iter2* iterator variable explicit and state *iter2.lastName*.

Both of the following invariant constraints are correct:

```
context Person
    inv: employer->forAll( employee->exists( p | p.lastName = name) )
    inv: employer->forAll( employee->exists( self.lastName = name) )
```

6.7 The Standard OCL Package

Each UML model that uses OCL constraints contains a predefined standard package called “UML_OCL.” This package is used by default in all other packages in the model to evaluate OCL expressions. This package contains all predefined OCL types and their features.

To extend the predefined OCL types, a modeler should define a separate package. The standard OCL package can be imported, and each OCL type can be extended with new features.

To specify that a package used the predefined OCL types from a user defined package instead of the standard package, the using package must define a Dependency with stereotype «OCL_Types» to the package that defines the extended OCL types.

A constraint on the user defined OCL package is that as a minimum all predefined OCL types with all of their features must be defined. The user defined package must be a proper extension to the standard OCL package.

6.8 Predefined OCL Types

This section contains all standard types defined within OCL, including all the properties defined on those types. Its signature and a description of its semantics define each property. Within the description, the reserved word ‘result’ is used to refer to the value that results from evaluating the property. In several places, post conditions are used to describe properties of the result. When there is more than one postcondition, all postconditions must be true.

6.8.1 Basic Types

The basic types used are Integer, Real, String, and Boolean. They are supplemented with OclExpression, OclType, and OclAny.

6.8.1.1 OclType

All types defined in a UML model, or pre-defined within OCL, have a type. This type is an instance of the OCL type called OclType. Access to this type allows the modeler limited access to the meta-level of the model. This can be useful for advanced modelers.

Properties of OclType, where the instance of OclType is called t*ype*.

type.name() : String

The name of *type*.

type.attributes() : Set(String)

The set of names of the attributes of *type*, as they are defined in the model.

type.associationEnds() : Set(String)

The set of names of the navigable associationEnds of *type*, as they are defined in the model.

type.operations() : Set(String)

The set of names of the operations of *type*, as they are defined in the model.

type.supertypes() : Set(OclType)

The set of all direct supertypes of *type*.
post: type.allSupertypes()->includesAll(result)

type.allSupertypes() : Set(OclType)

The transitive closure of the set of all supertypes of *type*.

type.allInstances() : Set(type)

The set of all instances of *type* and all its subtypes in existence at the snapshot at the time that the expression is evaluated.

6.8.1.2 OclAny

Within the OCL context, the type OclAny is the supertype of all types in the model and the basic predefined OCL type. The predefined OCL Collection types are not subtypes of OclAny. Properties of OclAny are available on each object in all OCL expressions.

All classes in a UML model inherit all properties defined on OclAny. To avoid name conflicts between properties in the model and the properties inherited from OclAny, all names on the properties of OclAny start with 'ocl.' Although theoretically there may still be name conflicts, they can be avoided. One can also use the oclAsType() operation to explicitly refer to the OclAny properties.

Properties of OclAny, where the instance of OclAny is called *object*.

object = (object2 : OclAny) : Boolean

True if *object* is the same object as *object2*.

object <> (object2 : OclAny) : Boolean

True if *object* is a different object from *object2*.
post: result = not (object = object2)

object.oclIsKindOf(type : OclType) : Boolean

True if *type* is one of the types of *object*, or one of the supertypes (transitive) of the types of *object*.

object.oclIsTypeOf(type : OclType) : Boolean

True if *type* is equal to one of the types of *object*.

object.oclAsType(type : OclType) : type

Results in *object*, but of known type *type*.
Results in Undefined if the actual type of *object* is not type or one of its subtypes.
pre : object.oclIsKindOf(type)
post: result = object
post: result.oclIsKindOf(type)

6.8.1.3

object.oclInState(state : OclState) : Boolean

Results in true if *object* is in the state *state*, otherwise results in false. The argument is a name of a state in the state machine corresponding with the class of *object*.

6.8.1.4

object.oclIsNew() : Boolean

Can only be used in a postcondition.
Evaluates to true if the *object* is created during performing the operation. That is it didn't exist at precondition time.

6.8.1.5 OclState

The type OclState is used as a parameter for the operation *oclInState*. There are no properties defined on OclState. One can only specify an OclState by using the name of the state, as it appears in a statemachine. These names can be fully qualified by the nested states and statemachine that contain them.

6.8.1.6 OclExpression

Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression. This type and its properties are used to define the semantics of properties that take an expression as one of their parameters: select, collect, forAll, etc.

An OclExpression includes the optional iterator variable and type and the optional accumulator variable and type.

Properties of OclExpression, where the instance of OclExpression is called *expression*.

expression.evaluationType() : OclType

The type of the object that results from evaluating *expression*.

6.8.1.7 Real

The OCL type Real represents the mathematical concept of real. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

Properties of Real, where the instance of Real is called *r*.

r = (r2 : Real) : Boolean

True if *r* is equal to *r2*.

r <> (r2 : Real) : Boolean

True if *r* is not equal to *r2*.
post: result = not (r = r2)

r + (r2 : Real) : Real

The value of the addition of *r* and *r2*.

r - (r2 : Real) : Real

The value of the subtraction of *r2* from *r*.

r * (r2 : Real) : Real

The value of the multiplication of *r* and *r2*.

- r : Real

The negative value of *r*.

r / (r2 : Real) : Real

The value of *r* divided by *r2*.

r.abs() : Real

The absolute value of *r*.
post: if r < 0 then result = - r else result = r endif

r.floor() : Integer

The largest integer which is less than or equal to *r*.
post: (result <= r) and (result + 1 > r)

r.round() : Integer

The integer that is closest to *r*. When there are two such integers, the largest one.
post: ((r - result) < r).abs() < 0.5) or ((r - result).abs() = 0.5 and (result > r))

r.max(r2 : Real) : Real

The maximum of *r* and *r2*.
post: if r >= r2 then result = r else result = r2 endif

r.min(r2 : Real) : Real

The minimum of *r* and *r2*.
post: if r <= r2 then result = r else result = r2 endif

r < (r2 : Real) : Boolean

True if *r1* is less than *r2*.

r > (r2 : Real) : Boolean

True if *r1* is greater than *r2*.
post: result = not (r <= r2)

r <= (r2 : Real) : Boolean

True if r1 is less than or equal to r2.
post: result = (r = r2) or (r < r2)

r >= (r2 : Real) : Boolean

True if r1 is greater than or equal to r2.
post: result = (r = r2) or (r > r2)

6.8.1.8 Integer

The OCL type Integer represents the mathematical concept of integer.

Properties of Integer, where the instance of Integer is called *i*.

i = (i2 : Integer) : Boolean

True if *i* is equal to *i2*.

- i : Integer

The negative value of *i*.

i + (i2 : Integer) : Integer

The value of the addition of *i* and *i2*.

i - (i2 : Integer) : Integer

The value of the subtraction of *i2* from *i*.

i * (i2 : Integer) : Integer

The value of the multiplication of *i* and *i2*.

i / (i2 : Integer) : Real

The value of *i* divided by *i2*.

i.abs() : Integer

The absolute value of *i*.
post: if i < 0 then result = - i else result = i endif

i.div(i2 : Integer) : Integer

The number of times that *i2* fits completely within *i*.
pre : i2 <> 0
post: if i / i2 >= 0 then result = (i / i2).floor() else result = -((-i/i2).floor()) endif

i.mod(i2 : Integer) : Integer

The result is *i* modulo *i2*.
post: result = i - (i.div(i2) * i2)

i.max(i2 : Integer) : Integer

The maximum of *i* an *i2*.
post: if i >= i2 then result = i else result = i2 endif

i.min(i2 : Integer) : Integer

The minimum of *i* an *i2*.
post: if i <= i2 then result = i else result = i2 endif

6.8.1.9 String

The OCL type String represents strings consisting of ASCII characters or multi-byte characters.

Properties of String, where the instance of String is called *string*.

string = (string2 : String) : Boolean

True if *string* and *string2* contain the same characters, in the same order.

string.size() : Integer

The number of characters in *string*.

string.concat(string2 : String) : String

The concatenation of *string* and *string2*.
post: result.size() = string.size() + string2.size()
post: result.substring(1, string.size()) = string
post: result.substring(string.size() + 1, result.size()) = string2

string.toUpper() : String

The value of *string* with all lowercase characters converted to uppercase characters.
post: result.size() = string.size()

string.toLower() : String

The value of *string* with all uppercase characters converted to lowercase characters.
post: result.size() = string.size()

string.substring(lower : Integer, upper : Integer) : String

The sub-string of *string* starting at character number *lower*, up to and including character number *upper*.

6.8.1.10 Boolean

The OCL type Boolean represents the common true/false values.

Features of Boolean, the instance of Boolean is called *b*.

b = (b2 : Boolean) : Boolean

Equal if *b* is the same as *b2*.

b or (b2 : Boolean) : Boolean

True if either *b* or *b2* is true.

b xor (b2 : Boolean) : Boolean

True if either *b* or *b2* is true, but not both.
post: (b or b2) and not (b = b2)

b and (b2 : Boolean) : Boolean

True if both *b1* and *b2* are true.

not b : Boolean

True if *b* is false.
post: if b then result = false else result = true endif

b implies (b2 : Boolean) : Boolean

True if *b* is false, or if *b* is true and *b2* is true.
post: (not b) or (b and b2)

if b then (expression1 : OclExpression)

else (expression2 : OclExpression) endif : expression1.evaluationType()

If *b* is true, the result is the value of evaluating *expression1*; otherwise, result is the value of evaluating *expression2*.

6.8.1.11 Enumeration

The OCL type Enumeration represents the enumerations defined in a UML model.

Features of Enumeration, the instance of Enumeration is called *enumeration*.

enumeration = (enumeration2 : Boolean) : Boolean

Equal if *enumeration* is the same as *enumeration2*.

enumeration <> (enumeration2 : Boolean) : Boolean

Equal if *enumeration* is not the same as *enumeration2*.
post: result = not (enumeration = enumeration2)

6.8.2 Collection-Related Types

The following sections define the properties on collections; that is, these properties are available on Set, Bag, and Sequence. As defined in this section, each collection type is actually a template with one parameter. 'T' denotes the parameter. A real collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types.

All collection operations with an OclExpression as parameter can have an iterator declarator.

6.8.2.1 Collection

Collection is the abstract supertype of all collection types in OCL. Each occurrence of an object in a collection is called an element. If an object occurs twice in a collection, there are two elements. This section defines the properties on Collections that have identical semantics for all collection subtypes. Some properties may be defined with the subtype as well, which means that there is an additional postcondition or a more specialized return value.

The definition of several common properties is different for each subtype. These properties are not mentioned in this section.

Properties of Collection, where the instance of Collection is called *collection*.

collection->size() : Integer

The number of elements in the collection *collection.*
post: result = collection->iterate(elem; acc : Integer = 0 | acc + 1)

collection->includes(object : OclAny) : Boolean

True if *object* is an element of *collection*, false otherwise.
post: result = (collection->count(object) > 0)

collection->excludes(object : OclAny) : Boolean

True if *object* is not an element of *collection*, false otherwise.
post: result = (collection->count(object) = 0)

collection->count(object : OclAny) : Integer

The number of times that *object* occurs in the collection *collection.*
post: result = collection->iterate(elem; acc : Integer = 0 |
if elem = object then acc + 1 else acc endif)

collection->includesAll(c2 : Collection(T)) : Boolean

Does *collection* contain all the elements of *c2* ?
post: result = c2->forAll(elem | collection->includes(elem))

collection->excludesAll(c2 : Collection(T)) : Boolean

Does *collection* contain none of the elements of *c2* ?
post: result = c2->forAll(elem | collection->excludes(elem))

collection->isEmpty() : Boolean

Is *collection* the empty collection?
post: result = (collection->size() = 0)

collection->notEmpty() : Boolean

Is *collection* not the empty collection?
post: result = (collection->size() <> 0)

collection->sum() : T

The addition of all elements in *collection*. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T and be both associative: (a+b)+c = a+(b+c), and commutative: a+b = b+a. Integer and Real fulfill this condition.

post: result = collection->iterate(elem; acc : T = 0 |
acc + elem)

collection->exists(expr : OclExpression) : Boolean

Results in true if *expr* evaluates to true for at least one element in *collection.*

post: result = collection->iterate(elem; acc : Boolean = false |
acc or expr)

collection->forAll(expr : OclExpression) : Boolean

Results in true if *expr* evaluates to true for each element in *collection*; otherwise, result is false.

post: result = collection->iterate(elem; acc : Boolean = true |
acc and expr)

collection->isUnique(expr : OclExpression) : Boolean

Results in true if *expr* evaluates to a different value for each element in *collection*; otherwise, result is false.

post: let values = collection->collect(expr) in
result = res->forAll(e | values->count(e) = 1)

collection->sortedBy(expr : OclExpression) : Sequence(T)

Results in the Sequence containing all elements of *collection*. The element for which *expr* has the lowest value comes first, and so on. The type of the *expr* expression must have the < operation defined. The < operation must return a Boolean value and must be transitive (i.e., if a < b and b < c, then a < c).

pre: expr.evaluationType().operations()->includes('<')
post: result->includesAll(collection) and collection->includesAll(result)

collection->iterate(expr : OclExpression) : expr.evaluationType()

Iterates over the collection. See Section 6.6.5, "Iterate Operation," on page 6-26 for a complete description. This is the basic collection operation with which the other collection operations can be described.

collection->any(expr : OclExpression) : T

Returns any element in the *collection* for which *expr* evaluates to true. If there is more than one element for which *expr* is true, one of them is returned. The precondition states that there must be at least one element fulfilling *expr*; otherwise, the result of this operation is Undefined.

pre: collection->exists(expr)
post collection->select(expr)->includes(result)

collection->one(expr : OclExpression) : Boolean

Results in true if there is exactly one element in the *collection* for which *expr* is true.
post: collection->select(expr)->size() = 1

6.8.2.2 Set

The Set is the mathematical set. It contains elements without duplicates. Features of Set, the instance of Set is called *set*.

set->union(set2 : Set(T)) : Set(T)

The union of *set* and *set2*.

post: result->forAll(elem | set->includes(elem) or set2->includes(elem))
post: set->forAll(elem | result->includes(elem))
post: set2->forAll(elem | result->includes(elem))

set->union(bag : Bag(T)) : Bag(T)

The union of *set* and *bag*.

post: result->forAll(elem |
result->count(elem) = set->count(elem) + bag->count(elem))
post: set->forAll(elem | result->includes(elem))
post: bag->forAll(elem | result->includes(elem))

set = (set2 : Set(T)) : Boolean

Evaluates to true if *set* and *set2* contain the same elements.

post: result = (set->forAll(elem | set2->includes(elem)) and
set2->forAll(elem | set->includes(elem)))

set->intersection(set2 : Set(T)) : Set(T)

The intersection of *set* and *set2*; that is, the set of all elements that are in both *set* and *set2*.

post: result->forAll(elem | set->includes(elem) and set2->includes(elem))
post: set->forAll(elem | set2->includes(elem) = result->includes(elem))
post: set2->forAll(elem | set->includes(elem) = result->includes(elem))

set->intersection(bag : Bag(T)) : Set(T)

The intersection of *set* and *bag*.
post: result = set->intersection(bag->asSet)

set – (set2 : Set(T)) : Set(T)

The elements of *set*, which are not in *set2*.

post: result->forAll(elem | set->includes(elem) and set2->excludes(elem))
post: set->forAll(elem | result->includes(elem) = set2->excludes(elem))

set->including(object : T) : Set(T)

The set containing all elements of *set* plus *object*.

post: result->forAll(elem | set->includes(elem) or (elem = object))
post: set->forAll(elem | result->includes(elem))
post: result->includes(object)

set->excluding(object : T) : Set(T)

The set containing all elements of *set* without *object*.

post: result->forAll(elem | set->includes(elem) and (elem <> object))
post: set->forAll(elem | result->includes(elem) = (object <> elem))
post: result->excludes(object)

set->symmetricDifference(set2 : Set(T)) : Set(T)

The sets containing all the elements that are in *set* or *set2*, but not in both.

post: result->forAll(elem | set->includes(elem) xor set2->includes(elem))
post: set->forAll(elem | result->includes(elem) = set2->excludes(elem))
post: set2->forAll(elem | result->includes(elem) = set->excludes(elem))

set->select(expr : OclExpression) : Set(T)

The subset of *set* for which *expr* is true.

post: result = set->iterate(elem; acc : Set(T) = Set{} |
if expr then acc->including(elem) else acc endif)

set->reject(expr : OclExpression) : Set(T)

The subset of *set* for which *expr* is false.
post: result = set->select(not expr)

set->collect(expr : OclExpression) : Bag(expr.evaluationType())

The Bag of elements that results from applying *expr* to every member of *set*.

post: result = set->iterate(elem; acc : Bag(expr.evaluationType()) = Bag{} |
acc->including(expr))

set->count(object : T) : Integer

The number of occurrences of *object* in *set*.
post: result <= 1

set->asSequence() : Sequence(T)

A Sequence that contains all the elements from *set*, in undefined order.

post: result->forAll(elem | set->includes(elem))
post: set->forAll(elem | result->count(elem) = 1)

set->asBag() : Bag(T)

The Bag that contains all the elements from *set*.

post: result->forAll(elem | set->includes(elem))
post: set->forAll(elem | result->count(elem) = 1)

6.8.2.3 Bag

A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag.

Properties of Bag, where the instance of Bag is called *bag*.

bag = (bag2 : Bag(T)) : Boolean

True if *bag* and *bag2* contain the same elements, the same number of times.

post: result = (bag->forAll(elem | bag->count(elem) = bag2->count(elem)) and
bag2->forAll(elem | bag2->count(elem) = bag->count(elem)))

bag->union(bag2 : Bag(T)) : Bag(T)

The union of *bag* and *bag2*.

post: result->forAll(elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
post: bag->forAll(elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
post: bag2->forAll(elem | result->count(elem) = bag->count(elem) + bag2->count(elem))

bag->union(set : Set(T)) : Bag(T)

The union of *bag* and *set*.

post: result->forAll(elem | result->count(elem) = bag->count(elem) + set->count(elem))
post: bag->forAll(elem |result->count(elem) = bag->count(elem) + set->count(elem))
post: set->forAll(elem |result->count(elem) = bag->count(elem) + set->count(elem))

bag->intersection(bag2 : Bag(T)) : Bag(T)

The intersection of *bag* and *bag2*.

post: result->forAll(elem | result->count(elem) = bag->count(elem).min(bag2->count(elem)))
post: bag->forAll(elem |result->count(elem) = bag->count(elem).min(bag2->count(elem)))
post: bag2->forAll(elem |result->count(elem) = bag->count(elem).min(bag2->count(elem)))

bag->intersection(set : Set(T)) : Set(T)

The intersection of *bag* and *set*.

post: result->forAll(elem | result->count(elem) = bag->count(elem).min(set->count(elem)))
post: bag->forAll(elem |result->count(elem) = bag->count(elem).min(set->count(elem)))
post: set->forAll(elem |result->count(elem) = bag->count(elem).min(set->count(elem)))

bag->including(object : T) : Bag(T)

The bag containing all elements of *bag* plus *object*.

```
post: result->forAll(elem |
if elem = object then
      result->count(elem) = bag->count(elem) + 1
   else
      result->count(elem) = bag->count(elem)
   endif)
post: bag->forAll(elem |
if elem = object then
      result->count(elem) = bag->count(elem) + 1
   else
      result->count(elem) = bag->count(elem)
   endif)
```

bag->excluding(object : T) : Bag(T)

The bag containing all elements of bag apart from all occurrences of *object*.

```
post: result->forAll(elem |
if elem = object then
      result->count(elem) = 0
   else
      result->count(elem) = bag->count(elem)
   endif)
post: bag->forAll(elem |
if elem = object then
      result->count(elem) = 0
   else
      result->count(elem) = bag->count(elem)
   endif)
```

bag->select(expr : OclExpression) : Bag(T)

The sub-bag of *bag* for which *expr* is true.

```
post: result = bag->iterate(elem; acc : Bag(T) = Bag{} |
       if expr then acc->including(elem) else acc endif)
```

bag->reject(expr : OclExpression) : Bag(T)

The sub-bag of *bag* for which *expr* is false.

```
post: result = bag->select(not expr)
```

bag->collect(expr: OclExpression) : Bag(expr.evaluationType())

The Bag of elements that results from applying *expr* to every member of *bag*.

```
post: result = bag->iterate(elem; acc : Bag(expr.evaluationType() ) = Bag{} |
            acc->including(expr) )
```

bag->count(object : T) : Integer

The number of occurrences of *object* in *bag*.

bag->asSequence() : Sequence(T)

A Sequence that contains all the elements from *bag*, in undefined order.

```
post: result->forAll(elem | bag->count(elem) = result->count(elem))
post: bag->forAll(elem | bag->count(elem) = result->count(elem))
```

bag->asSet() : Set(T)

The Set containing all the elements from *bag*, with duplicates removed.

```
post: result->forAll(elem | bag->includes(elem) )
post: bag->forAll(elem | result->includes(elem))
```

6.8.2.4 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once.

Properties of Sequence(T), where the instance of Sequence is called *sequence.*

sequence->count(object : T) : Integer

The number of occurrences of *object* in *sequence*.

sequence = (sequence2 : Sequence(T)) : Boolean

True if *sequence* contains the same elements as *sequence2* in the same order.

```
post: result = Sequence{1..sequence->size()}->forAll(index : Integer |
        sequence->at(index) = sequence2->at(index))
      and
      sequence->size() = sequence2->size()
```

sequence->union (sequence2 : Sequence(T)) : Sequence(T)

The sequence consisting of all elements in *sequence*, followed by all elements in *sequence2*.

```
post: result->size() = sequence->size() + sequence2->size()
post: Sequence{1..sequence->size()}->forAll(index : Integer |
      sequence->at(index) = result->at(index))
post: Sequence{1..sequence2->size()}->forAll(index : Integer |
      sequence2->at(index) =
                result->at(index + sequence->size() )))
```

sequence->append (object: T) : Sequence(T)

The sequence of elements, consisting of all elements of *sequence*, followed by *object.*

```
post: result->size() = sequence->size() + 1
post: result->at(result->size() ) = object
post:   Sequence{1..sequence->size() }->forAll(index : Integer |
        result->at(index) = sequence ->at(index))
```

sequence->prepend(object : T) : Sequence(T)

The sequence consisting of *object*, followed by all elements in *sequence.*

```
post: result->size = sequence->size() + 1
post: result->at(1) = object
post:   Sequence{1..sequence->size()}->forAll(index : Integer |
      sequence->at(index) = result->at(index + 1))
```

sequence->subSequence(lower : Integer, upper : Integer) : Sequence(T)

The sub-sequence of *sequence* starting at number *lower*, up to and including element number *upper*.

```
pre : 1 <= lower
pre : lower <= upper
pre : upper <= sequence->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
        result->at(index - lower + 1) =
                        sequence->at(index))
endif
```

sequence->at(i : Integer) : T

The *i-th* element of sequence.

```
pre : i >= 1 and i <= sequence->size()
```

sequence->first() : T

The first element in *sequence*.

```
post: result = sequence->at(1)
```

sequence->last() : T

The last element in *sequence.*

```
post: result = sequence->at(sequence->size() )
```

sequence->including(object : T) : Sequence(T)

The sequence containing all elements of *sequence* plus *object* added as the last element.

```
post: result = sequence.append(object)
```

sequence->excluding(object : T) : Sequence(T)

The sequence containing all elements of *sequence* apart from all occurrences of *object*. The order of the remaining elements is not changed.

```
post:result->includes(object) = false
post: result->size() = sequence->size() - sequence->count(object)
post: result = sequence->iterate(elem; acc : Sequence(T)
    = Sequence{}|
        if elem = object then acc else acc->append(elem) endif )
```

sequence->select(expression : OclExpression) : Sequence(T)

The subsequence of *sequence* for which *expression* is *true*.

```
post: result = sequence->iterate(elem; acc : Sequence(T) = Sequence{} |
       if expr then acc->including(elem) else acc endif)
```

sequence->reject(expression : OclExpression) : Sequence(T)

The subsequence of *sequence* for which *expression* is false.
post: result = sequence->select(not expr)

sequence->collect(expression : OclExpression) : Sequence(expression.evaluationType())

The Sequence of elements that results from applying *expression* to every member of *sequence*.

sequence->iterate(expr : OclExpression) : expr.evaluationType()

Iterates over the sequence. Iteration will be done from element at position 1 up until the element at the last position following the order of the sequence.

sequence->asBag() : Bag(T)

The Bag containing all the elements from *sequence*, including duplicates.

post: result->forAll(elem | sequence->count(elem) = result->count(elem))
post: sequence->forAll(elem | sequence->count(elem) = result->count(elem))

sequence->asSet() : Set(T)

The Set containing all the elements from *sequence*, with duplicated removed.

post: result->forAll(elem | sequence->includes(elem))
post: sequence->forAll(elem | result->includes(elem))

6.9 Grammar

This section describes the grammar for OCL expressions. An executable LL(1) version of this grammar is available on the OCL web site. (See http://www.software.ibm.com/ad/ocl).

The grammar description uses the EBNF syntax, where "|" means a choice, "?" optionality, and "*" means zero or more times, "+" means one or more times, and expressions delimited with "/*" and "*/" are definitions described with English words or sentences. In the description of *string*, the syntax for lexical tokens from the JavaCC parser generator is used. The "~" symbol denotes that none of the symbols following may be matched. It means "everything except the following."

```
oclFile              := ( "package" packageName
                          oclExpressions
                          "endpackage"
                        )+
packageName          := pathName
oclExpressions       := ( constraint )*
constraint           := contextDeclaration
                        ( ( "def" name? ":" letExpression* )
                          |
```

```
                                ( stereotype name? ":" oclExpression)
                              )+
contextDeclaration    := "context"
                              ( operationContext | classifierContext )
classifierContext     := (  name ":" name )
                              | name
operationContext      := name "::" operationName
                              "(" formalParameterList ")"
                              ( ":" returnType )?
stereotype            := ( "pre" | "post" | "inv" )
operationName         := name | "=" | "+" | "-" | "<"  | "<=" |
                              ">="   | ">" | "/" | "*" | "<>" |
                              "implies" | "not" | "or" | "xor" | "and"
formalParameterList   := ( name ":" typeSpecifier
                                ("," name ":" typeSpecifier )*
                              )?
typeSpecifier         := simpleTypeSpecifier
                              | collectionType
collectionType        := collectionKind
                              "(" simpleTypeSpecifier ")"
oclExpression         := (letExpression* "in")? expression
returnType            := typeSpecifier
expression            := logicalExpression
letExpression         := "let" name
                              ( "(" formalParameterList ")" )?
                              ( ":" typeSpecifier )?
                              "=" expression
ifExpression          := "if" expression
                              "then" expression
                              "else" expression
                              "endif"
logicalExpression     := relationalExpression
                              ( logicalOperator
                                relationalExpression
                              )*
relationalExpression  := additiveExpression
                              ( relationalOperator
                                additiveExpression
                              )?
additiveExpression    := multiplicativeExpression
                              ( addOperator
```

```
                              multiplicativeExpression
                            )*
multiplicativeExpression:= unaryExpression
                             ( multiplyOperator
                               unaryExpression
                             )*
unaryExpression        := ( unaryOperator
                              postfixExpression
                            )
                            | postfixExpression
postfixExpression      := primaryExpression
                            ( ("." | "->")propertyCall )*
primaryExpression      := literalCollection
                            | literal
                            | propertyCall
                            | "("  expression  ")"
                            | ifExpression
propertyCallParameters := "(" ( declarator )?
                            ( actualParameterList )? ")"
literal                := string
                            | number
                            | enumLiteral
enumLiteral            := name "::" name ( "::" name )*
simpleTypeSpecifier    := pathName
literalCollection      := collectionKind "{"
                            ( collectionItem
                              "," collectionItem )*
                            )?
                            "}"
collectionItem         := expression (".." expression )?
propertyCall           := pathName
                            ( timeExpression )?
                            ( qualifiers )?
                            ( propertyCallParameters )?
qualifiers             := "[" actualParameterList "]"
declarator             := name ( "," name )*
                            ( ":" simpleTypeSpecifier )?
                            ( ";" name ":" typeSpecifier "="
                              expression
                            )?
                            "|"
```

```
pathName              := name ( "::" name )*
timeExpression        := "@" "pre"
actualParameterList   := expression ("," expression)*
logicalOperator       := "and" | "or" | "xor" | "implies"
collectionKind        := "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator    := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator           := "+" |  "-"
multiplyOperator      := "*" | "/"
unaryOperator         := "-" | "not"
typeName              :=charForNameTop charForName*
name                  := charForNameTop charForName*
charForNameTop        := /* Characters except inhibitedChar
                           and ["0"-"9"]; the available
                           characters shall be determined by
                           the tool implementers ultimately.*/
charForName           := /* Characters except inhibitedChar; the
                           available characters shall be determined
                           by the tool implementers ultimately.*/
inhibitedChar     := " " | "\"" | "#" | "\'" | "(" | ")" |
                     "*" | "+" | "," | "-" | "." | "/" |
                     ":" | ";" | "<" | "=" | ">" | "@" |
                     "[" | "\\" | "]" | "{" | "|" | "}"
number                := ["0"-"9"] (["0"-"9"])*
                         ( "." ["0"-"9"] (["0"-"9"])* )?
                         ( ("e" | "E") ( "+" | "-" )? ["0"-"9"]
                           (["0"-"9"])*
                         )?
string                := "'"
                         (( ~["’","\\","\n","\r"] )
                          |("\\"
                             ( ["n","t","b","r","f","\\","’","\""]
                               | ["0"-"7"]
                                 ( ["0"-"7"] ( ["0"-"7"] )? )?
                             )
                           )
                         )*
                         "'"
```

UML Standard Elements *A*

Note – Changes based on the ISO version of UML 1.4.1 (formal/03-02-04) are in this font.

This appendix contains a list of the predefined standard elements for UML. The standard elements are stereotypes, constraints and tagged values. The names used for UML predefined standard elements are considered reserved words; modelers should not overload these names with different definitions. Each standard element is described in the chapter containing its base element.

Note – See the UML Semantics chapter for more details.

Standard Element Name	Applies to Base Element	Kind
	Package	Stereotype
	Package	Stereotype
	Package	Stereotype
	Package	Stereotype
«access»	Permission	Stereotype
«appliedProfile»	Package	Stereotype
association	Association	Constraint
«association»	AssociationEnd	Stereotype
«auxiliary»	Class	Stereotype
«become»	Flow	Stereotype
«call»	Usage	Stereotype

Standard Element Name	Applies to Base Element	Kind
complete	Generalization	Constraint
«copy»	Flow	Stereotype
«create»	BehavioralFeature	Stereotype
«create»	CallEvent	Stereotype
«create»	Usage	Stereotype
«derive»	Abstraction	Stereotype
derived	ModelElement	Tag
«destroy»	BehavioralFeature	Stereotype
«destroy»	CallEvent	Stereotype
~~destroyed~~	~~Association~~	~~Constraint~~
~~destroyed~~	~~Association~~	~~Constraint~~
disjoint	Generalization	Constraint
«document»	Artifact	Stereotype
documentation	Element	Tag
«executable»	Artifact	Stereotype
«facade»	Package	Stereotype
«file»	Artifact	Stereotype
«focus»	Class	Stereotype
«framework»	Package	Stereotype
«friend»	Permission	Stereotype
global	Association	Constraint
«global»	AssociationEnd	Stereotype
«implementation»	Class	Stereotype
«implementation»	Generalization	Stereotype
«implicit»	Association	Stereotype
«import»	Permission	Stereotype
incomplete	Generalization	Constraint
«instantiate»	Usage	Stereotype
«invariant»	Constraint	Stereotype
«library»	Artifact	Stereotype
local	Association	Constraint
«local»	AssociationEnd	Stereotype

Standard Element Name	Applies to Base Element	Kind
«metaclass»	Class	Stereotype
«metamodel»	Package	Stereotype
«modelLibrary»	Package	Stereotype
«modelLibrary»	Package	Stereotype
~~new~~	~~Association~~	~~Constraint~~
~~new~~	~~Association~~	~~Constraint~~
overlapping	Generalization	Constraint
parameter	Association	Constraint
«parameter»	AssociationEnd	Stereotype
persistence	Association	Tag
persistence	Attribute	Tag
persistence	Classifier	Tag
persistent	Association	Tag
«postcondition»	Constraint	Stereotype
«powertype»	Class	Stereotype
«precondition»	Constraint	Stereotype
«process»	Classifier	Stereotype
«profile»	Package	Stereotype
«realize»	Abstraction	Stereotype
«refine»	Abstraction	Stereotype
«requirement»	Comment	Stereotype
«responsibility»	Comment	Stereotype
self	Association	Constraint
«self»	AssociationEnd	Stereotype
semantics	Classifier	Tag
semantics	Operation	Tag
«send»	Usage	Stereotype
«signalflow»	ObjectFlowState	Stereotype
«source»	Artifact	Stereotype
«stateInvariant»	Constraint	Stereotype
«stub»	Package	Stereotype
«systemModel»	Package	Stereotype

Standard Element Name	Applies to Base Element	Kind
«table»	Artifact	Stereotype
«thread»	Classifier	Stereotype
«topLevel»	Package	Stereotype
«trace»	Abstraction	Stereotype
transient	Association	Constraint
transient	Association	Constraint
«type»	Class	Stereotype
~~usage~~	~~Association~~	~~Tag~~
~~«utility»~~	~~Classifier~~	~~Stereotype~~
xor	Association	Constraint

Action Language Examples B

This appendix shows mappings from fragments of specifications in existing action languages to UML Action Semantics models. The intent is to demonstrate by example that the Action Semantics model contains concepts and has a structure required to support real action languages.

The examples also aid the reader in understanding the models. By providing concrete examples in the familiar form of a textual language the reader can more readily see what some of the concepts actually mean. The reader should be aware that the mappings are not definitive and that the Action Semantics specification is normative.

B.1 The Action Languages

The Action Languages used for this mapping have been in use in system development for a number of years. All are Action Languages targeted at object modeling techniques and have primitives built in to support, for example, the creation and deletion of objects and links as well as the sending of signals and the invocation of operations. The example languages are:

- The Action Specification Language (ASL). A public domain language of which there have been several implementations. See “The Action Specification Language Reference Manual” available at www.kc.com.
- The BridgePoint Action Language (AL). An action language supported by the BridgePoint modeling tool. More information is available from www.projtech.com.
- The Kabira Action Semantics (Kabira AS). An action language for the ObjectSwitch middle-tier server suite. More information is available at www.kabira.com.
- The action language subset of SDL. An international standard widely used in the telecom industry. More information is available in ITU-T Recommendations Z.100 (SDL) and Z.109 (SDL UML profile).

The specification allows both for pure data and control flow oriented approaches and for traditional imperative languages. The languages used in this appendix are all examples of the latter style, but there are a number of features that map to individual actions connected by data and control flow. The mappings thus also provide examples of flow connections between actions.

The languages provide, neither individually nor collectively, constructs that make use of all of the Actions described in this document. However, they do cover most of the key features.

A mapping from the action language SDL to these action semantics is described in OMG document http://cgi.omg.org/cgi-bin/doc?ad/00-08-01, which is on the OMG web server.

B.2 Presentation of the Examples

This appendix starts with a series of examples, each usually of one source statement in size, of the ASL, AL and Kabira AS languages and concludes with a complete example of an SDL procedure.

The ASL, AL, and Kabira AS examples consist of:

- source text formulated in one (or more) of these languages exhibiting a fragment of a specification,
- an object diagram showing an instance of the Action Semantics model.

Each object diagram gives the meaning of the corresponding specification fragment in terms of the Action Semantics.

These object diagrams show M1 artifacts (i.e. actual user models) by displaying them as instances of classes at the M2 level (the UML Metamodel level). These object diagrams are not M0 objects.

These object diagrams may include model elements from the Core package of UML. These are included to aid in understanding the connection between the diagrams and the surface syntax. Where no confusion is likely to arise, model elements from the Core package are omitted.

This first series of examples illustrates many of the individual model elements of the Action Semantics: control structures (see Section B.3), object manipulation (see Section B.4), and messaging actions (see Section B.5).

For some of the examples there may be no direct equivalent construct in one or two of these languages. In such cases, this does not mean that the operation achieved by the example cannot be achieved in the other language(s), rather it means that it must be achieved by a more explicit and verbose model where the user strings several source language statements together to achieve the desired effect. For example, some language constructs can act directly on collections in some languages but not in others. In this case, the language without the direct support must employ an explicit loop to achieve the same effect. In such examples the resulting object diagram will look very different and so separate examples have been created.

Unlike ASL and AL that have implicit variable declarations and typing, Kabira AS requires explicit declaration that is, for the most part, not shown in the examples here. In the specification, local variables have an association with GroupAction providing for scoping within an action language. The examples do not show this association. Unless otherwise shown, all local variables in these examples have a multiplicity of 1..1.

B.3 Control Structures

These examples are of the traditional control structures and sequential logic present in the action languages. These map to the actions found in Section 2.20, “Composite Actions,” on page 2-228.

If-then-else Logic

The example in Figure B-1 shows a simple if-then-else involving a logical comparison. The semantics of the action language construct is that if and only if the value of `factor` is equal to (the integer) 2, then `some action 1` will be executed. In all other cases, `some action 2` will be executed.

.

ASL:	AL:	Kabira AS:
if factor = 2 then	if (factor == 2)	if (factor == 2) {
# Some action 1	// Some action 1	// Some action 1

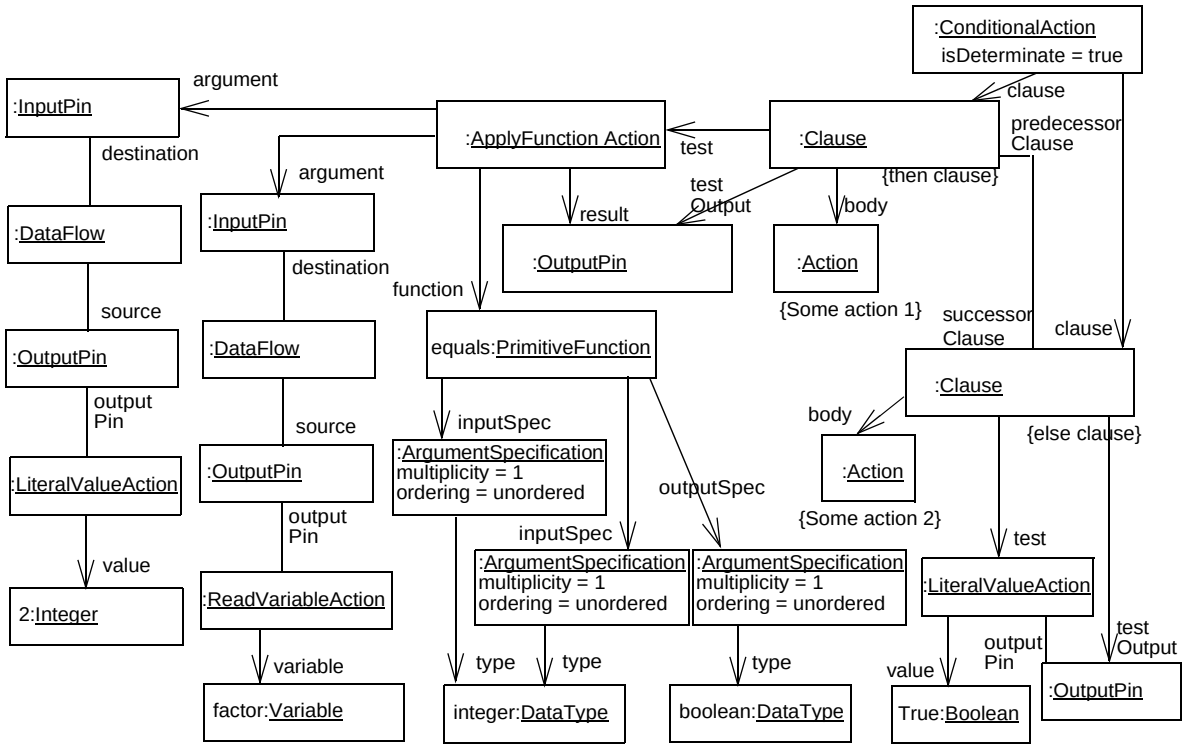


Figure B-1 If-then-else Logic

This maps to a general ConditionalAction within the action semantics. The conditional action has two clauses corresponding to the “then” and “else” branches of the if. These clauses are shown at the top right of the instance diagram. They have dummy actions as bodies for the purposes of this example.

The `then` clause has the actual logical comparison from the source action language attached to it as a `test`. The test action is an ApplyFunctionAction that applies the `equals` function to the two operands of the logical expression. The two operands come from a constants (LiteralValueAction) and a local variable (ReadVariableAction).

The `else` clause has a test action (LiteralValueAction) that always returns the value `TRUE`. The `else` clause is the “successor” of the `then` clause and so will be tested and hence executed only if the `then` clause failed. This arrangement preserves the `if-then-else` behavior of the source languages.

Multi-way Decision

All of the action languages support multi-way decisions in a single statement. ASL supports this through the use of a switch statement where a variable is compared against constant values whereas AL and the Kabira AS support the more general else-if construct.

The example shows a decision based on the value of a local variable realized both as a `switch` statement and as a `else-if` statement.

ASL:	AL:	Kabira AS:
switch factor	if (factor == 1)	if (factor == 1) {
case 1	// Some action 1	// Some action 1
# Some action 1	elif (factor == 2)	} else if (factor ==1) {

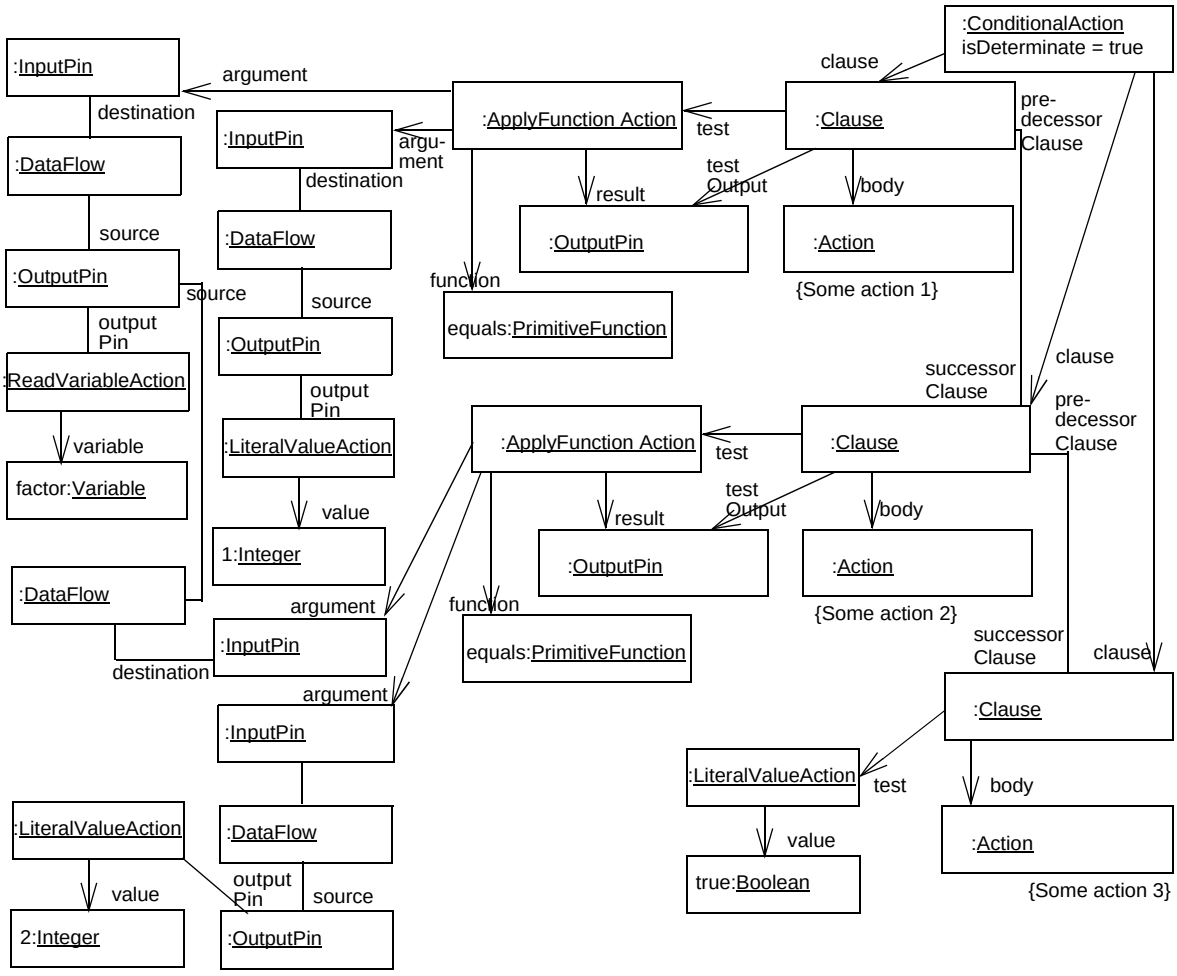


Figure B-2 Multi-way Decision

Note – In ASL, the switch statement has an implicit “break” at the end of every clause so that there is no “fall through” into the next clause. In addition, the execution of the switch terminates once any clause has executed and so the example shown is semantically identical to the else-if examples in AL and Kabira AS.

The details of the substructure of the PrimitiveFunction (input and output types) have been omitted to make the example clearer. The previous example shows what this would have looked like.

Note – The predecessor-successor control flows between the clauses that provide the correct semantics for both the ASL switch and the explicit if-then-else construct. However, an alternative mapping could have been shown in which the test on the third clause is replaced by a logical expression of the form “(factor != 1) && (factor !=2)” in which case the predecessor-successor control flows could have been omitted. This would have allowed all three tests to execute concurrently. However, the logic of the test specification would have meant that only one of the branches could ever execute. This would provide the same semantics as the example as shown.

B.4 Object Manipulation

These examples show the basic creation and manipulation of objects. The actions used by these language constructs are for the most part those described in the chapter on Read and Write Actions. Being local variable oriented languages, all of these operations use local variables of type object reference, or sometimes collections of object references.

Simple Object Creation

Figure B-3 shows an example of creating an object of the class `Customer`. The reference to the newly created object is then assigned to the local variable `new_customer`.

