MASTER EN SCIENCES INFORMATIQUES

Contribution à l'étude des bases de données d'atelier logiciel

Flener, Pierre; Jabas, Bernard

Award date:
1988

Awarding institution:
Universite de Namur

FACULTÉS
UNIVERSITAIRES
N. D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Contribution à l'étude des bases de données d'atelier logiciel

Promoteur

Monsieur **Axel van LAMSWEERDE**

Mémoire présenté pour l'obtention du grade

de Licencié et Maître en Informatique

par

Pierre FLENER et **Bernard JABAS**

Année académique 1987 - 1988

Adresse : rue Grandgagnage, 21, B-5000 NAMUR (Belgium) - ☎ 081 - 22 90 65 - Télex 59.222 FAC NAM-B - Telefax 081 - 23 03 91

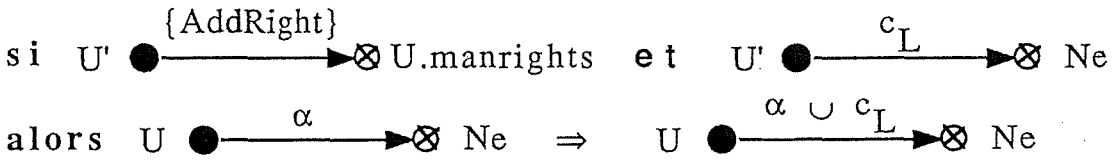
ERRATA

p.3 (1er alinéa) : substituer "analyse fonctionnelle" par "spécification des besoins"
et "analyse conceptuelle" par "conception";
p.43 (1er alinéa) : substituer "types d'événements" par "événements";
p.60/69 : substituer "atelier [logiciel]" par "environnement";
p.89 (spécification de OEImpliesNotN) : substituer "≢" par "::=";
p.91 (2e, 4e, 5e et 6e alinéa) : substituer "commande" par "primitive";
p.97 (5e alinéa) : substituer "droits propres ou effectifs" par "droits propres";
p.105/106 : substituer "[O_A[.E_A]]" par "[OE_A]"
et "[O_B[.E_B]]" par "[OE_B]";
p.106/108 : substituer "???Right" par "**I**???Right";
p.113 : supprimer la phrase suivant la figure 4.6;
p.116 (8e alinéa) : substituer "F:I:R.manattr.dep" par "F:I:R.man**rel**.dep";
p.120 : substituer "a U c" par "a U {c}"
et "a \ c" par "a \ {c}";
p.131/142 : le rôle "accesses" a une connectivité de "1-N" plutôt que de "1-1";
p.133 (11e alinéa) : substituer "programmes d'action d'une action" par "programmes d'action";
p.143/6 : substituer "CommitTime" par "BeforeCommit";
p.144 (figure 5.1) : substituer "SC" par "CV";
p.154 : substituer "caractère réalisable" par "caractère **effectif**";
p.C-3 : substituer "π" par "≠";
p.C-6 (2e alinéa) : substituer "de clauses actions ..." par "y est déjà incorporée".

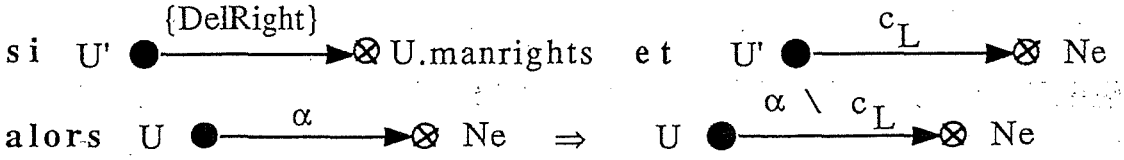
ADDENDUM

p.122 : ajouter les deux règles suivantes :

Règle_AddRight (U' , U , Ne) :



Règle_DelRight (U' , U , Ne) :



Résumé

Ce mémoire a trois objectifs. D'abord nous donnons une description détaillée et critique de l'environnement de programmation NOMADE. Ensuite nous développons un système de protection pour cet environnement. Finalement, nous proposons une extension du méta-modèle de l'atelier logiciel ALMA afin d'y incorporer les aspects dynamiques du cycle de vie d'un logiciel et nous élaborons l'architecture et le flux des données du moteur événements/actions sous-jacent.

Mots-clé : génie logiciel, ateliers logiciels, systèmes de protection, événement, action.

Abstract

The objective of this dissertation is threefold. First, we give a detailed and critical description of the programming environment NOMADE. Then we develop a capability-based protection system for this environment. Finally, we propose an extension of the meta-model of the software engineering environment ALMA in order to incorporate dynamic aspects of the software lifecycle and we elaborate the architecture and dataflow of the underlying event/action processor.

Keywords : software engineering, software engineering environments, protection systems, event, action.

Nous tenons à exprimer nos plus sincères remerciements à l'égard de toutes les personnes qui ont, de près ou de loin, contribué à la réalisation de ce travail.

Nous remercions plus particulièrement :

Monsieur Axel van Lamsweerde, promoteur de ce mémoire, qui nous a guidés et conseillés au cours de cette année académique;

Messieurs Jacky Estublier et Noureddine Belkhatir, ainsi que toute l'équipe de génie logiciel du Laboratoire de Génie Informatique de Grenoble, qui nous ont chaleureusement accueillis et encadrés lors de notre stage;

Monsieur Bruno Delcourt pour sa disponibilité à répondre à nos questions;

et nos collègues Nathalie Collart et Michel Joris pour les discussions fructueuses en matière de modèles entité-association étendus.

Nous exprimons également notre profonde gratitude :

à nos parents, à nos amis, et à nos collègues pour leurs encouragements et leurs soutiens tout au long de ce cycle d'études;

ainsi qu'à tous les professeurs de l'Institut d'Informatique pour avoir contribué à notre formation.

Pierre Flener et Bernard Jabas

Table des matières

PARTIE II : ELABORATION D'UN SYSTEME DE SECURITE POUR L'ENVIRONNEMENT DE PROGRAMMATION NOMADE 66

BIBLIOGRAPHIE

ANNEXES

Annexe A : Grammaire de NOMADE

Annexe B : Règles par défaut dans la désignation des objets/éléments

Annexe C : Fonctionnement des opérateurs d'héritage

Annexe D : Technique de vérification d'inclusion - Démonstrations et illustrations

Annexe E : Pseudocodes des algorithmes du système de protection de NOMADE

REPARTITION DU TRAVAIL DE CONCEPTION ET DE REDACTION

INTRODUCTION

INTRODUCTION

Ce mémoire s'inscrit dans le cadre des ateliers de développement de logiciels. Cette introduction met en évidence leurs enjeux, présente un bref aperçu des tendances actuelles et décrit succinctement quelques ateliers représentatifs existants.

I.1. Problèmes du "programming in the large"

Rappelons brièvement les principaux **problèmes** qui surgissent lors de l'élaboration de gros logiciels destinés à être produits en de multiples versions, par de multiples personnes et en des délais impartis :

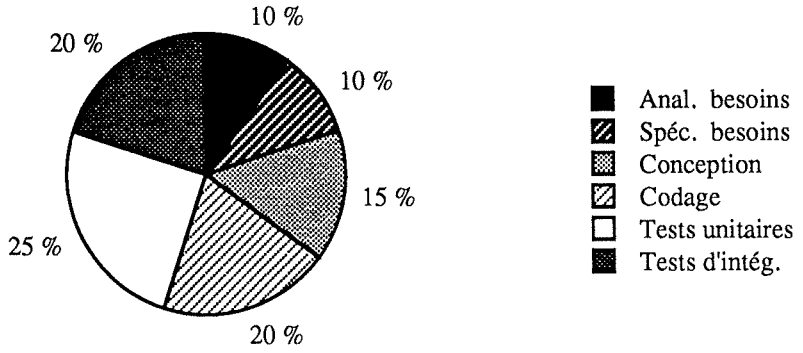
- tendance à accroître l'entropie du système, i.e. perte des structures, stratégies et méthodes clairement définies au départ;
- difficulté de contrôle des changements;
- quasi-absence de documentation/justification des décisions prises;
- manque de communication/coordination entre analystes et programmeurs;
- perte de l'information de l'état d'avancement du projet et de ses composants;
- manque de "traçabilité" entre composants;

etc.

Il en résulte généralement que les délais ne sont pas respectés (manque de productivité) et que le logiciel (s'il fonctionne !) est difficilement maintenable (manque de qualité).