ASL:

new_customer = create Customer

AL:

create object instance new_customer of Customer;

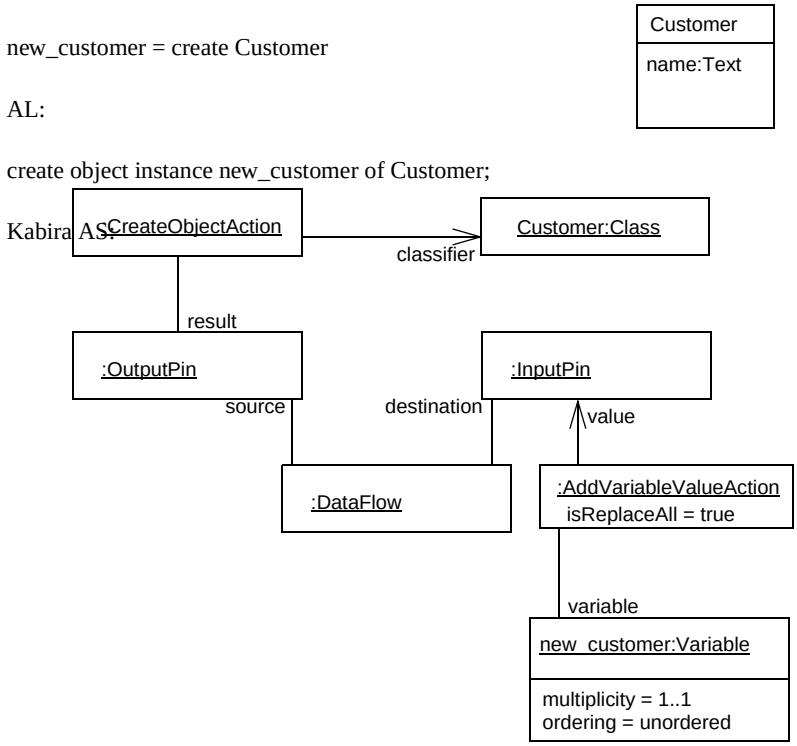


Figure B-3 Simple Object Creation

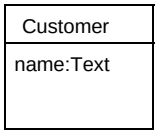
Note – AL actually uses a second form of the class name (the “key letter” captured as a UML tag) to identify the class. This is a minor syntactic detail and to avoid confusion the examples in this Appendix use the class name instead.

Object Creation with Attribute Assignment

Figure B-4 shows an example of creating an object of the class Customer with assignment of the value of the attribute `name` from the local variable `new_name`.

ASL:

new_customer = create Customer with name = new_name



AL:

create object instance new_customer of Customer;

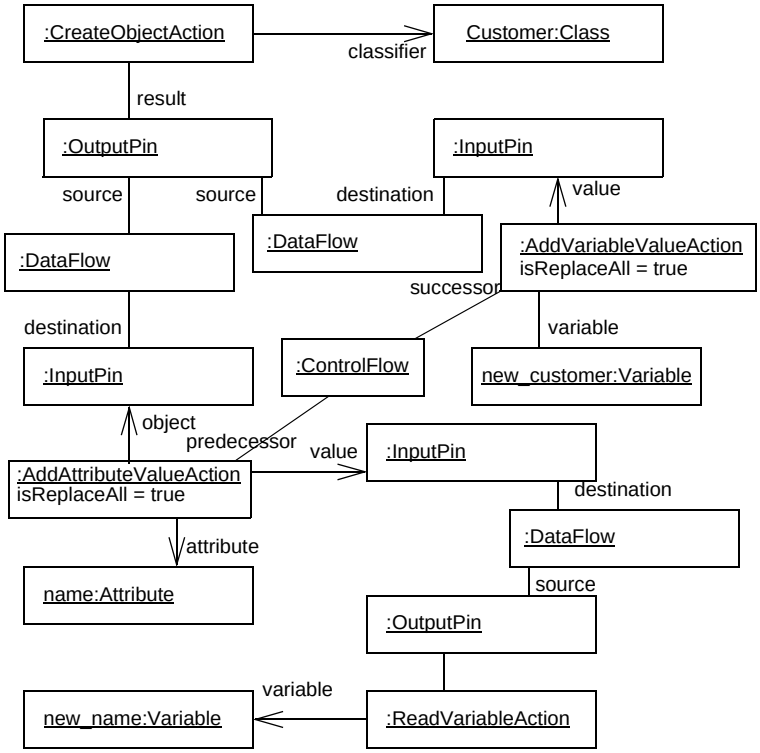


Figure B-4 Object Creation with Attribute Assignment

Note – With the AL example, a strict interpretation would have the attribute assignment via the reading of a local variable rather than through a data flow from the CreateObjectAction as shown. The example is semantically identical to the AL.

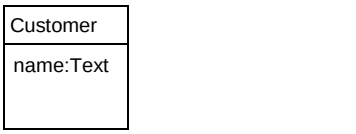
Object Destruction

Figure B-5 shows an example of destruction of an object. In the examples, the object is identified by a reference `my_customer`.

ASL:

delete my_customer

AL:



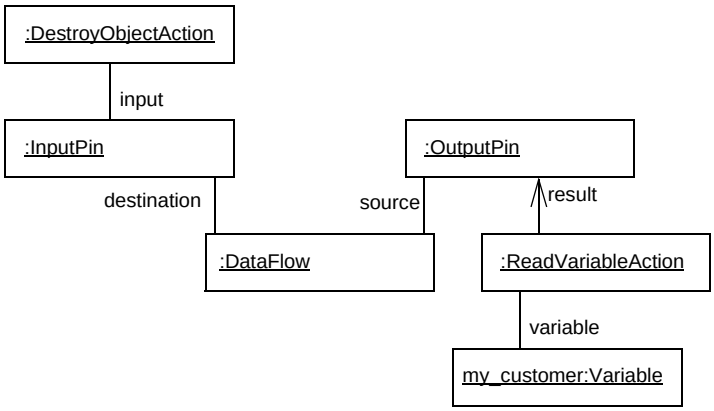


Figure B-5 Object Destruction

Writing of Attributes: Single Attribute, SIngle Object

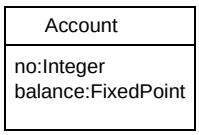
In this example, a value is written to a single attribute of a single object instance. The value comes from a local variable (`new_balance`), and the object is identified by an object reference local variable (`current_account`).

ASL:

current_account.balance = new_balance

AL:

current_account.balance = new_balance;



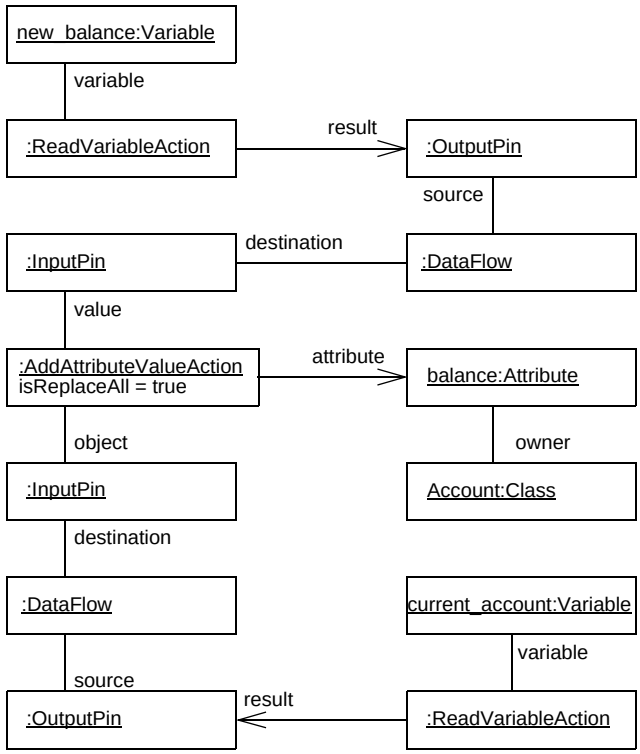


Figure B-6 Writing of Attributes (Single Attribute, Single Object)

Writing of Attributes: Multiple Attributes, Single Object

In this example, a values are written to a multiple attributes of a single object instance. The values come from local variables (`new_balance`, `todays_date`), and the object is identified by an object reference local variable (`current_account`). In ASL this can be achieved through a single language statement, but in AL and Kabira AS this must be achieved through multiple statements, each operating from the same object instance. This example shows ASL only.

.

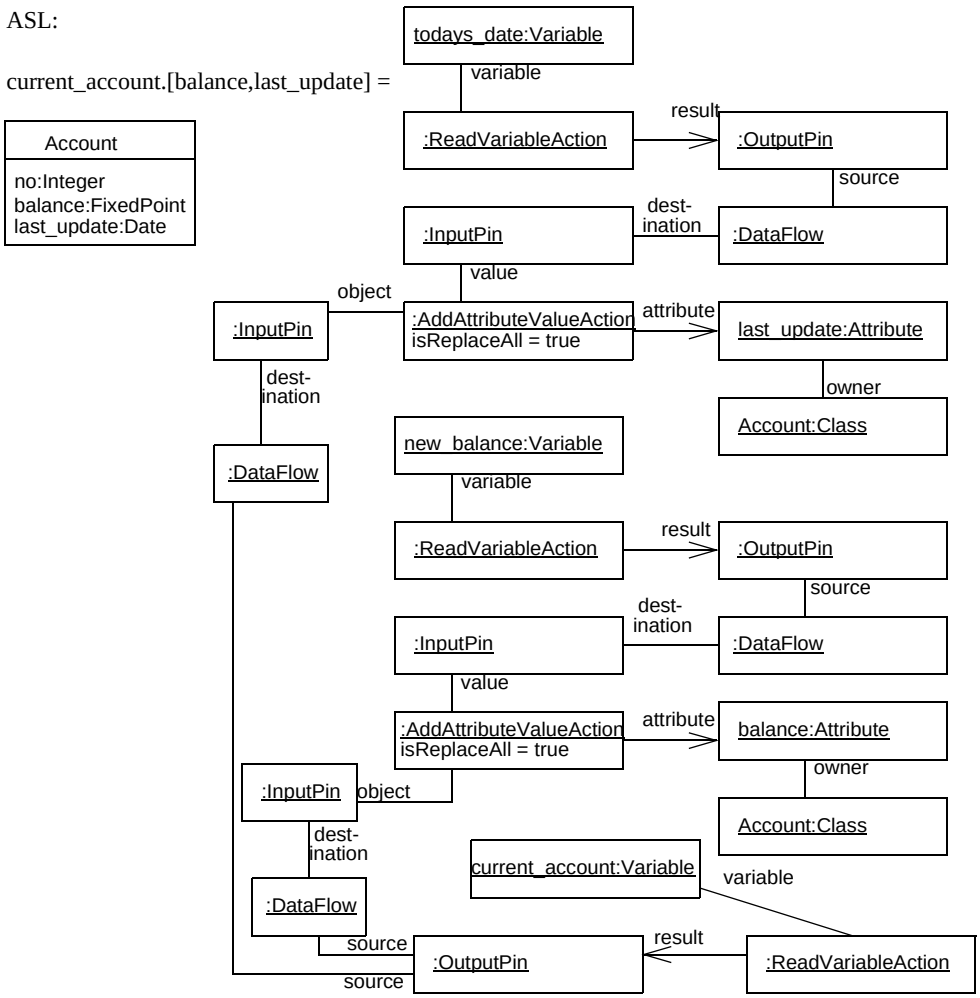


Figure B-7 Writing of Attributes (Multiple Attributes, Single Object)

Writing of Attributes: Single Attribute, Multiple Object

In this example, a value is written to an attribute of a multiple object instances. The value comes from a local variable (`todays_date`), and the objects are identified by an object reference collection local variable (`{reviewed_accts}`). In ASL this can be

achieved through a single language statement, but in AL and Kabira AS, this must be achieved through an explicit loop. The example shows ASL only and explicit loops are shown in other examples

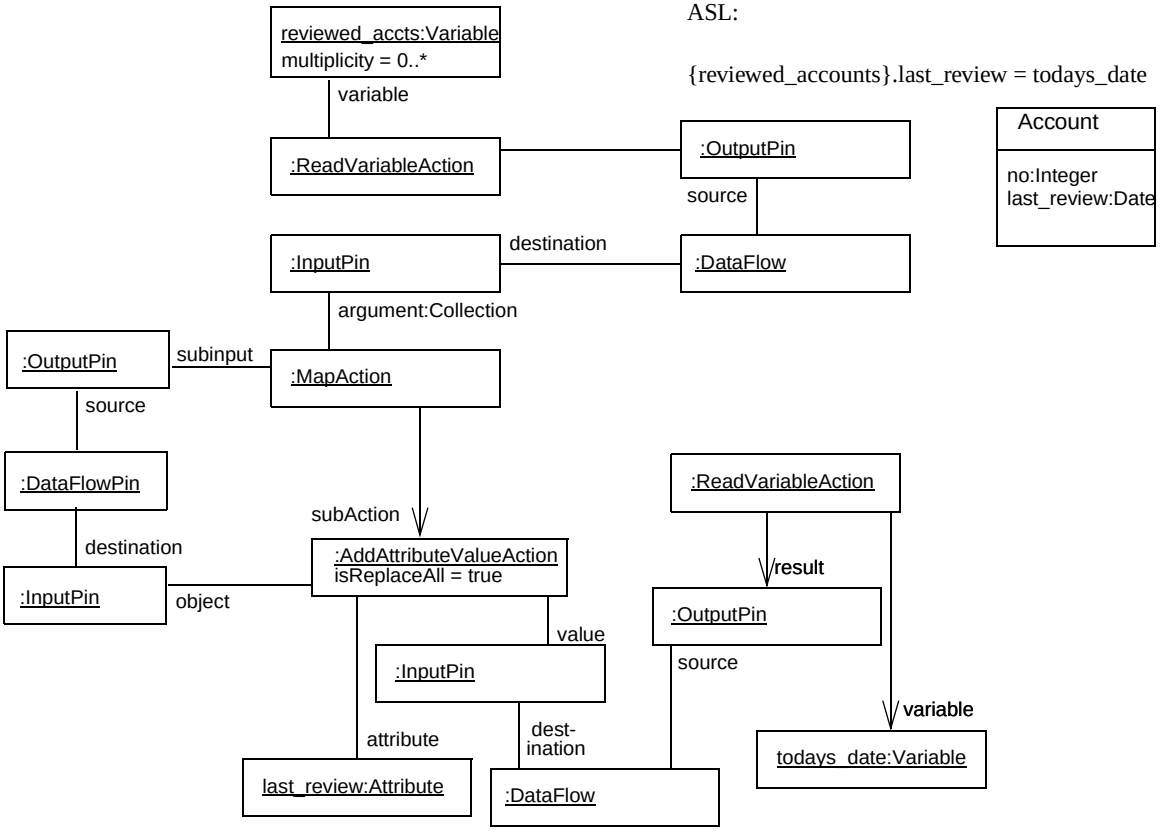


Figure B-8 Writing of Attributes (Single Attribute, Multiple Objects)

Obtaining a Selection of Objects

The ASL, AL, and Kabira AS languages provide facilities to select the extent of a class and store the result in a local variable that can then be used in other language constructs. In ASL and AL, the local variables can be singletons or collections.

In the first example, a selection is made from the `Account` class through to a simple logical condition. The condition results in a single object instance being selected and the reference assigned to a local variable `my_account`.

The top part of the instance diagram concerns the reading of the extent of the `Account` class and flowing the resulting collection into a filter action. The filter action produces an output written to the `my_account` local variable, shown on the right hand middle of the diagram. The remainder of the diagram concerns the subAction of the FilterAction. This is an ApplyFunctionAction that invokes the logical comparison of the two items

in the logical expression. One of the items is a constant (supplied by the LiteralValueAction) and the other is obtained by reading the value of the attribute of the instance being tested.

The second example shows a similar selection, but one that results in a collection that is assigned to a local variable. In this case, because the Kabira AS does not support local variables that are collections, we have not shown an example from this language. In Kabira AS such selections can be used directly as the inputs to loops. In that case the assignment to the output local variable would be replaced by a flow into a LoopAction.

ASL:

my_account = find-only Account where no = 42

AL:

select one my_account from instances of

Account where selected.no = 42;

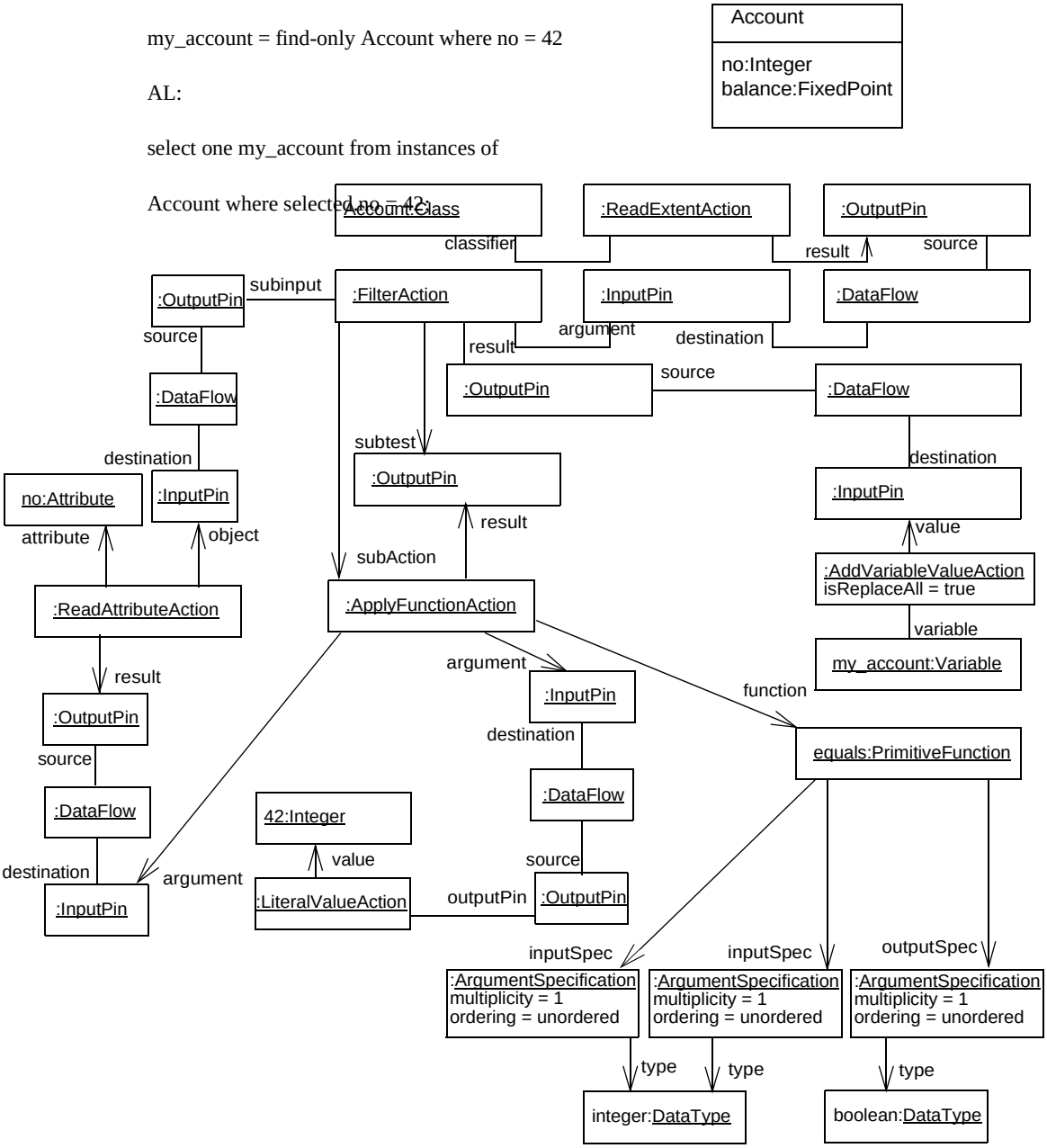


Figure B-9 Single Object Selection

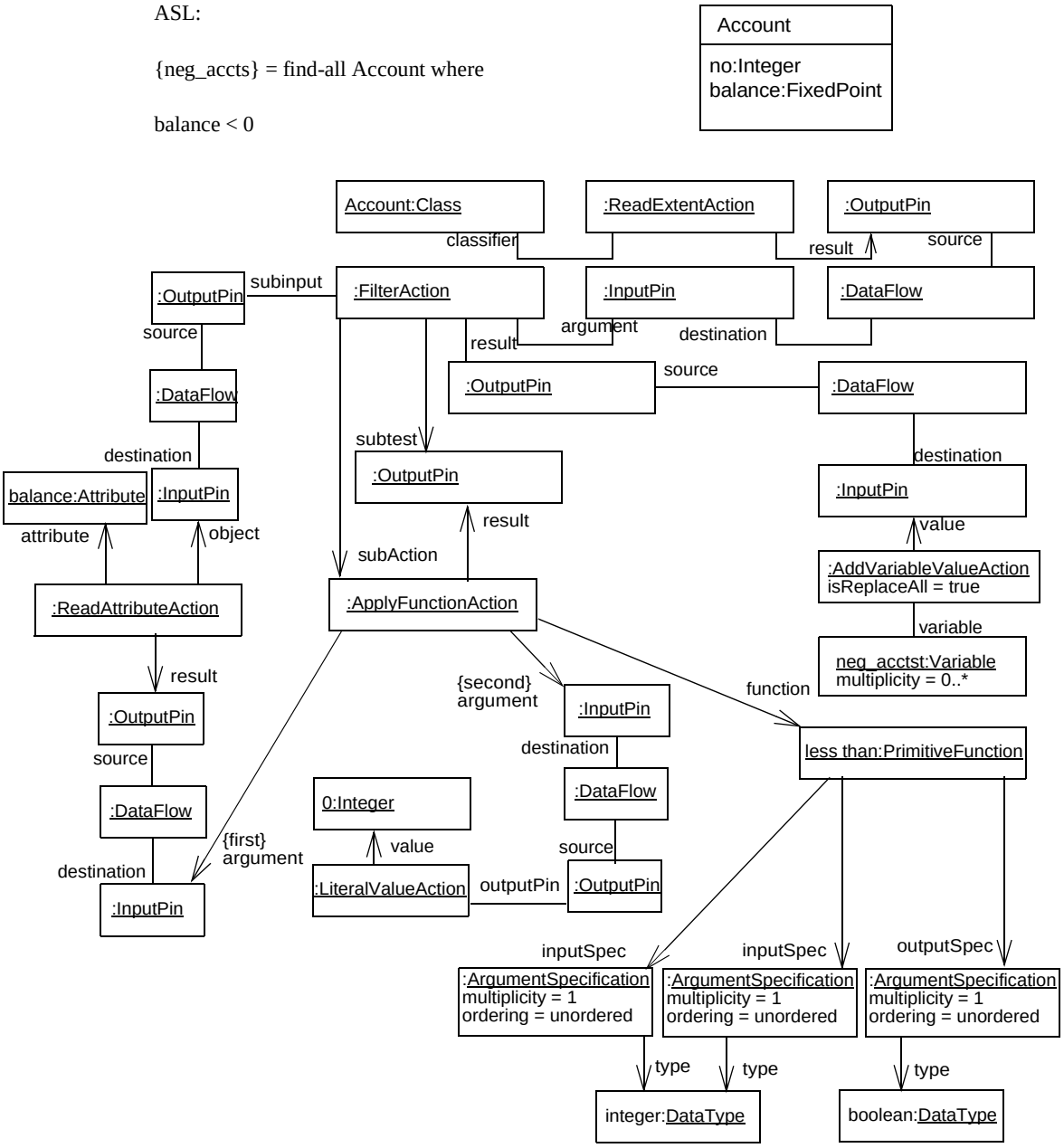


Figure B-10 Obtaining a Collection of Objects

Creating a Link

This example creates a link between two objects. The link is an instance of the binary association shown between the `Account` class and the `Customer` class. The objects are identified by the local variable object references (`the_customer`, `the_account`). In ASL and AL the syntax of the language is such that users must give all associations a

unique name (in this example, `R1`) to provide a unique reference to the association being manipulated. Kabira AS uses the association role for this purpose. Semantically, each of the three languages achieves the same effect.

In a reflexive association, it is necessary to know in which direction the link is intended. All three languages use role names to clarify this.

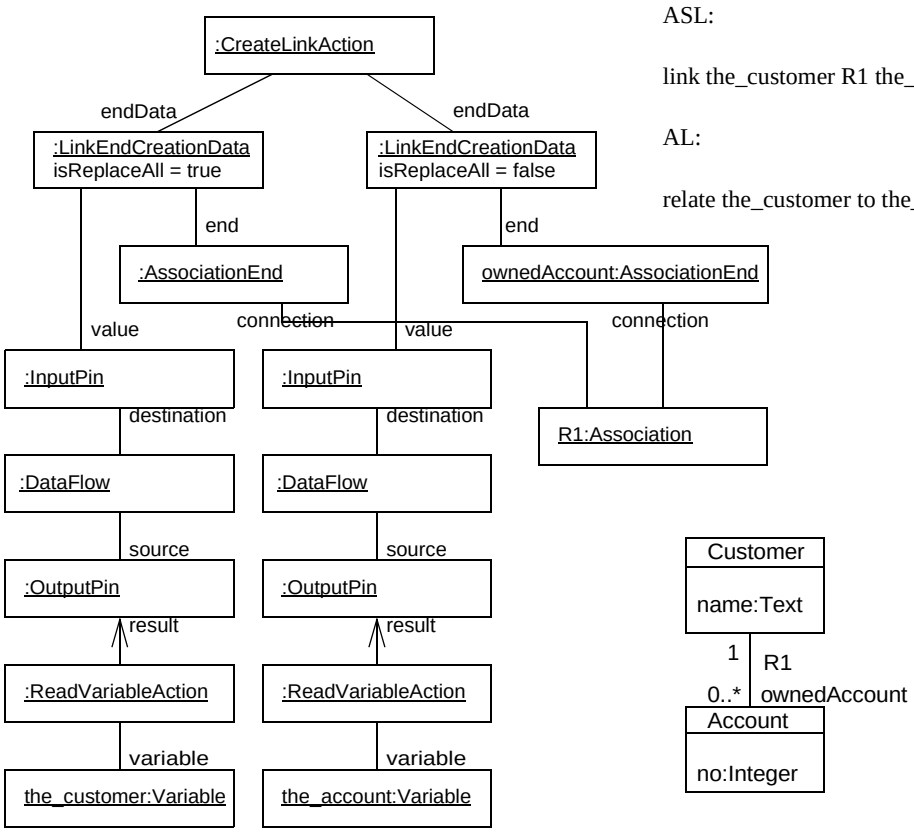


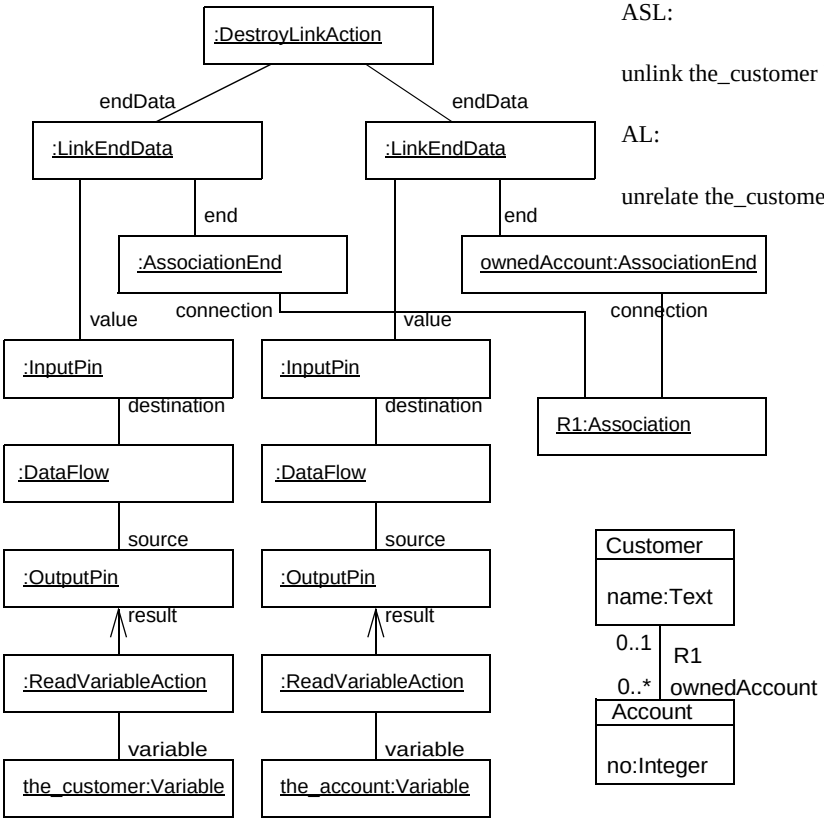
Figure B-11 Creating a Link

Note – This example shows a 1:0..* association. This means that there can be only one linked Customer for a given Account. In the Kabira AS, the semantics are such that if a Customer object was previously linked when the “relate” statement was executed, then an implicit “unrelate” (DestroyLinkAction) is performed prior to the CreateLinkAction. There can, however, be many Account objects for a given Customer object. The pattern of “isReplaceAll” values in the two LinkEndCreationData objects achieves this effect. In ASL, the semantics are slightly different. In that language, the modeler must ensure that any existing instance of Customer is explicitly unlinked from the instance of Account before the link is executed. Any failure to do this is regarded as an exception condition due to the violation of the multiplicity of the association. For ASL, therefore, a more accurate mapping of the semantics of “link” would have “isReplaceAll” false in both the instances of LinkEndCreationData.

Destroying a Link

This example destroys a link between two objects. The link is an instance of the binary association shown between the `Account` class and the `Customer` class. The link to be deleted is identified by the objects at either end that are identified by local variable object references (`the_customer`, `the_account`). In ASL and AL the syntax of the language is such that users must give all associations a unique name (in this example, `R1`) to provide a unique reference to the association being manipulated. Kabira AS uses the association role for this purpose. Semantically, each of the three languages achieves the same effect.

In a reflexive association, it is necessary to know in which direction the link is intended. All three languages use role names to clarify this.



ASL:

unlink the_customer R1 the_account

AL:

unrelate the_customer from the_account across R1;

Figure B-12 Destroying a Link

Navigating an Association to a Single Object

These action languages navigate an association to obtain the object references of linked objects, which are then used to perform some manipulation on the linked objects, such as calling an operation provided by the object.

In this example, a navigation starts from an instance of `Account` (specified by the object reference `the_account`) to obtain a reference to the Customer that owns it. The resulting reference is in a local variable called `owner`. Both ends of the association involved in the navigation are specified, but only one of them has an input pin. This pin takes the identity of the object at the starting end of the navigation.

In the model fragment shown, due to the multiplicity of the association, there will be only one instance of Customer obtained by the navigation.

ASL:

owner = the_account -> R1

AL:

select one owner related by

the_account -> Customer[R1];

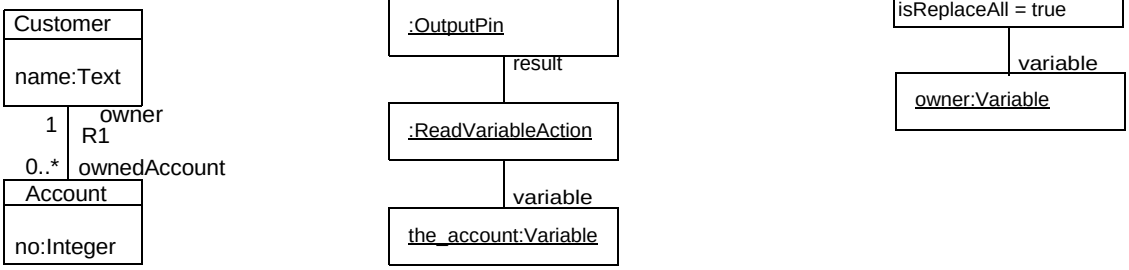


Figure B-13 Navigating an Association

Navigating an Association to Multiple Objects

These action languages navigate an association to obtain the object references of linked objects, which are then used to perform some manipulation on the linked objects, such as calling an operation provided by the object.

In this example, a navigation starts from an instance of `Customer` (specified by the object reference `the_customer`) to obtain references to all the Accounts owned by Customer. Due to the multiplicity of the association in the example, the result will be a collection.

In ASL and AL, local variables can be collections and so the example and mapping is very similar to those in the previous example (of navigation to a single instance). In Kabira AS, local variables cannot be collections and so the result of the navigation

must be used directly in an explicit loop. However, the effect of this loop is exactly that of a MapAction in the action semantics. The action enclosed in the Kabira AS loop is repeatedly executed with the local variable (the_account) being successively given a value corresponding to each instance in the collection of object references obtained by the navigation. This mapping is shown in the second diagram.

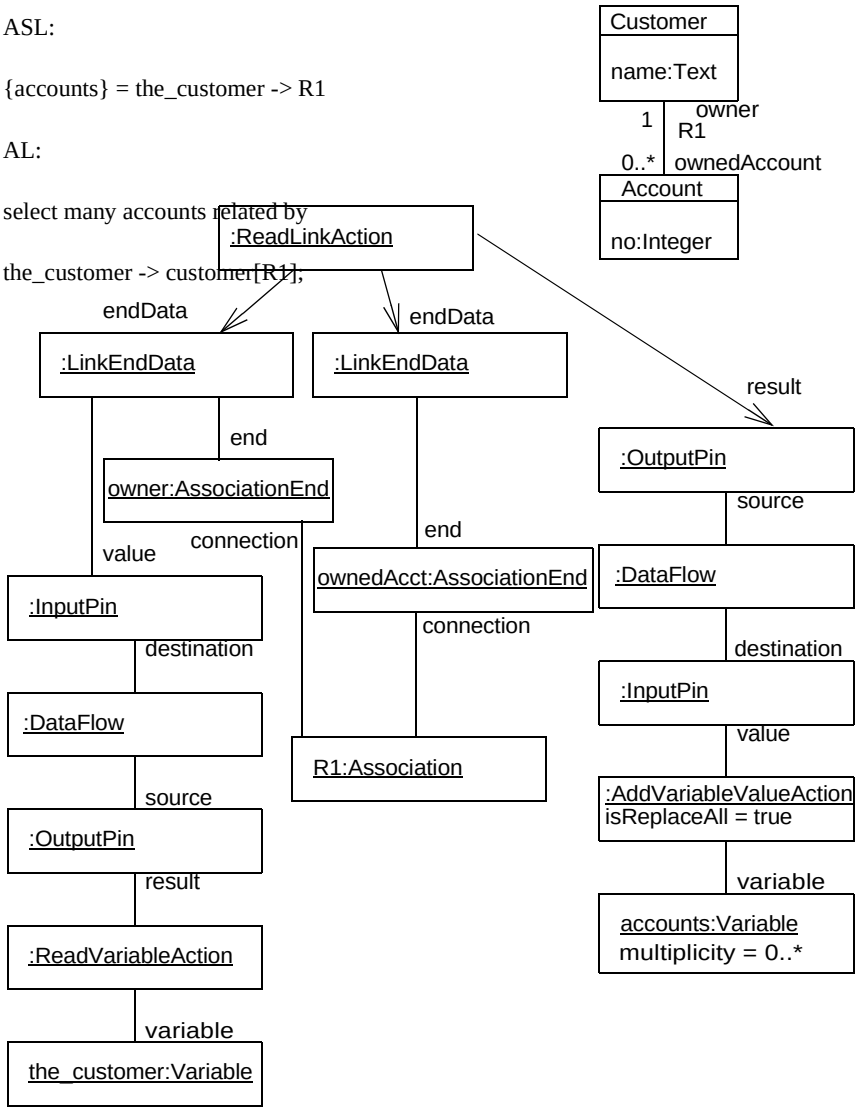


Figure B-14 Obtaining a Collection by Navigation

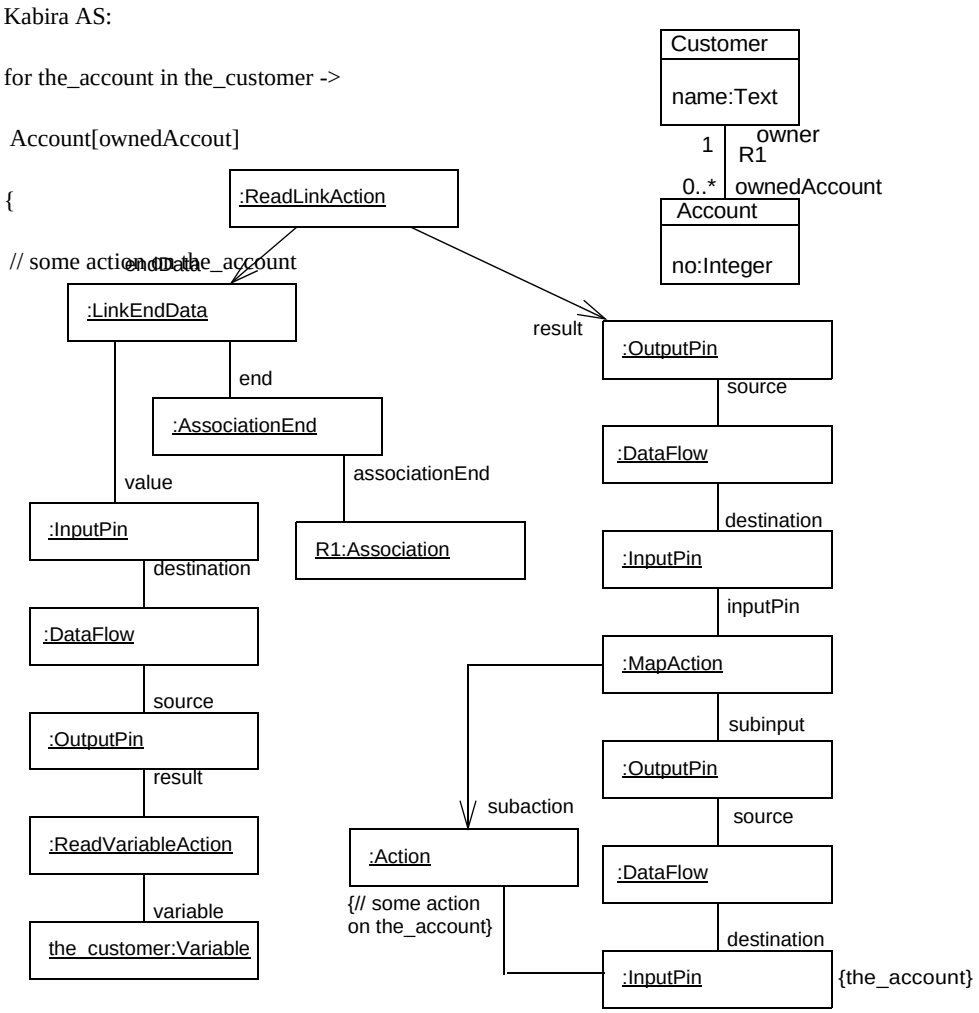


Figure B-15 Loop over a Navigation

B.5 Messaging Actions

This section covers the invocation of operations and the sending of signal events.

Invocation of an Instance Operation with no Parameters

In this example, an operation (validate) provided by the `Customer` class is invoked. The operation has no parameters, but applies to a specific instance of Customer (identified by the object reference local variable `my_customer`).

Note – The ASL syntax supports a particular syntax for the names of operations that makes their association with the class of the target object clear. This has not been used here to make the example straightforward.

ASL:

[]= validate[] on the_customer

AL:

the_customer.validate();

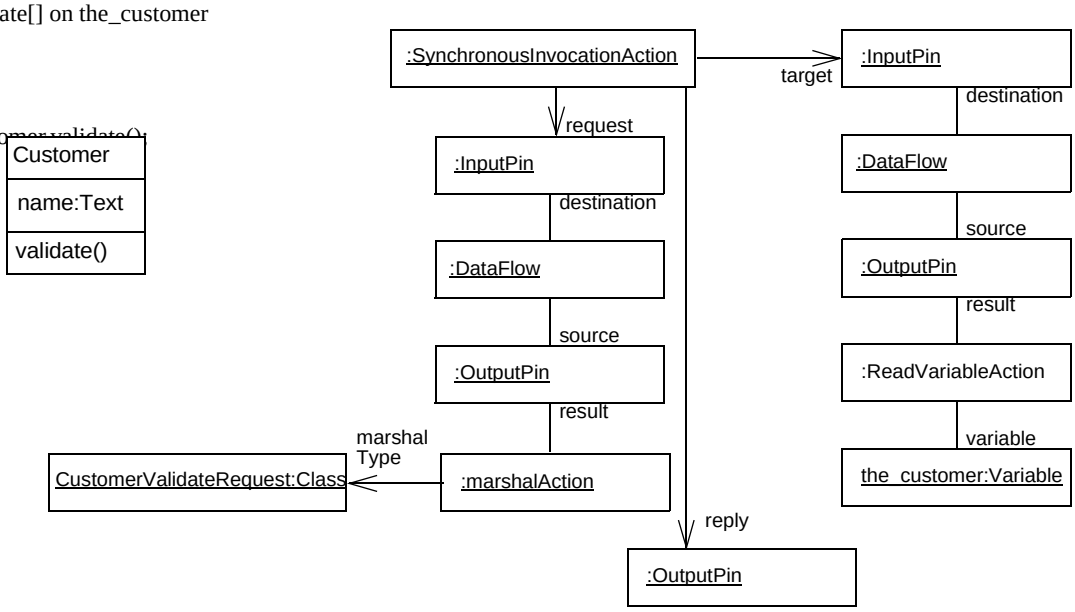


Figure B-16 Simple Operation Invocation using SynchronousInvocationAction

The example shows only the invocation actions and not the Effect Resolution. However, in all three languages the resolution is a simple operation lookup.

Note – In this example, there are no return parameters from the validate operation and so the output pin can be ignored, or it could be used as an input to a subsequent action in lieu of a control link.

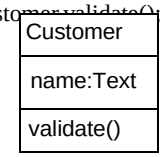
The above mapping uses the SynchronousInvocationAction. As is discussed in Section 2.24, “Messaging Actions,” on page 2-311; however, an alternative action, the CallOperationAction is also available. With this action, the semantics of marshalling data are implicit in the action. This alternative mapping is shown in Figure B-17.

ASL:

[]= validate[] on the_customer

AL:

the_customer.validate();



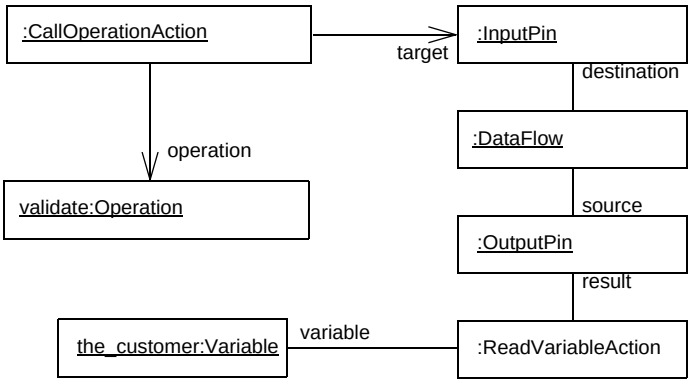


Figure B-17 Simple Operation Invocation using CallOperationAction

Invocation of an Instance Operation with Parameters

ASL:

[new_balance]= update_balance[amount] on the_account

AL:

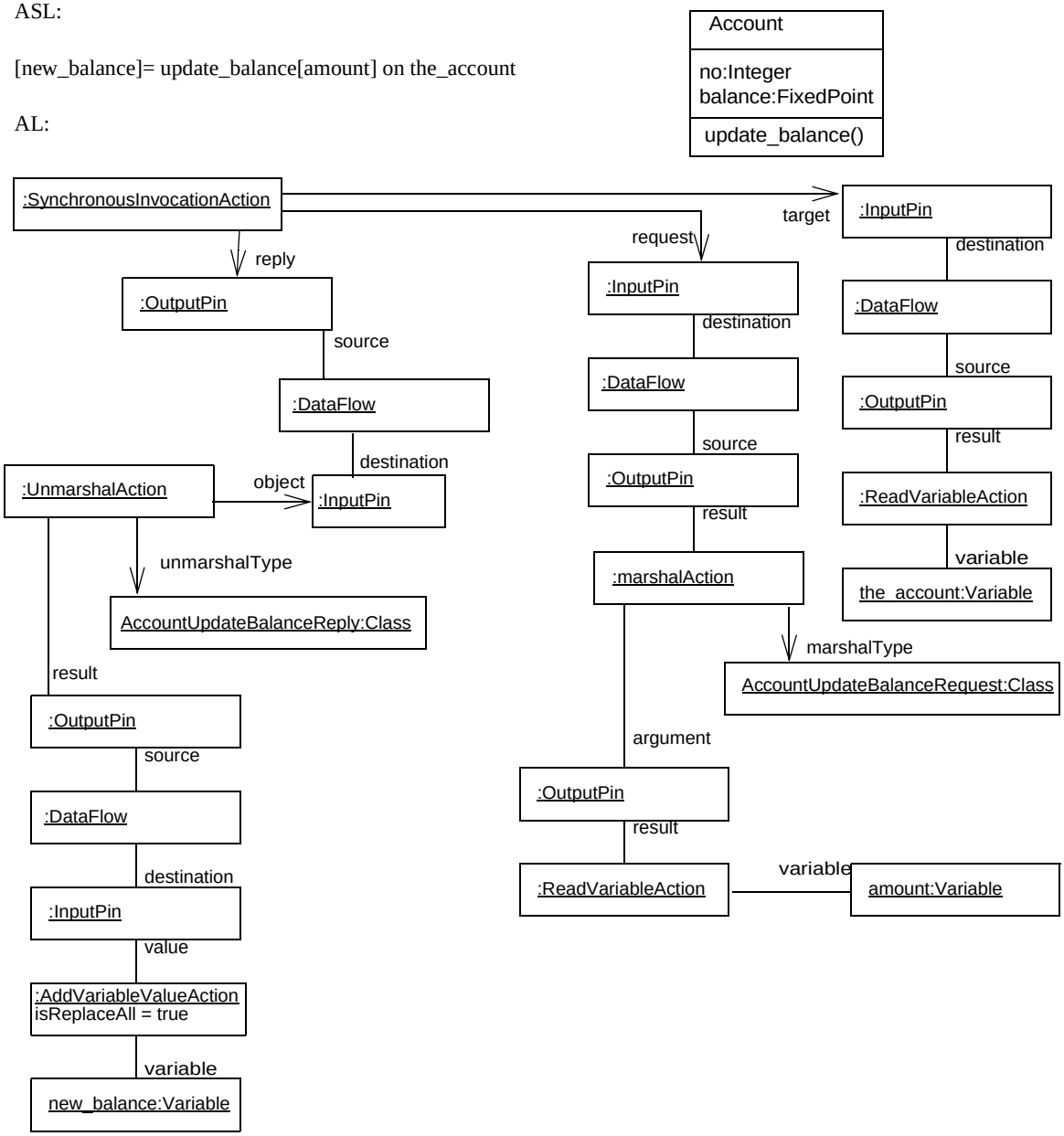


Figure B-18 Operation with Parameters using SynchronousInvocationAction

This example shows the invocation of an operation that takes an input parameter (amount) and updates the balance of an Account. The resulting balance is returned as an output parameter. In Figure B-18 this is shown using the SynchronousInvocationAction with explicit marshalling and unmarshalling of parameters. The equivalent mapping with CallOperationAction is shown in Figure B-19.