I.2. Du coût des erreurs

Des **études statistiques** ([Boehm 81]) ont mis en évidence la cruelle réalité que 67 à 75 % de l'effort total consacré à un logiciel relève du travail de maintenance. En analysant les différentes étapes du cycle de vie d'un logiciel (cf. par exemple [van Lamsweerde 85]), on constate la répartition moyenne suivante (compte non tenu de l'installation) :



Ceci confirme que la programmation n'est pas l'activité primordiale, d'autant plus que nous savons que seulement 36 % des erreurs commises sont des erreurs de codage (27 % constatées avant livraison, 9 % après), le reste (64 %) étant des erreurs d'analyse fonctionnelle ou de conception (19 % constatées avant livraison, 45 % après; ces dernières sont donc tenaces et coûteuses). De plus, les coûts relatifs de correction des erreurs sont de 66 % pour l'analyse fonctionnelle, 25 % pour l'analyse conceptuelle et 9 % pour le codage. Finalement, on s'accorde à reconnaître qu'environ 50 % du coût total d'un logiciel est à imputer à la correction de ses erreurs !

I.3. Génie logiciel et ateliers logiciels

Le **génie logiciel** est une discipline ayant pour but le développement de modèles, de méthodes et d'outils pour accroître les qualités des logiciels (fiabilité, performances, portabilité, convivialité, "maintenabilité", réutilisabilité, etc.), en augmentant la productivité des analystes et des programmeurs et en diminuant les coûts. Des ensembles cohérents et intégrés de tels outils sont communément appelés des **ateliers logiciels**.

Un certain consensus quant aux **qualités d'un atelier logiciel** commence à émerger ([Osterweil 81], [van Lamsweerde 82], [Taylor 87],...) :

- **Couverture du cycle de vie**

  Un bon atelier doit pouvoir couvrir la totalité du cycle de vie d'un logiciel, à la fois selon l'axe vertical (de la spécification à la maintenance), et selon l'axe horizontal (de l'élaboration de produit à la validation).

- **Intégration**

  On distingue deux concepts d'intégration suivant le point de vue sous lequel on se place :

  - du point de vue de l'utilisateur (programmeur), l'intégration se mesure par la facilité et la vitesse de passage d'un outil à l'autre;

  - du point de vue du concepteur, l'intégration se mesure par la facilité et la vitesse selon lesquelles on peut ajouter de nouveaux outils.

Deux stratégies (non exclusives) visent à assurer le caractère intégré (selon le point de vue du concepteur) d'un atelier :

- articuler les outils autour d'une base de données centralisant toutes les informations relatives au projet logiciel; cette stratégie est inspirée de la boîte à outils Unix;

- représenter tout objet manipulé dans l'atelier sous forme interne unique d'arbre de syntaxe abstraite.

- **Extensibilité**

L'atelier doit favoriser l'ajout de nouveaux outils et la redéfinition d'outils existants.

- **Portabilité**

Les interfaces abstraites ont pour objectif de conférer aux ateliers l'indépendance la plus grande possible par rapport à l'environnement matériel/logiciel sous-jacent. Elles permettent aux outils de l'atelier d'être implémentés en termes de leurs propres abstractions.

- **Généricité**

Etant donné la multitude de modèles, de méthodologies et de langages existants, il s'avère essentiel qu'un atelier puisse être automatiquement adaptable aux modèles, aux méthodologies et aux langages propres à un utilisateur particulier.

- **Support à la manipulation de textes formels**

Utilisation d'éditeurs structurels afin de garantir la validité syntaxique, voire une cohérence sémantique statique (contrôle de type par exemple) et dynamique (contrôle de l'initialisation de variables évaluées par exemple).

- **Uniformité d'utilisation**

Une interface homme/machine doit rendre l'utilisation agréable et unifiée.

- **Existence de mécanismes d'activation**

Ces mécanismes rendent la BD de projet active et lui permettent de réagir de manière autonome lors de la survenance d'événements.

- **Disponibilité d'un assistant "intelligent"**

Guidé par une connaissance des processus de développement et de maintenance de logiciels, il pilote l'utilisateur dans l'emploi des outils de l'atelier.