ASL:

[new_balance]= update_balance[amount] on the_account

AL:

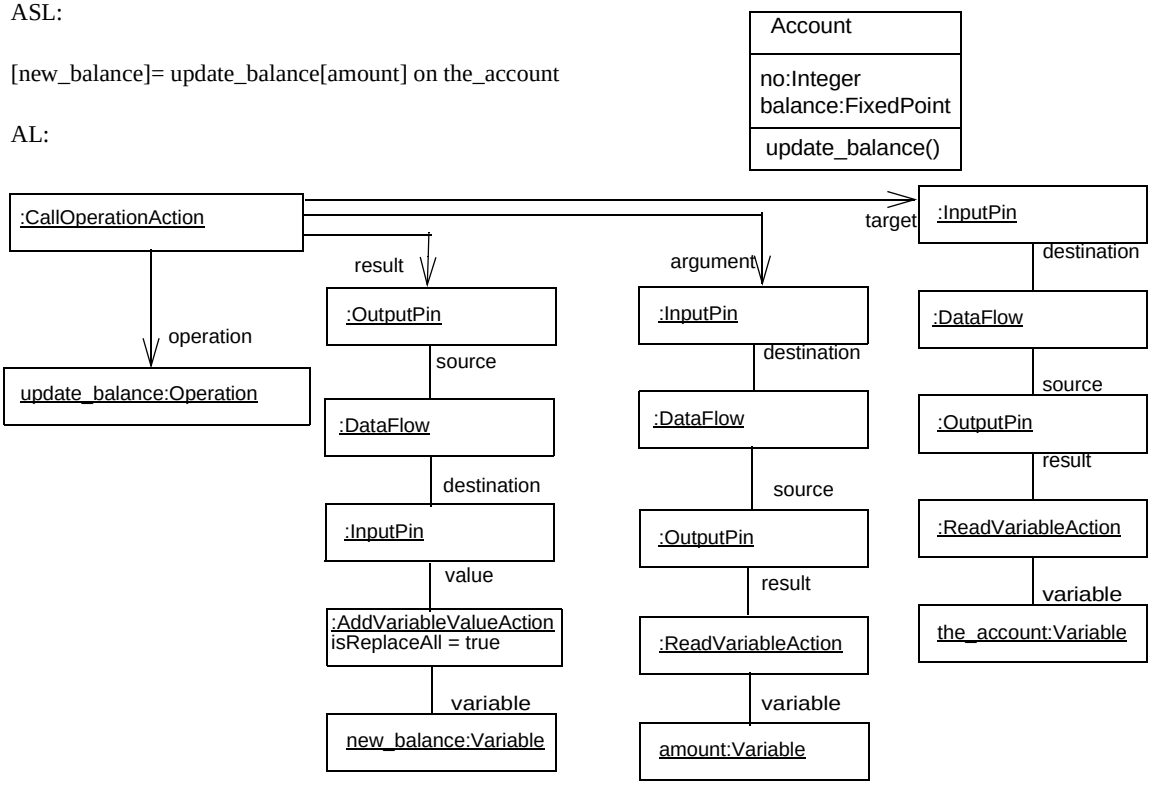


Figure B-19 Operation with Parameters using CallOperationAction

Sending of a Signal Event with no Parameters

This example shows the dispatching of a Signal Event to an object of the Account class.

ASL:

generate block_account() to the_account

AL:

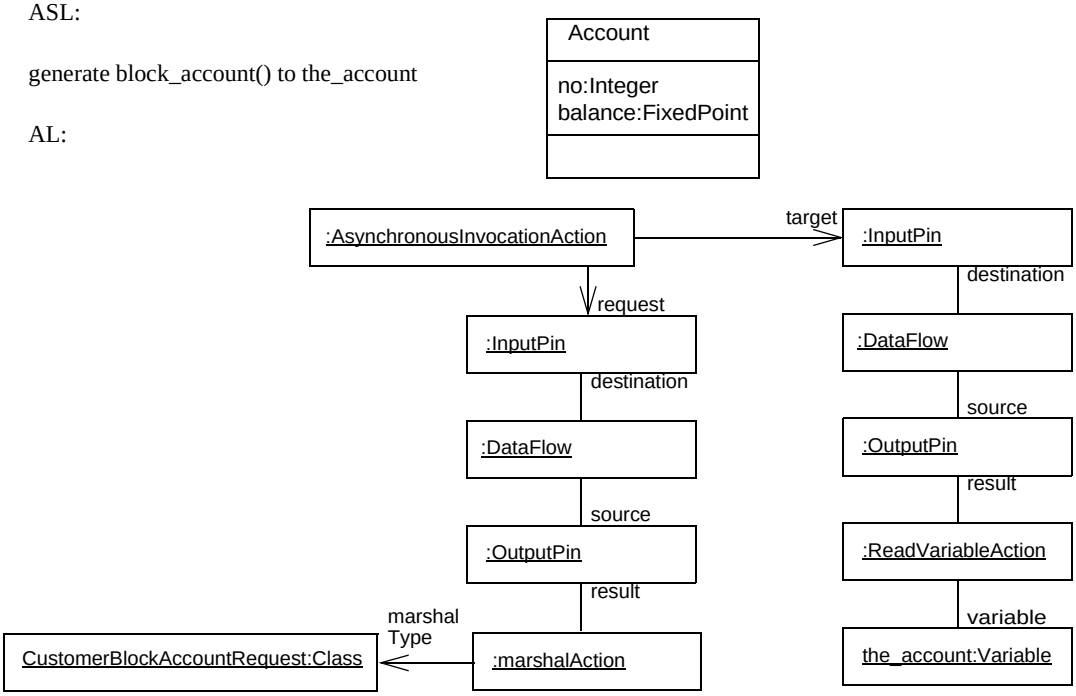


Figure B-20 Sending of Signal with no parameter using AsynchronousInvocationAction

In a similar way to the mapping of operation invocations discussed in the previous section, an alternative mapping for this uses the SendSignalAction, where the marshalling of data is implicit. This mapping is show in Figure B-21.

ASL:

generate block_account() to the_account

AL:

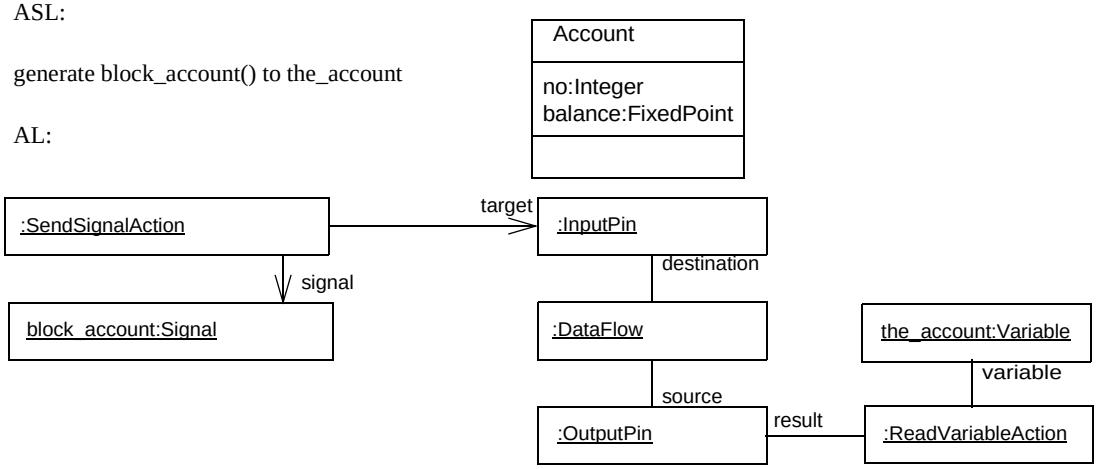


Figure B-21 Sending of Signal with no parameter using SendSignalAction

Note – The ASL and AL syntax actually requires a particular syntax for the names of signals that makes their association with the class of the target object clear. This syntax has not been used here to make the example more straightforward.

Sending of a Signal Event with Parameters

In a similar way to the previous example, this sends a signal to an object. In this case there is an additional parameter (period) that is sent with the signal.

ASL:

generate update_interest(period) to the_account

AL:

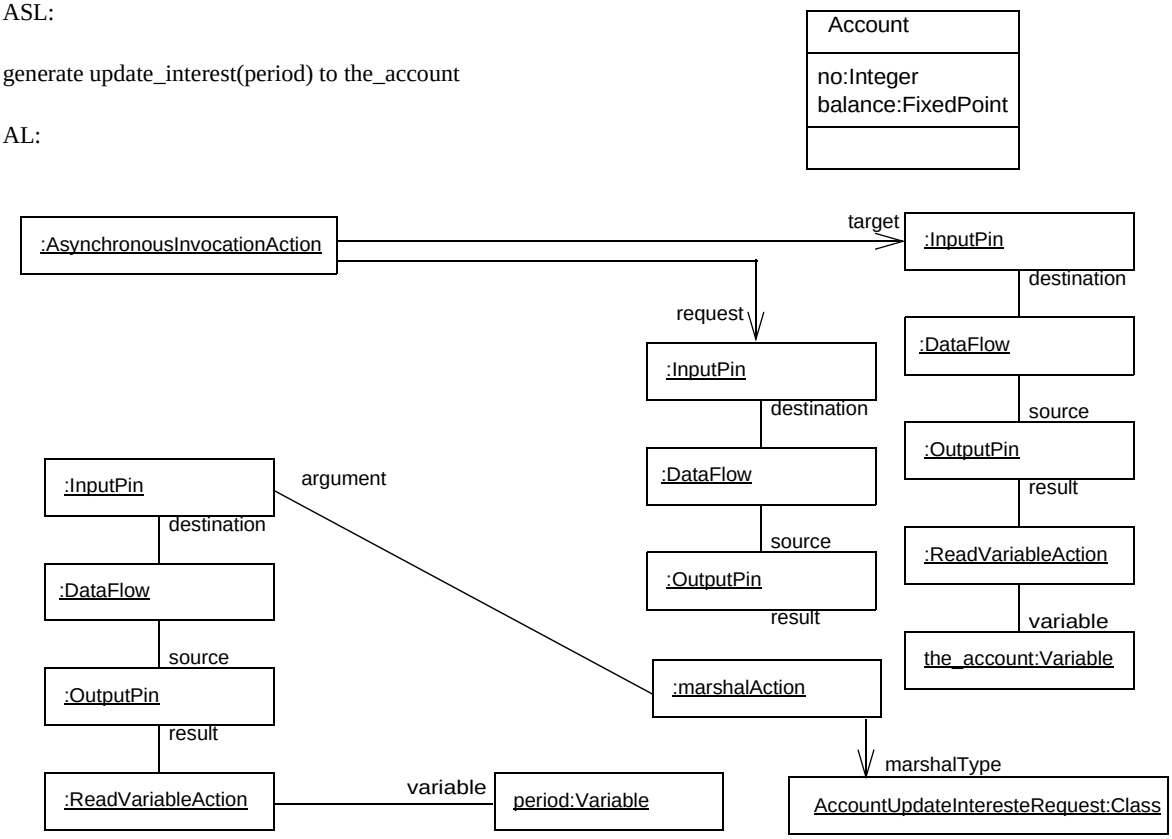


Figure B-22 Sending of Signal with Parameters using AsynchronousInvocationAction

Finally, the same example using the SendSignalAction is shown in Figure B-23.

ASL:

generate update_interest(period) to the_account

AL:

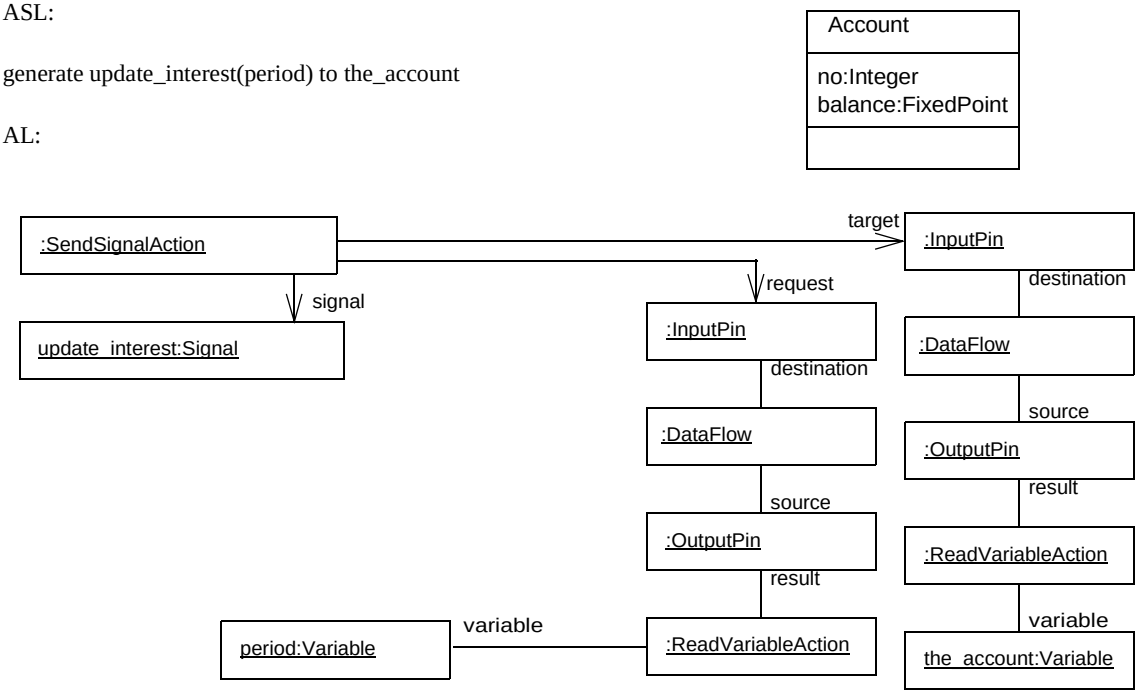


Figure B-23 Sending of Signal with Parameters using SendSignalAction

B.6 Complete Example: The FFT

This section shows a complete example of the specification of a procedure using actions. The purpose is to show how the various actions fit together to carry out a complete algorithm, as well as to illustrate some of the complicated actions through actual use. An informal notation is used to illustrate the examples, emphasizing the connectivity of the actions while suppressing minor mechanical details. This notation is not complete and is not intended to be a normative syntax, but merely to help get an intuitive feel for the action semantic constructs without becoming immersed in UML minutiae.

The example describes the Fast Fourier Transform (FFT), one of the most important algorithms discovered to date, both for its theoretical and practical consequences. This algorithm has revolutionized signal processing applications, and it has consequences for polynomial multiplication and other areas. Theoretically, its discovery showed that many algorithms could be much more efficient than intuition would suggest and led to major advances in complexity theory. The algorithm itself is also elegant both in theory and in implementation, with extreme malleability that allows it to be adapted in many different ways. In particular, it lends itself to a highly parallel implementation, which can be captured using these action semantics. The ability to preserve inherent concurrency is a major contribution of these action semantics.

This section does not attempt to explain the theory or derivation of the FFT. The reader is advised to consult a modern book on numerical algorithms for more information. Note also that the form of the algorithm presented here is not the most efficient form

used for computation. Even in algorithm books, the algorithm is usually presented in a more transparent form first, because the highly optimized forms can be tricky to understand. Such optimizations could, of course, be expressed in the action semantics.

B.6.1 The Fast Fourier Transform

The Discrete Fourier Transform (DFT) performs a complex-number transformation from the time or spatial domain into the frequency domain, according to the following formula:

$$B_j = \sum_{k=0}^{n-1} A_k W_n^k \qquad j = 0, 1, \ldots, n-1 \qquad \textbf{(EQ 1)}$$

where n is a power of 2 and W_n is the primary complex root of unity of order n:

$$W_n = e^{-2\pi i/n} \qquad \textbf{(EQ 2)}$$

Intuitively, this would seem to require $O(n^2)$ arithmetic operations (multiplications and additions) to compute. Remarkably, the Fast Fourier Transform (FFT) algorithm can perform this transformation in $log_2\ n$ stages of $O(n)$ operations, for a total complexity of $O(n \log n)$. Among many forms, the algorithm can be expressed in the following form:

$$S_{0,j} = A_j \qquad \textbf{(EQ 3)}$$

$$S_{m+1,2j} = S_{m,j} + S_{m,j+n/2} \qquad S_{m+1,2j+1} = S_{m,j} - S_{m,j+n/2} V_{m,j} \qquad j = 0, 1, 2\ldots, n/2-1 \qquad \textbf{(EQ 4)}$$

$$V_{m,j} = W_n^{\lfloor j/2^m \rfloor 2^m} \qquad \textbf{(EQ 5)}$$

$$B_j = S_{\log n,\, rev(j,\, n)} \qquad j = 0, 1, 2\ldots, n-1 \qquad \textbf{(EQ 6)}$$

where $rev(j,n)$ is the integer obtained by reversing the $log\ n$-bit binary representation of the integer j; for example, rev(6,8) is 3.

This formula can be understood as follows: There are $log\ n$ stages, each of which transforms a complex vector of n elements into another complex vector of n elements (Equation 4). The starting vector is the original vector to be transformed (Equation 3). During each stage, the pattern of computation is the same, except the multiplication factors V are “thinned” by a factor of 2 on each successive stage (Equation 5). The final complex vector is rearranged by swapping each element to its “bit-reversed” position to obtain the final complex vector result (Equation 6). These $log\ n$ stages must be performed sequentially; there is no concurrency across them.

Looking within each stage, Equation 4 indicates that two elements from one stage are used to compute two elements of the next stage. This pair of formulas is called a butterfly operation, after the shape of its data flow graph, which uses two input values to produce two output values. All $n/2$ butterfly operations can be performed in parallel, because there is no interaction among the input or output values of different ones. The FFT algorithm permits a high degree of concurrency. Examining the formula closely,

we see that each butterfly operation takes an element from the first half of the input vector and the corresponding element from the second half of the input vector, producing two successive values in the output vector for the next stage. There is no interaction between the n/2 butterfly operations in a stage, either on input values or output values. A stage can be viewed in vector terms as follows: A vector of *n* elements is cut in half (like a pack of cards) into two vectors of n/2 elements each. In parallel, each element of one half is combined with the corresponding element of the other half in a butterfly operation, yielding two values. If we form two half-vectors from the results, they must be shuffled together by alternating elements from each of the half-vectors (again, like a pack of cards) to form a full-size vector for the next stage of the computation. To summarize a stage: cut a full vector into two half vectors, map the butterfly operation concurrently onto each pair of half vectors to form a new pair of half vectors, and shuffle the two half vectors together to produce a new full vector.

There is one final detail. The multiplication factors change each stage. On the first pass, they are the full set of roots of unity. After each pass, the number of distinct values is reduced to form runs of length 2^m, where *m* is the pass. We call this a vector "thinning" operation, which forms runs by duplicating the first value of each sequence.

B.6.2 Illustrative Notation

Figure B-24 shows the action semantics constructs representing the FFT algorithm.

The notation represents an object diagram with some structural details suppressed or expressed as text. Actions are shown as rectangles; the kind of action is shown by a label (such as "LoopAction"), with an optional name and colon to label individual action instances that are part of other actions (for example, the Clause contains a GroupAction with the rolename "body"). Rectangles with names but no action names are ApplyFunctionActions; the name is the name of the applied function. Rectangles with values in them are LiteralValueActions. Nesting of actions indicates ownership of the contents by the containing action. Small squares represent pins. An input pin is a hollow square and an output pin is a filled square. Input and output pins of an action are placed on its outer border. Argument and result pins of a procedure are placed on its border (because they are viewed from the viewpoint of the procedure contents). Pins that are lined up inside it are meant to be an array of pins; a text label applies to the entire list. For example, "loopVariable" represents the array of 4 output pins near the top of LoopAction; the array of pins is a part of LoopAction with the rolename "loopVariable". (This obviously would not be precise enough for a complete notation.) Arrows from output pins to input pins represent data flow relationships.

There is one important action semantics construct that is not based on containment. A clause references an output pin within its embedded test action and an array of output pins within its embedded body action. These pins are owned by the embedded actions, not the clause. This reference is shown in the diagram by a hashed square within the clause connected by a solid line to the appropriate pins within the embedded actions. The square is hashed to indicate that it is not an actual pin but merely a reference. For example, the hashed pin in Clause labeled "testOutput" represents an association from Clause to the test Action with the rolename "testOutput." Clause does not duplicate the pin or own it directly.

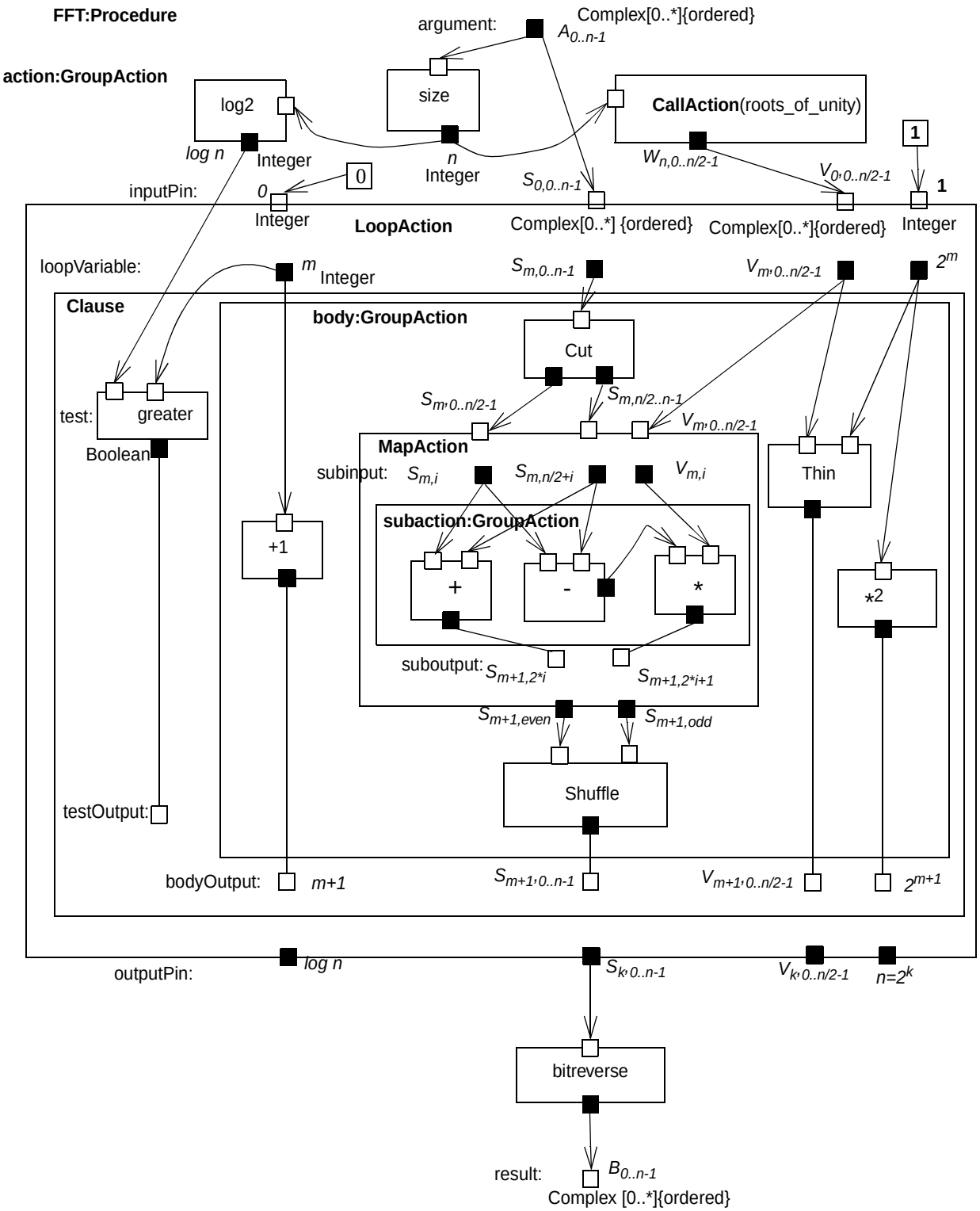


Figure B-24 FFT Algorithm

The consequences of various elements in a construct is explained in informal text. For a precise definition, consult the model and class descriptions. The purpose of this section is to show how the constructs might be used to build larger structures. The notation is suggestive, not precise, and is not meant to be normative.

Many pins have names in italics next to them, such as $A_{0..n-1}$. These are not UML constructs and do not appear in the model. They are merely labels corresponding to the mathematical equations to permit describing the pins in this discussion.

B.6.3 Discussion

The overall construct in the diagram is the procedure FFT. In a UML model, this procedure would be attached to an operation on a class as a method. This procedure takes one argument, a complex vector, and produces one result, another complex vector. The argument is modeled in the procedure as an output pin. This might seem strange; after all, the value is an input to the procedure. The mystery is explained because the argument is viewed from inside the procedure, where it appears as a value available to be used. From the viewpoint of the outside of the procedure, the value is an input, but from the inside, it is an output. Crossing the boundary changes the polarity, and we describe procedures from the inside.

The size of the vector must be a power of two. This could be expressed as a UML constraint, but it is not shown in the diagram. What if it is not a power of 2? Then the model is in error, and its semantics are undefined. Eventually exceptions will be added to the action semantics. When they are present, the procedure could contain a conditional action that would raise an exception on failure of the condition. However, it is permitted to have procedures that do not verify their preconditions, and such verification would be the responsibility of the entire model.

The argument value is used in two ways, as shown by the two arrows leaving the output pin *A*. Both arrows represent the same data value. One copy is used as an input to the loop action (the initial value of the loop variable *S*). The other copy is an input to the `size` PrimitiveFunction action. This action outputs the size of a collection. In some implementations, collections may be resolvable into simpler objects, but the implementations vary widely, and actions on collections can be mathematically defined, so they can be treated as primitive functions.

The size of the array, *n*, is also used in two ways. It is an input to the `log2` primitive function that outputs the base-2 logarithm of the size (remember that n must be a power of two). Why do we treat `log2` as a primitive function and not as an operation? It is easily defined mathematically. Moreover, it might well be directly implemented in hardware (square root is a built-in instruction in most floating point chips today). In any case, the algorithms to compute it are purely implementation and should not be included in a semantic specification of behavior. The purpose of specifying a behavior with action semantics is to understand its inherent constraints, not to make arbitrary implementation decisions that overspecify the behavior.

The other copy of the integer *n* is an argument to a call to the operation `roots_of_unity`. This operation returns the *n* complex roots of unity as a complex vector. The specification of this operation is given by Equation 2.

The main work of the algorithm is performed by a LoopAction. Two of the inputs to this action are the original argument vector and the vector of roots of unity. Within the loop, these vectors are loop variables that are recomputed each iteration. The other two inputs are the integers 0 and 1. These values initialize the loop variables *m* and 2^m. The loop variables are initialized by the inputs, but the loop variables are distinct from the pins that initialize them. The values of the loop variables are recomputed each iteration. If a value from outside the loop is used directly inside a loop, then its value is fixed over all the iterations of the loop. For example, the test action within the loop clause has *log n* and *m* as inputs. The former value is fixed during the loop. The latter value is a loop variable and it changes during each iteration. Obviously, at least one input to a loop test must be a loop variable or the loop will never terminate!

The test and body actions of a loop are sequential. The test must succeed before the body can be executed, and the body must complete before the test can be performed again on the updated loop variables. Within the loop body, however, there is a lot of concurrency. The actions `+1, thin, *2,` and the `cut-map-shuffle` sequence can all be performed concurrently. Expressing such concurrency is one of the main purposes of the actions semantics. This does not mean that an implementation must implement the concurrency using parallelism. It merely means that the implementation does not contain arbitrary constraints.

The primitive actions `+1` and `*2` perform simple unary arithmetic operations. These could be defined as primitive functions. The primitive action `thin` is a primitive vector function. It takes a vector and an integer *t* and copies every *t*'th value from the input array into a run of length *t*, so that the output vector is the same size as the input vector but has fewer unique values in it. This action could obviously be defined as a procedure, but any particular implementation is arbitrary.

`Cut` and `shuffle` are also operations on vectors. `Cut` takes a vector as input and produces two vectors as output, one of them the first half and the other the record half of the input vector. `Shuffle` takes two vectors as input and interleaves their values to produce a single vector containing all the values. These operations could be defined as loops, but by defining these operations as primitive functions we preserve the possibility of a highly parallel implementation (which depends heavily on the exact implementation).

The map action is probably the most complex action in the action semantics. In this example, it takes 3 input vectors, each of size *n/2* elements. All the vectors must be the same size, but they could contain values of different types (in this case, however, they all contain complex vectors). A tuple comprising one value from the same position in each vector is called a *slice*. The input vectors contain *n/2* slices (remember, they must all be the same size). Each slice is the input to a separate execution of the subaction GroupAction. The executions of the subaction are all concurrent. Note the 3 output pins at the top of the subaction. On each execution, each pin gets a value from the corresponding input vector. For example, $S_{m,i}$ is a value from the vector $S_{m,0..n/2-1}$. There is no explicit data flow from the inputs to the map variables. The connection is implicit, part of the definition of the map action, and the pins are of different rank in any case (one is a collection, the other a scalar). In any case, it does not represent a

single data flow. Rather, a loop represents an unbounded sequence of repetitions of its contents, each implicitly connected to the previous repetition by data flows. A loop is a finite representation of this infinite graph.

Each execution of the map subaction GroupAction performs 3 arithmetic operations to yield two complex values as output. Note the two hatched pins within the MapAction labeled `suboutput`. These are not actual pins owned by the MapAction. In the notation, they represent associations from the MapAction to output pins owned by the embedded subaction. They designate the outputs of the map action, but there is no need to physically copy the values to new output pins.

Each execution of the map subaction yields a pair of output values, one for each slice from the inputs. The output values are assembled concurrently into two output vectors, equal in size to the input vectors. The map action applies a subaction concurrently to each of the slices of the input to produce slices of the output. The output need not have the same number of vectors as the input, but each vector must be the same size.

The two output vectors produced by the map action are then shuffled together to produce a single vector of size *n*. This vector and the other three outputs of the loop action update the loop variables for the next iteration.

When the loop variable *m* is finally equal to *log n*, the loop test fails and the values of the loop variables are copied to the output pins of the loop action. Most of the output pins of the loop are ignored (they have served their purpose inside the loop), but the vector *S* serves as input to the `bitreverse` operation. This is another primitive function on a vector, defined by Equation 6. The output of this action, a complex vector of size *n*, becomes the result of the overall procedure.

Most published programs to compute FFT pay a lot of attention to rewriting vector values in place. It is possible to compute the function using the space of the input vector, but this requires some careful bookkeeping that obscures the basic algorithm itself. The inherent concurrency of the algorithm is obscured once a particular scan order is adopted. Specification of algorithms such as the FFT using the actions semantics avoids making implementation decisions not inherent in the basic algorithm. If parallel hardware is available (as in some signal processors), the specification can be implemented in parallel. If a sequential implementation is necessary, the actions semantics specification can be transformed in a straightforward manner to use read and write actions, expand the primitive vector functions, and so on. The point is not to perform such optimizations prematurely in the specification of the basic algorithm.

This example did not touch on every kind of action (for example, the conditional action did not occur, but it has many similarities to loop action), but it has illustrated how actions are connected together to define algorithms. It also did not touch on read and write actions and explicit control dependencies. These are more similar to traditional forms, however, and so less in need of explanation.

B.6.4 Implementation Using Memory Writes

Figure B-25 shows a possible implementation of the `bitreverse` function on an array in which the output values are written into the same array object, replacing the input values. This is obviously not a data flow operation. It is one of many possible implementations, although one that would often be used in an implementation of the FFT, because it is simple to do and does not consume extra memory space.

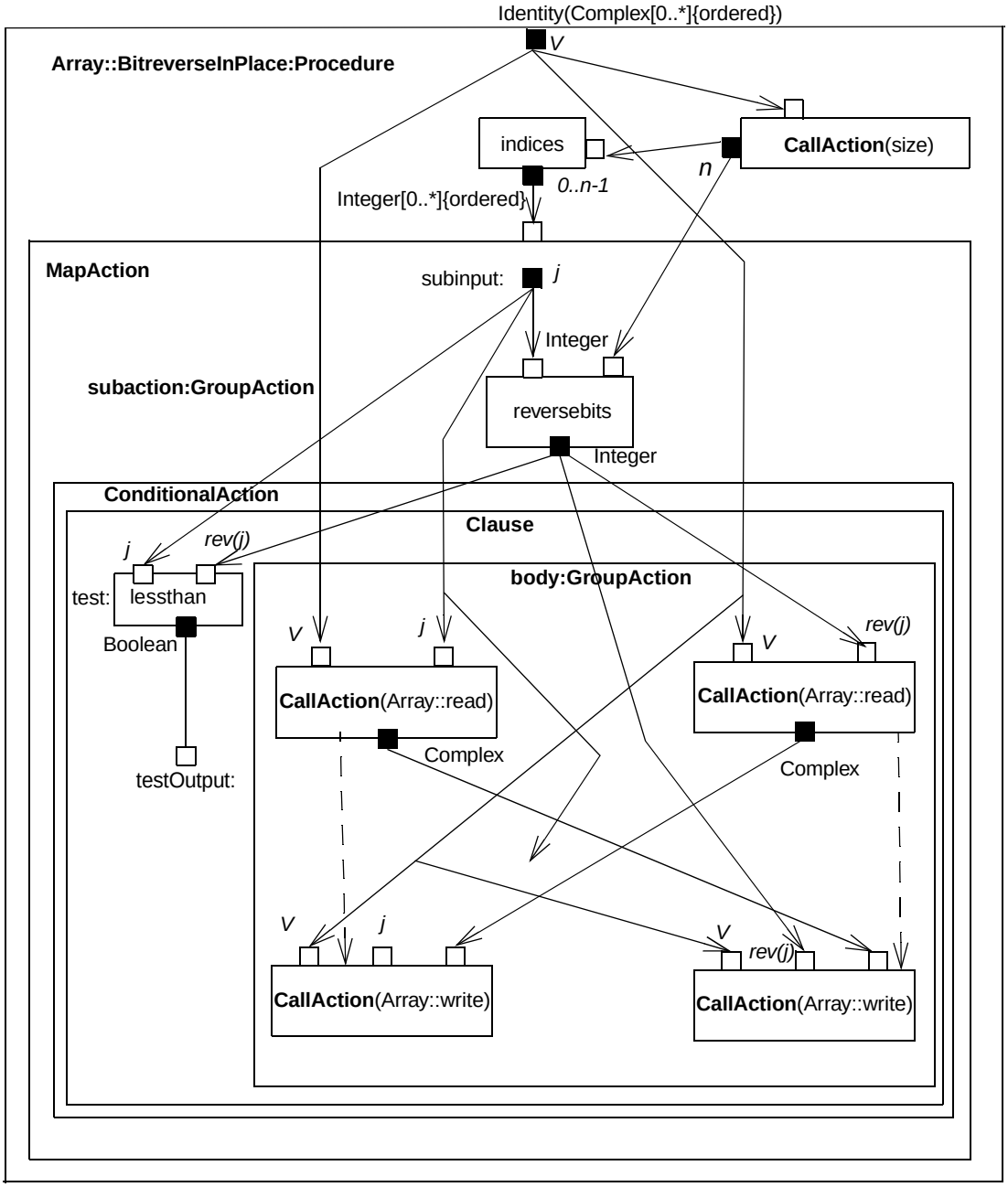


Figure B-25 Implementation of bitreverse using read and write operations

This procedure would be a method on the class `Array`. It would work for an array containing any kind of elements, including complex elements. `Carray` would be a subclass of `Array`.

This procedure has one input pin and no output pin. There are no outputs, because the operation does not generate a result. Rather, it operates on the existing array object whose identity is passed as input. In contrast to Figure B-24 showing the FFT algorithm in a fully data flow manner, the implementation of this procedure operates by side effects, that is, by modifying the state of existing objects.

The input to the procedure is an array *V*, containing elements of arbitrary type. The operation *size* obtains the size of the array, *n*. The primitive function *indices* generates an array containing the indices of array *V*, in this case the integers from 0 to *n-1*. The array of indices is the input to a map action. This means that the subaction within the map action is executed n times, once for each integer from 0 to *n-1*. The value for each execution of the subaction is available on the pin called subaction. Each execution is concurrent with the others.

The subaction comprises the *reversebits* primitive function and an embedded conditional action with a single clause. The *reversebits* primitive computes the function *rev(j,n)*, that is, it reverses the binary bits in an integer to obtain a new integer. One input to the *reversebits* primitive is the value *j*, that is, a value from the array of indices that was the input to the map action. This value is different for each concurrent execution of the subaction. The other input to the reversebits primitive is the value n, which is constant during all executions of the subaction, because it comes from outside the map action.

The conditional action contains a single clause. Its test condition tests whether the map variable *j* is less than its bit-reversed value. If so, the body is executed. If not, the subaction execution is complete—otherwise the same value would be swapped twice. This conditional action does not have clauses to cover every case. If the test fails, there is no other clause that succeeds. If a conditional action produces a data flow output, then at least one clause must be true in all circumstances; otherwise, the data flow output of the conditional would sometimes be undefined. However, in this case the conditional action has no outputs, so it is allowable to not cover all possible situations. In effect, the else clause is a null operation.

The body of the clause merely swaps the values in two cells of the array, those in position *j* and *rev(j,n)*. It does this by reading both values and then writing them back into the opposite positions. An array read operation takes an array *V* and an index *j* and extracts the value at the given position. This is very similar to a direct attribute read action, although a read attribute action has a single input, the identity of the object; the attribute designator is predefined as part of the action, not passed on an input pin. An array write operation takes 3 input values: the array, the index, and the value to write. It has no outputs. Write operations do not have results. They operate by side effects and represent dead ends for data flow values.

The same value of *V* is used 6 times within the overall procedure. At different times, the array may contain different element values, but it is the same array each time with the same identity. It is the identity of an object that is needed for a read or write operation. The identity is unaffected by the operations.

There is a complication. It is permissible to read or write two elements of an array concurrently, but it is not acceptable to write a new value in a cell before the old value has been read. Therefore it is necessary to add a control flow dependency between a

read operation and a write operation at the same position in the array, so that a value is not overwritten prematurely. This is shown in Figur eB-25 by dashed arrows from each read operation to the write operation on the same index value. Algorithms that use read and write operations often need explicit control flow dependencies, as opposed to algorithms in which values pass by data flow only. Note that there is an implicit control flow dependency from the contents of a procedure to the procedure itself—the procedure will not return to the caller until all internal executions have completed.

Glossary

This glossary defines the terms that are used to describe the Unified Modeling Language (UML) and the Meta Object Facility (MOF). In addition to UML and MOF specific terminology, it includes related terms from OMG standards and object-oriented analysis and design methods, as well as the domain of object repositories and meta data managers. Glossary entries are organized alphabetically and MOF specific entries are identified as '[MOF]'.

Notation Conventions

The entries in the glossary usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.

When one or more words in a multi-word term is enclosed in brackets, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.

The following conventions are used in this glossary:

- Contrast: <term>
  Refers to a term that has an opposed or substantively different meaning.
- See: <term>
  Refers to a related term that has a similar, but not synonymous meaning.
- Synonym: <term>
  Indicates that the term has the same meaning as another term, which is referenced.
- Acronym: <term>
  Indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.

This glossary defines the terms that are used to describe the Unified Modeling Language (UML) and the Meta Object Facility (MOF). In addition to UML and MOF specific terminology, it includes related terms from OMG standards and object-oriented analysis and design methods, as well as the domain of object repositories and meta data managers. Glossary entries are organized alphabetically and MOF specific entries are identified as '[MOF]'.

Glossary Terms

abstract class — A class that cannot be directly instantiated. Contrast: *concrete class.*

abstraction — The essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer.

action — The specification of an executable statement that is part of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or a value of an attribute. See: *procedure*.

action sequence — An expression that resolves to a sequence of actions.

action state — A state that represents the execution of an atomic action, typically the invocation of an operation.

activation — The execution of an action.

active class — A class whose instances are active objects. See: *active object.*

active object — An object that owns a thread and can initiate control activity. An instance of active class. See: *active class, thread.*

activity graph — A special case of a state machine that is used to model processes involving one or more classifiers. Contrast: *statechart diagram.*

actor [class] — A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

actual parameter — Synonym: *argument*.

aggregate [class] — A class that represents the "whole" in an aggregation (whole-part) relationship. See: *aggregation.*

aggregation — A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. See: *composition.*

analysis — The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses what to do, design focuses on how to do it. Contrast: *design*.

analysis time — Refers to something that occurs during an analysis phase of the software development process. See: *design time, modeling time.*

architecture — The organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

argument — A binding for a parameter that resolves to a run-time instance. Synonym: *actual parameter*. Contrast: *parameter*.

artifact — A physical piece of information that is used or produced by a software development process. Examples of Artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component. Synonym: *product*. Contrast: *component*.

association — The semantic relationship between two or more classifiers that specifies connections among their instances.

association class — A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end — The endpoint of an association, which connects the association to a classifier.

attribute — A feature within a classifier that describes a range of values that instances of the classifier may hold.

auxiliary class — A stereotyped class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. Auxiliary classes are typically used together with focus classes, and are particularly useful for specifying the secondary business logic or control flow of components during design. See also: *focus*.

behavior — The observable effects of an operation or event, including its results.

behavioral feature — A dynamic feature of a model element, such as an operation or method.

behavioral model aspect — A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association — An association between two classes. A special case of an n-ary association.

binding — The creation of a model element from a template by supplying arguments for the parameters of the template.

boolean An enumeration whose values are true and false.

boolean expression An expression that evaluates to a boolean value.

cardinality The number of elements in a set. Contrast: *multiplicity.*

child In a generalization relationship, the specialization of another element, the parent. See: *subclass, subtype*. Contrast: *parent*.

call state An action state that invokes an operation on a classifier.

class A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. See: *interface*.

classifier A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, datatypes, and components.

classification The assignment of an object to a classifier. See *dynamic classification, multiple classification* and *static classification*.

class diagram A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

client A classifier that requests a service from another classifier. Contrast: *supplier.*

collaboration The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction. See: *interaction.*

collaboration diagram A diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See: *sequence diagram*.

comment An annotation attached to an element or a collection of elements. A note has no semantics. Contrast: *constraint.*

compile time Refers to something that occurs during the compilation of a software module. See: *modeling time, run time*.

component A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component is typically specified by one or more classifiers (e.g., implementation classes) that reside on it, and may be implemented by one or more artifacts (e.g., binary, executable, or script files). Contrast: *artifact.*

component diagram A diagram that shows the organizations and dependencies among components.

composite [class] A class that is related to one or more classes by a composition relationship. See: *composition.*

composite aggregation Synonym: *composition*.

composite state A state that consists of either concurrent (orthogonal) substates or sequential (disjoint) substates. See: *substate*.

composition A form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts. Composition may be recursive.
Synonym: *composite aggregation*.

concrete class A class that can be directly instantiated. Contrast: *abstract class.*

concurrency The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. See: *thread.*

concurrent substate A substate that can be held simultaneously with other substates contained in the same composite state. See: *composite state.* Contrast: *disjoint substate.*

constraint A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. Constraints are one of three extensibility mechanisms in UML. See: *tagged value, stereotype.*

container 1. An instance that exists to contain other instances, and that provides operations to access or iterate over its contents. (for example, arrays, lists, sets).
2. A component that exists to contain other components.

containment hierarchy A namespace hierarchy consisting of model elements, and the containment relationships that exist between them. A containment hierarchy forms a graph.

context A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

datatype A descriptor of a set of values that lack identity and whose operations do not have side effects. Datatypes include primitive pre-defined types and user-definable types. Pre-defined types include numbers, string and time. User-definable types include enumerations.

defining model [MOF] The model on which a repository is based. Any number of repositories can have the same defining model.

delegation The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast: *inheritance*.

dependency A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

deployment diagram A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units. See: *component diagrams*.

derived element A model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

design The part of the software development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design time Refers to something that occurs during a design phase of the software development process. See: *modeling time*. Contrast: *analysis time*.

development process A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the following diagrams: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, component diagram, and deployment diagram.

disjoint substate A substate that cannot be held simultaneously with other substates contained in the same composite state. See: composite state. Contrast: *concurrent substate*.

distribution unit A set of objects or components that are allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

domain An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

dynamic classification A semantic variation of generalization in which an object may change its classifier. Contrast: *static classification*.

element An atomic constituent of a model.

entry action A procedure executed upon entering a state in a state machine regardless of the transition taken to reach that state.

enumeration A list of named values used as the range of a particular attribute type. For example, RGBColor = {red, green, blue}. Boolean is a predefined enumeration with values from the set {false, true}.

event — The specification of a significant occurrence that has a location in time and space. In the context of state diagrams, an event is an occurrence that can trigger a transition.

exit action — A procedure executed upon exiting a state in a state machine regardless of the transition taken to exit that state.

export — In the context of packages, to make an element visible outside its enclosing namespace. See: *visibility.* Contrast: *export* [OMA], *import*.

expression — A string that evaluates to a value of a particular type. For example, the expression "(7 + 5 * 3)" evaluates to a value of type number.

extend — A relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See *extension point*, *include*.

facade — A stereotyped package containing only references to model elements owned by another package. It is used to provide a 'public view' of some of the contents of a package.

feature — A property, like operation or attribute, which is encapsulated within a classifier, such as an interface, a class, or a datatype.

final state — A special kind of state signifying that the enclosing composite state or the entire state machine is completed.

fire — To execute a state transition. See: *transition.*

focus class — A stereotyped class that defines the core logic or control flow for one or more auxiliary classes that support it. Focus classes are typically used together with one or more auxiliary classes, and are particularly useful for specifying the core business logic or control flow of components during design. See also: *auxiliary.*

focus of control — A symbol on a sequence diagram that shows the period of time during which an object is performing a procedure, either directly or through a subordinate procedure.

formal parameter — Synonym: *parameter.*

framework — A stereotyped package that contains model elements which specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks. See: *pattern*.

generalizable element — A model element that may participate in a generalization relationship. See: *generalization*.

generalization — A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: *inheritance.*

guard condition — A condition that must be satisfied in order to enable an associated transition to fire.

implementation — A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation.

implementation class — A stereotyped class that specifies the implementation of a class in some programming language (e.g., C++, Smalltalk, Java) in which an instance may not have more than one class. An Implementation class is said to realize a type if it provides all of the operations defined for the type with the same behavior as specified for the type's operations. See also: *type*.

implementation inheritance — The inheritance of the implementation of a more general element. Includes inheritance of the interface. Contrast: *interface inheritance*.

import — In the context of packages, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: *export*.

include — A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location which is defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its structure (i.e., attributes or operations). See *extend*.

inheritance — The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See *generalization*.

initial state — A special kind of state signifying the source for a single transition to the default state of the composite state.

instance — An entity that has unique identity, a set of operations that can be applied to it, and state that stores the effects of the operations. See: *object*.

interaction — A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See *collaboration.*

interaction diagram — A generic term that applies to several types of diagrams that emphasize object interactions. These include collaboration diagrams and sequence diagrams.

interface — A named set of operations that characterize the behavior of an element.

interface inheritance — The inheritance of the interface of a more general element. Does not include inheritance of the implementation. Contrast: *implementation inheritance*.

internal transition — A transition signifying a response to an event without changing the state of an object.

layer — The organization of classifiers or packages at the same level of abstraction. A layer represents a horizontal slice through an architecture, whereas a partition represents a vertical slice. Contrast: *partition*.

link — A semantic connection among a tuple of objects. An instance of an association. See: *association*.

link end — An instance of an association end. See: *association end*.

message — A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation.

metaclass — A class whose instances are classes. Metaclasses are typically used to construct metamodels.

meta-metamodel — A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metamodel — A model that defines the language for expressing a model.

metaobject — A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method — The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

model — An abstraction of a physical system with a certain purpose. See: *physical system*.

[MOF] — Usage note: In the context of the MOF specification, which describes a meta-metamodel, for brevity the meta-metamodel is frequently to as simply the model.

model aspect — A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

model elaboration — The process of generating a repository type from a published model. Includes the generation of interfaces and implementations which allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model element — An element that is an abstraction drawn from the system being modeled. Contrast: *view element*.

[MOF] — In the MOF specification model elements are considered to be metaobjects.

model library — A stereotyped package that contains model elements which are intended to be reused by other packages. A model library differs from a profile in that a model library does not extend the metamodel using stereotypes and tagged definitions. A model library is analogous to a class library in some programming languages.

modeling time — Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. Usage note: When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns. See: *analysis time, design time*. Contrast: *run time*.

module — A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See: *component.*

multiple classification — A semantic variation of generalization in which an object may belong directly to more than one classifier. See: *static classification, dynamic classification*.

multiple inheritance — A semantic variation of generalization in which a type may have more than one supertype. Contrast: *single inheritance*.

multiplicity — A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast: *cardinality.*

multi-valued [MOF] — A model element with multiplicity defined whose Multiplicity Type:: upper attribute is set to a number greater than one. The term multi-valued does not pertain to the number of values held by an attribute, parameter, etc. at any point in time. Contrast: *single-valued*.

n-ary association — An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes. Contrast: *binary association*.

name — A string used to identify a model element.

namespace — A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: *name.*

node — A node is classifier that represents a run-time computational resource, which generally has at least a memory and often processing capability. Run-time objects and components may reside on nodes.

object — An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. See: *class, instance.*

object diagram — A diagram that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a collaboration diagram. See: *class diagram, collaboration diagram.*

object flow state — A state in an activity graph that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object lifeline — A line in a sequence diagram that represents the existence of an object over a period of time. See: *sequence diagram*.

operation — A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.

package — A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages.

parameter — The specification of a variable that can be changed, passed, or returned. A parameter may include a name, type, and direction. Parameters are used for operations, messages, and events. Synonyms: *formal parameter*. Contrast: *argument*.

parameterized element — The descriptor for a class with one or more unbound parameters. Synonym: *template*.

parent — In a generalization relationship, the generalization of another element, the child. See: *subclass, subtype*. Contrast: *child*.

participate — The connection of a model element to a relationship or to a reified relationship. For example, a class participates in an association, an actor participates in a use case.

partition — 1. activity graphs: A portion of an activity graphs that organizes the responsibilities for actions. See: *swimlane*.
2. architecture: A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice. Contrast: *layer*.

pattern — A template collaboration.

persistent object — An object that exists after the process or thread that created it has ceased to exist.

postcondition — A constraint that must be true at the completion of an operation.

precondition — A constraint that must be true when an operation is invoked.

primitive type — A pre-defined basic datatype without any substructure, such as an integer or a string.

procedure — A procedure is a coordinated set of actions that models a computation, such as an algorithm. See: *action*.

process — 1. A heavyweight unit of concurrency and execution in an operating system. Contrast: *thread*, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.
2. A software development process—the steps and guidelines by which to develop a system.
3. To execute an algorithm or otherwise handle something dynamically.

profile — A profile is a stereotyped package that contains model elements which have been customized for a specific domain or purpose using extension mechanisms, such as stereotypes, tagged definitions and constraints. A profile may also specify model libraries on which it depends and the metamodel subset that it extends.

projection — A mapping from a set to a subset of it.

property — A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML; others may be user defined. See: tagged value.

pseudo-state — A vertex in a state machine that has the form of a state, but doesn't behave as a state. Pseudo-states include initial and history vertices.

physical system — 1. The subject of a model.
2. A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast: system.

published model [MOF] — A model that has been frozen, and becomes available for instantiating repositories and for the support in defining other models. A frozen model's model elements cannot be changed.

qualifier — An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

receive [a message] — The handling of a stimulus passed from a sender instance. See: *sender, receiver.*

receiver [object] — The object handling a stimulus passed from a sender object. Contrast: *sender.*

reception — A declaration that a classifier is prepared to react to the receipt of a signal.

reference — 1. A denotation of a model element.
2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym: *pointer.*

refinement — A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relationship — A semantic connection among model elements. Examples of relationships include associations and generalizations.

repository A facility for storing object models, interfaces, and implementations.

requirement A desired feature, property, or behavior of a system.

responsibility A contract or obligation of a classifier.

reuse The use of a pre-existing artifact.

role The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

run time The period of time during which a computer program executes. Contrast: *modeling time*.

scenario A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction or the execution of a use case instance. See: *interaction.*

schema [MOF] In the context of the MOF, a schema is analogous to a package which is a container of model elements. Schema corresponds to an MOF package. Contrast: *metamodel, package*.

semantic variation point A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send [a message] The passing of a stimulus from a sender instance to a receiver instance. See: *sender, receiver.*

sender [object] The object passing a stimulus to a receiver object. Contrast: *receiver.*

sequence diagram A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See: *collaboration diagram.*

signal The specification of an asynchronous stimulus communicated between instances. Signals may have parameters.

signature The name and parameters of a behavioral feature. A signature may include an optional returned parameter.

single inheritance A semantic variation of generalization in which a type may have only one supertype. Synonym: *multiple inheritance* [OMA]. Contrast: *multiple inheritance*.

single valued [MOF] A model element with multiplicity defined is single valued when its Multiplicity Type:: upper attribute is set to one. The term single-valued does not pertain to the number of values held by an attribute, parameter, etc., at any point in time, since a single-valued attribute (for instance, with a multiplicity lower bound of zero) may have no value. Contrast: *multi-valued*.

specification A declarative description of what something is or does. Contrast: *implementation*.

state A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast: *state* [OMA].

statechart diagram A diagram that shows a state machine. See: *state machine*.

state machine A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.

static classification A semantic variation of generalization in which an object may not change classifier. Contrast: *dynamic classification.*

stereotype A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extensibility mechanisms in UML. See: *constraint, tagged value.*

stimulus The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See: *message*.

string A sequence of text characters. The details of string representation depend on implementation, and may include character sets that support international characters and graphics.

structural feature A static feature of a model element, such as an attribute.

structural model aspect A model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

subactivity state A state in an activity graph that represents the execution of a non-atomic sequence of steps that has some duration.

subclass In a generalization relationship, the specialization of another class; the superclass. See: *generalization*. Contrast: *superclass.*

submachine state A state in a state machine which is equivalent to a composite state but its contents is described by another state machine.

substate A state that is part of a composite state. See: *concurrent state, disjoint state*.

subpackage A package that is contained in another package.

subsystem — A grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements. See *package*. See: *physical system*.

subtype — In a generalization relationship, the specialization of another type; the supertype. See: *generalization*. Contrast: *supertype*.

superclass — In a generalization relationship, the generalization of another class; the subclass. See: *generalization*. Contrast: *subclass*.

supertype — In a generalization relationship, the generalization of another type; the subtype. See: *generalization*. Contrast: *subtype*.

supplier — A classifier that provides services that can be invoked by others. Contrast: *client*.

swimlane — A partition on a activity diagram for organizing the responsibilities for actions. Swimlanes typically correspond to organizational units in a business model. See: *partition*.

synch state — A vertex in a state machine used for synchronizing the concurrent regions of a state machine.

system — A top-level subsystem in a model. Contrast: physical system.

tagged value — The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML; others may be user defined. Tagged values are one of three extensibility mechanisms in UML. See: *constraint, stereotype.*

template — Synonym: *parameterized element*.

thread [of control] — A single path of execution through a program, a dynamic model, or some other representation of control flow. Also, a stereotype for the implementation of an active object as lightweight process. See *process*.

time event — An event that denotes the time elapsed since the current state was entered. See: *event*.

time expression — An expression that resolves to an absolute or relative value of time.

top level — A stereotype of package denoting the top-most package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces “see” outwards. For example, opLevel subsystem represents the top of the subsystem containment hierarchy.

trace — A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other.

transient object — An object that exists only during the execution of the process or thread that created it.

transition — A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire.

type — A stereotyped class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type may not contain any methods, maintain its own thread of control, or be nested. However, it may have attributes and associations. Although an object may have at most one implementation class, it may conform to multiple different types. See also: *implementation class* Contrast: *interface*.

type expression — An expression that evaluates to a reference to one or more types.

uninterpreted — A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation. See: *any* [CORBA].

usage — A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

use case [class] — The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. See: *use case instances.*

use case diagram — A diagram that shows the relationships among actors and use cases within a system.

use case instance — The performance of a sequence of actions being specified in a use case. An instance of a use case. See: *use case class*.

use case model — A model that describes a system's functional requirements in terms of use cases.

utility — A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience.

value — An element of a type domain.

vertex — A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See: *state, pseudo-state.*

view — A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

view element — A view element is a textual and/or graphical projection of a collection of model elements.

view projection — A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

visibility — An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

Index

A

abstract class 3-38
abstract operation 3-46
abstract syntax section 2-9
Abstraction 2-17, 2-18
access 3-62, 3-91
access (Permission) 2-47, 2-193, 2-197
accessing a package 3-62
accessing elements 2-46
Action 2-103
action
  definition 2-211
  specifying 2-211
action (Message) 2-119
action descriptions
  conventions 2-207
action execution
  status 2-215
action expression 3-145
action foundation 2-204, 2-211
action language mapping B-1
action state 3-158
action-object flow relationships 3-163
ActionSequence 2-103
ActionState 2-171, 2-176, 2-178
activation 3-108, 3-110
activator (Message)2-119
active clas s2-26
active object 3-128
active state 2-155
active state configuratio n2-156
activity diagra m3-155
activity grap h3-155
Activity Graphs Package 2-170, 2-181
activity in a state 2-156
activity state 3-158, 3-159
ActivityGraph 2-172, 2-175, 2-178
Actor 2-131, 2-133, 2-135
actor 3-97
actor relationshi p3-99
addition (Include) 2-132
addOnly (ChangeableKind) 2-22, 2-49, 2-87
addOnly (keyword) 3-73
adornment
  on association 3-68
  order 3-74
after (keyword)3-143
aggregate (AggregationKind) 2-22, 2-86
aggregation 3-72
aggregation (AssociationEnd) 2-22, 2-66
AggregationKind 2-86
alias (ElementImport) 2-183
angle bracket
  for binding argument 3-55
annotatedElement (Comment) 2-30
architecture of metamodel 2-4
ArgListsExpression 2-86
argument (Binding) 2-25
argument (Stimulus) 2-103
argument lis t3-133
arrow
  dashed
    for constraint 3-27
    for dependency 3-90
    for extend 3-98
    for flow relationshi p3-65
    for include 3-98
    for instance of 3-93
    for object flow 3-163
    for realizatio n3-49
    for retur n3-112
  solid
    for call 3-112
    for generalization 3-86
    for message 3-111
    for navigation 3-73
    for transition 3-145
Artifacts 1-2

development project 1-2
UML-defining Artifacts 1-2
artifacts
UML-defining 1-2
Association 2-19, 2-51, 2-64
association 3-68
association (AssociationEnd) 2-24
association (Classifier) 2-28
association (keyword) 3-84
association (Link) 2-99
association (LinkEnd) 2-100
association class 3-69, 3-77
association end 3-68, 3-71
association name 3-68
association role 3-125
AssociationClass 2-21, 2-52, 2-67
AssociationEnd 2-21, 2-53, 2-64
associationEnd (Attribute) 2-24
associationEnd (LinkEnd) 2-100
AssociationEndRole 2-115, 2-120
AssociationRole 2-116, 2-120
Attribute 2-24, 2-53
attribute 3-38, 3-41
in object 3-65
attribute writing B-9, B-10
AttributeLink 2-97, 2-103
available in put and output 2-229
availableContents (ClassifierRole) 2-116
availableFeature (ClassifierRole) 2-116
availableQualifier (AssociationEndRole) 2-115

B

Bag 6-40
bar
for stub state 3-152
for stubbed transition 3-148
for synchronization, fork, join 3-146
base (AssociationEndRole) 2-115
base (AssociationRole) 2-116
base (ClassifierRole) 2-116
base (Extend) 2-131
base (Include) 2-132
baseClass (Stereotype) 2-78
Basic Values and Types 6-7
become (Flow) 2-37
become (keyword) 3-65
before-after methods 2-318
behavior
of operation as note 3-46
Behavioral Elements Package 2-92
BehavioralFeature 2-24, 2-53
binary association 3-68
bind (keyword) 3-91
Binding 2-25, 2-42, 2-54, 2-73
binding 3-54
blocked 2-315
body (Comment) 2-30
body (Constraint) 2-32, 2-77
body (Expression) 2-87
body (Mapping) 2-88
body (Method) 2-41
boldface
for class name 3-36
for compartment name 3-39
for special list elements 3-37
Boolean 2-86, 6-35
Boolean property 3-30
BooleanExpression 2-86
bound (SynchState) 2-149, 2-164
bound element 3-54
braces
for constraint 3-27, 3-28
for property string 3-29, 3-38, 3-42
branch 3-159, 3-161
branch point 3-150
break 2-334
bull's eye
for final state 3-140

C

call 3-108
call (Usage) 2-51
call event 2-319, 3-143
CallAction 2-98, 2-104
CallConcurrencyKind 2-86
CallEvent 2-142
CallState 2-173, 2-176
chain of transitions 3-150
changeability 3-73
changeability (AssociationEnd) 2-22
changeability (Attribute) 2-49
changeable (ChangeableKind) 2-22, 2-49, 2-87
ChangeableKind 2-87
ChangeEvent 2-143
changeExpression (ChangeEvent) 2-143
child (Generalization) 2-39
choice (PseudostateKind) 2-91, 2-146
circle
bull's eye
for final state 3-140
filled
for initial state 3-140
for history state 3-148
for interface 3-51
for junction 3-150
for synch state 3-154
Class 2-26, 2-55, 2-67
class 3-35
declared in another class 3-82
class description
conventions 2-208
class diagram 3-34
class in state 3-65
class scope
attribute 3-43
operation 3-45
Classifier 2-27, 2-55
classifier 3-35
classifier (Instance) 2-99
classifier (ScopeKind) 2-36, 2-49, 2-91
classifier role 3-124
ClassifierInState 2-173

ClassifierRole 2-116, 2-120
clause 2-231
  loop 2-232
client (Dependency) 2-33
clientDependency (ModelElement) 2-41
Collaboration 2-117, 2-121, 2-123, 2-124
collaboration 3-114, 3-121
collaboration diagra m3-114, 3-116
collaboration role 3-124
collaborationMultiplicity (AssociationEndRole) 2-115
Collaborations Package 2-111
Collect Operation 6-24
Collection 6-36
collection action 2-206
Collection Operations 6-22
Collection Type Hierarchy and Type Conformance Rules 6-21
Collection-Related Typed 6-36
Collections 6-19
Collections of Collections 6-21
colon
  for return typ e3-44
  for sequence expressio n3-131
  for type 3-42, 3-45, 3-53, 3-61, 3-65, 3-72, 3-81, 3-124, 3-173, 3-175
Combining Properties 6-14
Comment 2-30, 2-57, 6-11
comment 3-26, 3-28
Common Behavior Package 2-93
communication association 3-97, 3-99
communication relationship 2-132, 2-137
communicationConnection (Message) 2-119
communicationLink (Stimulus) 2-103
compartment 3-38
  name 3-39
  special 3-36
complete
  status 2-215
complete (Generalization) 2-39
complete (keyword) 3-87
completion event 2-159
completion transition 2-159
complex transition 3-146, 3-147
Component 2-30, 2-58
component 3-174
  on node 3-174
component diagram 3-169
ComponentInstance 2-98, 2-104
composite (AggregationKind) 2-22, 2-66, 2-86
composite object 3-67
composite state 3-140, 3-154
CompositeState 2-143, 2-150, 2-156
composition 3-67, 3-81
Compound transition 2-158
computation action 2-206
computation actions 2-287
concurrency
  in state machine 2-162
  maximized 2-204
  of operation 3-45
  synchronizing 2-163
concurrency (Operation) 2-45
concurrent (CallConcurrencyKind) 2-45, 2-87
concurrent lifeline s3-109
concurrent substate 3-140
condition (Extend) 2-131
condition event 3-143
conditional fork 3-169
conflict 2-162
connection (Association) 2-20
connection (Link) 2-99
constant
  enumeration 3-57
constrainedElement (Constraint) 2-32, 2-77
constrainedStereotype (Constraint) 2-77
Constraint 2-31, 2-58, 2-72, 2-80
constraint 3-26, 3-28
  as list element 3-27
constraint (ModelElement) 2-41, 2-77
constraint languag e2-10, 2-82, 3-27
container (StateVertex)2-148
contents (Partition) 2-174
context 3-115
context (Exception) 2-98
context (Interaction) 2-118, 2-119
context (Signal) 2-103
context (StateMachine) 2-147
continue 2-334
control flow 2-204
  on group action 2-230
control flow icon 3-130
control flow type 3-134
control icon s3-165
conventions
  action descriptions 2-207
  class description 2-208
copy (Flow) 2-37
copy (keyword) 3-65
copying composite 2-66
create (BehavioralFeature) 2-25
create (CallEvent) 2-143
create (Usage) 2-51
CreateAction 2-98, 2-104
creating a link B-14
creation 3-109, 3-115, 3-134
cross
  for destruction 3-109
cube
  for node 3-173

D

data flow 2-204
data flow relationship 2-179
Data Types Foundation Package 2-85
DataType 2-32, 2-58, 2-73, 2-85
DataValue 2-98, 2-104, 2-109
decision, *See* branch
deepHistory (PseudostateKind) 2-91, 2-145, 2-157
default entry 2-156
defaultElement (TemplateParameter) 2-50
defaultValue (Parameter) 2-46
defer (keyword)3-166
deferrableEvent (State) 2-147

deferred event 2-156, 2-157, 3-166
delegation 2-318
Dependency 2-33, 2-59, 2-73
dependency 3-90
  subsystem 3-21
deployment diagram 3-171
deploymentLocation (Component) 2-31
Derivation 2-17
derivation 3-91
derive (Abstraction) 2-18
derive (keyword) 3-91
derived (ModelElement) 2-42
derived element 3-93
descriptor 2-69
design patter n3-117
destination state 3-146
destroy (BehavioralFeature) 2-25
destroy (CallEvent) 2-143
DestroyAction 2-98
destroyed (Instance) 2-99
destroyed (keyword) 3-115
destroyed (Link) 2-100
destroying a link B-16
destroying composite 2-66
destruction 3-109, 3-115, 3-134
development project 1-2
diamond
  filled
    for composition 3-81
  for aggregation 3-72
  for branch or merge 3-160
  for merge 3-150
  for n-ary association 3-79
direct receive and repl y2-319
discriminator 2-40, 3-86, 3-87
discriminator (Generalization)2-39
disjoint (Generalization) 2-40
disjoint (keyword) 3-87
disjoint substate 3-140
dispatchAction (Stimulus) 2-103
do activity 2-156
doActivity (State) 2-147
document (Component) 2-19
documentation (Element) 2-33
dog-eared rectangle
  for note 3-13
dot
  for navigation 3-13
  for sequence expressio n3-131
double colon
  for pathname 3-36, 3-61
double dot
  for integer ran g 3-75
dynamic choice point 3-151
dynamic concurrency 3-168
dynamicArguments (ActionState) 2-172
dynamicArguments (SubactivityState) 2-175
dynamicMultiplicity (ActionState) 2-172
dynamicMultiplicity (SubactivityState) 2-175

E

effect (Transition) 2-150
elapsed-time even t3-143
Element 2-33, 2-59
element property 3-29
ElementImport 2-183, 2-187
ElementOwnership 2-33, 2-59
ElementResidence 2-34, 2-59
ellipse
  dashed
    for collaboration 3-118
  for use case 3-96
ellipsis
  for generalization 3-86
  for missing element 3-39
else (keyword) 3-151
enabled transition 2-159
entering a concurrent composite stat e2-157
entry (ActionState) 2-172
entry (State) 2-147
entry action 2-155
entry stub state 3-152
Enumeration 2-34, 2-59, 6-35
enumeration 3-57
enumeration (EnumerationLiteral) 2-35
enumeration literal 3-57
Enumeration Types 6-8
EnumerationLiteral 2-35, 2-59
equal sign
  for attribute value 3-65
  for default value 3-45, 3-53
  for initial value 3-42
  for tagged value 3-29
Event 2-144, 2-155
event 3-142
event processing 2-161
event signature 3-145
example
  Fast Fourier Transform algorithm B-26
examples
  use of action semantics B-1
examples section 3-5
Exception 2-98, 2-104, 2-110
exception
  called procedure 2-315
executable (Component) 2-19, 2-31
executing
  status 2-215
execution engine 2-203
Exists Operation 6-26
exit (State) 2-147
exit action 2-155
exit stub state 3-152
exiting a concurrent state 2-157
exiting a non-concurrent state 2-157
Expression 2-87
expression 3-11
expression (Guard) 2-145
expression (ProgrammingLanguageDataType) 2-48
Extend 2-131, 2-133, 2-139

extend 3-98
extend (UseCase) 2-133
extendedElement (Stereotype) 2-78
extensibility mechanism 3-29, 3-31
extension (Extend) 2-131
Extension Mechanisms Foundation Package 2-73
extension point 3-96
extension points compartment 3-96
ExtensionPoint 2-131, 2-134
extensionPoint (Extend) 2-131
extensionPoint (UseCase) 2-133
extent of classifier
  reading 2-260

F

facade (Package) 2-185
facade (stereotype) 3-17
Facility Implementation Requirements 5-24
factored transition path 3-150
false (Boolean) 2-86
Fast Fourier transform (FFT) B-26
Feature 2-35, 2-59
feature 3-38
feature (Classifier) 2-28
Features on Types Themselves 6-19
FFT (fast Fourier transform) B-26
file (Component) 2-19
final state 3-140
FinalState 2-144, 2-151, 2-157
fire a transition 2-161
Flow 2-36
flow relationship 3-65
focus of control 3-108, 3-110
font usage 3-8
ForAll Operation 6-25
fork (PseudostateKind) 2-91, 2-145
fork of control 3-146
formalism 2-8
Foundation packag e2-11
four-layer metamodel architecture 2-4
framework (Package) 2-185, 2-193
framework (stereotype) 3-17
friend (Permission) 2-47
frozen (ChangeableKind) 2-22, 2-49, 2-87
frozen (keyword) 3-43, 3-73
full descriptor 2-69

G

GeneralizableElement 2-37, 2-59, 2-69
Generalization 2-38, 2-60, 2-69
  of package 2-193
  of subsystem 2-196
  of use case 2-138
generalization 3-86
  constraints on 3-87
  use case 3-98
generalization (GeneralizableElement) 2-38
Geometry 2-88
global (AssociationEnd) 2-24
global (keyword) 3-85
global (LinkEnd) 2-100
Goals 1-4
Grammar for OC L6-45
graphic construct s3-6
graphic marker 3-32
group action 2-204, 2-230
group property 3-39
Guard 2-144, 2-151, 2-159
guard (Transition) 2-150
guard condition 3-145
guarded (CallConcurrencyKind) 2-45, 2-87
guillemets
  for keyword 3-11
  for stereotype 3-31, 3-38

H

Harel statechart 2-169
hidden element 3-39
high-level transitio n2-158
history
  deep 2-157
  shallow 2-157
history state 3-148
host
  execution 2-217
hyperlink 3-7

I

icon
  for stereotype 3-32, 3-38
icon (Stereotype) 2-78
icons 3-6
If-then-else logic B-3
implementation (Generalization) 2-39
implementation class
  and type 3-49
implementation diagra m3-169
ImplementationClass 2-60
implementationClass (Class) 2-27, C-16
implementationLocation (ModelElement) 2-41
import 3-62, 3-91
import (Permission) 2-47, 2-192, 2-197
imported element 3-17
importedElement (Package) 2-185
importing a package 3-62
importing elements 2-46
in (ParameterDirectionKind) 2-46, 2-90
Include 2-132, 2-134, 2-138
include
  a use case 3-98
include (keyword) 3-152
include (UseCase) 2-133
incoming (StateVertex) 2-148
incomplete (Generalization) 2-40
incomplete (keyword) 3-87
indeterminacy
  conditional 2-231
Industry Trends 1-3
Inheritance 2-69
inheritance

request resolution 2-318
inheritance relationship 2-38
initial (PseudostateKind) 2-91, 2-145
initial state 3-140
initial value
of attribute 3-42
initialValue (Attribute) 2-24
inout (ParameterDirectionKind) 2-46, 2-90
input
available 2-229
input event ico n3-165
input pin 2-212
input signature 2-330
Instance 2-98, 2-104
instance 3-14, 3-93
of classifier 3-93
instance (LinkEnd) 2-100
instance (ScopeKind) 2-36, 2-49, 2-91
instance level collaboration 3-115
instantiable subsystem 3-19
instantiate (Usage) 2-51
Instantiation 2-70
inState (ClassifierInState) 2-173
Integer 2-88, 6-33
Interaction 2-118, 2-123, 2-128
interaction 3-123
interaction (Collaboration) 2-117, 2-118
interaction (Message) 2-119
Interface 2-40, 2-60, 2-71
use cas e2-137
interface 3-50
on subsystem 3-21
interface specifier 3-72
internal transition 2-159
internal transition compartment 3-138
internalTransition (State) 2-147
invariant (Constraint) 2-32
Invariants 6-5
invisible hyperlink 3-7
isAbstract (GeneralizableElement) 2-37
isAbstract (Operation) 2-45
isAbstract (Reception) 2-102
isActive (Class) 2-26
isConcurrent (CompositeState) 2-144
isDynamic (ActionState) 2-172
isDynamic (SubactivityState) 2-175
isInstantiable (Subsystem) 2-186
isLeaf (GeneralizableElement)2-37
isLeaf (Operation)2-45
isLeaf (Reception)2-102
isList flag 2-214
isNavigable (AssociationEnd) 2-22
isQuery (BehavioralFeature) 2-25
isRegion (CompositeState) 2-144
isRoot (GeneralizableElement) 2-37
isRoot (Operation) 2-45
isRoot (Reception) 2-102
isSpecification (ElementOwnership) 2-34, 2-183
isSynch (ObjectFlowState) 2-174
italics
for abstract class 3-38
for abstract operation 3-46
Iterate Operation 6-26
iteration indicator 3-132

J

join (PseudostateKind) 2-91, 2-145
join of contro l3-146
jump type 2-332
junction 3-150
junction (PseudostateKind) 2-91, 2-145

K

keyword 3-11
kind (Parameter) 2-46
kind (PseudoState) 2-146

L

label 3-10
language (Expression) 2-87
layer, metamodel 2-4
library (Component) 2-19
lifeline 3-102, 3-108
line 3-68
dashed
for association class 3-78
for lifeline 3-109
solid
for actor-use cas e3-98
for association 3-68
for association class 3-78
for communication association 3-99
Link 2-99, 2-106, 2-110
link 3-84
creation B-14
destroying B-16
identity 2-256
LinkEnd 2-100, 2-106
linkEnd (Instance) 2-99
LinkObject 2-100, 2-106
list compartment 3-38
literal
of enumeration type 3-57
literal (Enumeration) 2-35
local (AssociationEnd) 2-24
local (keyword) 3-85
local (LinkEnd) 2-100
location (ExtensionPoint) 2-132
LocationReference 2-88

M

many 3-75
Mapping 2-88
mapping (Abstraction) 2-18
mapping of languages 2-202
mapping section 3-6
MappingExpression 2-88
marshalling 2-214
Message 2-119, 2-123
message 3-111, 3-130
message (Interaction) 2-118, 2-119

message label 3-131
message name 3-133
Message Sequence Chart notation 3-102
messaging examples B-19
metaclass 3-57
metaclass (Classifier) 2-28
meta-metamodel laye r2-5
metamodel (Model) 2-183, 2-184
metamodel laye r2-5
Method 2-40, 2-60
method 3-47
minus sign
  for private visibility 3-42
Missing Rolenames 6-14
Model 2-183, 2-187, 2-196
model 3-24
model laye r2-5
model management 3-16
Model Management Package 2-181
model organization 3-16
ModelElement 2-41, 2-61, 2-77, 2-80
multiobject 3-127
Multiplicity 2-89
multiplicity 3-75
  of association end 3-71
  of attribute 3-42
  of qualified association 3-76
  on dynamic concurrenc y3-169
multiplicity (AssociationEnd) 2-22
multiplicity (AssociationRole) 2-116
multiplicity (Attribute) 2-49
multiplicity (ClassifierRole) 2-116
MultiplicityRange 2-89
multi-way decision B-5
mustIsolate flag 2-234

N

Name 2-90
name 3-9
name (Association) 2-20
name (AssociationEnd) 2-23
name (BehavioralFeature) 2-25
name (Feature) 2-36
name (ModelElement) 2-41
name (Parameter) 2-46
name compartmen t3-38, 3-138
named compartment 3-39
Namespace 2-43, 2-62
namespace (ModelElement) 2-42
n-ary association 3-79
natural languag e2-10, 2-82
navigability 2-65, 3-72
navigating an association B-17
navigation arrow 3-73
Navigation from Association Classes 6-16
Navigation over Associations with Multiplicity Zero or One 6-14
Navigation through Qualified Associations 6-16
Navigation to Association Types 6-15
nested state 3-140
nesting
  for composition 3-81
new (Instance) 2-99
new (keyword) 3-115
new (Link) 2-100
Node 2-44, 2-63
node 3-173, 3-174
NodeInstance 2-101, 2-107
none (AggregationKind) 2-22, 2-86
notation section 3-5
note 3-13, 3-28
Notes section 2-10

O

Object 2-101, 2-107, 2-109
object 3-64, 3-124, 3-163
  lifeline 3-108
  playing role 3-125
Object Constraint Language 6-1, 6-2
object creation
  with attribute assignmen tB-7
object destruction B-8
object diagram 3-35
object flow 3-163
object in state 3-163
Object Message Sequence Chart notation 3-102
ObjectFlowState 2-173, 2-177, 2-179
Objects and Properties 6-12
occurrence
  jump 2-333
OCL 2-9, 2-82, 3-27
OCL - Legend 6-3
OCL (Language) 2-88
OCL expression 3-12
OCL Grammar 6-45
OCL Uses 6-3
OclAny 6-30, 6-31
OclExpression 6-31
OclType 6-29
Operation 2-44, 2-63, 2-71
operation 3-38, 3-44, 3-47
  pointer to 2-320
operation (CallEvent) 2-142
operation lookup 2-318
ordered (keyword) 3-71
ordered (OrderingKind) 2-22, 2-49, 2-90
ordering 3-42, 3-71
ordering (AssociationEnd) 2-22, 2-49
OrderingKind 2-90
out (ParameterDirectionKind) 2-46, 2-90
outgoing (StateVertex) 2-148
output
  available 2-229
output event ico n3-165
output pin 2-212
output signature 2-330
overlapping (Generalization) 2-40
overlapping (keyword) 3-87
ownedElement (Collaboration) 2-117
ownedElement (Namespace) 2-43
ownedInstance (Instance) 2-99
ownedLink (Instance) 2-99

owner (Feature)2-36
owner (Instance)2-99
owner (Link)2-99
ownerScope (Feature) 2-36
ownership of elements 2-192

P

Package 2-184, 2-187, 2-192
package 3-16
package (VisibilityKind) 2-23, 2-34, 2-36, 2-92
package structure of UML 2-6
Parameter 2-45, 2-63
parameter (AssociationEnd)2-24
parameter (BehavioralFeature)2-25
parameter (Event)2-144
parameter (keyword)3-84
parameter (LinkEnd)2-100
parameter (ObjectFlowState)2-174
parameter list 3-45
ParameterDirectionKind 2-90
parameterized class 3-52
parent (Generalization) 2-39
parentheses
  for argument lis t3-13
  for parameter list 3-44, 3-139, 3-145
participant (AssociationEnd) 2-23
participation (in a use case) 3-97, 3-99
Partition 2-174
partition (ActivityGraph) 2-173
passive clas s2-26
path 3-7, 3-61
  for association 3-68
path (symbol) 3-6, 3-7
pathname 3-61
Pathnames for Packages and Properties 6-17
Pattern 2-129
pattern 3-117
pentagon
  for signal receipt 3-165
  for signal sending 3-165
Permission 2-46
persistence (Association) 2-20
persistence (Attribute) 2-49
persistence (Classifier) 2-29
persistent (Instance) 2-99
pin valu e2-215
plus sign
  for containment tree 3-17
  for public visibility 3-42
postcondition (Constraint) 2-32
pound sign
  for protected visibili t 3-42
powertype 3-61
powertype (Classifier) 2-28
powertype (Generalization) 2-39
powertypeRange (Classifier) 2-28
Pre and Postcondition s6-6
Precedence Rules 6-10
precondition (Constraint) 2-32
predecessor 3-108, 3-131
predecessor (Message)2-119
Predefined Features on All Objects 6-18
Predefined OCL Type s6-29
presentation (ModelElement) 2-42
presentation options 3-5, 3-8
PresentationElement 2-47, 2-63, 2-71
Previous Values in Postconditions 6-21
Primitive 2-47, 2-63
priority of transition 2-163
private (keyword) 3-42
private (VisibilityKind) 2-23, 2-34, 2-36, 2-92
procedural sequence diagram 3-108
procedure execution
  status 2-216
ProcedureExpression 2-90
Process 1-8
process (Classifier) 2-28
Programming Languages 1-7
ProgrammingLanguageDataType 2-47
pronged rectangle
  for component 3-175
propagation semantics 2-66
Properties 6-12
  Association Ends and Navigation 6-13
  Attributes 6-12
  Operations 6-12
property 3-29
property string 3-29, 3-39
protected (keyword) 3-42
protected (VisibilityKind) 2-23, 2-34, 2-36, 2-92
protocol state machine 2-165
PseudoState 2-145, 2-151, 2-178
PseudostateKind 2-90
public (keyword) 3-42
public (VisibilityKind) 2-23, 2-34, 2-36, 2-92

Q

qualifier 2-66, 3-72, 3-76
qualifier (AssociationEn d 2-23
qualifierValue (LinkEnd) 2-100
query 3-45

R

range 3-75
read action 2-205
ready
  status 2-215
Real 6-31
Realization 2-17
realization
  of interface by classifier 3-51
realization element 3-19
realization relationship 3-49
realize (Abstraction) 2-18
receive
  direct 2-319
  request 2-317
receiver (Message) 2-119
receiver (Stimulus) 2-103
Reception 2-102, 2-107
reception (Signal) 2-103

rectangle
dog-eared
for note 3-13
pronged
for component 3-175
rounded ends
for action state 3-158
for state 3-138
for subactivity state 3-159
solid
for active class 3-128
for association class 3-78
for class 3-36
for object 3-64
for qualifier 3-76
stacked
for multiobject 3-127
tabbed
for package 3-16
thin
for activation lifeline 3-110
recurrence 3-132
reference to another package 3-36
referenceState (StubState) 2-148
referencing elements 2-46
refine (Abstraction) 2-18
refine (keyword) 3-91
Refinement 2-17, 2-73
refinement 3-91
refinement of state machine 2-166
Relationship 2-33, 2-48
remote procedure cal l2-315
reply 2-315
direct 2-319
representedClassifier (Collaboration)2-117
representedOperation (Collaboration) 2-117
request
performing 2-316
request object 2-214, 2-316
requiredTag (Stereotype) 2-78
resident (Component) 2-31
resident (ComponentInstance) 2-98
resident (NodeInstance) 2-101
resolution
OO traditional 2-330
responsibility (Comment) 2-30
return (ParameterDirectionKind) 2-46, 2-90
return type expressio n3-45
return value 3-132
ReturnAction 2-102, 2-108
Re-typing or Casing 6-10
right arrow
for special operation 3-13
role 3-15
rolename 3-72
run to completion 2-161

S

Scope 1-6
ScopeKind 2-91
segment descriptor 2-69
Select and Reject Operations 6-22
selecting a subset of objects B-11
Self 6-4
self (AssociationEnd)2-24
self (keyword)3-85
self (LinkEnd)2-100
Semantics 2-82, 2-192
semantics (Classifier) 2-29
semantics (Operation) 2-45
semantics of state machines 2-154
Semantics Package 2-192
semantics sectio n2-10, 3-5
semaphore 2-164
send (Usage) 2-51
SendAction 2-102, 2-108
sender (Message)2-119
sender (Stimulus)2-103
sending a signal B-23
Sequence 6-43
sequence diagram 3-102, 3-106
sequence expressio n3-131
sequence number 3-115, 3-131
sequential (CallConcurrencyKind) 2-45, 2-87
sequential substate 3-140
Set 6-38
shallowHistory (PseudostateKind) 2-91, 2-145, 2-157
Shorthand for Collect 6-25
Signal 2-102, 2-108, 2-110
signal 3-143
declaration 3-143
signal (Reception) 2-102
signal (SignalEvent) 2-146
signal receipt ico n3-165
signal sending icon 3-165
SignalEvent 2-146
signalflow (ObjectFlowState) 2-174
signature 3-132
action 2-212
simple object creation B-6
simple transition 3-145
SimpleState 2-146
slash
for action expression 3-145
for derived element 3-93
for predecessor 3-131
for rol e3-124
slot (Instance) 2-99
sorted (keyword) 3-71
sorted (OrderingKind) 2-22, 2-49, 2-90
source (Transition) 2-150
source state 3-146
specialization (GeneralizableElement) 2-38
specification (AssociationEnd) 2-23
specification (Method) 2-41
specification (Reception) 2-102
specification element 3-19
specification level collaboration 3-115, 3-116
specifiedEnd (Classifier) 2-28
square brackets
for attribute multiplicity 3-42, 3-43
for condition clause 3-132

for guard condition 3-139, 3-145
for selection 3-13
for state 3-65, 3-163
standard elements sectio n2-10
star
for iteration indicator 3-132
for multiplicity 3-75
State 2-146, 2-155
state 3-137
composite 3-154
of object 3-65
state machine
request handling 2-317
state machine refinement 2-166
State Machines Package 2-140
statechart 2-169
statechart diagram 3-135
StateMachine 2-147, 2-152, 2-161
semantics 2-154
StateVertex 2-148
Stereotype 2-78, 2-81
stereotype 3-31, 3-57
class 3-38
object 3-65
stereotype (ModelElement) 2-77
stereotypeConstraint (Stereotype) 2-78
Stereotypes iii-xxix, 6-2
stick arrowhead
for control flow 3-112
stick man figure
for use cas e3-97
Stimulus 2-103, 2-108, 2-110
stimulus 3-111, 3-130
String 2-91, 6-34
string 3-7, 3-8, 3-10
StructuralFeature 2-48, 2-63
stub (Package) 2-185
stub (stereotype) 3-17
stub state 3-152
stubbed transition 3-148
StubState 2-148, 2-164
style guidelines 3-5
subactivity state 3-159
SubactivityState 2-174, 2-175, 2-178, 2-180
submachine (SubactivityState) 2-175
submachine (SubmachineState) 2-149
submachine state 3-152
SubmachineState 2-148, 2-153, 2-158
subordinate use case 2-137
substate 3-140
Subsystem 2-103, 2-109, 2-186, 2-190, 2-194
subsystem 3-19, 3-21
subtyping and state machine 2-167
subvertex (CompositeState) 2-143
superordinate use cas e2-137
supplier (Dependency)2-33
supplierDependency (ModelElement) 2-42
suppressed element 3-39
swimlane 3-161
synch state 3-154, 3-168
synchronization 3-154
synchronization ba r3-146
synchronization fork and join 2-163
synchronous request
state machine 2-319
SynchState 2-149, 2-153, 2-163
system boundary 3-94
systemModel (Model) 2-184
systemModel (stereotype) 3-24

T

tabbed rectangle
for package 3-16
table (Component) 2-19
tagged value 3-29
TaggedValue 2-78, 2-81, 2-82
taggedValue (ModelElement) 2-77
target (Transition) 2-150
targetScope (AssociationEnd) 2-23
targetScope (Attribute) 2-49
taxonomic relationship 2-38, 3-86
template 2-41, 2-42, 2-61, 2-72, 3-52
collaboration 2-129
TemplateParameter 2-50
templateParameter (ModelElement)2-42
TerminateAction 2-103, 2-109
thread (Classifier) 2-28
tiling (a state) 3-140
time dimension 3-102
time even t3-143
time expressio n3-113, 3-145
time in t e r 3-103
TimeEvent 2-149
TimeExpression 2-91
timing constraint 3-103, 3-113
Tools 1-7
tools, interactive 3-7
top (StateMachine) 2-147
topLevel (Package) 2-185
topLevel (stereotype) 3-17
Trace 2-17, 2-64, 2-73
trace (Abstraction) 2-18, 2-197
trace (keyword) 3-91
transient (Instance) 2-99
transient (keyword) 3-115
transient (Link) 2-100
Transition 2-149, 2-153, 2-158, 2-180
execution 2-160
firing rules 2-163
transition 3-145
chain 3-150
complex 3-146, 3-147
constraint 3-145
execution of request s2-319
name 3-113
simple 3-145
string 3-145
stubbed 3-148
time 3-113
to composite state 3-147
transition (StateMachine) 2-148
triangle

for generalization 3-86
for realizatio n3-49
trigger (Transition) 2-150
true (Boolean) 2-86
two-dimensional symbols 3-6
Type 2-64
type
and implementation class 3-49
type (Attribute) 2-49
type (Class) 2-26, 2-27, 3-173, C-3, C-7
type (ClassifierInState) 2-173
type (ObjectFlowState) 2-174
type (Parameter) 2-46
Type Conformance 6-8
TypeExpression 2-92
type-instance correspondenc e3-14
Types 6-8

U

UML
relationship to 2-201
UML - defined 1-1
UML and other modeling language s1-8
UML Extension for Business Modelin g4-9
UML Extension for Objectory Process for Software Engineering 4-1
UML features 1-9
undefined semantics 2-201
Undefined Values 6-11
underlining
for class sco p 3-43, 3-45
for instances 3-14
for object 3-64, 3-125
UninterpretedAction 2-103
unlimited multiplicity 3-75
UnlimitedInteger 2-92
unmarshalling 2-214
unordered (keyword) 3-71
unordered (OrderingKind) 2-22, 2-49, 2-90
Usage 2-50, 2-64, 2-73
usage dependency 3-91
use (keyword) 3-91
use case 3-96
use case diagram 3-94
use case relationship 3-97
Use Cases Package 2-129
UseCase 2-132, 2-134, 2-136
description 2-137
instance 2-137
UseCaseInstance 2-133, 2-135
user object layer 2-5
Using Pathnames for Packages and Properties 6-17
utility (Classifier) 2-28
utility (keyword) 3-56

V

value (AttributeLink) 2-97
variable manipulation actions 2-259
visibility
of association 3-73
of attribute 3-42
of operation 3-45
of package elemen t3-17
visibility (ElementImport) 2-183
visibility (ElementOwnership) 2-34
visibility (ElementResidence) 2-34
visibility (Feature) 2-36
VisibilityKind 2-92

W

waiting
status 2-215
well-formedness rules sectio n2-9
when (keyword) 3-143
when (TimeEvent) 2-149
write action 2-205
writing attributes B-9, B-10

X

X
for destruction 3-109
xor (Association)2-20
xor association 3-69

UML, v1.5
Reference Sheet

UML 1.5 is a combination of the UML 1.4 specification and the Action Semantics for the UML specification. UML 1.5 also incorporates resolutions from UML 1.4.1 (see ad/02-06-21) and corrects an internalization issue. OMG documents used to create this version include:

UML Action Semantics:

- Revised submission document: ad/01-08-04
- FTF Final Adopted Specification: ptc/02-01-09
- FTF Report: ptc/02-09-01
- Convenience document: ptc/02-09-02

UML 1.4.1:

- Convenience document: ad/02-06-22
- RTF Report: ad/02-06-18

UML/ISO:

Additionally, this formal specification incorporates editorial changes that were recommended by ISO. These changes are in this font and are marked with change bars.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:eff="http://efficient.citi.tudor.lu">
  - <xsd:simpleType name="Type_Statut">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="NonTraité" />
        <xsd:enumeration value="Conforme" />
        <xsd:enumeration value="Accord" />
        <xsd:enumeration value="Rejet" />
        <xsd:enumeration value="Refus" />
        <xsd:enumeration value="EnCours" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:element name="Demande_de_consultation__">
    - <xsd:complexType>
      - <xsd:sequence>
          <xsd:element name="Valider_Proposition" type="Valider_Proposition"
            minOccurs="1" maxOccurs="1" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  - <xsd:complexType name="Valider_Proposition">
    - <xsd:sequence>
        <xsd:element name="Nom_Proposition" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="IndiceProposition" type="xsd:integer" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="date_emission" type="xsd:date" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="echeance" type="xsd:date" minOccurs="0" maxOccurs="1" />
        <xsd:element name="Statut" type="Type_Statut" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="Processus" type="Processus" minOccurs="1"
          maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>
  - <xsd:complexType name="Document">
    - <xsd:sequence>
        <xsd:element name="ID_Document" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="Nom_Doc" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="Description" type="xsd:string" minOccurs="0"
          maxOccurs="1" />
        <xsd:element name="Reference" type="xsd:string" minOccurs="1"
          maxOccurs="unbounded" />
        <xsd:element name="Auteur" type="xsd:string" minOccurs="0"
          maxOccurs="unbounded" />
        <xsd:element name="Editeur" type="xsd:string" minOccurs="0"
          maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  - <xsd:complexType name="Objet_Projet">
    - <xsd:sequence>
        <xsd:element name="T_Objet" type="xsd:string" minOccurs="0" maxOccurs="1" />
        <xsd:element name="Root" type="Root" minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>
  - <xsd:complexType name="Root">
    - <xsd:sequence>
        <xsd:element name="GlobalID" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="NomProjet" type="xsd:string" minOccurs="1"
```

```
    maxOccurs="1" />
      <xsd:element name="Associe_Document" type="Document" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Processus">
  - <xsd:sequence>
      <xsd:element name="Objet_Projet" type="Objet_Projet" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xpdl:Package xmlns:xpdl="http://www.wfmc.org/2002/XPDL1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:vx="http://www.an.danet.de/2002/XPDL-Extensions1.0"
    xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
    http://www.wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd" Id="IFC3-4-
    DemandeValidation-package" Name="IFC3-4-DemandeValidation-package">
  - <xpdl:PackageHeader>
      <xpdl:XPDLVersion>1.0</xpdl:XPDLVersion>
      <xpdl:Vendor>CRP Henri Tudor</xpdl:Vendor>
      <xpdl:Created>May 31, 2004 9:29:20 PM CEST</xpdl:Created>
    </xpdl:PackageHeader>
  - <xpdl:RedefinableHeader PublicationStatus="RELEASED">
      <xpdl:Author />
    </xpdl:RedefinableHeader>
    <xpdl:ConformanceClass GraphConformance="NON_BLOCKED" />
  - <xpdl:WorkflowProcesses>
    - <xpdl:WorkflowProcess Id="IFC3-4-DemandeValidation" Name="IFC3-4-
        DemandeValidation" AccessLevel="PUBLIC">
        <xpdl:ProcessHeader />
      - <xpdl:RedefinableHeader PublicationStatus="RELEASED">
          <xpdl:Author />
        </xpdl:RedefinableHeader>
      - <xpdl:DataFields>
        - <xpdl:DataField Id="procInstId" Name="procInstId" IsArray="FALSE">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
          </xpdl:DataField>
        - <xpdl:DataField Id="initialSender" Name="initialSender" IsArray="FALSE">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
          </xpdl:DataField>
        - <xpdl:DataField Id="messageRootName" Name="messageRootName"
            IsArray="FALSE">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
          </xpdl:DataField>
        - <xpdl:DataField Id="Demande_de_validation_ Conforme_message"
            Name="Demande_de_validation_ Conforme_message" IsArray="FALSE">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
          </xpdl:DataField>
        - <xpdl:DataField Id="Demande_de_validation_ Conforme_recipient"
            Name="Demande_de_validation_ Conforme_recipient" IsArray="FALSE">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
            <xpdl:InitialValue>Gestionnaire du modele</xpdl:InitialValue>
          </xpdl:DataField>
        - <xpdl:DataField Id="Demande_de_validation_ Conforme_sender"
            Name="Demande_de_validation_ Conforme_sender" IsArray="FALSE">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
            <xpdl:InitialValue>Modele</xpdl:InitialValue>
          </xpdl:DataField>
        - <xpdl:DataField Id="Demande_de_validation_ Conforme_schema"
            Name="Demande_de_validation_ Conforme_schema" IsArray="FALSE">
          - <xpdl:DataType>
```

```
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Demande_de_validation_
        Conforme.xsd</xpdl:InitialValue>
    </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_ Conforme_constraints"
    Name="Demande_de_validation_ Conforme_constraints"
    IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Demande_de_validation_ Conforme</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_ Conforme_errorText"
    Name="Demande_de_validation_ Conforme_errorText" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_NonTraite_message"
    Name="Demande_de_validation_NonTraite_message" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_NonTraite_recipient"
    Name="Demande_de_validation_NonTraite_recipient" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_NonTraite_sender"
    Name="Demande_de_validation_NonTraite_sender" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Ingenieur stabilite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_NonTraite_schema"
    Name="Demande_de_validation_NonTraite_schema" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>

    <xpdl:InitialValue>Demande_de_validation_NonTraite.xsd</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_NonTraite_constraints"
    Name="Demande_de_validation_NonTraite_constraints"
    IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Demande_de_validation_NonTraite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_NonTraite_errorText"
    Name="Demande_de_validation_NonTraite_errorText" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Demande_de_consultation___message"
    Name="Demande_de_consultation___message" IsArray="FALSE">
```

```
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_consultation___recipient"
      Name="Demande_de_consultation___recipient" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Modele</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_consultation___sender"
      Name="Demande_de_consultation___sender" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Gestionnaire du modele</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_consultation___schema"
      Name="Demande_de_consultation___schema" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Demande_de_consultation__.xsd</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_consultation___constraints"
      Name="Demande_de_consultation___constraints" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Demande_de_consultation__</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_consultation___errorText"
      Name="Demande_de_consultation___errorText" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Rejet_demande_de_validation___message"
      Name="Rejet_demande_de_validation___message" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Rejet_demande_de_validation___recipient"
      Name="Rejet_demande_de_validation___recipient" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Ingenieur stabilite</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Rejet_demande_de_validation___sender"
      Name="Rejet_demande_de_validation___sender" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Modele</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Rejet_demande_de_validation___schema"
      Name="Rejet_demande_de_validation___schema" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
```

```
      <xpdl:InitialValue>Rejet_demande_de_validation__.xsd</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Rejet_demande_de_validation___constraints"
      Name="Rejet_demande_de_validation___constraints" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Rejet_demande_de_validation__</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Rejet_demande_de_validation___errorText"
      Name="Rejet_demande_de_validation___errorText" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_Proposition_accessibles_message"
      Name="Objets_Proposition_accessibles_message" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_Proposition_accessibles_recipient"
      Name="Objets_Proposition_accessibles_recipient" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Modele</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_Proposition_accessibles_sender"
      Name="Objets_Proposition_accessibles_sender" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Modele</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_Proposition_accessibles_schema"
      Name="Objets_Proposition_accessibles_schema" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Objets_Proposition_accessibles.xsd</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_Proposition_accessibles_constraints"
      Name="Objets_Proposition_accessibles_constraints" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
      <xpdl:InitialValue>Objets_Proposition_accessibles</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_Proposition_accessibles_errorText"
      Name="Objets_Proposition_accessibles_errorText" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_propositions_transfert_message"
      Name="Objets_propositions_transfert_message" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  - <xpdl:DataField Id="Objets_propositions_transfert_recipient"
```

```
Name="Objets_propositions_transfert_recipient" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Gestionnaire du modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Objets_propositions_transfert_sender"
    Name="Objets_propositions_transfert_sender" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Objets_propositions_transfert_schema"
    Name="Objets_propositions_transfert_schema" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Objets_propositions_transfert.xsd</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Objets_propositions_transfert_constraints"
    Name="Objets_propositions_transfert_constraints" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Objets_propositions_transfert</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Objets_propositions_transfert_errorText"
    Name="Objets_propositions_transfert_errorText" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_proposition_nonTraite_message"
    Name="Refus_proposition_nonTraite_message" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_proposition_nonTraite_recipient"
    Name="Refus_proposition_nonTraite_recipient" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_proposition_nonTraite_sender"
    Name="Refus_proposition_nonTraite_sender" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Gestionnaire du modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_proposition_nonTraite_schema"
    Name="Refus_proposition_nonTraite_schema" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Refus_proposition_nonTraite.xsd</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_proposition_nonTraite_constraints"
    Name="Refus_proposition_nonTraite_constraints" IsArray="FALSE">
  - <xpdl:DataType>
```

```
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Refus_proposition_nonTraite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_proposition_nonTraite_errorText"
    Name="Refus_proposition_nonTraite_errorText" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_de_proposition_transfert_message"
    Name="Refus_de_proposition_transfert_message" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_de_proposition_transfert_recipient"
    Name="Refus_de_proposition_transfert_recipient" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Ingenieur stabilite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_de_proposition_transfert_sender"
    Name="Refus_de_proposition_transfert_sender" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_de_proposition_transfert_schema"
    Name="Refus_de_proposition_transfert_schema" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Refus_de_proposition_transfert.xsd</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_de_proposition_transfert_constraints"
    Name="Refus_de_proposition_transfert_constraints" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Refus_de_proposition_transfert</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Refus_de_proposition_transfert_errorText"
    Name="Refus_de_proposition_transfert_errorText" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_nonTraite_message"
    Name="Accord_proposition_nonTraite_message" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_nonTraite_recipient"
    Name="Accord_proposition_nonTraite_recipient" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Modele</xpdl:InitialValue>
  </xpdl:DataField>
```

```
- <xpdl:DataField Id="Accord_proposition_nonTraite_sender"
    Name="Accord_proposition_nonTraite_sender" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Gestionnaire du modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_nonTraite_schema"
    Name="Accord_proposition_nonTraite_schema" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Accord_proposition_nonTraite.xsd</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_nonTraite_constraints"
    Name="Accord_proposition_nonTraite_constraints" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Accord_proposition_nonTraite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_nonTraite_errorText"
    Name="Accord_proposition_nonTraite_errorText" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_traite_message"
    Name="Accord_proposition_traite_message" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_traite_recipient"
    Name="Accord_proposition_traite_recipient" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Ingenieur stabilite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_traite_sender"
    Name="Accord_proposition_traite_sender" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Modele</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_traite_schema"
    Name="Accord_proposition_traite_schema" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Accord_proposition_traite.xsd</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_traite_constraints"
    Name="Accord_proposition_traite_constraints" IsArray="FALSE">
  - <xpdl:DataType>
      <xpdl:BasicType Type="STRING" />
    </xpdl:DataType>
    <xpdl:InitialValue>Accord_proposition_traite</xpdl:InitialValue>
  </xpdl:DataField>
- <xpdl:DataField Id="Accord_proposition_traite_errorText"
    Name="Accord_proposition_traite_errorText" IsArray="FALSE">
```

```
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
</xpdl:DataField>
- <xpdl:DataField Id="Objets_proposition_modification_message"
  Name="Objets_proposition_modification_message" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
</xpdl:DataField>
- <xpdl:DataField Id="Objets_proposition_modification_recipient"
  Name="Objets_proposition_modification_recipient" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
  <xpdl:InitialValue>Modele</xpdl:InitialValue>
</xpdl:DataField>
- <xpdl:DataField Id="Objets_proposition_modification_sender"
  Name="Objets_proposition_modification_sender" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
  <xpdl:InitialValue>Ingenieur stabilite</xpdl:InitialValue>
</xpdl:DataField>
- <xpdl:DataField Id="Objets_proposition_modification_schema"
  Name="Objets_proposition_modification_schema" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>

    <xpdl:InitialValue>Objets_proposition_modification.xsd</xpdl:InitialValue>
</xpdl:DataField>
- <xpdl:DataField Id="Objets_proposition_modification_constraints"
  Name="Objets_proposition_modification_constraints" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
  <xpdl:InitialValue>Objets_proposition_modification</xpdl:InitialValue>
</xpdl:DataField>
- <xpdl:DataField Id="Objets_proposition_modification_errorText"
  Name="Objets_proposition_modification_errorText" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
</xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_Modification_message"
  Name="Demande_de_validation_Modification_message" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
</xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_Modification_recipient"
  Name="Demande_de_validation_Modification_recipient"
  IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
  </xpdl:DataType>
  <xpdl:InitialValue>Gestionnaire du modele</xpdl:InitialValue>
</xpdl:DataField>
- <xpdl:DataField Id="Demande_de_validation_Modification_sender"
  Name="Demande_de_validation_Modification_sender" IsArray="FALSE">
- <xpdl:DataType>
    <xpdl:BasicType Type="STRING" />
```

```
      </xpdl:DataType>
      <xpdl:InitialValue>Modele</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_validation_Modification_schema"
      Name="Demande_de_validation_Modification_schema" IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>

        <xpdl:InitialValue>Demande_de_validation_Modification.xsd</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_validation_Modification_constraints"
      Name="Demande_de_validation_Modification_constraints"
      IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>

        <xpdl:InitialValue>Demande_de_validation_Modification</xpdl:InitialValue>
    </xpdl:DataField>
  - <xpdl:DataField Id="Demande_de_validation_Modification_errorText"
      Name="Demande_de_validation_Modification_errorText"
      IsArray="FALSE">
    - <xpdl:DataType>
        <xpdl:BasicType Type="STRING" />
      </xpdl:DataType>
    </xpdl:DataField>
  </xpdl:DataFields>
- <xpdl:Participants>
  - <xpdl:Participant Id="efficient" Name="efficient">
      <xpdl:ParticipantType Type="ROLE" />
    </xpdl:Participant>
  </xpdl:Participants>
- <xpdl:Applications>
  - <xpdl:Application Id="MessageInformer">
    - <xpdl:FormalParameters>
      - <xpdl:FormalParameter Id="messageName" Mode="IN">
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      - <xpdl:FormalParameter Id="schema" Mode="IN">
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      - <xpdl:FormalParameter Id="recipient" Mode="INOUT">
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      - <xpdl:FormalParameter Id="sender" Mode="INOUT">
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      - <xpdl:FormalParameter Id="message" Mode="INOUT">
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      - <xpdl:FormalParameter Id="messageRootName" Mode="INOUT">
```

```
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      </xpdl:FormalParameters>
    - <xpdl:ExtendedAttributes>
      - <xpdl:ExtendedAttribute Name="Implementation">
          <vx:ToolAgent Class="efficient.animator.MessageInformerTool"
            Execution="SYNCHR" XMLParameterMode="USE_W3C_DOM" />
        </xpdl:ExtendedAttribute>
      </xpdl:ExtendedAttributes>
    </xpdl:Application>
  - <xpdl:Application Id="MessageHandler">
    - <xpdl:FormalParameters>
      - <xpdl:FormalParameter Id="messageName" Mode="IN">
        - <xpdl:DataType>
            <xpdl:BasicType Type="STRING" />
          </xpdl:DataType>
        </xpdl:FormalParameter>
      </xpdl:FormalParameters>
    - <xpdl:ExtendedAttributes>
      - <xpdl:ExtendedAttribute Name="Implementation">
          <vx:ToolAgent Class="efficient.animator.MessageHandlerTool"
            Execution="SYNCHR" XMLParameterMode="USE_W3C_DOM" />
        </xpdl:ExtendedAttribute>
      </xpdl:ExtendedAttributes>
    </xpdl:Application>
  </xpdl:Applications>
- <xpdl:Activities>
  - <xpdl:Activity Id="Introduire__une__demande__de__validation"
      Name="Introduire__une__demande__de__validation">
      <xpdl:Description>Communication</xpdl:Description>
    - <xpdl:Implementation>
        <xpdl:No />
      </xpdl:Implementation>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="initial" Name="initial">
      <xpdl:Description>initial state</xpdl:Description>
      <xpdl:Route />
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        - <xpdl:Split Type="XOR">
          - <xpdl:TransitionRefs>
              <xpdl:TransitionRef
                Id="_609020c_1084028389640_25006_194" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="_609020c_1084041644169_565354_713"
      Name="_609020c_1084041644169_565354_713">
      <xpdl:Description>Business Activity</xpdl:Description>
    - <xpdl:Implementation>
        <xpdl:No />
      </xpdl:Implementation>
```

```
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="_609020c_1084041725085_928465_735"
      Name="_609020c_1084041725085_928465_735">
      <xpdl:Description>final state</xpdl:Description>
      <xpdl:Route />
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="_609020c_1084044545571_673193_895"
      Name="_609020c_1084044545571_673193_895">
      <xpdl:Description>Human Choice</xpdl:Description>
    - <xpdl:Implementation>
        <xpdl:No />
      </xpdl:Implementation>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="_609020c_1084044620989_805516_917"
      Name="_609020c_1084044620989_805516_917">
      <xpdl:Description>final state</xpdl:Description>
      <xpdl:Route />
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="_609020c_1084045476920_289436_1024"
      Name="_609020c_1084045476920_289436_1024">
      <xpdl:Description>Human Choice</xpdl:Description>
    - <xpdl:Implementation>
        <xpdl:No />
      </xpdl:Implementation>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="_609020c_1084393175311_84032_236"
      Name="_609020c_1084393175311_84032_236">
      <xpdl:Route />
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        - <xpdl:Split Type="AND">
          - <xpdl:TransitionRefs>
              <xpdl:TransitionRef
                Id="_609020c_1084395048375_782405_354" />
              <xpdl:TransitionRef
                Id="_609020c_1084393354168_342793_250" />
```

```
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084393175311_84032_236_route"
    Name="_609020c_1084393175311_84032_236_route">
    <xpdl:Description>Route state needed by a deferred
      choice</xpdl:Description>
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084607106379_63506_81_reception" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084393380727_718345_260"
    Name="_609020c_1084393380727_718345_260">
    <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084395007316_693386_319"
    Name="_609020c_1084395007316_693386_319">
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="AND">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084395054834_508401_355" />
            <xpdl:TransitionRef
              Id="_609020c_1084395011361_464636_321" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084395007316_693386_319_route"
    Name="_609020c_1084395007316_693386_319_route">
    <xpdl:Description>Route state needed by a deferred
      choice</xpdl:Description>
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084607106379_63506_81_reception" />
```

```
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084395041495_332562_352"
    Name="_609020c_1084395041495_332562_352">
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084395059430_735720_365" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084045522816_3898_1046"
    Name="_609020c_1084045522816_3898_1046">
    <xpdl:Description>final state</xpdl:Description>
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084030518341_700532_323"
    Name="_609020c_1084030518341_700532_323">
    <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084036605794_205207_629"
    Name="_609020c_1084036605794_205207_629">
    <xpdl:Description>Human Choice</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084041507032_959447_679"
    Name="_609020c_1084041507032_959447_679">
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="AND">
        - <xpdl:TransitionRefs>
```

```
            <xpdl:TransitionRef
              Id="_609020c_1084041597792_550882_703" />
            <xpdl:TransitionRef
              Id="_609020c_1084392623588_670060_181" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084041507032_959447_679_route"
    Name="_609020c_1084041507032_959447_679_route">
    <xpdl:Description>Route state needed by a deferred
      choice</xpdl:Description>
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084044047575_65599_835_reception" />
            <xpdl:TransitionRef
              Id="_609020c_1084045286987_106270_982_reception" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084606899792_245429_44"
    Name="_609020c_1084606899792_245429_44">
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084606929625_367812_67" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Donner__acces__a__la__proposition"
    Name="Donner__acces__a__la__proposition">
    <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084031028995_851646_375"
    Name="_609020c_1084031028995_851646_375">
    <xpdl:Description>Human Choice</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
```

```
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084032346650_737054_427"
    Name="_609020c_1084032346650_737054_427">
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="AND">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084035785865_409106_551" />
            <xpdl:TransitionRef
              Id="_609020c_1084032564823_255398_453" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084032346650_737054_427_route"
    Name="_609020c_1084032346650_737054_427_route">
    <xpdl:Description>Route state needed by a deferred
      choice</xpdl:Description>
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084044106039_323595_837_reception" />
            <xpdl:TransitionRef
              Id="_609020c_1084036244815_24097_585_reception" />
            <xpdl:TransitionRef
              Id="_609020c_1084606905500_321229_46_reception" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084036217035_542454_573"
    Name="_609020c_1084036217035_542454_573">
    <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="AND">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084041444892_875518_649" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084036515264_659807_605"
    Name="_609020c_1084036515264_659807_605">
    <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
```

```
- <xpdl:TransitionRestrictions>
  - <xpdl:TransitionRestriction>
      <xpdl:Join Type="XOR" />
    </xpdl:TransitionRestriction>
  </xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="Transmettre__refus__proposition"
  Name="Transmettre__refus__proposition">
  <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084045131443_643956_958"
  Name="_609020c_1084045131443_643956_958">
  <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084032548901_789479_450"
  Name="_609020c_1084032548901_789479_450">
  <xpdl:Route />
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="AND">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084032889781_516914_497" />
            <xpdl:TransitionRef
              Id="_609020c_1084606917487_485034_47" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="_609020c_1084395793556_316949_375"
  Name="_609020c_1084395793556_316949_375">
  <xpdl:Description>Communication</xpdl:Description>
  - <xpdl:Implementation>
      <xpdl:No />
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="Demande_de_validation_ Conforme_information"
  Name="Demande_de_validation_ Conforme_information">
  - <xpdl:Implementation>
```

```
    - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>'Demande_de_validation_
            Conforme'</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
            Conforme_schema</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
            Conforme_recipient</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
            Conforme_sender</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
            Conforme_message</xpdl:ActualParameter>

          <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Demande_de_validation_ Conforme_reception"
    Name="Demande_de_validation_ Conforme_reception">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>'Demande_de_validation_
            Conforme'</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Demande_de_validation_ Conforme_check"
    Name="Demande_de_validation_ Conforme_check">
  - <xpdl:Implementation>
    - <xpdl:SubFlow Id="_checker">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
            Conforme_message</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
            Conforme_recipient</xpdl:ActualParameter>
          <xpdl:ActualParameter>Demande_de_validation_
```

```
          Conforme_sender</xpdl:ActualParameter>
            <xpdl:ActualParameter>Demande_de_validation_
              Conforme_schema</xpdl:ActualParameter>
            <xpdl:ActualParameter>Demande_de_validation_
              Conforme_constraints</xpdl:ActualParameter>
            <xpdl:ActualParameter>Demande_de_validation_
              Conforme_errorText</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:SubFlow>
      </xpdl:Implementation>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="AND" />
        - <xpdl:Split Type="XOR">
          - <xpdl:TransitionRefs>
              <xpdl:TransitionRef
                Id="_609020c_1084606905500_321229_46_ok" />
              <xpdl:TransitionRef
                Id="_609020c_1084606905500_321229_46_nok" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Demande_de_validation_NonTraite_information"
      Name="Demande_de_validation_NonTraite_information">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
        - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Demande_de_validation_NonTraite'</xpdl:ActualParamet

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_schema</xpdl:Actua

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_recipient</xpdl:Actu

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_sender</xpdl:Actual

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_message</xpdl:Actu

            <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
    - <xpdl:StartMode>
        <xpdl:Automatic />
      </xpdl:StartMode>
    - <xpdl:FinishMode>
        <xpdl:Automatic />
      </xpdl:FinishMode>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Demande_de_validation_NonTraite_reception"
      Name="Demande_de_validation_NonTraite_reception">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
        - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Demande_de_validation_NonTraite'</xpdl:ActualParamet
```

```
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Manual />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Demande_de_validation_NonTraite_check"
    Name="Demande_de_validation_NonTraite_check">
  - <xpdl:Implementation>
    - <xpdl:SubFlow Id="_checker">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_message</xpdl:Actu

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_recipient</xpdl:Actu

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_sender</xpdl:ActualI

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_schema</xpdl:Actua

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_constraints</xpdl:A

            <xpdl:ActualParameter>Demande_de_validation_NonTraite_errorText</xpdl:Actu
        </xpdl:ActualParameters>
      </xpdl:SubFlow>
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084031153875_833858_387_ok" />
            <xpdl:TransitionRef
              Id="_609020c_1084031153875_833858_387_nok" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Demande_de_consultation___information"
    Name="Demande_de_consultation___information">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
      - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Demande_de_consultation__'</xpdl:ActualParameter>

            <xpdl:ActualParameter>Demande_de_consultation___schema</xpdl:ActualParam

            <xpdl:ActualParameter>Demande_de_consultation___recipient</xpdl:ActualPara
```

```
            <xpdl:ActualParameter>Demande_de_consultation___sender</xpdl:ActualParameter>
                <xpdl:ActualParameter>Demande_de_consultation___message</xpdl:ActualPara
                <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
            </xpdl:ActualParameters>
          </xpdl:Tool>
        </xpdl:Implementation>
      - <xpdl:StartMode>
          <xpdl:Automatic />
        </xpdl:StartMode>
      - <xpdl:FinishMode>
          <xpdl:Automatic />
        </xpdl:FinishMode>
      - <xpdl:TransitionRestrictions>
        - <xpdl:TransitionRestriction>
            <xpdl:Join Type="XOR" />
          </xpdl:TransitionRestriction>
        </xpdl:TransitionRestrictions>
      </xpdl:Activity>
    - <xpdl:Activity Id="Demande_de_consultation___reception"
        Name="Demande_de_consultation___reception">
      - <xpdl:Implementation>
        - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
          - <xpdl:ActualParameters>
                <xpdl:ActualParameter>'Demande_de_consultation__'</xpdl:ActualParameter>
            </xpdl:ActualParameters>
          </xpdl:Tool>
        </xpdl:Implementation>
      - <xpdl:StartMode>
          <xpdl:Manual />
        </xpdl:StartMode>
      - <xpdl:FinishMode>
          <xpdl:Automatic />
        </xpdl:FinishMode>
      - <xpdl:TransitionRestrictions>
        - <xpdl:TransitionRestriction>
            <xpdl:Join Type="XOR" />
          </xpdl:TransitionRestriction>
        </xpdl:TransitionRestrictions>
      </xpdl:Activity>
    - <xpdl:Activity Id="Demande_de_consultation___check"
        Name="Demande_de_consultation___check">
      - <xpdl:Implementation>
        - <xpdl:SubFlow Id="_checker">
          - <xpdl:ActualParameters>
              <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
              <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>
                <xpdl:ActualParameter>Demande_de_consultation___message</xpdl:ActualPara
                <xpdl:ActualParameter>Demande_de_consultation___recipient</xpdl:ActualPara
                <xpdl:ActualParameter>Demande_de_consultation___sender</xpdl:ActualParame
                <xpdl:ActualParameter>Demande_de_consultation___schema</xpdl:ActualParam
                <xpdl:ActualParameter>Demande_de_consultation___constraints</xpdl:ActualPa
                <xpdl:ActualParameter>Demande_de_consultation___errorText</xpdl:ActualPara
            </xpdl:ActualParameters>
          </xpdl:SubFlow>
```

```
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084036222333_311314_575_ok" />
            <xpdl:TransitionRef
              Id="_609020c_1084036222333_311314_575_nok" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Rejet_demande_de_validation___information"
    Name="Rejet_demande_de_validation___information">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
      - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Rejet_demande_de_validation__'</xpdl:ActualParameter

            <xpdl:ActualParameter>Rejet_demande_de_validation___schema</xpdl:ActualPa

            <xpdl:ActualParameter>Rejet_demande_de_validation___recipient</xpdl:ActualF

            <xpdl:ActualParameter>Rejet_demande_de_validation___sender</xpdl:ActualPar

            <xpdl:ActualParameter>Rejet_demande_de_validation___message</xpdl:ActualF

            <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Rejet_demande_de_validation___reception"
    Name="Rejet_demande_de_validation___reception">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
      - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Rejet_demande_de_validation__'</xpdl:ActualParameter
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
```

```
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Rejet_demande_de_validation___check"
      Name="Rejet_demande_de_validation___check">
    - <xpdl:Implementation>
      - <xpdl:SubFlow Id="_checker">
        - <xpdl:ActualParameters>
            <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
            <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

              <xpdl:ActualParameter>Rejet_demande_de_validation___message</xpdl:ActualP

              <xpdl:ActualParameter>Rejet_demande_de_validation___recipient</xpdl:ActualP

              <xpdl:ActualParameter>Rejet_demande_de_validation___sender</xpdl:ActualPar

              <xpdl:ActualParameter>Rejet_demande_de_validation___schema</xpdl:ActualPa

              <xpdl:ActualParameter>Rejet_demande_de_validation___constraints</xpdl:Actu

              <xpdl:ActualParameter>Rejet_demande_de_validation___errorText</xpdl:Actual
          </xpdl:ActualParameters>
        </xpdl:SubFlow>
      </xpdl:Implementation>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="AND" />
        - <xpdl:Split Type="XOR">
          - <xpdl:TransitionRefs>
              <xpdl:TransitionRef
                Id="_609020c_1084044106039_323595_837_ok" />
              <xpdl:TransitionRef
                Id="_609020c_1084044106039_323595_837_nok" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Objets_Proposition_accessibles_information"
      Name="Objets_Proposition_accessibles_information">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
        - <xpdl:ActualParameters>

              <xpdl:ActualParameter>'Objets_Proposition_accessibles'</xpdl:ActualParameter>

              <xpdl:ActualParameter>Objets_Proposition_accessibles_schema</xpdl:ActualPara

              <xpdl:ActualParameter>Objets_Proposition_accessibles_recipient</xpdl:ActualPa

              <xpdl:ActualParameter>Objets_Proposition_accessibles_sender</xpdl:ActualPara

              <xpdl:ActualParameter>Objets_Proposition_accessibles_message</xpdl:ActualPa

              <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
    - <xpdl:StartMode>
```

```
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Objets_Proposition_accessibles_reception"
    Name="Objets_Proposition_accessibles_reception">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
      - <xpdl:ActualParameters>

          <xpdl:ActualParameter>'Objets_Proposition_accessibles'</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Manual />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Objets_Proposition_accessibles_check"
    Name="Objets_Proposition_accessibles_check">
  - <xpdl:Implementation>
    - <xpdl:SubFlow Id="_checker">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

            <xpdl:ActualParameter>Objets_Proposition_accessibles_message</xpdl:ActualPa

            <xpdl:ActualParameter>Objets_Proposition_accessibles_recipient</xpdl:ActualPa

            <xpdl:ActualParameter>Objets_Proposition_accessibles_sender</xpdl:ActualPara

            <xpdl:ActualParameter>Objets_Proposition_accessibles_schema</xpdl:ActualPar

            <xpdl:ActualParameter>Objets_Proposition_accessibles_constraints</xpdl:Actua

            <xpdl:ActualParameter>Objets_Proposition_accessibles_errorText</xpdl:ActualP
        </xpdl:ActualParameters>
      </xpdl:SubFlow>
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084036244815_24097_585_ok" />
            <xpdl:TransitionRef
```

```
            Id="_609020c_1084036244815_24097_585_nok" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Objets_propositions_transfert_information"
      Name="Objets_propositions_transfert_information">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
        - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Objets_propositions_transfert'</xpdl:ActualParameter>

            <xpdl:ActualParameter>Objets_propositions_transfert_schema</xpdl:ActualParar

            <xpdl:ActualParameter>Objets_propositions_transfert_recipient</xpdl:ActualPara

            <xpdl:ActualParameter>Objets_propositions_transfert_sender</xpdl:ActualParam

            <xpdl:ActualParameter>Objets_propositions_transfert_message</xpdl:ActualPara

            <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
    - <xpdl:StartMode>
        <xpdl:Automatic />
      </xpdl:StartMode>
    - <xpdl:FinishMode>
        <xpdl:Automatic />
      </xpdl:FinishMode>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Objets_propositions_transfert_reception"
      Name="Objets_propositions_transfert_reception">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
        - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Objets_propositions_transfert'</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
    - <xpdl:StartMode>
        <xpdl:Manual />
      </xpdl:StartMode>
    - <xpdl:FinishMode>
        <xpdl:Automatic />
      </xpdl:FinishMode>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Objets_propositions_transfert_check"
      Name="Objets_propositions_transfert_check">
    - <xpdl:Implementation>
```

```
- <xpdl:SubFlow Id="_checker">
  - <xpdl:ActualParameters>
      <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
      <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

        <xpdl:ActualParameter>Objets_propositions_transfert_message</xpdl:ActualPara

        <xpdl:ActualParameter>Objets_propositions_transfert_recipient</xpdl:ActualPara

        <xpdl:ActualParameter>Objets_propositions_transfert_sender</xpdl:ActualParam

        <xpdl:ActualParameter>Objets_propositions_transfert_schema</xpdl:ActualParar

        <xpdl:ActualParameter>Objets_propositions_transfert_constraints</xpdl:ActualP

        <xpdl:ActualParameter>Objets_propositions_transfert_errorText</xpdl:ActualPar
    </xpdl:ActualParameters>
  </xpdl:SubFlow>
</xpdl:Implementation>
- <xpdl:TransitionRestrictions>
  - <xpdl:TransitionRestriction>
      <xpdl:Join Type="AND" />
    - <xpdl:Split Type="XOR">
      - <xpdl:TransitionRefs>
          <xpdl:TransitionRef
            Id="_609020c_1084041465983_210179_669_ok" />
          <xpdl:TransitionRef
            Id="_609020c_1084041465983_210179_669_nok" />
        </xpdl:TransitionRefs>
      </xpdl:Split>
    </xpdl:TransitionRestriction>
  </xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="Refus_proposition_nonTraite_information"
  Name="Refus_proposition_nonTraite_information">
- <xpdl:Implementation>
  - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
    - <xpdl:ActualParameters>

        <xpdl:ActualParameter>'Refus_proposition_nonTraite'</xpdl:ActualParameter>

        <xpdl:ActualParameter>Refus_proposition_nonTraite_schema</xpdl:ActualParam

        <xpdl:ActualParameter>Refus_proposition_nonTraite_recipient</xpdl:ActualParar

        <xpdl:ActualParameter>Refus_proposition_nonTraite_sender</xpdl:ActualParame

        <xpdl:ActualParameter>Refus_proposition_nonTraite_message</xpdl:ActualParar

        <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
      </xpdl:ActualParameters>
    </xpdl:Tool>
  </xpdl:Implementation>
- <xpdl:StartMode>
    <xpdl:Automatic />
  </xpdl:StartMode>
- <xpdl:FinishMode>
    <xpdl:Automatic />
  </xpdl:FinishMode>
- <xpdl:TransitionRestrictions>
  - <xpdl:TransitionRestriction>
      <xpdl:Join Type="XOR" />
    </xpdl:TransitionRestriction>
```

```
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Refus_proposition_nonTraite_reception"
    Name="Refus_proposition_nonTraite_reception">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
      - <xpdl:ActualParameters>

          <xpdl:ActualParameter>'Refus_proposition_nonTraite'</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Refus_proposition_nonTraite_check"
    Name="Refus_proposition_nonTraite_check">
  - <xpdl:Implementation>
    - <xpdl:SubFlow Id="_checker">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

            <xpdl:ActualParameter>Refus_proposition_nonTraite_message</xpdl:ActualParar

            <xpdl:ActualParameter>Refus_proposition_nonTraite_recipient</xpdl:ActualParar

            <xpdl:ActualParameter>Refus_proposition_nonTraite_sender</xpdl:ActualParame

            <xpdl:ActualParameter>Refus_proposition_nonTraite_schema</xpdl:ActualParam

            <xpdl:ActualParameter>Refus_proposition_nonTraite_constraints</xpdl:ActualPa

            <xpdl:ActualParameter>Refus_proposition_nonTraite_errorText</xpdl:ActualPara
        </xpdl:ActualParameters>
      </xpdl:SubFlow>
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084044047575_65599_835_ok" />
            <xpdl:TransitionRef
              Id="_609020c_1084044047575_65599_835_nok" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Refus_de_proposition_transfert_information"
    Name="Refus_de_proposition_transfert_information">
  - <xpdl:Implementation>
```

```
- <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
  - <xpdl:ActualParameters>

      <xpdl:ActualParameter>'Refus_de_proposition_transfert'</xpdl:ActualParameter>

      <xpdl:ActualParameter>Refus_de_proposition_transfert_schema</xpdl:ActualPar

      <xpdl:ActualParameter>Refus_de_proposition_transfert_recipient</xpdl:ActualPa

      <xpdl:ActualParameter>Refus_de_proposition_transfert_sender</xpdl:ActualPara

      <xpdl:ActualParameter>Refus_de_proposition_transfert_message</xpdl:ActualPa

      <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
    </xpdl:ActualParameters>
  </xpdl:Tool>
</xpdl:Implementation>
- <xpdl:StartMode>
    <xpdl:Automatic />
</xpdl:StartMode>
- <xpdl:FinishMode>
    <xpdl:Automatic />
</xpdl:FinishMode>
- <xpdl:TransitionRestrictions>
  - <xpdl:TransitionRestriction>
      <xpdl:Join Type="XOR" />
    </xpdl:TransitionRestriction>
</xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="Refus_de_proposition_transfert_reception"
  Name="Refus_de_proposition_transfert_reception">
- <xpdl:Implementation>
  - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
    - <xpdl:ActualParameters>

        <xpdl:ActualParameter>'Refus_de_proposition_transfert'</xpdl:ActualParameter>
      </xpdl:ActualParameters>
    </xpdl:Tool>
</xpdl:Implementation>
- <xpdl:StartMode>
    <xpdl:Manual />
</xpdl:StartMode>
- <xpdl:FinishMode>
    <xpdl:Automatic />
</xpdl:FinishMode>
- <xpdl:TransitionRestrictions>
  - <xpdl:TransitionRestriction>
      <xpdl:Join Type="XOR" />
    </xpdl:TransitionRestriction>
</xpdl:TransitionRestrictions>
</xpdl:Activity>
- <xpdl:Activity Id="Refus_de_proposition_transfert_check"
  Name="Refus_de_proposition_transfert_check">
- <xpdl:Implementation>
  - <xpdl:SubFlow Id="_checker">
    - <xpdl:ActualParameters>
        <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
        <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

        <xpdl:ActualParameter>Refus_de_proposition_transfert_message</xpdl:ActualPa

        <xpdl:ActualParameter>Refus_de_proposition_transfert_recipient</xpdl:ActualPa
```

```
            <xpdl:ActualParameter>Refus_de_proposition_transfert_sender</xpdl:ActualParamet

                <xpdl:ActualParameter>Refus_de_proposition_transfert_schema</xpdl:ActualPar

                <xpdl:ActualParameter>Refus_de_proposition_transfert_constraints</xpdl:Actua

                <xpdl:ActualParameter>Refus_de_proposition_transfert_errorText</xpdl:ActualP
            </xpdl:ActualParameters>
          </xpdl:SubFlow>
        </xpdl:Implementation>
      - <xpdl:TransitionRestrictions>
        - <xpdl:TransitionRestriction>
            <xpdl:Join Type="AND" />
          - <xpdl:Split Type="XOR">
            - <xpdl:TransitionRefs>
                <xpdl:TransitionRef
                  Id="_609020c_1084044577186_781054_907_ok" />
                <xpdl:TransitionRef
                  Id="_609020c_1084044577186_781054_907_nok" />
              </xpdl:TransitionRefs>
            </xpdl:Split>
          </xpdl:TransitionRestriction>
        </xpdl:TransitionRestrictions>
      </xpdl:Activity>
    - <xpdl:Activity Id="Accord_proposition_nonTraite_information"
        Name="Accord_proposition_nonTraite_information">
      - <xpdl:Implementation>
        - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
          - <xpdl:ActualParameters>

                <xpdl:ActualParameter>'Accord_proposition_nonTraite'</xpdl:ActualParameter>

                <xpdl:ActualParameter>Accord_proposition_nonTraite_schema</xpdl:ActualParar

                <xpdl:ActualParameter>Accord_proposition_nonTraite_recipient</xpdl:ActualPara

                <xpdl:ActualParameter>Accord_proposition_nonTraite_sender</xpdl:ActualParam

                <xpdl:ActualParameter>Accord_proposition_nonTraite_message</xpdl:ActualPara

                <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
            </xpdl:ActualParameters>
          </xpdl:Tool>
        </xpdl:Implementation>
      - <xpdl:StartMode>
          <xpdl:Automatic />
        </xpdl:StartMode>
      - <xpdl:FinishMode>
          <xpdl:Automatic />
        </xpdl:FinishMode>
      - <xpdl:TransitionRestrictions>
        - <xpdl:TransitionRestriction>
            <xpdl:Join Type="XOR" />
          </xpdl:TransitionRestriction>
        </xpdl:TransitionRestrictions>
      </xpdl:Activity>
    - <xpdl:Activity Id="Accord_proposition_nonTraite_reception"
        Name="Accord_proposition_nonTraite_reception">
      - <xpdl:Implementation>
        - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
          - <xpdl:ActualParameters>

                <xpdl:ActualParameter>'Accord_proposition_nonTraite'</xpdl:ActualParameter>
```

```
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Accord_proposition_nonTraite_check"
    Name="Accord_proposition_nonTraite_check">
  - <xpdl:Implementation>
    - <xpdl:SubFlow Id="_checker">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

            <xpdl:ActualParameter>Accord_proposition_nonTraite_message</xpdl:ActualPara

            <xpdl:ActualParameter>Accord_proposition_nonTraite_recipient</xpdl:ActualPara

            <xpdl:ActualParameter>Accord_proposition_nonTraite_sender</xpdl:ActualParam

            <xpdl:ActualParameter>Accord_proposition_nonTraite_schema</xpdl:ActualParar

            <xpdl:ActualParameter>Accord_proposition_nonTraite_constraints</xpdl:ActualP

            <xpdl:ActualParameter>Accord_proposition_nonTraite_errorText</xpdl:ActualPar
        </xpdl:ActualParameters>
      </xpdl:SubFlow>
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084045286987_106270_982_ok" />
            <xpdl:TransitionRef
              Id="_609020c_1084045286987_106270_982_nok" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Accord_proposition_traite_information"
    Name="Accord_proposition_traite_information">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
      - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Accord_proposition_traite'</xpdl:ActualParameter>

            <xpdl:ActualParameter>Accord_proposition_traite_schema</xpdl:ActualParamete

            <xpdl:ActualParameter>Accord_proposition_traite_recipient</xpdl:ActualParamet
```

```
            <xpdl:ActualParameter>Accord_proposition_traite_sender</xpdl:ActualParameter>

                <xpdl:ActualParameter>Accord_proposition_traite_message</xpdl:ActualParamet

                <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
            </xpdl:ActualParameters>
          </xpdl:Tool>
        </xpdl:Implementation>
      - <xpdl:StartMode>
          <xpdl:Automatic />
        </xpdl:StartMode>
      - <xpdl:FinishMode>
          <xpdl:Automatic />
        </xpdl:FinishMode>
      - <xpdl:TransitionRestrictions>
        - <xpdl:TransitionRestriction>
            <xpdl:Join Type="XOR" />
          </xpdl:TransitionRestriction>
        </xpdl:TransitionRestrictions>
      </xpdl:Activity>
    - <xpdl:Activity Id="Accord_proposition_traite_reception"
        Name="Accord_proposition_traite_reception">
      - <xpdl:Implementation>
        - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
          - <xpdl:ActualParameters>

                <xpdl:ActualParameter>'Accord_proposition_traite'</xpdl:ActualParameter>
            </xpdl:ActualParameters>
          </xpdl:Tool>
        </xpdl:Implementation>
      - <xpdl:StartMode>
          <xpdl:Manual />
        </xpdl:StartMode>
      - <xpdl:FinishMode>
          <xpdl:Automatic />
        </xpdl:FinishMode>
      - <xpdl:TransitionRestrictions>
        - <xpdl:TransitionRestriction>
            <xpdl:Join Type="XOR" />
          </xpdl:TransitionRestriction>
        </xpdl:TransitionRestrictions>
      </xpdl:Activity>
    - <xpdl:Activity Id="Accord_proposition_traite_check"
        Name="Accord_proposition_traite_check">
      - <xpdl:Implementation>
        - <xpdl:SubFlow Id="_checker">
          - <xpdl:ActualParameters>
              <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
              <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

                <xpdl:ActualParameter>Accord_proposition_traite_message</xpdl:ActualParamet

                <xpdl:ActualParameter>Accord_proposition_traite_recipient</xpdl:ActualParamet

                <xpdl:ActualParameter>Accord_proposition_traite_sender</xpdl:ActualParameter

                <xpdl:ActualParameter>Accord_proposition_traite_schema</xpdl:ActualParamete

                <xpdl:ActualParameter>Accord_proposition_traite_constraints</xpdl:ActualParam

                <xpdl:ActualParameter>Accord_proposition_traite_errorText</xpdl:ActualParame
            </xpdl:ActualParameters>
          </xpdl:SubFlow>
```

```
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084045519602_393923_1036_ok" />
            <xpdl:TransitionRef
              Id="_609020c_1084045519602_393923_1036_nok" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Objets_proposition_modification_information"
    Name="Objets_proposition_modification_information">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
      - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Objets_proposition_modification'</xpdl:ActualParameter>

            <xpdl:ActualParameter>Objets_proposition_modification_schema</xpdl:ActualPa

            <xpdl:ActualParameter>Objets_proposition_modification_recipient</xpdl:ActualP

            <xpdl:ActualParameter>Objets_proposition_modification_sender</xpdl:ActualPar

            <xpdl:ActualParameter>Objets_proposition_modification_message</xpdl:ActualP

            <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Objets_proposition_modification_reception"
    Name="Objets_proposition_modification_reception">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
      - <xpdl:ActualParameters>

            <xpdl:ActualParameter>'Objets_proposition_modification'</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Manual />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
```

```
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="XOR" />
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Objets_proposition_modification_check"
      Name="Objets_proposition_modification_check">
    - <xpdl:Implementation>
      - <xpdl:SubFlow Id="_checker">
        - <xpdl:ActualParameters>
            <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
            <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

              <xpdl:ActualParameter>Objets_proposition_modification_message</xpdl:ActualP

              <xpdl:ActualParameter>Objets_proposition_modification_recipient</xpdl:ActualP

              <xpdl:ActualParameter>Objets_proposition_modification_sender</xpdl:ActualPar

              <xpdl:ActualParameter>Objets_proposition_modification_schema</xpdl:ActualPa

              <xpdl:ActualParameter>Objets_proposition_modification_constraints</xpdl:Actu

              <xpdl:ActualParameter>Objets_proposition_modification_errorText</xpdl:Actuall
          </xpdl:ActualParameters>
        </xpdl:SubFlow>
      </xpdl:Implementation>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
          <xpdl:Join Type="AND" />
        - <xpdl:Split Type="XOR">
          - <xpdl:TransitionRefs>
              <xpdl:TransitionRef
                Id="_609020c_1084607106379_63506_81_ok" />
              <xpdl:TransitionRef
                Id="_609020c_1084607106379_63506_81_nok" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  - <xpdl:Activity Id="Demande_de_validation_Modification_information"
      Name="Demande_de_validation_Modification_information">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="MessageInformer" Type="APPLICATION">
        - <xpdl:ActualParameters>

              <xpdl:ActualParameter>'Demande_de_validation_Modification'</xpdl:ActualParan

              <xpdl:ActualParameter>Demande_de_validation_Modification_schema</xpdl:Act

              <xpdl:ActualParameter>Demande_de_validation_Modification_recipient</xpdl:A

              <xpdl:ActualParameter>Demande_de_validation_Modification_sender</xpdl:Actu

              <xpdl:ActualParameter>Demande_de_validation_Modification_message</xpdl:A

              <xpdl:ActualParameter>messageRootName</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
    - <xpdl:StartMode>
```

```
      <xpdl:Automatic />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Demande_de_validation_Modification_reception"
    Name="Demande_de_validation_Modification_reception">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="MessageHandler" Type="APPLICATION">
      - <xpdl:ActualParameters>

          <xpdl:ActualParameter>'Demande_de_validation_Modification'</xpdl:ActualParan
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
  - <xpdl:StartMode>
      <xpdl:Manual />
    </xpdl:StartMode>
  - <xpdl:FinishMode>
      <xpdl:Automatic />
    </xpdl:FinishMode>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="XOR" />
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  </xpdl:Activity>
- <xpdl:Activity Id="Demande_de_validation_Modification_check"
    Name="Demande_de_validation_Modification_check">
  - <xpdl:Implementation>
    - <xpdl:SubFlow Id="_checker">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>

            <xpdl:ActualParameter>Demande_de_validation_Modification_message</xpdl:A

            <xpdl:ActualParameter>Demande_de_validation_Modification_recipient</xpdl:A

            <xpdl:ActualParameter>Demande_de_validation_Modification_sender</xpdl:Actu

            <xpdl:ActualParameter>Demande_de_validation_Modification_schema</xpdl:Act

            <xpdl:ActualParameter>Demande_de_validation_Modification_constraints</xpdl

            <xpdl:ActualParameter>Demande_de_validation_Modification_errorText</xpdl:A
        </xpdl:ActualParameters>
      </xpdl:SubFlow>
    </xpdl:Implementation>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
        <xpdl:Join Type="AND" />
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef
              Id="_609020c_1084606923346_181986_57_ok" />
            <xpdl:TransitionRef
```

```
            Id="_609020c_1084606923346_181986_57_nok" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    </xpdl:Activity>
  </xpdl:Activities>
- <xpdl:Transitions>
    <xpdl:Transition Id="_609020c_1084028389640_25006_194" From="initial"
      To="Introduire__une__demande__de__validation" />
    <xpdl:Transition Id="_609020c_1084032722590_750295_477"
      From="Introduire__une__demande__de__validation"
      To="Demande_de_validation_NonTraite_information" />
    <xpdl:Transition Id="_609020c_1084041727899_775052_737"
      From="_609020c_1084041644169_565354_713"
      To="_609020c_1084041725085_928465_735" />
    <xpdl:Transition Id="_609020c_1084393267814_526342_238"
      From="_609020c_1084044545571_673193_895"
      To="_609020c_1084393175311_84032_236" />
    <xpdl:Transition Id="_609020c_1084395011361_464636_321"
      From="_609020c_1084395007316_693386_319"
      To="_609020c_1084045522816_3898_1046" />
    <xpdl:Transition Id="_609020c_1084393354168_342793_250"
      From="_609020c_1084393175311_84032_236"
      To="_609020c_1084044620989_805516_917" />
    <xpdl:Transition Id="_609020c_1084395048375_782405_354"
      From="_609020c_1084393175311_84032_236"
      To="_609020c_1084395041495_332562_352" />
    <xpdl:Transition Id="_609020c_1084394812936_890832_308"
      From="_609020c_1084393380727_718345_260"
      To="Objets_proposition_modification_information" />
    <xpdl:Transition Id="_609020c_1084395029968_904734_332"
      From="_609020c_1084045476920_289436_1024"
      To="_609020c_1084395007316_693386_319" />
    <xpdl:Transition Id="_609020c_1084395054834_508401_355"
      From="_609020c_1084395007316_693386_319"
      To="_609020c_1084395041495_332562_352" />
    <xpdl:Transition Id="_609020c_1084395059430_735720_365"
      From="_609020c_1084395041495_332562_352"
      To="_609020c_1084393380727_718345_260" />
    <xpdl:Transition Id="_609020c_1084035627578_7894_529"
      From="_609020c_1084030518341_700532_323"
      To="Demande_de_consultation___information" />
    <xpdl:Transition Id="_609020c_1084041515574_399938_681"
      From="_609020c_1084036605794_205207_629"
      To="_609020c_1084041507032_959447_679" />
    <xpdl:Transition Id="_609020c_1084041597792_550882_703"
      From="_609020c_1084041507032_959447_679"
      To="Refus_proposition_nonTraite_information" />
    <xpdl:Transition Id="_609020c_1084392623588_670060_181"
      From="_609020c_1084041507032_959447_679"
      To="Accord_proposition_nonTraite_information" />
    <xpdl:Transition Id="_609020c_1084606929625_367812_67"
      From="_609020c_1084606899792_245429_44"
      To="_609020c_1084030518341_700532_323" />
    <xpdl:Transition Id="_609020c_1084032394549_887544_440"
      From="_609020c_1084031028995_851646_375"
      To="_609020c_1084032346650_737054_427" />
    <xpdl:Transition Id="_609020c_1084032564823_255398_453"
      From="_609020c_1084032346650_737054_427"
      To="_609020c_1084032548901_789479_450" />
    <xpdl:Transition Id="_609020c_1084032889781_516914_497"
      From="_609020c_1084032548901_789479_450"
      To="Donner__acces__a__la__proposition" />
    <xpdl:Transition Id="_609020c_1084035785865_409106_551"
```

```
From="_609020c_1084032346650_737054_427"
    To="Rejet_demande_de_validation___information" />
  <xpdl:Transition Id="_609020c_1084036286866_903768_595"
    From="Donner__acces__a__la__proposition"
    To="Objets_Proposition_accessibles_information" />
  <xpdl:Transition Id="_609020c_1084041444892_875518_649"
    From="_609020c_1084036217035_542454_573"
    To="_609020c_1084036515264_659807_605" />
  <xpdl:Transition Id="_609020c_1084041461496_70868_659"
    From="_609020c_1084036515264_659807_605"
    To="Objets_propositions_transfert_information" />
  <xpdl:Transition Id="_609020c_1084044201166_341542_861"
    From="Transmettre__refus__proposition"
    To="Refus_de_proposition_transfert_information" />
  <xpdl:Transition Id="_609020c_1084392724673_805668_219"
    From="_609020c_1084045131443_643956_958"
    To="Accord_proposition_traite_information" />
  <xpdl:Transition Id="_609020c_1084606917487_485034_47"
    From="_609020c_1084032548901_789479_450"
    To="Demande_de_validation_ Conforme_information" />
  <xpdl:Transition Id="_609020c_1084607122702_327102_82"
    From="_609020c_1084395793556_316949_375"
    To="Demande_de_validation_Modification_information" />
  <xpdl:Transition Id="_609020c_1084031153875_833858_387_reception"
    From="Demande_de_validation_NonTraite_information"
    To="Demande_de_validation_NonTraite_reception" />
  <xpdl:Transition Id="_609020c_1084031153875_833858_387"
    From="Demande_de_validation_NonTraite_reception"
    To="Demande_de_validation_NonTraite_check" />
- <xpdl:Transition Id="_609020c_1084031153875_833858_387_ok"
    From="Demande_de_validation_NonTraite_check"
    To="_609020c_1084031028995_851646_375">
    <xpdl:Condition
      Type="CONDITION">Demande_de_validation_NonTraite_errorText==""</xpdl:Condi
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084031153875_833858_387_nok"
    From="Demande_de_validation_NonTraite_check"
    To="Demande_de_validation_NonTraite_information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084036222333_311314_575_reception"
    From="Demande_de_consultation___information"
    To="Demande_de_consultation___reception" />
  <xpdl:Transition Id="_609020c_1084036222333_311314_575"
    From="Demande_de_consultation___reception"
    To="Demande_de_consultation___check" />
- <xpdl:Transition Id="_609020c_1084036222333_311314_575_ok"
    From="Demande_de_consultation___check"
    To="_609020c_1084036217035_542454_573">
    <xpdl:Condition
      Type="CONDITION">Demande_de_consultation___errorText==""</xpdl:Condition>
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084036222333_311314_575_nok"
    From="Demande_de_consultation___check"
    To="Demande_de_consultation___information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084036244815_24097_585_reception"
    From="Objets_Proposition_accessibles_information"
    To="Objets_Proposition_accessibles_reception" />
  <xpdl:Transition Id="_609020c_1084036244815_24097_585"
    From="Objets_Proposition_accessibles_reception"
    To="Objets_Proposition_accessibles_check" />
- <xpdl:Transition Id="_609020c_1084036244815_24097_585_ok"
    From="Objets_Proposition_accessibles_check"
```

```
To="_609020c_1084036217035_542454_573">
    <xpdl:Condition
      Type="CONDITION">Objets_Proposition_accessibles_errorText==""</xpdl:Condition
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084036244815_24097_585_nok"
    From="Objets_Proposition_accessibles_check"
    To="Objets_Proposition_accessibles_information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084041465983_210179_669_reception"
    From="Objets_propositions_transfert_information"
    To="Objets_propositions_transfert_reception" />
  <xpdl:Transition Id="_609020c_1084041465983_210179_669"
    From="Objets_propositions_transfert_reception"
    To="Objets_propositions_transfert_check" />
- <xpdl:Transition Id="_609020c_1084041465983_210179_669_ok"
    From="Objets_propositions_transfert_check"
    To="_609020c_1084036605794_205207_629">
    <xpdl:Condition
      Type="CONDITION">Objets_propositions_transfert_errorText==""</xpdl:Condition>
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084041465983_210179_669_nok"
    From="Objets_propositions_transfert_check"
    To="Objets_propositions_transfert_information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084044047575_65599_835"
    From="Refus_proposition_nonTraite_reception"
    To="Refus_proposition_nonTraite_check" />
- <xpdl:Transition Id="_609020c_1084044047575_65599_835_ok"
    From="Refus_proposition_nonTraite_check"
    To="Transmettre__refus__proposition">
    <xpdl:Condition
      Type="CONDITION">Refus_proposition_nonTraite_errorText==""</xpdl:Condition>
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084044047575_65599_835_nok"
    From="Refus_proposition_nonTraite_check"
    To="Refus_proposition_nonTraite_information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084044047575_65599_835_route"
    From="Refus_proposition_nonTraite_information"
    To="_609020c_1084041507032_959447_679_route" />
- <xpdl:Transition Id="_609020c_1084044047575_65599_835_reception"
    From="_609020c_1084041507032_959447_679_route"
    To="Refus_proposition_nonTraite_reception">

      <xpdl:Condition>messageRootName=="Refus_proposition_nonTraite"</xpdl:Conditio
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084044106039_323595_837"
    From="Rejet_demande_de_validation___reception"
    To="Rejet_demande_de_validation___check" />
- <xpdl:Transition Id="_609020c_1084044106039_323595_837_ok"
    From="Rejet_demande_de_validation___check"
    To="_609020c_1084041644169_565354_713">
    <xpdl:Condition
      Type="CONDITION">Rejet_demande_de_validation___errorText==""</xpdl:Conditio
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084044106039_323595_837_nok"
    From="Rejet_demande_de_validation___check"
    To="Rejet_demande_de_validation___information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084044106039_323595_837_route"
```

```
From="Rejet_demande_de_validation___information"
    To="_609020c_1084032346650_737054_427_route" />
- <xpdl:Transition Id="_609020c_1084044106039_323595_837_reception"
    From="_609020c_1084032346650_737054_427_route"
    To="Rejet_demande_de_validation___reception">

      <xpdl:Condition>messageRootName=="Rejet_demande_de_validation__"</xpdl:Con
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084044577186_781054_907_reception"
    From="Refus_de_proposition_transfert_information"
    To="Refus_de_proposition_transfert_reception" />
  <xpdl:Transition Id="_609020c_1084044577186_781054_907"
    From="Refus_de_proposition_transfert_reception"
    To="Refus_de_proposition_transfert_check" />
- <xpdl:Transition Id="_609020c_1084044577186_781054_907_ok"
    From="Refus_de_proposition_transfert_check"
    To="_609020c_1084044545571_673193_895">
    <xpdl:Condition
      Type="CONDITION">Refus_de_proposition_transfert_errorText==""</xpdl:Condition
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084044577186_781054_907_nok"
    From="Refus_de_proposition_transfert_check"
    To="Refus_de_proposition_transfert_information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084045286987_106270_982"
    From="Accord_proposition_nonTraite_reception"
    To="Accord_proposition_nonTraite_check" />
- <xpdl:Transition Id="_609020c_1084045286987_106270_982_ok"
    From="Accord_proposition_nonTraite_check"
    To="_609020c_1084045131443_643956_958">
    <xpdl:Condition
      Type="CONDITION">Accord_proposition_nonTraite_errorText==""</xpdl:Condition>
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084045286987_106270_982_nok"
    From="Accord_proposition_nonTraite_check"
    To="Accord_proposition_nonTraite_information">
    <xpdl:Condition Type="OTHERWISE" />
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084045286987_106270_982_route"
    From="Accord_proposition_nonTraite_information"
    To="_609020c_1084041507032_959447_679_route" />
- <xpdl:Transition Id="_609020c_1084045286987_106270_982_reception"
    From="_609020c_1084041507032_959447_679_route"
    To="Accord_proposition_nonTraite_reception">

      <xpdl:Condition>messageRootName=="Accord_proposition_nonTraite"</xpdl:Conditi
  </xpdl:Transition>
  <xpdl:Transition Id="_609020c_1084045519602_393923_1036_reception"
    From="Accord_proposition_traite_information"
    To="Accord_proposition_traite_reception" />
  <xpdl:Transition Id="_609020c_1084045519602_393923_1036"
    From="Accord_proposition_traite_reception"
    To="Accord_proposition_traite_check" />
- <xpdl:Transition Id="_609020c_1084045519602_393923_1036_ok"
    From="Accord_proposition_traite_check"
    To="_609020c_1084045476920_289436_1024">
    <xpdl:Condition
      Type="CONDITION">Accord_proposition_traite_errorText==""</xpdl:Condition>
  </xpdl:Transition>
- <xpdl:Transition Id="_609020c_1084045519602_393923_1036_nok"
    From="Accord_proposition_traite_check"
    To="Accord_proposition_traite_information">
    <xpdl:Condition Type="OTHERWISE" />
```

```
    </xpdl:Transition>
    <xpdl:Transition Id="_609020c_1084606905500_321229_46"
      From="Demande_de_validation_ Conforme_reception"
      To="Demande_de_validation_ Conforme_check" />
  - <xpdl:Transition Id="_609020c_1084606905500_321229_46_ok"
      From="Demande_de_validation_ Conforme_check"
      To="_609020c_1084606899792_245429_44">
      <xpdl:Condition Type="CONDITION">Demande_de_validation_
        Conforme_errorText==""</xpdl:Condition>
    </xpdl:Transition>
  - <xpdl:Transition Id="_609020c_1084606905500_321229_46_nok"
      From="Demande_de_validation_ Conforme_check"
      To="Demande_de_validation_ Conforme_information">
      <xpdl:Condition Type="OTHERWISE" />
    </xpdl:Transition>
    <xpdl:Transition Id="_609020c_1084606905500_321229_46_route"
      From="Demande_de_validation_ Conforme_information"
      To="_609020c_1084032346650_737054_427_route" />
  - <xpdl:Transition Id="_609020c_1084606905500_321229_46_reception"
      From="_609020c_1084032346650_737054_427_route"
      To="Demande_de_validation_ Conforme_reception">
      <xpdl:Condition>messageRootName=="Demande_de_validation_
        Conforme"</xpdl:Condition>
    </xpdl:Transition>
    <xpdl:Transition Id="_609020c_1084607106379_63506_81_reception"
      From="Objets_proposition_modification_information"
      To="Objets_proposition_modification_reception" />
    <xpdl:Transition Id="_609020c_1084607106379_63506_81"
      From="Objets_proposition_modification_reception"
      To="Objets_proposition_modification_check" />
  - <xpdl:Transition Id="_609020c_1084607106379_63506_81_ok"
      From="Objets_proposition_modification_check"
      To="_609020c_1084395793556_316949_375">
      <xpdl:Condition
        Type="CONDITION">Objets_proposition_modification_errorText==""</xpdl:Conditio
    </xpdl:Transition>
  - <xpdl:Transition Id="_609020c_1084607106379_63506_81_nok"
      From="Objets_proposition_modification_check"
      To="Objets_proposition_modification_information">
      <xpdl:Condition Type="OTHERWISE" />
    </xpdl:Transition>
    <xpdl:Transition Id="_609020c_1084606923346_181986_57_reception"
      From="Demande_de_validation_Modification_information"
      To="Demande_de_validation_Modification_reception" />
    <xpdl:Transition Id="_609020c_1084606923346_181986_57"
      From="Demande_de_validation_Modification_reception"
      To="Demande_de_validation_Modification_check" />
  - <xpdl:Transition Id="_609020c_1084606923346_181986_57_ok"
      From="Demande_de_validation_Modification_check"
      To="_609020c_1084606899792_245429_44">
      <xpdl:Condition
        Type="CONDITION">Demande_de_validation_Modification_errorText==""</xpdl:Co
    </xpdl:Transition>
  - <xpdl:Transition Id="_609020c_1084606923346_181986_57_nok"
      From="Demande_de_validation_Modification_check"
      To="Demande_de_validation_Modification_information">
      <xpdl:Condition Type="OTHERWISE" />
    </xpdl:Transition>
  </xpdl:Transitions>
</xpdl:WorkflowProcess>
- <xpdl:WorkflowProcess AccessLevel="PRIVATE" Id="_checker" Name="Check"
  xsi:schemaLocation="http://www.wfmc.org/2002/XPDL1.0
  http://www.wfmc.org/standards/docs/TC-1025_schema_10_xpdl.xsd">
  - <xpdl:ProcessHeader DurationUnit="D">
```

```
      <xpdl:Created>2003-10-27</xpdl:Created>
    </xpdl:ProcessHeader>
    <xpdl:RedefinableHeader PublicationStatus="UNDER_TEST" />
  - <xpdl:FormalParameters>
    - <xpdl:FormalParameter Id="procInstId" Index="0" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="initialSender" Index="1" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="message" Index="2" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="recipient" Index="3" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="sender" Index="4" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="schema" Index="5" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="constraints" Index="6" Mode="IN">
      - <xpdl:DataType>
          <!-- <xpdl:SchemaType/>  -->
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="errorText" Index="7" Mode="OUT">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    </xpdl:FormalParameters>
  - <xpdl:Participants>
    - <xpdl:Participant Id="efficient" Name="efficient">
        <xpdl:ParticipantType Type="ROLE" />
      </xpdl:Participant>
    </xpdl:Participants>
  - <xpdl:Applications>
    - <xpdl:Application Id="XMLSchemaChecker">
      - <xpdl:FormalParameters>
        - <xpdl:FormalParameter Id="messageName" Mode="IN">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
            </xpdl:DataType>
          </xpdl:FormalParameter>
        - <xpdl:FormalParameter Id="schemaName" Mode="IN">
          - <xpdl:DataType>
              <xpdl:BasicType Type="STRING" />
```

```
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="procInstId" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="result" Mode="OUT">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    </xpdl:FormalParameters>
  - <xpdl:ExtendedAttributes>
    - <xpdl:ExtendedAttribute Name="Implementation">
      - <vx:ToolAgent Class="JavaTool" Execution="SYNCHR"
          XMLParameterMode="USE_W3C_DOM">
          <vx:Property
            Name="class">efficient.animator.XMLSchemaChecker</vx:Property>
          <vx:Property Name="method">check</vx:Property>
        </vx:ToolAgent>
      </xpdl:ExtendedAttribute>
    </xpdl:ExtendedAttributes>
  </xpdl:Application>
- <xpdl:Application Id="ErrorSender">
  - <xpdl:FormalParameters>
    - <xpdl:FormalParameter Id="isSchema" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="BOOLEAN" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="messageName" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="sender" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="schemaName" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="errorText" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="procInstId" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    </xpdl:FormalParameters>
  - <xpdl:ExtendedAttributes>
    - <xpdl:ExtendedAttribute Name="Implementation">
      - <vx:ToolAgent Class="JavaTool" Execution="SYNCHR"
          XMLParameterMode="USE_W3C_DOM">
          <vx:Property
```

```
        Name="class">efficient.animator.ErrorSender</vx:Property>
          <vx:Property Name="method">send</vx:Property>
        </vx:ToolAgent>
      </xpdl:ExtendedAttribute>
    </xpdl:ExtendedAttributes>
  </xpdl:Application>
- <xpdl:Application Id="XlinkitConstraintChecker">
  - <xpdl:FormalParameters>
    - <xpdl:FormalParameter Id="messageName" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="constraints" Mode="IN">
      - <xpdl:DataType>
          <!-- <xpdl:SchemaType/>  -->
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="procInstId" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="result" Mode="OUT">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    </xpdl:FormalParameters>
  - <xpdl:ExtendedAttributes>
    - <xpdl:ExtendedAttribute Name="Implementation">
      - <vx:ToolAgent Class="JavaTool" Execution="SYNCHR"
          XMLParameterMode="USE_W3C_DOM">
          <vx:Property
            Name="class">efficient.animator.XlinkitConstraintChecker</vx:Property>
          <vx:Property Name="method">check</vx:Property>
        </vx:ToolAgent>
      </xpdl:ExtendedAttribute>
    </xpdl:ExtendedAttributes>
  </xpdl:Application>
- <xpdl:Application Id="XMLMessageSender">
  - <xpdl:FormalParameters>
    - <xpdl:FormalParameter Id="messageName" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="recipient" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="procInstId" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
      </xpdl:FormalParameter>
    - <xpdl:FormalParameter Id="initialSender" Mode="IN">
      - <xpdl:DataType>
          <xpdl:BasicType Type="STRING" />
        </xpdl:DataType>
```

```
        </xpdl:FormalParameter>
      </xpdl:FormalParameters>
    - <xpdl:ExtendedAttributes>
      - <xpdl:ExtendedAttribute Name="Implementation">
        - <vx:ToolAgent Class="JavaTool" Execution="SYNCHR"
            XMLParameterMode="USE_W3C_DOM">
            <vx:Property
              Name="class">efficient.animator.XMLMessageSender</vx:Property>
            <vx:Property Name="method">send</vx:Property>
          </vx:ToolAgent>
        </xpdl:ExtendedAttribute>
      </xpdl:ExtendedAttributes>
    </xpdl:Application>
  </xpdl:Applications>
- <xpdl:Activities>
  - <xpdl:Activity Id="_schemachecker" Name="XMLSchemaChecker">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="XMLSchemaChecker" Type="APPLICATION">
        - <xpdl:ActualParameters>
            <xpdl:ActualParameter>message</xpdl:ActualParameter>
            <xpdl:ActualParameter>schema</xpdl:ActualParameter>
            <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
            <xpdl:ActualParameter>errorText</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
      <xpdl:Performer>efficient</xpdl:Performer>
    - <xpdl:TransitionRestrictions>
      - <xpdl:TransitionRestriction>
        - <xpdl:Split Type="XOR">
          - <xpdl:TransitionRefs>
              <xpdl:TransitionRef Id="_check_edge_2" />
              <xpdl:TransitionRef Id="_check_edge_1" />
            </xpdl:TransitionRefs>
          </xpdl:Split>
        </xpdl:TransitionRestriction>
      </xpdl:TransitionRestrictions>
    - <xpdl:ExtendedAttributes>
        <xpdl:ExtendedAttribute Name="ParticipantID" Value="efficient" />
        <xpdl:ExtendedAttribute Name="XOffset" Value="141" />
        <xpdl:ExtendedAttribute Name="YOffset" Value="85" />
      </xpdl:ExtendedAttributes>
    </xpdl:Activity>
  - <xpdl:Activity Id="_schemaerrorsender" Name="SchemaErrorSender">
    - <xpdl:Implementation>
      - <xpdl:Tool Id="ErrorSender" Type="APPLICATION">
        - <xpdl:ActualParameters>
            <xpdl:ActualParameter>true</xpdl:ActualParameter>
            <xpdl:ActualParameter>message</xpdl:ActualParameter>
            <xpdl:ActualParameter>sender</xpdl:ActualParameter>
            <xpdl:ActualParameter>schema</xpdl:ActualParameter>
            <xpdl:ActualParameter>errorText</xpdl:ActualParameter>
            <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          </xpdl:ActualParameters>
        </xpdl:Tool>
      </xpdl:Implementation>
      <xpdl:Performer>efficient</xpdl:Performer>
    - <xpdl:ExtendedAttributes>
        <xpdl:ExtendedAttribute Name="ParticipantID" Value="efficient" />
        <xpdl:ExtendedAttribute Name="XOffset" Value="331" />
        <xpdl:ExtendedAttribute Name="YOffset" Value="80" />
      </xpdl:ExtendedAttributes>
```

```
</xpdl:Activity>
- <xpdl:Activity Id="_xlinkitchecker" Name="XlinkitConstraintChecker">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="XlinkitConstraintChecker" Type="APPLICATION">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>message</xpdl:ActualParameter>
          <xpdl:ActualParameter>constraints</xpdl:ActualParameter>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>errorText</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
    <xpdl:Performer>efficient</xpdl:Performer>
  - <xpdl:TransitionRestrictions>
    - <xpdl:TransitionRestriction>
      - <xpdl:Split Type="XOR">
        - <xpdl:TransitionRefs>
            <xpdl:TransitionRef Id="_check_edge_4" />
            <xpdl:TransitionRef Id="_check_edge_3" />
          </xpdl:TransitionRefs>
        </xpdl:Split>
      </xpdl:TransitionRestriction>
    </xpdl:TransitionRestrictions>
  - <xpdl:ExtendedAttributes>
      <xpdl:ExtendedAttribute Name="ParticipantID" Value="efficient" />
      <xpdl:ExtendedAttribute Name="XOffset" Value="141" />
      <xpdl:ExtendedAttribute Name="YOffset" Value="195" />
    </xpdl:ExtendedAttributes>
  </xpdl:Activity>
- <xpdl:Activity Id="_constrainterrorsender" Name="ConstraintErrorSender">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="ErrorSender" Type="APPLICATION">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>false</xpdl:ActualParameter>
          <xpdl:ActualParameter>message</xpdl:ActualParameter>
          <xpdl:ActualParameter>sender</xpdl:ActualParameter>
          <xpdl:ActualParameter>schema</xpdl:ActualParameter>
          <xpdl:ActualParameter>errorText</xpdl:ActualParameter>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
    <xpdl:Performer>efficient</xpdl:Performer>
  - <xpdl:ExtendedAttributes>
      <xpdl:ExtendedAttribute Name="ParticipantID" Value="efficient" />
      <xpdl:ExtendedAttribute Name="XOffset" Value="351" />
      <xpdl:ExtendedAttribute Name="YOffset" Value="205" />
    </xpdl:ExtendedAttributes>
  </xpdl:Activity>
- <xpdl:Activity Id="_messagesender" Name="XMLMessageSender">
  - <xpdl:Implementation>
    - <xpdl:Tool Id="XMLMessageSender" Type="APPLICATION">
      - <xpdl:ActualParameters>
          <xpdl:ActualParameter>message</xpdl:ActualParameter>
          <xpdl:ActualParameter>recipient</xpdl:ActualParameter>
          <xpdl:ActualParameter>procInstId</xpdl:ActualParameter>
          <xpdl:ActualParameter>initialSender</xpdl:ActualParameter>
        </xpdl:ActualParameters>
      </xpdl:Tool>
    </xpdl:Implementation>
    <xpdl:Performer>efficient</xpdl:Performer>
  - <xpdl:ExtendedAttributes>
```

```
            <xpdl:ExtendedAttribute Name="ParticipantID" Value="efficient" />
            <xpdl:ExtendedAttribute Name="XOffset" Value="141" />
            <xpdl:ExtendedAttribute Name="YOffset" Value="290" />
          </xpdl:ExtendedAttributes>
        </xpdl:Activity>
      </xpdl:Activities>
    - <xpdl:Transitions>
      - <xpdl:Transition From="_schemachecker" Id="_check_edge_2"
          To="_xlinkitchecker">
          <xpdl:Condition Type="CONDITION">errorText==""</xpdl:Condition>
        - <xpdl:ExtendedAttributes>
            <xpdl:ExtendedAttribute Name="RoutingType" Value="NOROUTING" />
          </xpdl:ExtendedAttributes>
        </xpdl:Transition>
      - <xpdl:Transition From="_schemachecker" Id="_check_edge_1"
          To="_schemaerrorsender">
          <xpdl:Condition Type="OTHERWISE" />
        - <xpdl:ExtendedAttributes>
            <xpdl:ExtendedAttribute Name="RoutingType" Value="NOROUTING" />
          </xpdl:ExtendedAttributes>
        </xpdl:Transition>
      - <xpdl:Transition From="_xlinkitchecker" Id="_check_edge_4"
          To="_messagesender">
          <xpdl:Condition Type="CONDITION">errorText==""</xpdl:Condition>
        - <xpdl:ExtendedAttributes>
            <xpdl:ExtendedAttribute Name="RoutingType" Value="NOROUTING" />
          </xpdl:ExtendedAttributes>
        </xpdl:Transition>
      - <xpdl:Transition From="_xlinkitchecker" Id="_check_edge_3"
          To="_constrainterrorsender">
          <xpdl:Condition Type="OTHERWISE" />
        - <xpdl:ExtendedAttributes>
            <xpdl:ExtendedAttribute Name="RoutingType" Value="NOROUTING" />
          </xpdl:ExtendedAttributes>
        </xpdl:Transition>
      </xpdl:Transitions>
    - <xpdl:ExtendedAttributes>
        <xpdl:ExtendedAttribute Name="StartOfWorkflow"
          Value="efficient;_schemachecker;161;10;NOROUTING" />
        <xpdl:ExtendedAttribute Name="EndOfWorkflow"
          Value="efficient;_schemaerrorsender;500;80;NOROUTING" />
        <xpdl:ExtendedAttribute Name="EndOfWorkflow"
          Value="efficient;_constrainterrorsender;500;220;NOROUTING" />
        <xpdl:ExtendedAttribute Name="EndOfWorkflow"
          Value="efficient;_messagesender;160;380;NOROUTING" />
        <xpdl:ExtendedAttribute Name="ParticipantVisualOrder" Value="efficient;" />
      </xpdl:ExtendedAttributes>
    </xpdl:WorkflowProcess>
  </xpdl:WorkflowProcesses>
</xpdl:Package>
```

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:eff="http://efficient.citi.tudor.lu">
  - <xsd:simpleType name="Type_Role">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="GestionaireMaquette" />
        <xsd:enumeration value="Maitre ouvrage" />
        <xsd:enumeration value="Architecte" />
        <xsd:enumeration value="IngenieurStabilite" />
        <xsd:enumeration value="Entrepreneur" />
        <xsd:enumeration value="IngenieurEco" />
        <xsd:enumeration value="IngenieurElectricite" />
        <xsd:enumeration value="Consultant" />
        <xsd:enumeration value="Def_Utilisateur" />
        <xsd:enumeration value="Non_Defini" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Statut">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="NonTraité" />
        <xsd:enumeration value="Conforme" />
        <xsd:enumeration value="Accord" />
        <xsd:enumeration value="Rejet" />
        <xsd:enumeration value="Refus" />
        <xsd:enumeration value="EnCours" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:element name="Demande_de_validation_Conforme">
    - <xsd:complexType>
      - <xsd:sequence>
          <xsd:element name="Valider_Proposition" type="Valider_Proposition"
            minOccurs="1" maxOccurs="1" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  - <xsd:complexType name="Valider_Proposition">
    - <xsd:sequence>
        <xsd:element name="Nom_Proposition" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="IndiceProposition" type="xsd:integer" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="date_emission" type="xsd:date" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="echeance" type="xsd:date" minOccurs="0" maxOccurs="1" />
        <xsd:element name="Statut" type="Type_Statut" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="Processus" type="Processus" minOccurs="1"
          maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>
  - <xsd:complexType name="Document">
    - <xsd:sequence>
        <xsd:element name="ID_Document" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="Nom_Doc" type="xsd:string" minOccurs="1"
          maxOccurs="1" />
        <xsd:element name="Description" type="xsd:string" minOccurs="0"
          maxOccurs="1" />
        <xsd:element name="Reference" type="xsd:string" minOccurs="1"
          maxOccurs="unbounded" />
        <xsd:element name="Auteur" type="xsd:string" minOccurs="0"
          maxOccurs="unbounded" />
        <xsd:element name="Editeur" type="xsd:string" minOccurs="0"
```

```
    maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Objet_Projet">
  - <xsd:sequence>
      <xsd:element name="T_Objet" type="xsd:string" minOccurs="0" maxOccurs="1" />
      <xsd:element name="Root" type="Root" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Personne">
  - <xsd:sequence>
      <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Nom" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Prenom" type="xsd:string" minOccurs="0" maxOccurs="1" />
      <xsd:element name="Role_Acteur" type="Role_Acteur" minOccurs="1"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Root">
  - <xsd:sequence>
      <xsd:element name="GlobalID" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="NomProjet" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Associe_Document" type="Document" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Processus">
  - <xsd:sequence>
      <xsd:element name="Assigner_Acteur" type="Acteur" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Objet_Projet" type="Objet_Projet" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Role_Acteur">
  - <xsd:sequence>
      <xsd:element name="Role" type="Type_Role" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Acteur">
  - <xsd:sequence>
      <xsd:element name="Objet_Projet" type="Objet_Projet" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Personne" type="Personne" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

ebXML Technical Architecture Specification v1.0.4

ebXML Technical Architecture Project Team

16 February 2001

1 Status of this Document

This document is a FINAL DRAFT for the *eBusiness* community. Distribution of this document is unlimited. This document will go through the formal Quality Review Process as defined by the ebXML Requirements Document. The formatting for this document is based on the Internet Society's Standard RFC format.

This version:
ebXML_TA_v1.0.4.doc

***Latest version*:**
ebXML_TA_v1.0.4.doc

Previous version:
ebXML_TA_v1.0.3.doc

2 ebXML Technical Architecture Participants

We would like to recognize the following for their significant participation in the development of this document.

Team Lead: Anders Grangard, EDI France

Editors: Brian Eisenberg, DataChannel
Duane Nickull, XML Global Technologies

Participants: Colin Barham, TIE
Al Boseman, ATPCO
Christian Barret, GIP-MDS
Dick Brooks, Group 8760
Cory Casanave, DataAccess Technologies
Robert Cunningham, Military Traffic Management Command, US Army
Christopher Ferris, Sun Microsystems
Peter Kacandes, Sun Microsystems
Kris Ketels, SWIFT

Copyright © ebXML 2001. All Rights Reserved.

Piming Kuo, Worldspan
Kyu-Chul Lee, Chungnam National University
Henry Lowe, OMG
Matt MacKenzie, XML Global Technologies
Melanie McCarthy, General Motors
Stefano Pagliani, Sun Microsystems
Bruce Peat, eProcessSolutions
John Petit, KPMG Consulting
Mark Heller, MITRE
Scott Hinkelman, IBM
Karsten Riemer, Sun Microsystems
Lynne Rosenthal, NIST
Nikola Stojanovic, Encoda Systems, Inc.
Jeff Sutor, Sun Microsystems
David RR Webber, XML Global Technologies

Copyright © ebXML 2001. All Rights Reserved.

3 Introduction

3.1 Summary of Contents of Document

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in RFC 2119 [Bra97].

The following conventions are used throughout this document:

- *Capitalized Italics* words are defined in the ebXML Glossary.
- [NOTES: are used to further clarify the discussion or to offer additional suggestions and/or resources]

EBXML TECHNICAL ARCHITECTURE SPECIFICATION V1.0.3 1

1 STATUS OF THIS DOCUMENT 1
2 EBXML TECHNICAL ARCHITECTURE PARTICIPANTS 1
3 INTRODUCTION 3
3.1 Summary of Contents of Document 3
3.2 Audience and Scope 4
3.3 Related Documents 5
3.4 Normative References 5
4 DESIGN OBJECTIVES 5
4.1 Problem Description & Goals for ebXML 5
4.2 Caveats and Assumptions 6
4.3 Design Conventions for ebXML Specifications 6
5 EBXML SYSTEM OVERVIEW 7
6 EBXML RECOMMENDED MODELING METHODOLOGY 9
6.1 Overview 9
6.2 ebXML Business Operational View 11
6.3 ebXML Functional Service View 13
7 EBXML FUNCTIONAL PHASES 14
7.1 Implementation Phase 14
7.2 Discovery and Retrieval Phase 14
7.3 Run Time Phase 15
8 EBXML INFRASTRUCTURE 16
8.1 Trading Partner Information [CPP and CPA's] 16
8.1.1 Introduction 16
8.1.2 CPP Formal Functionality 16
8.1.3 CPA Formal Functionality 16
8.1.4 CPP Interfaces 17
8.1.5 CPA Interfaces 18
8.1.6 Non-Normative Implementation Details [CPP and CPA's] 18
8.2 Business Process and Information Modeling 19
8.2.1 Introduction 19

Copyright © ebXML 2001. All Rights Reserved.

8.2.2 Formal Functionality 21
8.2.3 Interfaces 21
8.2.4 Non-Normative Implementation Details 22
8.3 Core Components and Core Library Functionality *23*
8.3.1 Introduction 23
8.3.2 Formal Functionality 23
8.3.3 Interfaces 23
8.3.4 Non-Normative Implementation Details 24
8.4 Registry Functionality *25*
8.4.1 Introduction 25
8.4.2 Formal Functionality 25
8.4.3 Interfaces 27
8.4.4 Non-Normative Implementation Details 27
8.5 Messaging Service Functionality *28*
8.5.1 Introduction 28
8.5.2 Formal Functionality 30
8.5.3 Interfaces 30
8.5.4 Non-Normative Implementation Details 31
9 CONFORMANCE 32
9.1 Introduction *32*
9.2 Conformance to ebXML *32*
9.3 Conformance to the Technical Architecture Specification *33*
9.4 General Framework of Conformance Testing *33*
10.0 SECURITY CONSIDERATIONS 33
10.1 Introduction *34*
DISCLAIMER 34
COPYRIGHT STATEMENT 34
APPENDIX A: EXAMPLE EBXML BUSINESS SCENARIOS 35
SCENARIO 1 : TWO TRADING PARTNERS SET-UP AN AGREEMENT AND RUN THE ASSOCIATED EXCHANGE 35
SCENARIO 2: THREE OR MORE PARTIES SET-UP A BUSINESS PROCESS IMPLEMENTING A SUPPLY-CHAIN AND RUN THE ASSOCIATED EXCHANGES 36
SCENARIO 3 : A COMPANY SETS UP A PORTAL WHICH DEFINES A BUSINESS PROCESS INVOLVING THE USE OF EXTERNAL BUSINESS SERVICES 37
SCENARIO 4: THREE OR MORE TRADING PARTNERS CONDUCT BUSINESS USING SHARED BUSINESS PROCESSES AND RUN THE ASSOCIATED EXCHANGES 38

3.2 Audience and Scope

This document is intended primarily for the ebXML project teams to help guide their work. Secondary audiences include, but are not limited to: software implementers, international standards bodies, and other industry organizations.

This document describes the underlying architecture for ebXML. It provides a high level overview of ebXML and describes the relationships, interactions, and basic functionality

Copyright © ebXML 2001. All Rights Reserved.

of ebXML. It SHOULD be used as a roadmap to learn: (1) what ebXML is, (2) what problems ebXML solves, and (3) core ebXML functionality and architecture.

3.3 Related Documents

As mentioned above, other documents provide detailed definitions of the components of ebXML and of their inter-relationship. They include ebXML specifications on the following topics:

1. Requirements
2. Business Process and Information Meta Model
3. Core Components
4. Registry and Repository
5. Trading Partner Information
6. Messaging Services

These specifications are available for download at http://www.ebxml.org.

3.4 Normative References

The following standards contain provisions that, through reference in this text, constitute provisions of this specification. At the time of publication, the editions indicated below were valid. All standards are subject to revision, and parties to agreements based on this specification are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

ISO/IEC 14662: Open-edi Reference Model
ISO 11179/3 Metadata Repository
ISO 10646: Character Encoding
ISO 8601:2000 Date/Time/Number Data typing
OASIS Registry/Repository Technical Specification
RFC 2119: Keywords for use in RFC's to Indicate Requirement Levels
UN/CEFACT Modeling Methodology (UMM)
W3C XML v1.0 Second Edition Specification

4 Design Objectives

4.1 Problem Description & Goals for ebXML

For over 25 years *Electronic Data Interchange (EDI)* has given companies the prospect of eliminating paper documents, reducing costs, and improving efficiency by exchanging business information in electronic form. Ideally, companies of all sizes could conduct *eBusiness* in a completely ad hoc fashion, without prior agreement of any kind. But this vision has not been realized with *EDI*; only large companies are able to afford to

Copyright © ebXML 2001. All Rights Reserved.

implement it, and much *EDI*-enabled *eBusiness* is centered around a dominant enterprise that imposes proprietary integration approaches on its *Trading Partners*.

In the last few years, the *Extensible Markup Language (XML)* has rapidly become the first choice for defining data interchange formats in new *eBusiness* applications on the Internet. Many people have interpreted the *XML* groundswell as evidence that "*EDI* is dead" – made completely obsolete by the *XML* upstart -- but this view is naïve from both business and technical standpoints.

EDI implementations encode substantial experience in *Business Processes*, and companies with large investments in *EDI* integration will not abandon them without good reason. *XML* enables more open, more flexible business transactions than *EDI*. *XML* might enable more flexible and innovative "eMarketplace" business models than *EDI*. But the challenges of designing M*essages* that meet *Business Process* requirements and standardizing their semantics are independent of the syntax in which the M*essages* are encoded.

The ebXML specifications provide a framework in which *EDI's* substantial investments in *Business Processes* can be preserved in an architecture that exploits *XML's* new technical capabilities.

Please consult the ebXML Requirements Specification, available at http://www.ebxml.org, for additional information on the underlying goals of ebXML.

4.2 Caveats and Assumptions

This specification is designed to provide a high level overview of ebXML, and as such, does not provide the level of detail required to build *ebXML Applications*, components, and related services. Please refer to each of the respective ebXML specifications to get the level of detail.

4.3 Design Conventions for ebXML Specifications

In order to enforce a consistent capitalization and naming convention across all ebXML specifications "Upper Camel Case" (*UCC*) and "Lower Camel Case" (*LCC*) Capitalization styles SHALL be used. *UCC* style capitalizes the first character of each word and compounds the name. *LCC* style capitalizes the first character of each word except the first word.

1) ebXML DTD, XML Schema and *XML* instance documents SHALL have the effect of producing ebXML *XML* instance documents such that:

- Element names SHALL be in *UCC* convention (example: <UpperCamelCaseElement/>).
- Attribute names SHALL be in *LCC* convention (example: <UpperCamelCaseElement lowerCamelCaseAttribute="Whatever"/>).

Copyright © ebXML 2001. All Rights Reserved.

2) When *UML* and *Object Constrained Language (OCL)* are used to specify ebXML artifacts Capitalization naming SHALL follow the following rules:

- Class, *Interface*, Association, Package, State, Use Case, Actor names SHALL use UCC convention (examples: ClassificationNode, Versionable, Active, InsertOrder, Buyer).
- Attribute, Operation, Role, Stereotype, Instance, Event, Action names SHALL use LCC convention (examples: name, notifySender, resident, orderArrived).

3) General rules for all names are:

- Acronyms SHOULD be avoided, but in cases where they are used, the capitalization SHALL remain (example: XMLSignature).
- Underscore (_), periods (.) and dashes (-) MUST NOT be used (don't use: header.manifest, stock_quote_5, commercial-transaction, use HeaderManifest, stockQuote5, CommercialTransaction instead).

5 ebXML System Overview

Figure 1 below shows a high-level use case scenario for two *Trading Partners*, first configuring and then engaging in a simple business transaction and interchange. This model is provided as an example of the process and steps that may be required to configure and deploy *ebXML Applications* and related architecture components. These components can be implemented in an incremental manner. The ebXML specifications are not limited to this simple model, provided here as quick introduction to the concepts. Specific ebXML implementation examples are described in Appendix A.

The conceptual overview described below introduces the following concepts and underlying architecture:

1. A standard mechanism for describing a *Business Process* and its associated information model.
2. A mechanism for registering and storing *Business Process and Information Meta Models* so they can be shared and reused.
3. Discovery of information about each participant including:
    - The *Business Processes* they support.
    - The *Business Service Interfaces* they offer in support of the *Business Process*.
    - The *Business Messages* that are exchanged between their respective *Business Service Interfaces*.
    - The technical configuration of the supported transport, security and encoding protocols.
4. A mechanism for registering the aforementioned information so that it may be discovered and retrieved.

Copyright © ebXML 2001. All Rights Reserved.

5. A mechanism for describing the execution of a mutually agreed upon business arrangement which can be derived from information provided by each participant from item 3 above. (*Collaboration Protocol Agreement – CPA*)
6. A standardized business *Messaging Service* framework that enables interoperable, secure and reliable exchange of M*essages* between *Trading Partners*.
7. A mechanism for configuration of the respective *Messaging Services* to engage in the agreed upon *Business Process* in accordance with the constraints defined in the business arrangement.

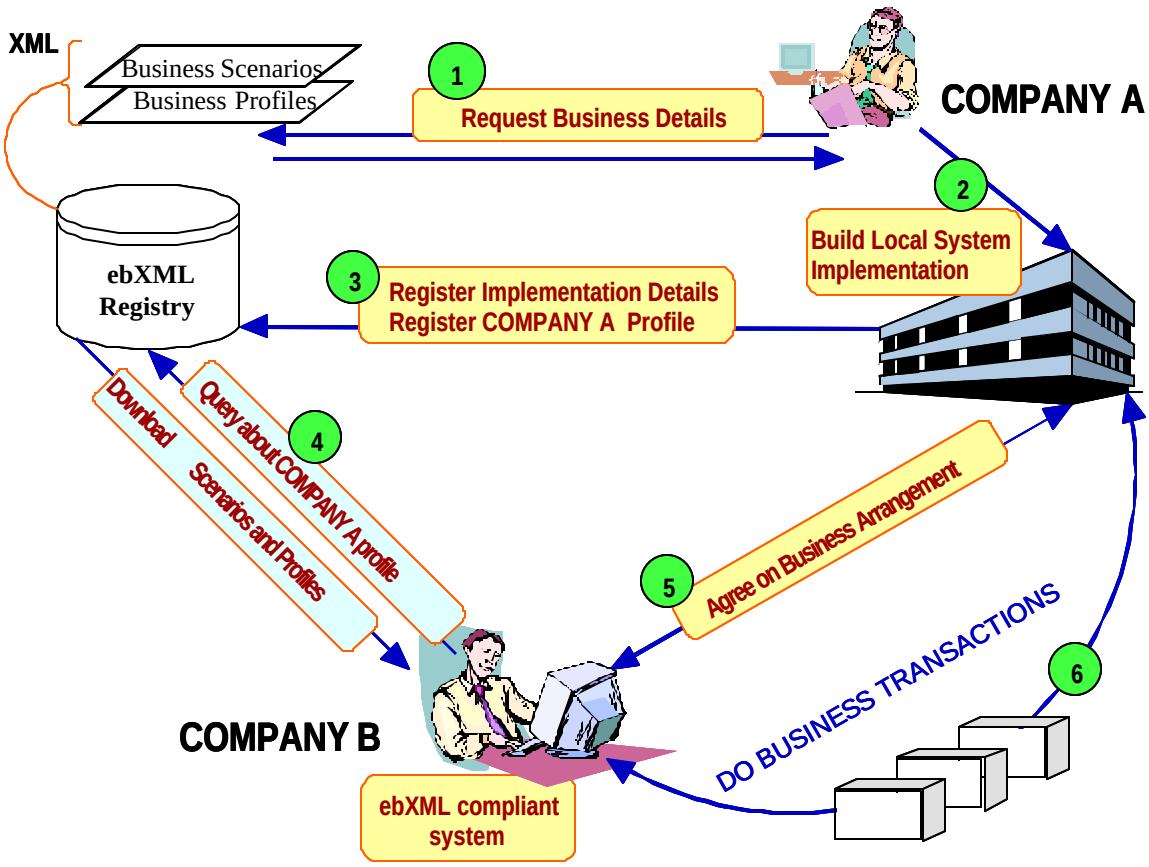


Figure 1 *- a high level overview of the interaction of two companies conducting eBusiness using ebXML.*

In Figure 1, Company A has become aware of an ebXML *Registry* that is accessible on the Internet (Figure 1, step 1). Company A, after reviewing the contents of the ebXML *Registry*, decides to build and deploy its own ebXML compliant application (Figure 1, step 2). Custom software development is not a necessary prerequisite for ebXML participation. ebXML compliant applications and components may also be commercially available as shrink-wrapped solutions.

Company A then submits its own *Business Profile* information (including implementation details and reference links) to the ebXML *Registry* (Figure 1, step 3). The business profile submitted to the ebXML *Registry* describes the company's ebXML capabilities and constraints, as well as its supported business scenarios. These business scenarios are

Copyright © ebXML 2001. All Rights Reserved.

XML versions of the *Business Processes* and associated information bundles (e.g. a sales tax calculation) in which the company is able to engage. After receiving verification that the format and usage of a business scenario is correct, an acknowledgment is sent to Company A (Figure 1, step 3).

Company B discovers the business scenarios supported by Company A in the ebXML *Registry* (Figure 1, step 4). Company B sends a request to Company A stating that they would like to engage in a business scenario using ebXML (Figure 1, step 5). Company B acquires an ebXML compliant shrink-wrapped application.

Before engaging in the scenario Company B submits a proposed business arrangement directly to Company A's ebXML compliant software *Interface*. The proposed business arrangement outlines the mutually agreed upon business scenarios and specific agreements. The business arrangement also contains information pertaining to the messaging requirements for transactions to take place, contingency plans, and security-related requirements (Figure 1, step 5). Company A then accepts the business agreement. Company A and B are now ready to engage in *eBusiness* using ebXML (Figure 1, step 6).

6 ebXML Recommended Modeling Methodology

Business Process and Information Modeling is not mandatory. However, if implementers and users select to model *Business Processes and Information*, then they SHALL use the *UN/CEFACT Modeling Methodology (UMM)* that utilizes *UML*.

6.1 Overview

While business practices from one organization to another are highly variable, most activities can be decomposed into *Business Processes* which are more generic to a specific type of business. This analysis through the modeling process will identify *Business Process and Information Meta Models* that are likely candidates for standardization. The ebXML approach looks for standard reusable components from which to construct interoperable and components.

The *UN/CEFACT Modeling Methodology (UMM)* uses the following two views to describe the relevant aspects of *eBusiness* transactions. This model is based upon the Open-edi Reference Model, ISO/IEC 14662.

Copyright © ebXML 2001. All Rights Reserved.

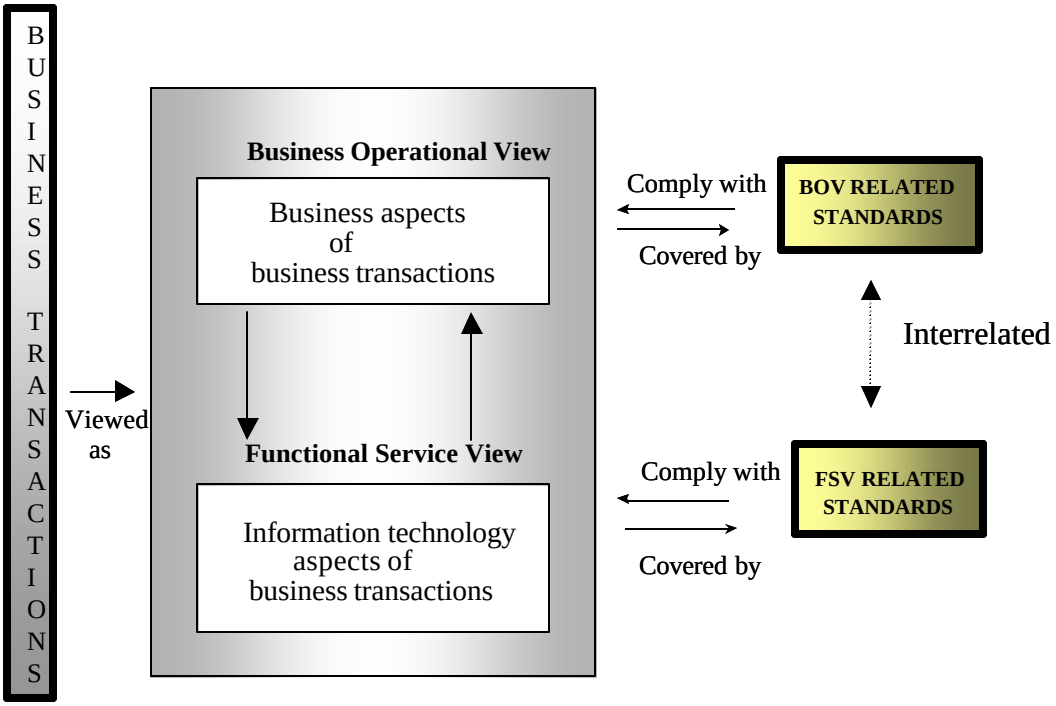


Figure 2 *- ebXML Recommended Modeling Methodology*

The *UN/CEFACT Modeling Methodology (UMM)* is broken down into the *Business Operational View (BOV)* and the supporting *Functional Service View (FSV)* described above. The assumption for ebXML is that the *FSV* serves as a reference model that MAY be used by commercial software vendors to help guide them during the development process. The underlying goal of the *UN/CEFACT Modeling Methodology (UMM)* is to provide a clear distinction between the operational and functional views, so as to ensure the maximum level of system interoperability and backwards compatibility with legacy systems (when applicable). As such, the resultant *BOV*-related standards provide the *UN/CEFACT Modeling Methodology (UMM)* for constructing *Business Process and Information Meta Models* for ebXML compliant applications and components.

The *BOV* addresses:

a) The semantics of business data in transactions and associated data interchanges
b) The architecture for business transactions, including:
   - operational conventions;
   - agreements and arrangements;
   - mutual obligations and requirements.

These specifically apply to the business needs of ebXML *Trading Partners*.

The *FSV* addresses the supporting services meeting the mechanistic needs of ebXML. It focuses on the information technology aspects of:

- Functional capabilities;
- *Business Service Interfaces*;

Copyright © ebXML 2001. All Rights Reserved.

- Protocols and *Messaging Services.*

This includes, but is not limited to:

- Capabilities for implementation, discovery, deployment and run time scenarios;
- User *Interfaces*;
- Data transfer infrastructure *Interfaces*;
- *Protocols* for enabling interoperability of *XML* vocabulary deployments from different organizations.

6.2 ebXML Business Operational View

The modeling techniques described in this section are not mandatory requirements for participation in ebXML compliant business transactions.

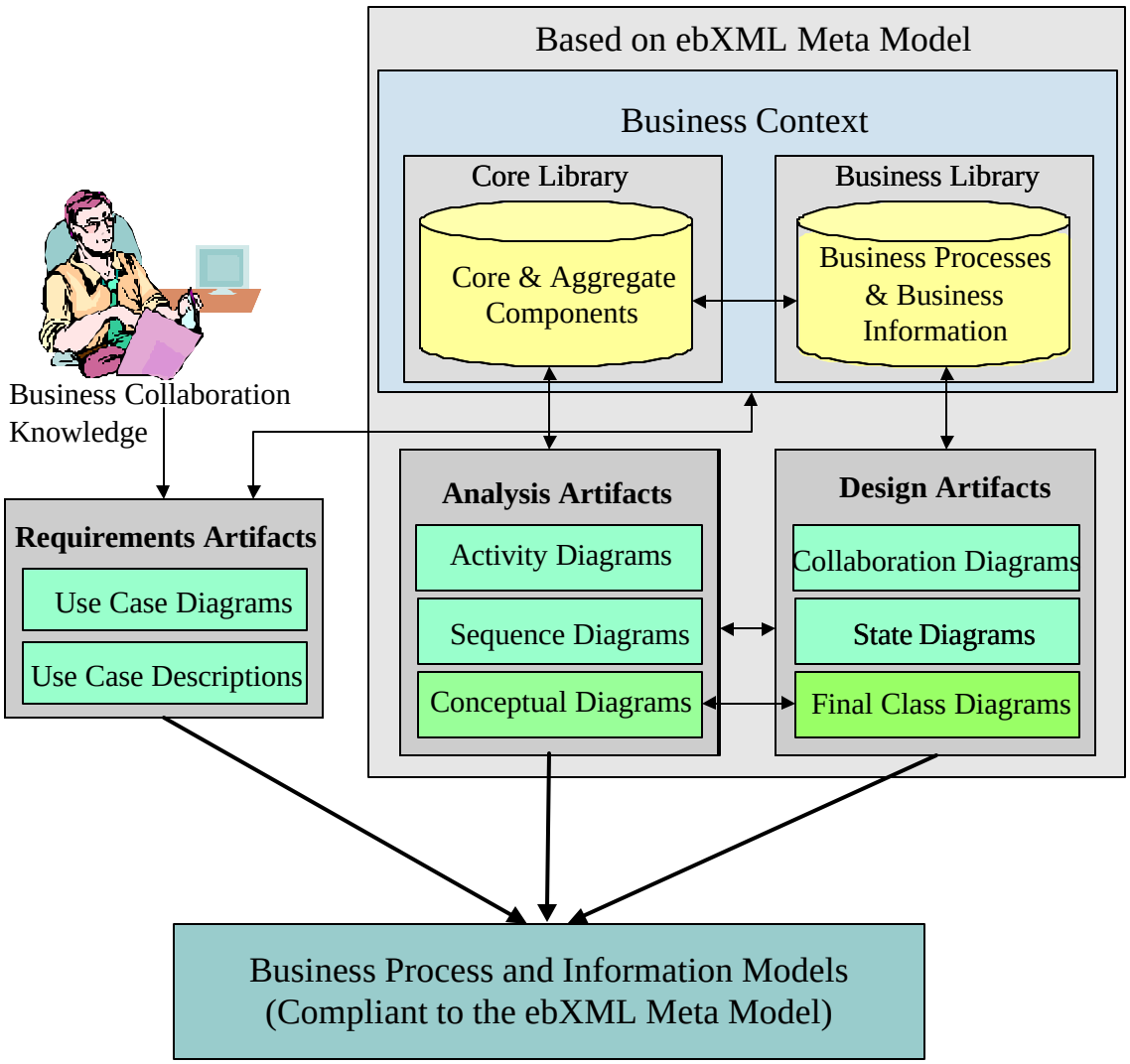


***Figure 3** – detailed representation of the Business Operational View*

Copyright © ebXML 2001. All Rights Reserved.

In Figure 3 above, *Business Collaboration Knowledge* is captured in a *Core Library*. The *Core Library* contains data and process definitions, including relationships and cross-references, as expressed in business terminology that MAY be tied to an accepted industry classification scheme or taxonomy. The *Core Library* is the bridge between the specific business or industry language and the knowledge expressed by the models in a more generalized context neutral language.

The first phase defines the requirements artifacts that describe the problem using *Use Case Diagrams and Descriptions*. If *Core Library* entries are available from an ebXML compliant *Registry* they will be utilized, otherwise new *Core Library* entries will be created and registered in an ebXML compliant *Registry*.

The second phase (analysis) will create activity and sequence diagrams (as defined in the *UN/CEFACT Modeling Methodology* specification) describing the *Business Processes*. *Class Diagrams* will capture the associated information parcels (business documents). The analysis phase reflects the business knowledge contained in the *Core Library*. No effort is made to force the application of object-oriented principles. The class diagram is a free structured data diagram. Common *Business Processes* in the Business Library MAY be referenced during the process of creating analysis and design artifacts.

The design phase is the last step of standardization, which MAY be accomplished by applying object-oriented principles based on the *UN/CEFACT Modeling Methodology*. In addition to generating collaboration diagrams, a state diagram MAY also be created. The class view diagram from the analysis phase will undergo harmonization to align it with other models in the same industry and across others.

In ebXML, interoperability is achieved by applying *Business Information Objects* across all class models. *Business Processes* are created by applying the *UN/CEFACT Modeling Methodology (UMM)* which utilizes a common set of *Business Information Objects* and *Core Components.*

Copyright © ebXML 2001. All Rights Reserved.

6.3 ebXML Functional Service View

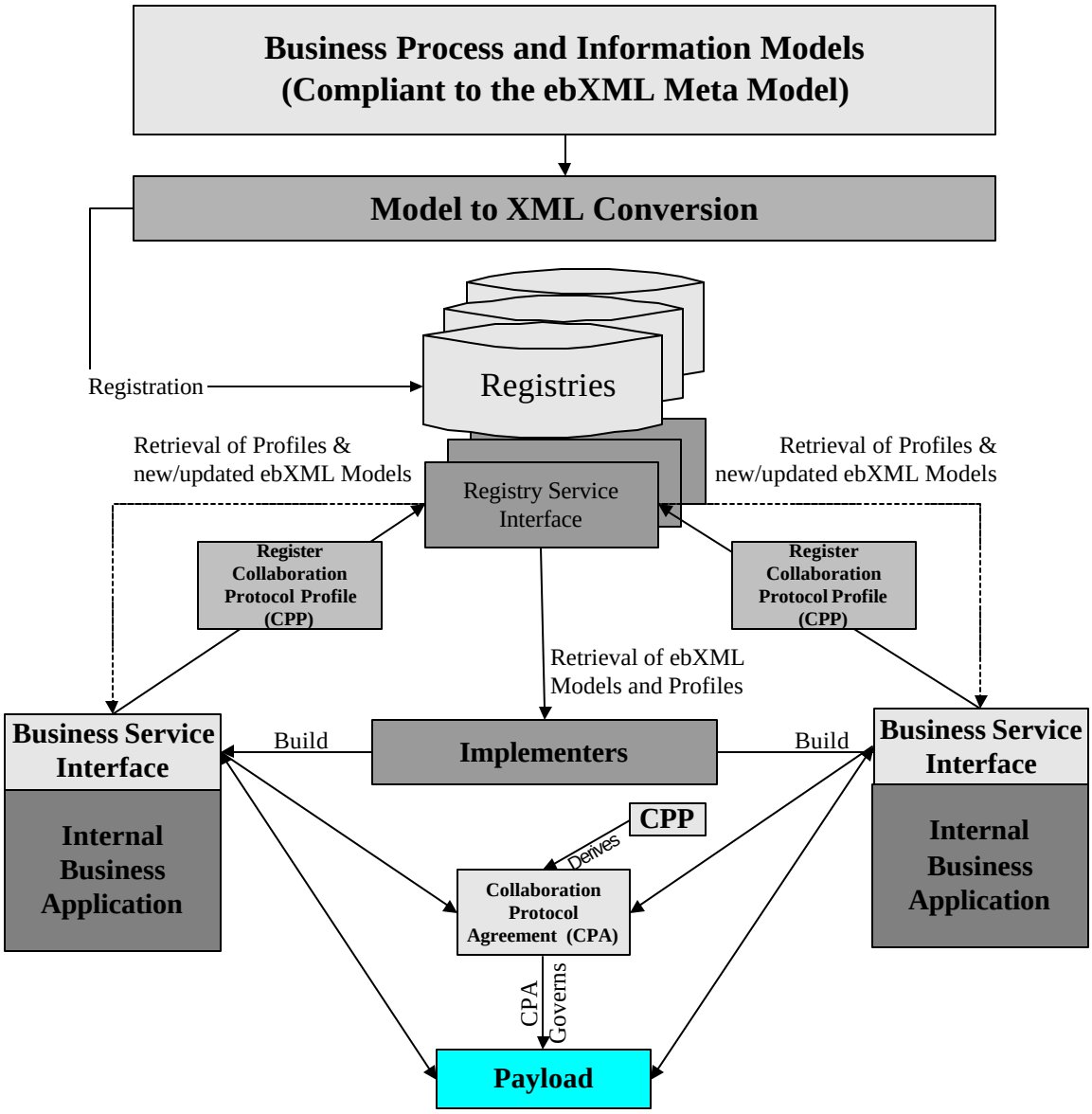


Figure 4 - *ebXML Functional Service View*

As illustrated in Figure 4 above, the ebXML *Registry Service* serves as the storage facility for the *Business Process and Information Models*, the *XML*-based representations of those models, *Core Components*, and *Collaboration Protocol Profiles*. The *Business Process and Information Meta Models* MAY be stored in modeling syntax, however they MAY be also stored as *XML* syntax in the *Registry*. This *XML*-based business information SHALL be expressed in a manner that allows discovery down to the atomic data level via a consistent methodology.

Copyright © ebXML 2001. All Rights Reserved.

The underlying ebXML Architecture is distributed in such a manner to minimize the potential for a single point of failure within the ebXML infrastructure. This specifically refers to *Registry Services* (see Registry Functionality, Section 8.4 for details of this architecture).

7 ebXML Functional Phases

7.1 Implementation Phase

The implementation phase deals specifically with the procedures for creating an application of the ebXML infrastructure. A *Trading Partner* wishing to engage in an ebXML compliant transaction SHOULD first acquire copies of the ebXML Specifications. The *Trading Partner* studies these specifications and subsequently downloads the *Core Library* and the *Business Library*. The *Trading Partner* MAY also request other *Trading Partners' Business Process* information (stored in their business profile) for analysis and review. Alternatively, the Trading Partner MAY implement ebXML by utilizing 3rd party applications. The *Trading Partner* can also submit its own *Business Process* information to an ebXML compliant *Registry Service*.

Figure 5 below, illustrates a basic interaction between an ebXML *Registry Service* and a *Trading Partner*.

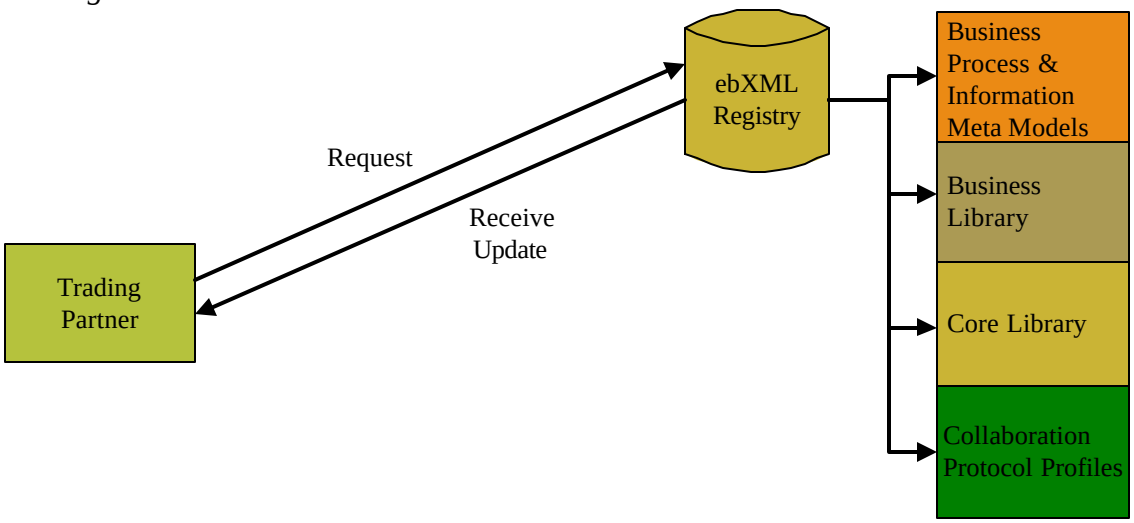


Figure 5 - *Functional Service View: Implementation Phase*

7.2 Discovery and Retrieval Phase

The Discovery and Retrieval Phase covers all aspects of the discovery of ebXML related resources. A *Trading Partner* who has implemented an ebXML *Business Service Interface* can now begin the process of discovery and retrieval (Figure 6 below). One possible discovery method may be to request the *Collaboration Protocol Profile* of another *Trading Partner*. Requests for updates to *Core Libraries*, *Business Libraries* and updated or new *Business Process and Information Meta Models* SHOULD be supported

Copyright © ebXML 2001. All Rights Reserved.

by an ebXML *Business Service Interface*. This is the phase where *Trading Partners* discover the meaning of business information being requested by other *Trading Partners.*

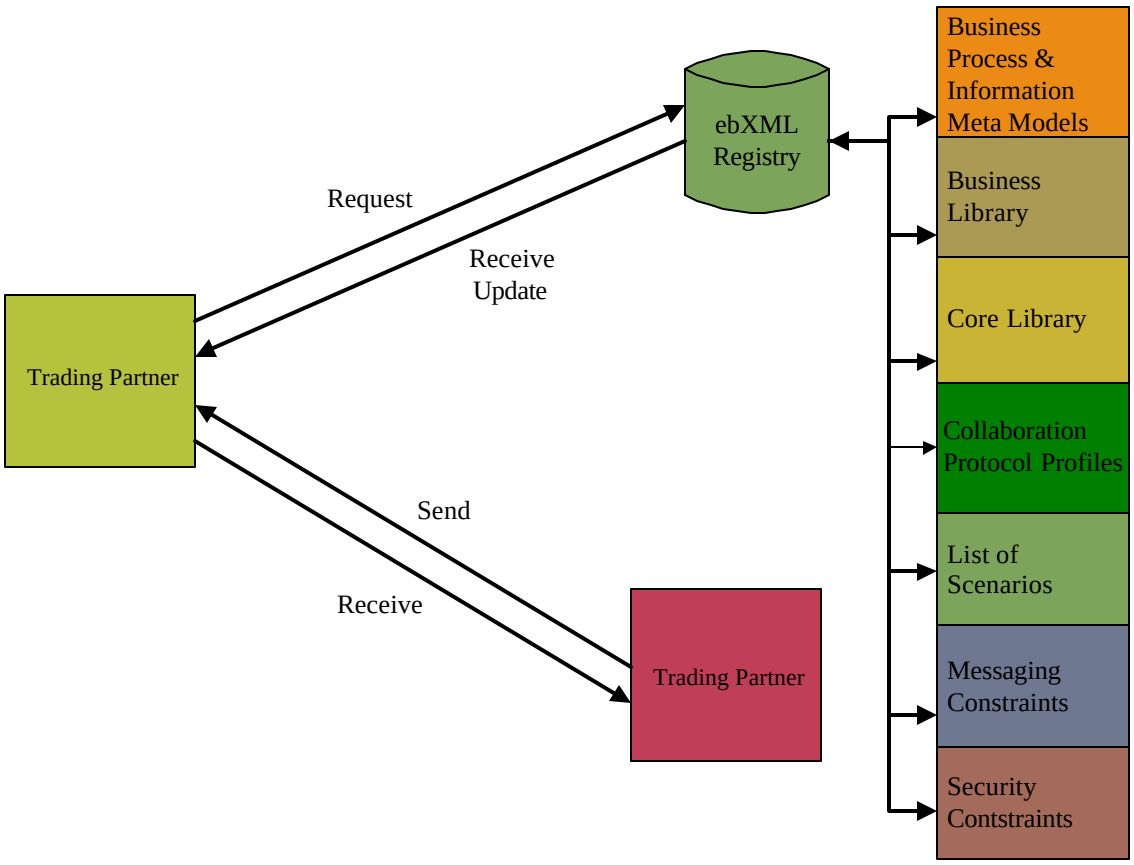


Figure 6 - *Functional Service View: Discovery and Retrieval Phase*

7.3 Run Time Phase

The run time phase covers the execution of an ebXML scenario with the actual associated ebXML transactions. In the Run Time Phase, ebXML *Messages* are being exchanged between *Trading Partners* utilizing the ebXML *Messaging Service.*

For example, an ebXML *CPA* is a choreographed set of business *Message* exchanges linked together by a well-defined choreography using the ebXML *Messaging Service*.

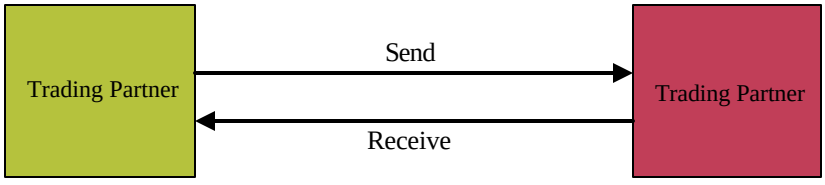


Figure 7 - *Functional Service View: Run Time Phase*

Copyright © ebXML 2001. All Rights Reserved.

[NOTE: There is no run time access to the *Registry*. If it becomes necessary to make calls to the *Registry* during the run time, this SHOULD be considered as a reversion to the Discovery and Retrieval Phase.]

8 ebXML Infrastructure

8.1 Trading Partner Information [CPP and CPA's]

8.1.1 Introduction

To facilitate the process of conducting *eBusiness*, potential *Trading Partners* need a mechanism to publish information about the *Business Processes* they support along with specific technology implementation details about their capabilities for exchanging business information. This is accomplished through the use of a *Collaboration Protocol Profile (CPP)*. The *CPP* is a document which allows a *Trading Partner* to express their supported *Business Processes* and *Business Service Interface* requirements in a manner where they can be universally understood by other ebXML compliant *Trading Partners*.

A special business agreement called a *CPA* is derived from the intersection of two or more *CPP's*. The *CPA* serves as a formal handshake between two or more *Trading Partners* wishing to conduct business transactions using ebXML.

8.1.2 CPP Formal Functionality

The *CPP* describes the specific capabilities that a *Trading Partner* supports as well as the *Service Interface* requirements that need to be met in order to exchange business documents with that *Trading Partner*. The *CPP* contains essential information about the *Trading Partner* including, but not limited to: contact information, industry classification, supported *Business Processes*, *Interface* requirements and *Messaging Service* requirements. *CPP's* MAY also contain security and other implementation specific details. Each ebXML compliant *Trading Partner* SHOULD register their *CPP(s)* in an ebXML compliant *Registry Service*, thus providing a discovery mechanism that allows *Trading Partners* to (1) find one another, (2) discover the *Business Process* that other *Trading Partners* support.

The *CPP* definition SHALL provide for unambiguous selection of choices in all instances where there may be multiple selections (e.g. HTTP or SMTP transport).

8.1.3 CPA Formal Functionality

A *Collaboration Protocol Agreement* (*CPA*) is a document that represents the intersection of two *CPP's* and is mutually agreed upon by both Trading Partners who wish to conduct *eBusiness* using ebXML.

A *CPA* describes: (1) the *Messaging Service* and (2) the *Business Process* requirements that are agreed upon by two or more *Trading Partners*. Conceptually, ebXML supports a three level view of narrowing subsets to arrive at *CPA's* for transacting *eBusiness*. The

Copyright © ebXML 2001. All Rights Reserved.

outer-most scope relates to all of the capabilities that a *Trading Partner* can support, with a subset of what a *Trading Partner* "will" actually support.

A *CPA* contains the *Messaging Service Interface* requirements as well as the implementation details pertaining to the mutually agreed upon *Business Processes* that both *Trading Partners* agree to use to conduct *eBusiness*. *Trading Partners* may decide to register their *CPA's* in an ebXML compliant *Registry Service*, but this is not a mandatory part of the *CPA* creation process.

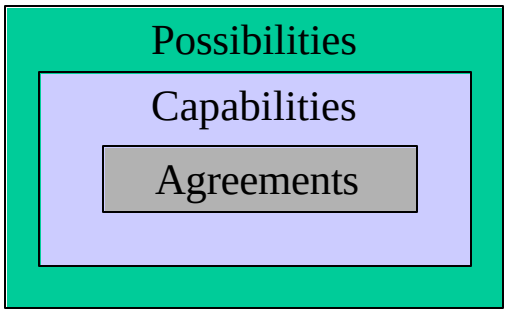


Figure 8 - *Three level view of CPA's*

Business Collaborations are the first order of support that can be claimed by ebXML *Trading Partners*. This "claiming of support" for specific *Business Collaborations* is facilitated by a distinct profile defined specifically for publishing, or advertising in a directory service, such as an ebXML *Registry* or other available service. Figure 9 below outlines the scope for *Collaboration Protocol Agreements* within ebXML.

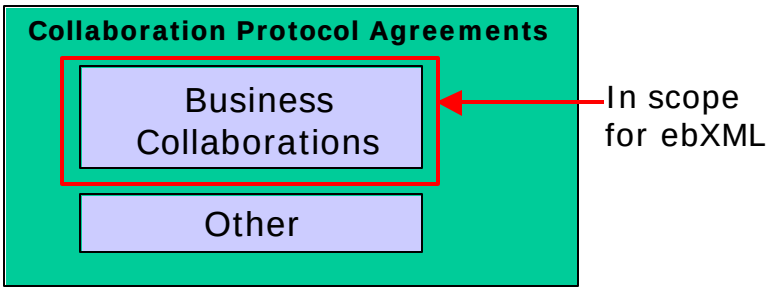


Figure 9 - *Scope for CPA's*

The *CPA-CPP* specification includes a non-normative appendix that discusses *CPA* composition and negotiation and includes advice as to composition and negotiation procedures.

8.1.4 CPP Interfaces

Interface to Business Processes

Copyright © ebXML 2001. All Rights Reserved.

A *CPP* SHALL be capable of referencing one or more *Business Processes* supported by the *Trading Partner* owning the *CPP* instance. The *CPP* SHALL reference the Roles within a *Business Process* that the user is capable of assuming. An example of a Role could be the notion of a "Seller" and "Buyer" within a "Purchasing" *Business Process.*

The *CPP* SHALL be capable of being stored and retrieved from an ebXML *Registry* Mechanism

A *CPP* SHOULD also describe binding details that are used to build an ebXML *Message Heade*r.

8.1.5 CPA Interfaces

A *CPA* governs the *Business Service Interface* used by a *Trading* Partner to constrain the *Business Service Interface* to a set of parameters agreed to by all *Trading Partners* who will execute such an agreement.

CPA's have *Interface*s to *CPP's* in that the *CPA* is derived through a process of mutual negotiation narrowing the *Trading Partners* capabilities (*CPP*) into what the *Trading Partner* "will" do (*CPA*).

A *CPA* must reference to a specific *Business Process* and the interaction requirements needed to execute that *Business Process*.

A *CPA* MAY be stored in a *Registry* mechanism, hence an implied ability to be stored and retrieved is present.

8.1.6 Non-Normative Implementation Details [CPP and CPA's]

A *CPA* is negotiated after the Discovery and Retrieval Phase and is essentially a snapshot of the *Messaging Services* and *Business Process* related information that two or more *Trading Partners* agree to use to exchange business information. If any parameters contained within an accepted *CPA* change after the agreement has been executed, a new *CPA* SHOULD be negotiated between *Trading Partners*.

In some circumstances there may be a need or desire to describe casual, informal or implied *CPA's*.

An eventual goal of ebXML is to facilitate fully automated *CPA* generation. In order to meet this goal, a formal methodology SHOULD be specified for the *CPA* negotiation process.

Copyright © ebXML 2001. All Rights Reserved.

8.2 Business Process and Information Modeling

8.2.1 Introduction

The ebXML *Business Process and Information Meta Model* is a mechanism that allows *Trading Partners* to capture the details for a specific business scenario using a consistent modeling methodology. A *Business Process* describes in detail how *Trading Partners* take on roles, relationships and responsibilities to facilitate interaction with other *Trading Partners* in shared collaborations. The interaction between roles takes place as a choreographed set of business transactions. Each business transaction is expressed as an exchange of electronic *Business Documents*. *Business Documents* MAY be composed from re-useable *Business Information Objects* (see "Relationships to Core Components" under 8.2.3 "*Interfaces*" below). At a lower level, *Business Processes* can be composed of re-useable *Core Processes*, and *Business Information Objects* can be composed of re-useable *Core Components*.

The ebXML *Business Process and Information Meta Model* supports requirements, analysis and design viewpoints that provide a set of semantics (vocabulary) for each viewpoint and forms the basis of specification of the artifacts that are required to facilitate *Business Process* and information integration and interoperability.

An additional view of the *Meta Model*, the *Specification Schema*, is also provided to support the direct specification of the set of elements required to configure a runtime system in order to execute a set of ebXML business transactions. By drawing out modeling elements from several of the other views, the *Specification Schema* forms a semantic subset of the ebXML *Business Process and Information Meta Model*. The *Specification Schema* is available in two stand-alone representations, a *UML* profile, and a DTD.

The relationship between the ebXML *Business Process and Information Meta Model* and the ebXML *Specification Schema* can be shown as follows:

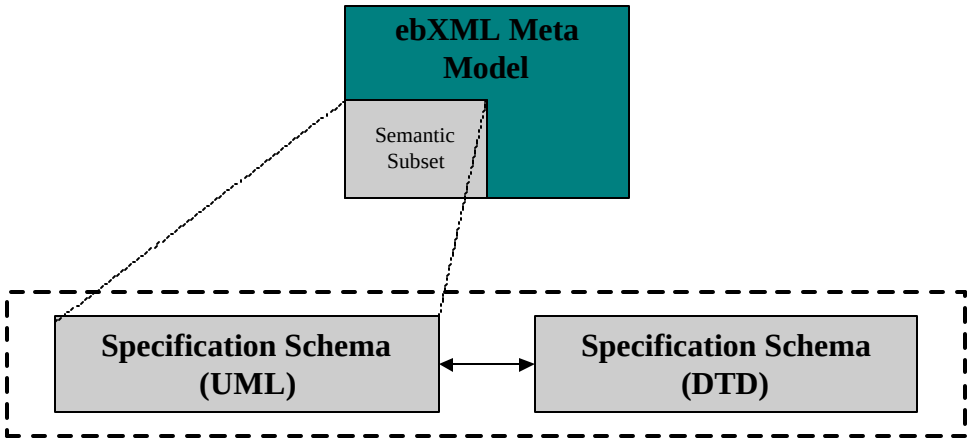


***Figure 10** - ebXML Meta Model - Semantic Subset*

Copyright © ebXML 2001. All Rights Reserved.

The *Specification Schema* supports the specification of business transactions and the choreography of business transactions into *Business Collaborations*. Each *Business Transaction* can be implemented using one of many available standard patterns. These patterns determine the actual exchange of M*essages* and signals between *Trading Partners* to achieve the required electronic transaction. To help specify the patterns *the Specification Schema* is accompanied by a set of standard patterns, and a set of modeling elements common to those patterns. The full specification of a *Business Process* consists of a *Business Process and Information Meta Model* specified against the *Specification Schema* and an identification of the desired pattern(s). This information serves as the primary input for the formation of *Collaboration Protocol Profiles* (*CPP's*) and *CPA's*. This can be shown as follows:

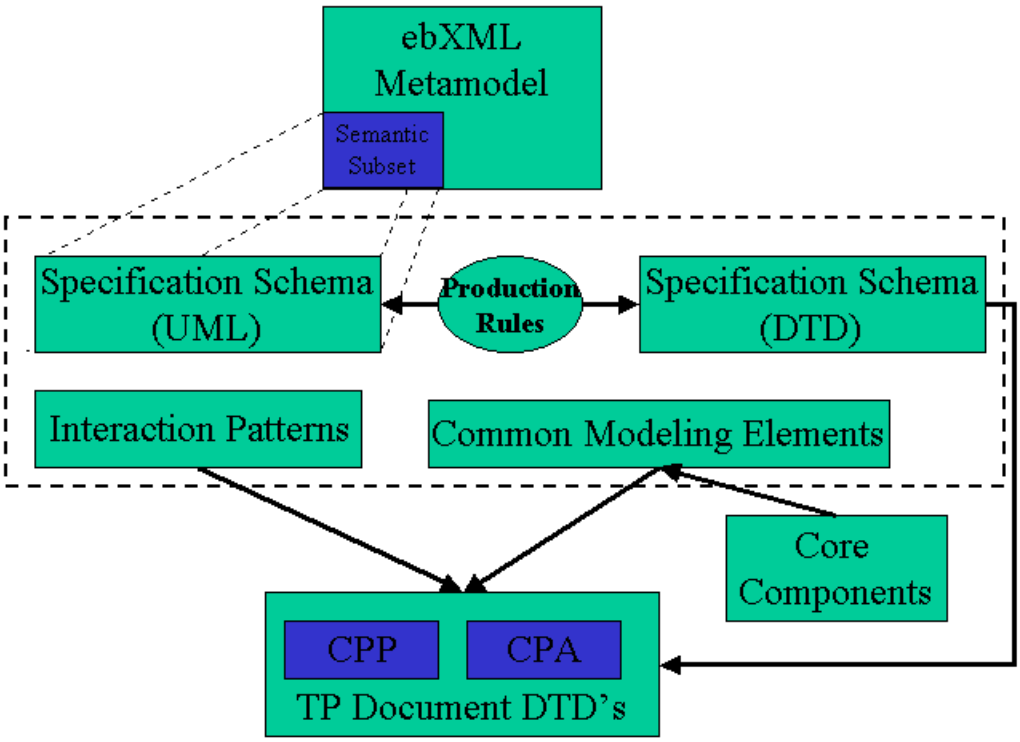


***Figure 11** - ebXML Meta Model*

There are no formal requirements to mandate the use of a modeling language to compose new *Business Processes*, however, if a modeling language is used to develop *Business Processes*, it SHALL be the *Unified Modeling Language (UML)*. This mandate ensures that a single, consistent modeling methodology is used to create new *Business Processes.* One of the key benefits of using a single consistent modeling methodology is that it is possible to compare models to avoid duplication of existing *Business Processes*.

Copyright © ebXML 2001. All Rights Reserved.

To further facilitate the creation of consistent *Business Processes* and information models, ebXML will define a common set of *Business Processes* in parallel with a *Core Library*. It is possible that users of the ebXML infrastructure may wish to extend this set or use their own *Business Processes.*

8.2.2 Formal Functionality

The representation of a *Business Process* document instance SHALL be in a form that will allow both humans and applications to read the information. This is necessary to facilitate a gradual transition to full automation of business interactions.

The *Business Process* SHALL be storable and retrievable in a *Registry* mechanism. *Business Processes* MAY be registered in an ebXML *Registry* in order to facilitate discovery and retrieval.

To be understood by an application, a *Business Process* SHALL be expressible in *XML* syntax. A *Business Process* MAY be constructed as an *Business Process and Information Meta Model* or an *XML* representation of that model. *Business Processes* are capable of expressing the following types of information:

- Choreography for the exchange of document instances. (e.g. the choreography of necessary *Message* exchanges between two Trading Partners executing a "Purchasing" ebXML transaction.)
- References to *Business Process and Information Meta Model* or *Business Documents* (possibly *DTD's* or *Schemas*) that add structure to business data.
- Definition of the roles for each participant in a *Business Process.*

A *Business Process*:

- Provides the contextual constraints for using *Core Components*
- Provides the framework for establishing *CPAs*
- Specifies the domain owner of a *Business Process*, along with relevant contact information.

  [NOTE: the above lists are not inclusive.]

8.2.3 Interfaces

Relationship to CPP and CPA

The *CPP* instance of a *Trading Partner* defines that partner's functional and technical capability to support zero, one, or more *Business Processes* and one or more roles in each process.

The agreement between two *Trading Partners* defines the actual conditions under which the two partners will conduct business transactions together. The *Interface* between the *Business Process*, its *Information Meta Model*, and the *CPA* is the part of the *Business Process* document. This MAY be instantiated as an *XML* document representing the business transactional and collaboration layers of the *Business Process and Information Meta Model*. The expression of the sequence of commercial transactions in *XML* is shared between the *Business Process* and *Trading Partner Information* models.

Copyright © ebXML 2001. All Rights Reserved.

Relationship to Core Components

A *Business Process* instance SHOULD specify the constraints for exchanging business data with other *Trading Partners*. The business information MAY be comprised of components of the ebXML *Core Library*. A *Business Process* document SHALL reference the *Core Components* directly or indirectly using a *XML* document that references the appropriate Business and Information Models and/or Business Documents (possibly DTD's or Schemas). The mechanism for interfacing with the *Core Components* and *Core Library* SHALL be by way of a unique identifier for each component.

Relationship to ebXML Messaging

A *Business Process* instance SHALL be capable of being transported from a *Registry Service* to another *Registry Service* via an ebXML *Message*. It SHALL also be capable of being transported between a *Registry* and a users application via the ebXML *Messaging Service*.

Relationship to a Registry System

A *Business Process* instance intended for use within the ebXML infrastructure SHALL be retrievable through a *Registry* query, and therefore, each *Business Process* SHALL contain a unique identifier.

8.2.4 Non-Normative Implementation Details

The exact composition of *Business Information Objects* or a *Business Document* is guided by a set of contexts derived from the *Business Process.* The modeling layer of the architecture is highlighted in green in Figure 12 below.

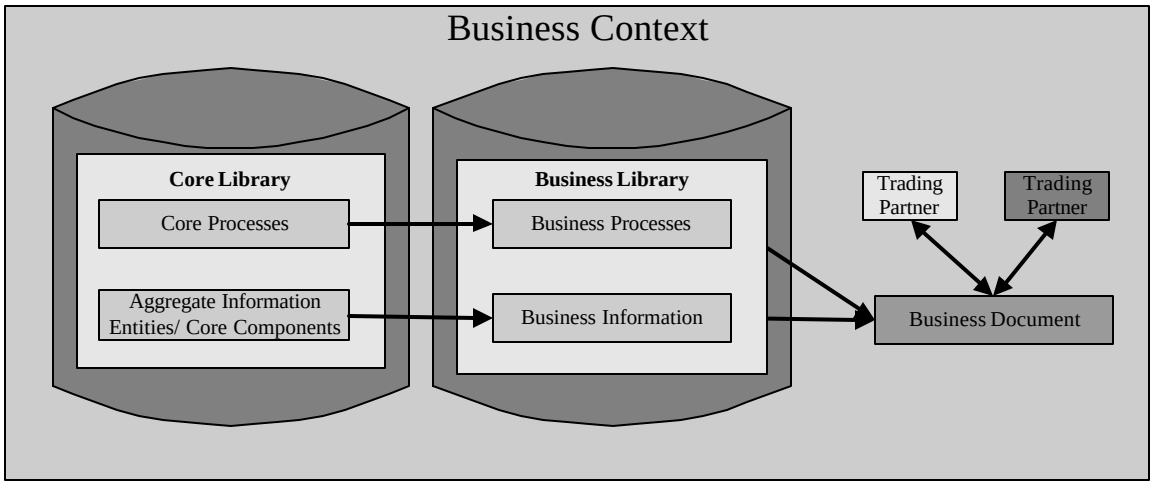


Figure 12 – *ebXML Business Process and Information Modeling layer*

ebXML *Business Process and Information Meta Model* MAY be created following the recommended UN/CEFACT *Modeling Methodology* (*UMM*), or MAY be arrived at in any other way, as long as they comply with the ebXML *Business Process and Information Meta Model.*

Copyright © ebXML 2001. All Rights Reserved.

8.3 Core Components and Core Library Functionality

8.3.1 Introduction

A *Core Component* captures information about a real world business concept, and the relationships between that concept, other *Business Information Objects,* and a contextual description that describes how a *Core* or *Aggregate Information Entity* may be used in a particular ebXML *eBusiness* scenario.

A *Core Component* can be either an individual piece of business information, or a natural "go-together" family of *Business Information Objects* that may be assembled into *Aggregate Information Entities*.

The ebXML Core Components project team SHALL define an initial set of *Core Components.* ebXML users may adopt and/or extend components from the ebXML *Core Library.*

8.3.2 Formal Functionality

As a minimum set of requirements, *Core Components* SHALL facilitate the following functionality:

Core Components SHALL be storable and retrievable using an ebXML *Registry* Mechanism.

Core Components SHALL capture and hold a minimal set of information to satisfy *eBusiness* needs.

Core Components SHALL be capable of being expressed in *XML* syntax.

A *Core Component* SHALL be capable of containing:

- Another *Core Component* in combination with one or more individual pieces of *Business Information Objects.*
- Other *Core Components* in combination with zero or more individual pieces of *Business Information Objects.*

A *Core Component* SHALL be able to be uniquely identified.

8.3.3 Interfaces

A *Core Component* MAY be referenced indirectly or directly from a *Business Document* instance. The *Business Process* MAY specify a single or group of *Core Components* as required or optional information as part of a *Business Document* instance.

Copyright © ebXML 2001. All Rights Reserved.

A *Core Component* SHALL interface with a *Registry* mechanism by way of being storable and retrievable in such a mechanism.

A *Core Component* MAY interface with an *XML* Element from another *XML* vocabulary by the fact it is bilaterally or unilaterally referenced as a semantic equivalent.

8.3.4 Non-Normative Implementation Details

A *Core Component* MAY contain attribute(s) or be part of another *Core Component*, thus specifying the precise context or combination of contexts in which it is used.

The process of aggregating *Core Components* for a specific business context, shall include a means to identify the placement of a *Core Component* within another *Core Component*. It MAY also be a combination of structural contexts to facilitate *Core Component* re-use at different layers within another *Core Component* or *Aggregate Information Entity*. This is referred to as *Business Context*.

Context MAY also be defined using the *Business Process and Information Meta Model*, which defines the instances of *Business Information Objects* in which the *Core Component* occurs.

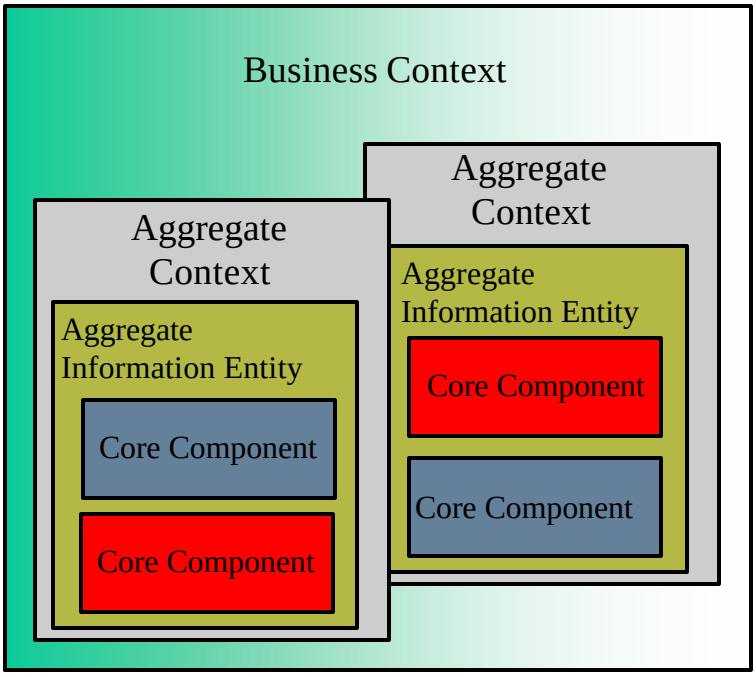


Figure 13 - *Business Context defined in terms of Aggregate Context, Aggregate Information Entities, and Core Components*

The pieces of *Business Information Objects*, or *Core Components*, within a generic *Core Component* may be either mandatory, or optional. A *Core Component* in a specific

Copyright © ebXML 2001. All Rights Reserved.

context or combination of contexts (aggregate or business context) may alter the fundamental mandatory/optional cardinality.

8.4 Registry Functionality

8.4.1 Introduction

An ebXML *Registry* provides a set of services that enable the sharing of information between *Trading Partners*. A *Registry* is a component that maintains an interface to metadata for a registered item. Access to an ebXML *Registry* is provided through *Interfaces* (APIs) exposed by *Registry Services*.

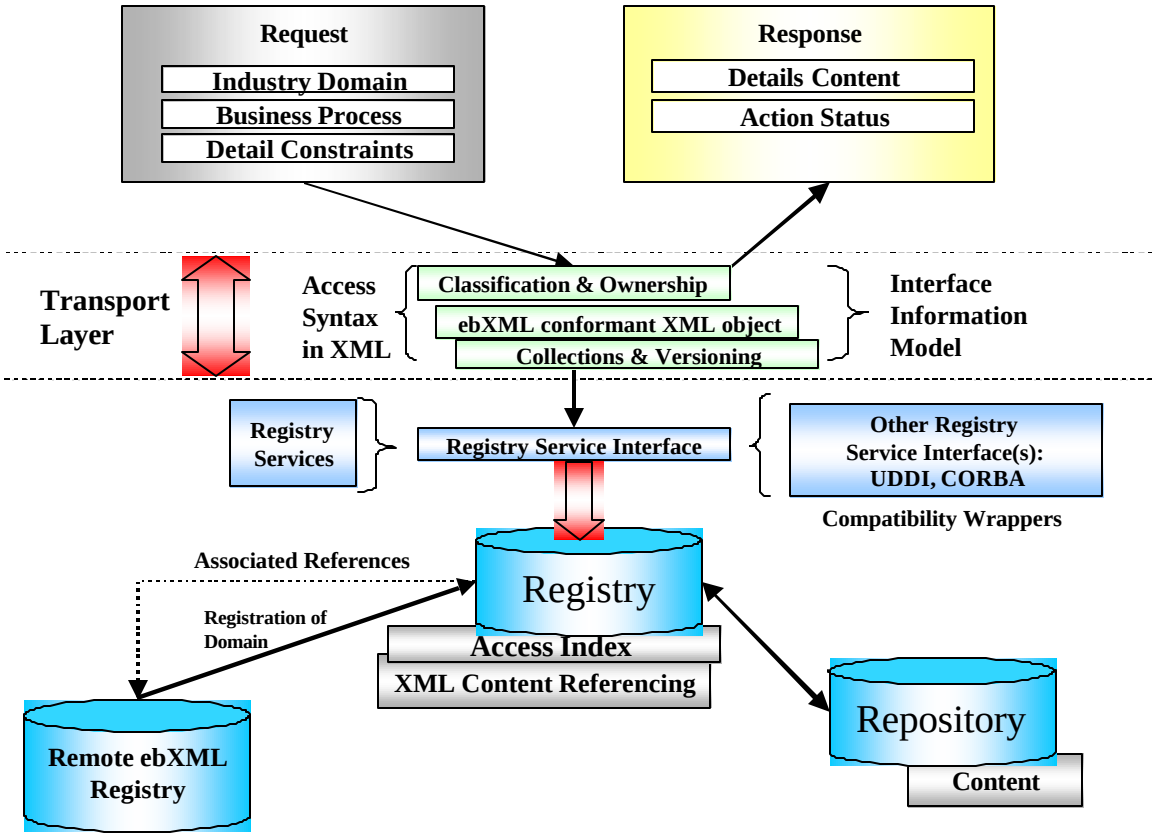


***Figure 14** - Overall Registry Architecture.*

8.4.2 Formal Functionality

A *Registry* SHALL accommodate the storage of items expressed in syntax using multi-byte character sets.

Each *Registry Item*, at each level of granularity as defined by the *Submitting Organization*, MUST be uniquely identifiable. This is essential to facilitate application-to-Registry queries.

Copyright © ebXML 2001. All Rights Reserved.

A *Registry* SHALL return either zero or one positive matches in response to a contextual query for a unique identifier. In such cases where two or more positive results are displayed for such queries, an error message SHOULD be reported to the *Registry Authority.*

A *Registry Item* SHALL be structured to allow information associations that identify, name, describe it, give its administrative and access status, define its persistence and mutability, classify it according to pre-defined classification schemes, declare its file representation type, and identify the submitting and responsible organizations.

The *Registry Interface* serves as an application-to-registry access mechanism. Human-to-registry interactions SHALL be built as a layer over a *Registry Interface* (e.g. a Web browser) and not as a separate *Interface*.

The *Registry Interface* SHALL be designed to be independent of the underlying network protocol stack (e.g. HTTP/SMTP over TCP/IP). Specific instructions on how to interact with the *Registry Interface* MAY be contained in the payload of the ebXML *Message*.

The processes supported by the *Registry* MAY also include:

- A special *CPA* between the *Registry* and *Registry Clients.*
- A set of functional processes involving the *Registry* and *Registry Clients.*
- A set of *Business Messages* exchanged between a *Registry Client* and the *Registry* as part of a specific *Business Process*.
- A set of primitive *Interface* mechanisms to support the *Business Messages* and associated query and response mechanisms.
- A special *CPA* for orchestrating the interaction between ebXML compliant Registries.
- A set of functional processes for *Registry*-to-*Registry* interactions.
- A set of error responses and conditions with remedial actions.

To facilitate the discovery process, browse and drill down queries MAY be used for human interactions with a *Registry* (e.g. via a Web browser). A user SHOULD be able to browse and traverse the content based on the available *Registry* classification schemes.

Registry Services exist to create, modify, and delete *Registry Items* and their metadata.

Appropriate security protocols MAY be deployed to offer authentication and protection for the *Repository* when accessed by the *Registry*.

Unique Identifiers (*UIDs*) SHALL be assigned to all items within an ebXML *Registry System*. *UID* keys are REQUIRED references for all ebXML content. *Universally Unique Identifiers (UUIDs)* MAY be used to ensure that *Registry* entries are truly globally unique, and thus when systems query a *Registry* for a *UUID*, one and only one result SHALL be retrieved.

Copyright © ebXML 2001. All Rights Reserved.

To facilitate semantic recognition of *Business Process and Information Meta Models*, the *Registry Service* SHALL provide a mechanism for incorporating human readable descriptions of *Registry* items. Existing *Business Process and Information Meta Models* (e.g. RosettaNet PIPs) and *Core Components* SHALL be assigned *UID* keys when they are registered in an ebXML compliant *Registry Service*. These *UID* keys MAY be implemented in physical *XML* syntax in a variety of ways. These mechanisms MAY include, but are not limited to:

- A pure explicit reference mechanism (example: URN:*UID* method),
- A referential method (example: URI:*UID* / namespace:*UID*),
- An object-based reference compatible with W3C Schema (*example* URN:complextype name), and
- A datatype based reference (example: ISO 8601:2000 Date/Time/Number datatyping and then legacy datatyping).

Components in ebXML MUST facilitate multilingual support. A *UID* reference is particularly important here as it provides a language neutral reference mechanism. To enable multilingual support, the ebXML specification SHALL be compliant with Unicode and ISO/IEC 10646 for character set and UTF-8 or UTF-16 for character encoding.

8.4.3 Interfaces

ebXML Messaging:
The query syntax used by the *Registry* access mechanisms is independent of the physical implementation of the backend system.

The ebXML *Messaging Service* MAY serve as the transport mechanism for all communication into and out of the *Registry*.

Business Process:
Business Processes are published and retrieved via ebXML *Registry Services*.

Core Components:
Core Components are published and retrieved via ebXML *Registry Services*.

Any item with metadata: *XML* elements provide standard metadata about the item being managed through ebXML *Registry Services*. Since ebXML Registries are distributed each *Registry* MAY interact with and cross-reference another ebXML *Registry*.

8.4.4 Non-Normative Implementation Details

The *Business Process and Information Meta Models* within a *Registry* MAY be stored according to various classification schemes.

The existing ISO11179/3 work on *Registry* implementations MAY be used to provide a model for the ebXML *Registry* implementation.

Copyright © ebXML 2001. All Rights Reserved.

Registry Items and their metadata MAY also be addressable as *XML* based URI references using only HTTP for direct access.

Examples of extended *Registry Services* functionality may be deferred to a subsequent phase of the ebXML initiative. This includes, but is not limited to transformation services, workflow services, quality assurance services and extended security mechanisms.

A *Registry Service* MAY have multiple deployment models as long as the *Registry Interfaces* are ebXML compliant.

The *Business Process and Information Meta Model* for an ebXML *Registry Service* may be an extension of the existing OASIS Registry/Repository Technical Specification, specifically tailored for the storage and retrieval of business information, whereas the OASIS model is a superset designed for handling extended and generic information content.

8.5 Messaging Service Functionality

8.5.1 Introduction

The ebXML *Message Service* mechanism provides a standard way to exchange business M*essages* among ebXML *Trading Partners*. The ebXML *Messaging Service* provides a reliable means to exchange business M*essages* without relying on proprietary technologies and solutions. An ebXML *Message* contains structures for a *Message Header* (necessary for routing and delivery) and a *Payload* section.

The ebXML *Messaging Service* is conceptually broken down into three parts: (1) an abstract *Service Interface*, (2) functions provided by the *Messaging Service Layer*, and (3) the mapping to underlying transport service(s). The relation of the abstract *Interface*, *Messaging Service Layer*, and transport service(s) are shown in Figure 15 below.

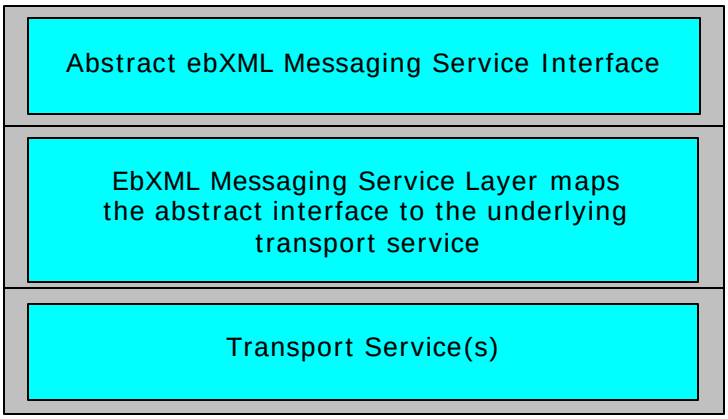


***Figure 15** - ebXML Messaging Service*

Copyright © ebXML 2001. All Rights Reserved.

The following diagram depicts a logical arrangement of the functional modules that exist within the ebXML *Messaging Services* architecture. These modules are arranged in a manner to indicate their inter-relationships and dependencies. This architecture diagram illustrates the flexibility of the ebXML *Messaging Service*, reflecting the broad spectrum of services and functionality that may be implemented in an ebXML system.

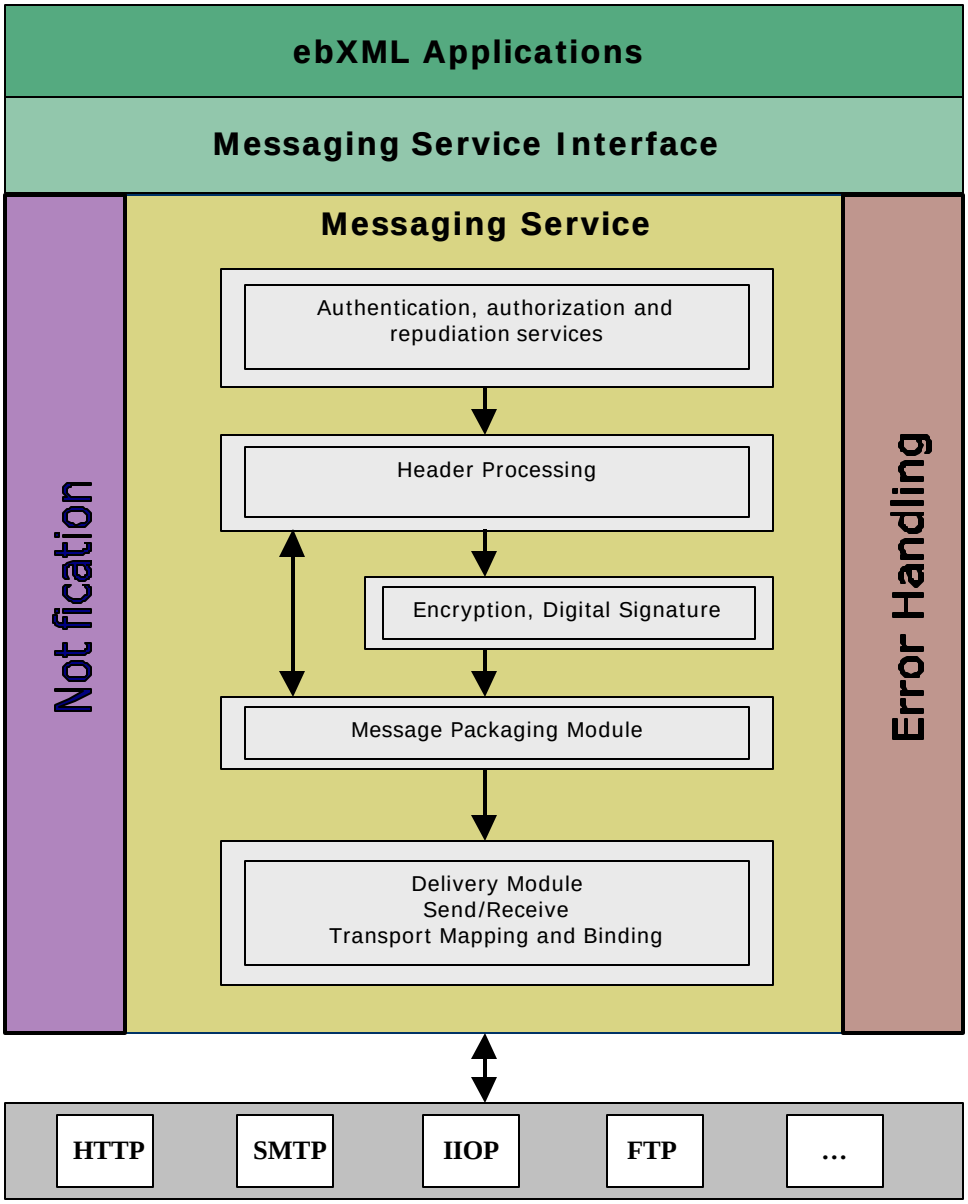


Figure 16 *- The Messaging Service Architecture*

Copyright © ebXML 2001. All Rights Reserved.

8.5.2 Formal Functionality

The ebXML *Messaging Service* provides a secure, consistent and reliable mechanism to exchange ebXML *Messages* between users of the ebXML infrastructure over various transport *Protocols* (possible examples include SMTP, HTTP/S, FTP, etc.).

The ebXML *Messaging Service* prescribes formats for all *Messages* between distributed ebXML *Components* including *Registry* mechanisms and compliant user *Applications*.

The ebXML *Messaging Service* does not place any restrictions on the content of the payload.

The ebXML *Messaging Service* supports simplex (one-way) and request/response (either synchronous or asynchronous) *Message* exchanges.

The ebXML *Messaging Service* supports sequencing of payloads in instances where multiple payloads or multiple *Messages* are exchanged between *Trading Partners*.

The ebXML *Messaging Service Layer* enforces the "rules of engagement" as defined by two *Trading Partners* in a *Collaboration Protocol Agreement* (including, but not limited to security and *Business Process* functions related to *Message* delivery). The *Collaboration Protocol Agreement* defines the acceptable behavior by which each *Trading Partner* agrees to abide. The definition of these ground rules can take many forms including formal *Collaboration Protocol Agreements*, interactive agreements established at the time a business transaction occurs (e.g. buying a book online), or other forms of agreement. There are *Messaging Service Layer* functions that enforce these ground rules. Any violation of the ground rules result in an error condition, which is reported using the appropriate means.

The ebXML *Messaging Service* performs all security related functions including:

- Identification
- Authentication (verification of identity)
- Authorization (access controls)
- Privacy (encryption)
- Integrity (message signing)
- Non-repudiation
- Logging

8.5.3 Interfaces

The ebXML *Messaging Service* provides ebXML with an abstract *Interface* whose functions, at an abstract level, include:

- Send – send an ebXML *Message* – values for the parameters are derived from the ebXML *Message Headers*.
- Receive – indicates willingness to receive an ebXML *Message*.
- Notify – provides notification of expected and unexpected events.

Copyright © ebXML 2001. All Rights Reserved.

- Inquire – provides a method of querying the status of the particular ebXML *Message* interchange.

The ebXML *Messaging Service* SHALL interface with internal systems including:

- Routing of received *Messages* to internal systems
- Error notification

The ebXML *Messaging Service* SHALL help facilitate the *Interface* to an ebXML *Registry*.

8.5.4 Non-Normative Implementation Details

ebXML Message Structure and Packaging

Figure 17 below illustrates the logical structure of an ebXML *Message*.

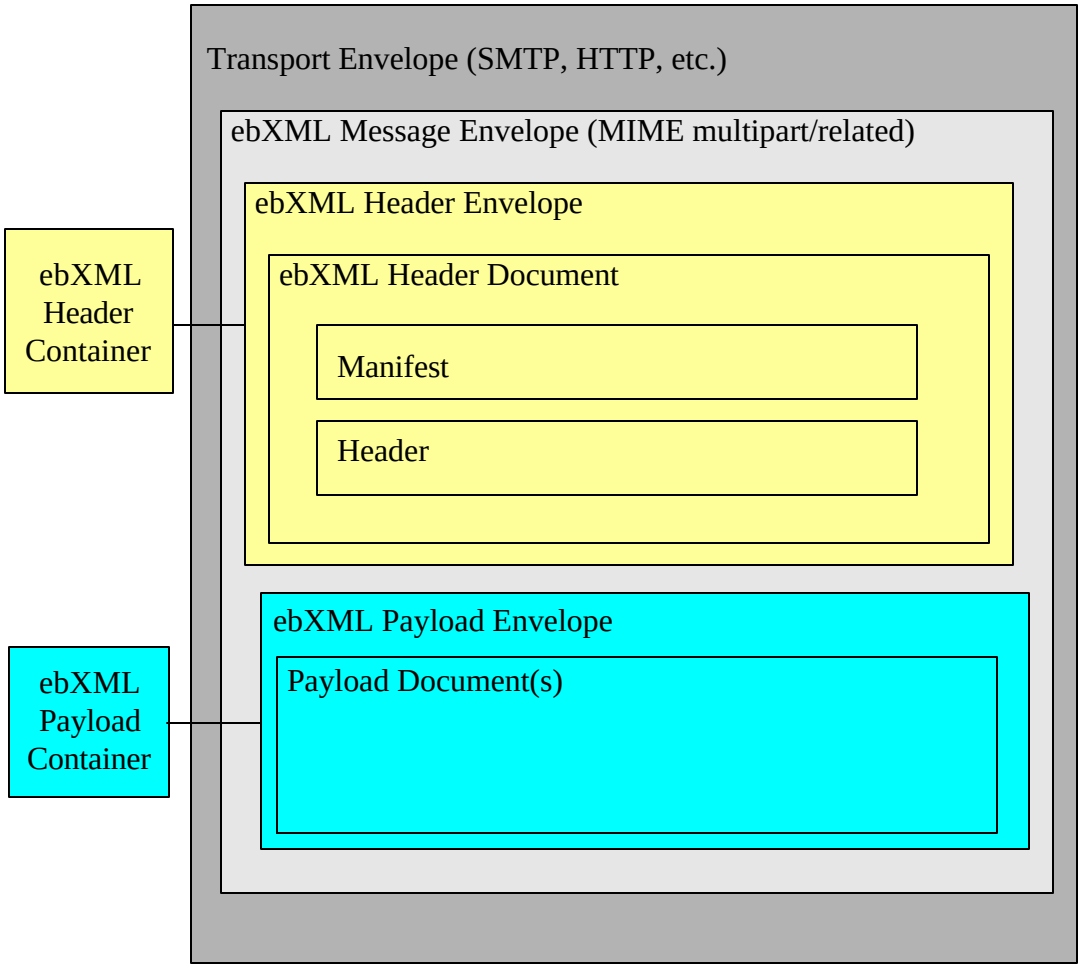


Figure 17 - *ebXML Message Structure*

Copyright © ebXML 2001. All Rights Reserved.

An ebXML *Message* consists of an optional transport *Protocol* specific outer *Communication Protocol Envelope* and a *Protocol* independent ebXML *Message Envelope*. The ebXML *Message Envelope* is packaged using the MIME multipart/related content type. MIME is used as a packaging solution because of the diverse nature of information exchanged between *Partners* in *eBusiness* environments. For example, a complex *Business Transaction* between two or more *Trading Partners* might require a payload that contains an array of business documents (*XML* or other document formats), binary images, or other related Business Information.

9 Conformance

9.1 Introduction

This clause specifies the general framework, concepts and criteria for *Conformance* to ebXML, including an overview of the conformance strategy for ebXML, guidance for addressing conformance in each ebXML technical specification, and the conformance clause specific to the Technical Architecture specification. Except for the Technical Architecture Specification, this clause does not define the conformance requirements for each of the ebXML technical specifications – the latter is the purview of the technical specifications.

The objectives of this section are to:

a) Ensure a common understanding of conformance and what is required to claim conformance to this family of specifications;
b) Ensure that conformance is consistently addressed in each of the component specifications;
c) Promote interoperability and open interchange of *Business Processes* and *Messages*;
d) Encourage the use of applicable conformance test suites as well as promote uniformity in the development of conformance test suites.

Conformance to ebXML is defined in terms of conformance to the ebXML infrastructure and conformance to each of the technical specifications for ebXML. The primary purpose of conformance to ebXML is to increase the probability of successful interoperability between implementations and the open interchange of *XML* business documents and *Messages*. Successful interoperability and open interchange is more likely to be achieved if implementations conform to the requirements in the ebXML specifications.

9.2 Conformance to ebXML

ebXML Conformance is defined as conformance to an ebXML system that is comprised of all the architectural components of the ebXML infrastructure and satisfies at least the minimum conformance requirements for each of the ebXML technical specifications,

Copyright © ebXML 2001. All Rights Reserved.

including the functional and *Interface* requirements in this Technical Architecture specification.

In the context of ebXML, an implementation is said to exhibit conformance if it complies with the requirements of each applicable ebXML technical specification. The conformance requirements are stated in the conformance clause of each technical specification of ebXML. The conformance clause specifies explicitly all the requirements that have to be satisfied to claim conformance to that specification. These requirements MAY be applied and grouped at varying levels within each specification.

9.3 Conformance to the Technical Architecture Specification

This section details the conformance requirements for claiming conformance to the Technical Architecture specification.

In order to conform to this specification, each ebXML technical specification:

a) SHALL support all the functional and *Interface* requirements defined in this specification that are applicable to that technical specification;
b) SHALL NOT specify any requirements that would contradict or cause non-conformance to ebXML or any of its components;
c) MAY contain a conformance clause that adds requirements that are more specific and limited in scope than the requirements in this specification;
d) SHALL only contain requirements that are testable.

A conforming implementation SHALL satisfy the conformance requirements of the applicable parts of this specification and the appropriate technical specification(s).

9.4 General Framework of Conformance Testing

The objective of conformance testing is to determine whether an implementation being tested conforms to the requirements stated in the relative ebXML specification. Conformance testing enables vendors to implement compatible and interoperable systems built on the ebXML foundations. ebXML *Implementations* and *Applications* SHOULD be tested to available test suites to verify their conformance to ebXML Specifications as soon as test suites are available.

Publicly available test suites from vendor neutral organizations such as OASIS and NIST SHOULD be used to verify the conformance of ebXML *Implementations, Applications,* and *Components* claiming conformance to ebXML. Open source reference implementations MAY be available to allow vendors to test their products for *Interface* compatibility, conformance, and interoperability.

10.0 Security Considerations

Copyright © ebXML 2001. All Rights Reserved.

10.1 Introduction

A comprehensive *Security Model* for ebXML will be expressed in a separate document. The *Security Model* will be applied to the entire *ebXML Infrastructure*, with the underlying goal of best meeting the needs of users of ebXML.

The Security Model will comply with security needs specified in the ebXML Requirements Document.

Disclaimer

The views and specification expressed in this document are those of the authors and are not necessarily those of their employers. The authors and their employers specifically disclaim responsibility for any problems arising from correct or incorrect implementation or use of this design.

Copyright Statement

Copyright © ebXML 2001. All Rights Reserved.

This document and translations of it MAY be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation MAY be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself MAY not be modified in any way, such as by removing the copyright notice or references to ebXML, UN/CEFACT, or OASIS, except as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by ebXML or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and ebXML DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright © ebXML 2001. All Rights Reserved.

Appendix A: Example ebXML Business Scenarios

Definition

This set of scenarios defines how ebXML compliant software could be used to implement popular, well-known *eBusiness* models.

Scope

These scenarios are oriented to properly position the ebXML specifications as a convenient mean for companies to properly run electronic business over the Internet using open standards. They bridge the specifications to real life uses.

Audience

Companies planning to use ebXML compliant software will benefit from these scenarios because they will show how these companies may be able to implement popular business scenarios onto the ebXML specifications.

List

a) Two *Trading Partners* set-up an agreement and run the associated electronic exchange.
b) Three or more *Trading Partners* set-up a *Business Process* implementing a supply-chain and run the associated exchanges
c) A Company sets up a Portal that defines a *Business Process* involving the use of external business services.
d) Three or more *Trading Partners* conduct business using shared *Business Processes* and run the associated exchanges.

Scenario 1 : Two Trading Partners set-up an agreement and run the associated exchange

In this scenario:

- Each *Trading Partner* defines its own Profile (*CPP*).
  Each Profile references:
  - One or more existing *Business Process* found in the ebXML *Registry*
  - One of more *Message* Definitions. Each *Message* definition is built from reusable components (*Core Components*) found in the ebXML *Registry*

  Each Profile (*CPP*) defines:
  - The business transactions that the *Trading Partner* is able to engage into
  - The Technical protocol (like HTPP, SMTP etc) and the technical properties (such as special encryption, validation, authentication) that the *Trading Partner* supports in the engagement
  - The *Trading Partners* acknowledge each other profile and create a *CPA*.
- The *Trading Partners* implement the respective part of the Profile. This is done:
  - Either by creating/configuring a *Business Service Interface*.
  - Or properly upgrading the legacy software running at their side

  In both cases, this step is about :
  - Plugging the Legacy into the ebXML technical infrastructure as specified by the *Messaging Service*.
  - Granting that the software is able to properly engage the stated conversations

Copyright © ebXML 2001. All Rights Reserved.

- Granting that the exchanges semantically conform to the agreed upon *Message* Definitions
- Granting that the exchanges technically conform with the underlying ebXML *Messaging Service.*

- The *Trading Partners* start exchanging *Messages* and performing the agreed upon commercial transactions.

Scenario 2: Three or more parties set-up a Business Process implementing a supply-chain and run the associated exchanges

The simple case of a supply-chain involving two *Trading Partners* can be redefined in terms of the Scenario 1.

Here we are dealing with situations where more *Trading Partners* are involved. We consider a supply chain of the following type:

What fundamentally differs from Scenario 1 is that "*Trading Partner* 2" is engaged at the same time with two different *Trading Partners*. The assumption is that the "state" of the local portion of the *Business Process* is managed by each *Trading Partner*, i.e. that each *Trading Partner* is fully responsible of the Business Transaction involving it ("*Trading Partner* 3" only knows about "*Trading Partner* 2", "*Trading Partner* 2" knows about "*Trading Partner* 3" and "*Trading Partner* 1", "*Trading Partner* 1" knows about "*Trading Partner* 2").

In this scenario:

- Each *Trading Partner* defines its own Profile (*CPP*). Each Profile (*CPP*) references:
  - One or more existing *Business Process* found in the ebXML *Registry*
  - One of more *Message* Definitions. Each *Message* definition is built from reusable components (*Core Components*) found in the ebXML *Registry*

  Each Profile (*CPP*) defines:
  - The Commercial Transactions that the *Trading Partner* is able to engage into. "*Trading Partner* 2" must be able to support at least 2 Commercial Transactions.
  - The Technical protocol (like HTPP, SMTP etc) and the technical properties (such as special encryption, validation, authentication) that the *Trading Partner* supports in the engagement. As to "*Trading Partner* 2", the technical requirements for the exchanges with "*Trading Partner* 1" and "*Trading Partner* 3" may be different. In such case, "*Trading Partner* 2" must be able to support different protocols and/or properties.

Copyright © ebXML 2001. All Rights Reserved.

- The *Trading Partners* acknowledge each other profile and create the relevant *CPA*. (at least 2 in this Scenario).
- "*Trading Partner* 2" is engaged in 2 *CPA's*

- The *Trading Partners* implement the respective part of the Profile. This is done:
  - Either by creating/configuring a *Business Service Interface*.
  - Or properly upgrading the legacy software running at their side.

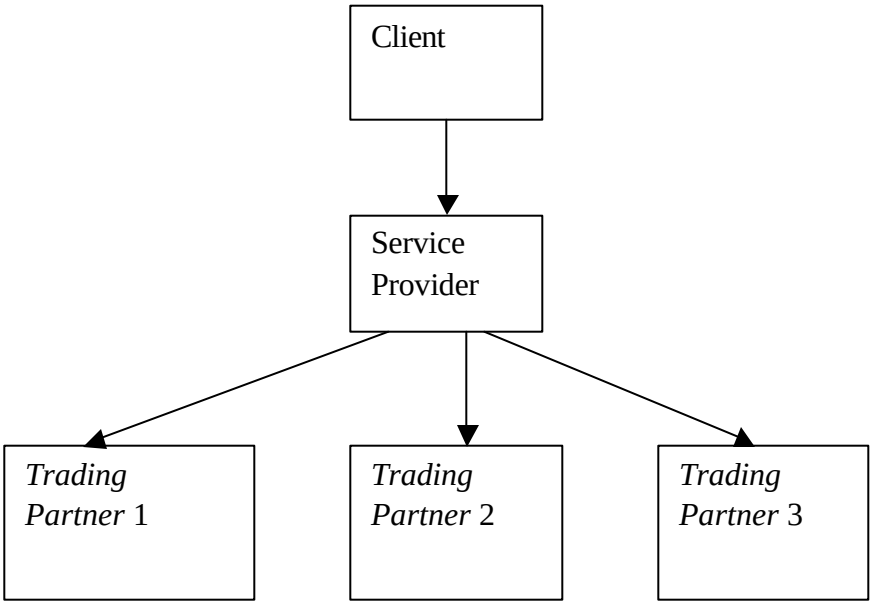
  In both cases, this step is about:
  - Plugging the Legacy into the ebXML technical infrastructure as specified by the *Messaging Service*
  - Granting that the software is able to properly engage the stated conversations
  - Granting that the exchanges semantically conform to the agreed upon ebXML *Message* definitions
  - Granting that the exchanges technically conform with the underlying ebXML *Messaging Service*.
  - "*Trading Partner* 2" may need to implement a complex *Business Service Interface* in order to be able to engage with different *Trading Partners*.
- The *Trading Partners* start exchanging *Messages* and performing the agreed upon commercial transactions.
  - "*Trading Partner* 3" places an order at "*Trading Partner* 2"
  - "*Trading Partner* 2" (eventually) places an order with "*Trading Partner* 1"
  - "*Trading Partner* 1" fulfills the order
  - "*Trading Partner* 2" fulfill the order

Scenario 3 : A Company sets up a Portal which defines a Business Process involving the use of external business services

This is the Scenario describing a Service Provider. A "client" asks the Service Provider for a Service. The Service Provider fulfills the request by properly managing the exchanges with other *Trading Partners* that provide information to build the final answer.

Copyright © ebXML 2001. All Rights Reserved.

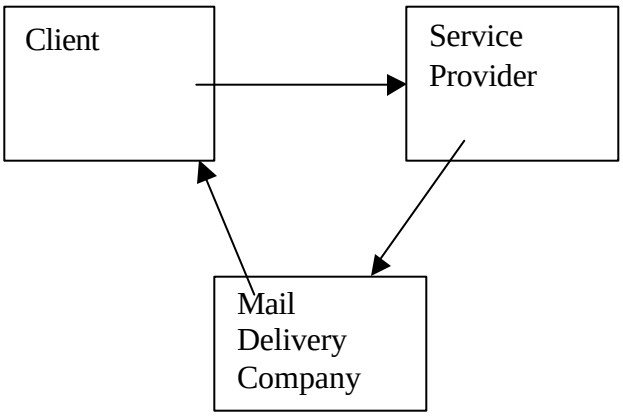
In the simplest case, this Scenario could be modeled as follows :



This is an evolution of Scenario 2. The Description of this scenario is omitted.

Scenario 4: Three or more Trading Partners conduct business using shared Business Processes and run the associated exchanges

This Scenario is about 3 or more *Trading Partners* having complex relationships. An example of this is the use of an external delivery service for delivering goods.



In this Scenario, each *Trading Partner* is involved with more than one other *Trading Partner* but the relationship is not linear. The product or good that is ordered by the *Client* with a Service Provider is delivered by a 3rd party.

Copyright © ebXML 2001. All Rights Reserved.

In this scenario:

- Each *Trading Partner* defines its own Profile (*CPP*). Each Profile (*CPP*) references:
  - One or more existing *Business Process* found in the ebXML *Registry*
  - One of more *Registry* Definitions. Each *Registry* definition is built from reusable components (*Core Components*) found in the ebXML *Registry*

  Each Profile (*CPP*) defines:
  - The Commercial Transactions that the *Trading Partner* is able to engage into. In this case, each *Trading Partner* must be able to support at least 2 Commercial Transactions.
  - The Technical protocol (like HTPP, SMTP etc) and the technical properties (such as special encryption, validation, authentication) that the *Trading Partner* supports in the engagement.
    In case the technical infrastructure underlying the different exchanges differes, each *Trading Partner* must be able to support different protocols and/or properties. (an example is that the order is done through a Web Site and the delivery is under the form of an email).
  - The *Trading Partners* acknowledge each other profile and create a *CPA.* Each *Trading Partner*, in this Scenario, must be able to negotiate at least 2 Agreements.

  Each *Trading Partner* is enagaged in 2 Agreements (*CPA*).
- The *Trading Partners* implement the respective part of the Profile. This is done:
  - Either by creating/configuring a *Business Service Interface*.
  - Or properly upgrading the legacy software running at their side

  In both cases, this step is about:
  - Plugging the Legacy into the ebXML technical infrastructure as specified by the *Messaging Service.*
  - Granting that the software is able to properly engage the stated conversations
  - Granting that the exchanges semantically conform to the agreed upon *Message* definitions.
  - Granting that the exchanges technical conform with the underlying ebXML *Messaging Service.*
  - All *Trading Partners* may need to implement complex *Business Service Interfaces* to accommodate the differences in the *CPA's* with different *Trading Partners*.
- The *Trading Partners* start exchanging *Messages* and performing the agreed upon commercial transactions.
  - The *Client* places an Order at the Service Provider.
  - The Service Provider Acknowledges the Order with The *Client*.
  - The Service Provider informs the Mail Delivery Service about a good to be delivered at the *Client*
  - The Mail Delivery Service delivers the good at the *Client*
  - The *Clients* notifies the Service Provider that the good is received.

Copyright © ebXML 2001. All Rights Reserved.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:eff="http://efficient.citi.tudor.lu">
  - <xsd:simpleType name="Type_Role">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="GestionaireMaquette" />
        <xsd:enumeration value="Maitre ouvrage" />
        <xsd:enumeration value="Architecte" />
        <xsd:enumeration value="IngenieurStabilite" />
        <xsd:enumeration value="Entrepreneur" />
        <xsd:enumeration value="IngenieurEco" />
        <xsd:enumeration value="IngenieurElectricite" />
        <xsd:enumeration value="Consultant" />
        <xsd:enumeration value="Non_Defini" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Statut">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="NonTraité" />
        <xsd:enumeration value="Conforme" />
        <xsd:enumeration value="Accord" />
        <xsd:enumeration value="Rejet" />
        <xsd:enumeration value="Refus" />
        <xsd:enumeration value="EnCours" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Dalle">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Dalle_Etage" />
        <xsd:enumeration value="Dalle_Toiture" />
        <xsd:enumeration value="Dalle_Palier" />
        <xsd:enumeration value="Def_Utilisateur" />
        <xsd:enumeration value="Non_Defini" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Poutre">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Poutre_Classique" />
        <xsd:enumeration value="Solive" />
        <xsd:enumeration value="Linteau" />
        <xsd:enumeration value="Poutre_T" />
        <xsd:enumeration value="Def_Utilisateur" />
        <xsd:enumeration value="Non_Defini" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Colonne">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Colonne_Classique" />
        <xsd:enumeration value="Def_Utilisateur" />
        <xsd:enumeration value="Non_Defini" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Mur">
    - <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Standard" />
        <xsd:enumeration value="Polygonal" />
        <xsd:enumeration value="Section_Var" />
        <xsd:enumeration value="Def_Utilisateur" />
        <xsd:enumeration value="Non-defini" />
      </xsd:restriction>
    </xsd:simpleType>
  - <xsd:simpleType name="Type_Struct_Spatiale">
```

```
  - <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Complex" />
      <xsd:enumeration value="Element" />
      <xsd:enumeration value="Partiel" />
    </xsd:restriction>
  </xsd:simpleType>
- <xsd:simpleType name="Type_Espace">
  - <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Interieur" />
      <xsd:enumeration value="Exterieur" />
      <xsd:enumeration value="Non_Defini" />
    </xsd:restriction>
  </xsd:simpleType>
- <xsd:element name="Objets_propositions_transfert">
  - <xsd:complexType>
    - <xsd:sequence>
        <xsd:element name="Validation" type="Validation" minOccurs="1"
          maxOccurs="1" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
- <xsd:complexType name="Root">
  - <xsd:sequence>
      <xsd:element name="GlobalID" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="NomProjet" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Associe_Class" type="Classificationx" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Associe_Document" type="Documentx" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Associe_Librairie" type="Lib_Objets" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Produit">
  - <xsd:sequence>
      <xsd:element name="Sit_Spaciale" type="xsd:string" minOccurs="0"
        maxOccurs="1" />
      <xsd:element name="Représentation" type="xsd:string" minOccurs="0"
        maxOccurs="1" />
      <xsd:element name="Est_Assigne" type="Objet_Projet" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Objet_Projet" type="Objet_Projet" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Element">
  - <xsd:sequence>
      <xsd:element name="Est_Contenu_Dans_Batiment" type="Batiment"
        minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="Est_Contenu_Dans_Espace" type="Espace" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Est_Contenu_Dans_Etage" type="Etage" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Est_Contenu_Dans_Site" type="Site" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Limite_Spatiale" type="Espace" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Produit" type="Produit" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Composant_Batiment">
```

```
  - <xsd:sequence>
      <xsd:element name="Materiau" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Connecte" type="Element" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Element" type="Element" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Est_Evide_par" type="Elt_Ouverture" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Est_transforme_par" type="Elt_Saillant" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Remplit" type="Elt_Ouverture" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Dalle">
  - <xsd:sequence>
      <xsd:element name="T_Dalle" type="Type_Dalle" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Perimetre_Dalle" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_Dalle" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Volume_Dalle" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Poids_Dalle" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Composant_Batiment" type="Composant_Batiment"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Colonne">
  - <xsd:sequence>
      <xsd:element name="T_Colonne" type="Type_Colonne" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Longueur_Colonne" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Section" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Surface_Dev" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Volume_Colonne" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Poids_Colonne" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Composant_Batiment" type="Composant_Batiment"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Poutre">
  - <xsd:sequence>
      <xsd:element name="T_Poutre" type="Type_Poutre" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Longueur_Poutre" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Section_Colonne" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_Dev" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Volume_Poutre" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Poids_Poutre" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Composant_Batiment" type="Composant_Batiment"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
```

```
  </xsd:complexType>
- <xsd:complexType name="Mur">
  - <xsd:sequence>
      <xsd:element name="T_Mur" type="Type_Mur" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Longueur_Mur" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Section_Mur" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_ElevG" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_ElevD" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Volume_Mur" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Composant_Batiment" type="Composant_Batiment"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Structure_Spatiale">
  - <xsd:sequence>
      <xsd:element name="Composition" type="Type_Struct_Spatiale" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Nom" type="xsd:string" minOccurs="0" maxOccurs="1" />
      <xsd:element name="Produit" type="Produit" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Espace">
  - <xsd:sequence>
      <xsd:element name="T_Espace" type="Type_Espace" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Hauteur_Dalle_Dalle" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Hauteur_Dalle_Plafond" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Hauteur_Sol_Plafond" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Perimetre_Espace" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_sol" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Volume_Espace" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Structure_Spatiale" type="Structure_Spatiale"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Batiment">
  - <xsd:sequence>
      <xsd:element name="Hauteur_Totale" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Emprise_Sol" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_Habitable" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Volume_Batiment" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Structure_Spatiale" type="Structure_Spatiale"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Site">
  - <xsd:sequence>
      <xsd:element name="ID_Pays" type="xsd:integer" minOccurs="1"
```

```
    maxOccurs="1" />
      <xsd:element name="Latitude" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Longitude" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Site_Adresse" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="SurfaceProjetee" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Perimetre" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Structure_Spatiale" type="Structure_Spatiale"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Etage">
  - <xsd:sequence>
      <xsd:element name="Niveau" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Hauteur_Totale" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_Totale" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Surface_Net" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Structure_Spatiale" type="Structure_Spatiale"
        minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Acteur">
  - <xsd:sequence>
      <xsd:element name="Objet_Projet" type="Objet_Projet" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Personne" type="Personne" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Personne">
  - <xsd:sequence>
      <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Nom" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Prenom" type="xsd:string" minOccurs="0" maxOccurs="1" />
      <xsd:element name="Role_Acteur" type="Role_Acteur" minOccurs="1"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Role_Acteur">
  - <xsd:sequence>
      <xsd:element name="Role" type="Type_Role" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Processus">
  - <xsd:sequence>
      <xsd:element name="Assigner_Acteur" type="Acteur" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Assigner_Produit_Colonne" type="Colonne" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Assigner_Produit_Dalle" type="Dalle" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Assigner_Produit_Mur" type="Mur" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Assigner_Produit_Poutre" type="Poutre" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
```

```
- <xsd:complexType name="Validation">
  - <xsd:sequence>
      <xsd:element name="Nom_Proposition" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="IndiceProposition" type="xsd:integer" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="AccesProposition" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="date_emission" type="xsd:date" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="echeance" type="xsd:date" minOccurs="0" maxOccurs="1" />
      <xsd:element name="Statut" type="Type_Statut" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Processus" type="Processus" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Objet_Projet">
  - <xsd:sequence>
      <xsd:element name="Root" type="Root" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Elt_Soustrait">
  - <xsd:sequence>
      <xsd:element name="Elt_Modifiant" type="Elt_Modifiant" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Elt_Ouverture">
  - <xsd:sequence>
      <xsd:element name="Surface" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Elt_Soustrait" type="Elt_Soustrait" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Elt_Add">
  - <xsd:sequence>
      <xsd:element name="Elt_Modifiant" type="Elt_Modifiant" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Elt_Saillant">
  - <xsd:sequence>
      <xsd:element name="Elt_Add" type="Elt_Add" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Elt_Modifiant">
  - <xsd:sequence>
      <xsd:element name="Element" type="Element" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Classificationx">
  - <xsd:sequence>
      <xsd:element name="Source" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Edition" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Edition_Date" type="xsd:date" minOccurs="0"
        maxOccurs="1" />
      <xsd:element name="Nom_Class" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Lib_Objets">
  - <xsd:sequence>
```

```
      <xsd:element name="Nom" type="xsd:string" minOccurs="1" maxOccurs="1" />
      <xsd:element name="Version" type="xsd:string" minOccurs="0" maxOccurs="1" />
      <xsd:element name="Date_Version" type="xsd:date" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Reference" type="xsd:string" minOccurs="1"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
- <xsd:complexType name="Documentx">
  - <xsd:sequence>
      <xsd:element name="ID_Document" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Nom_Doc" type="xsd:string" minOccurs="1"
        maxOccurs="1" />
      <xsd:element name="Description" type="xsd:string" minOccurs="0"
        maxOccurs="1" />
      <xsd:element name="Reference" type="xsd:string" minOccurs="1"
        maxOccurs="unbounded" />
      <xsd:element name="Auteur" type="xsd:string" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="Editeur" type="xsd:string" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```