- **Réutilisabilité de composants**

Les composants de l'atelier doivent être réutilisables à différents endroits.

- **Distribution transparente**

Distribution transparente sur un réseau local de stations de travail hétérogènes.

etc.

Remarquons qu'étant donné la relative jeunesse du domaine et parce que beaucoup d'aspects du problème ne sont que partiellement explorés, aucun atelier logiciel existant à ce jour ne peut prétendre à toutes ces qualités.

I.4. Ateliers existants

Une base de données d'atelier centralise toutes les informations relatives aux différentes étapes du cycle de vie couvertes par l'atelier, pour un modèle sous-jacent de cycle de vie particulier. Ces informations concernent les objets logiciels (spécifications, modules, etc.) et leurs inter-relations.

Nous pouvons classer les ateliers d'après leur modèle de structuration de ces différentes informations. Trois types de modèles sont utilisés pour structurer les informations d'un projet logiciel :

- modèles ERA (entité-relation-attribut) de haut niveau : ALMA, SKB, PMDB;
- modèles ERA de bas niveau : PCTE;
- modèles orientés objets : SODOS.

Nous donnons ici une brève description de ces ateliers.

- **ALMA** ([van Lamsweerde 86,88b]) - ALMA (*Atelier Logiciel sur Machine Abstraite*) constitue un atelier générique pouvant être instancié à n'importe quelle méthodologie, modèle ou langage. Des outils autonomes s'articulent autour d'une BD de projet (basée sur un modèle ERA étendu) contenant les informations de tout le cycle de vie d'un logiciel. Des textes formels sont manipulés via une connaissance de leurs syntaxes abstraites. Une très grande portabilité a été atteinte via l'utilisation massive d'interfaces abstraites. Remarque : une présentation un peu plus détaillée d'ALMA se trouve dans la section 2.1. de la partie III de ce travail.

- **SKB** ([Meyer 85a]) - Le projet SKB (*Software Knowledge Base*) vise à fournir un ensemble intégré d'outils articulés autour d'une BD centrale, qui est structurée en termes d'un modèle ERA binaire. Le système est paramétrable pour être indépendant de toute méthode, de tout langage et de tout système d'exploitation. La couverture de tout le cycle de vie d'un projet logiciel est ainsi possible. Une originalité réside dans le fait que cette BD ne contient que des **descripteurs** des objets logiciels du monde réel, et non pas ces objets eux-mêmes. Il en résulte une portabilité et une simplicité maximales de l'atelier, mais il devient aussi impossible de garantir la cohérence entre la BD et la réalité. Une autre caractéristique intéressante est que la définition de **contraintes** que doivent vérifier les relations inter-objets permet d'enregistrer des propriétés sémantiques. Suite à des violations de telles contraintes, des **actions** seront déclenchées par un **démon**.

- **PMDB** ([Penedo 85,87]) - Le projet PMDB (*Project Master DataBase*) propose un noyau de modèle (du type ERA binaire) permettant la gestion des données générées au cours du cycle de vie complet d'un logiciel. Au niveau des méthodologies cependant, rien n'est imposé. La notion d'historique y joue un rôle primordial. On y prévoit l'intégration de composants actifs (vérifiant des contraintes d'intégrité et assurant la cohérence des données), ainsi que d'un assistant "intelligent" dirigé par une base de règles et qui guidera l'utilisateur à travers l'emploi des différents outils. L'importance du concept de vue et la nécessité de distribution/partitionnement de la BD y sont reconnues.

- **PCTE** ([Gallo 86]) - Le projet PCTE (*Portable Common Tool Environment*) propose un standard international servant de structure d'accueil pour des ateliers logiciels afin de faciliter le développement et l'intégration de nouveaux outils. La couche OMS (Object Management System) est basée sur un modèle ERA binaire assez pauvre et offre des primitives pour le stockage et l'accès aux objets ainsi que pour la gestion des dépendances inter-objets. Des caractéristiques importantes sont la possibilité de distribution transparente de la BD sur un réseau à postes de travail graphiques, les bonnes performances d'accès (en raison de la granularité élevée des objets) et l'extensibilité dynamique des objets;

- **SODOS** ([Horowitz 86]) - Le système SODOS (*Software Documentation Support Environment*) aide à la définition et à la manipulation de documents élaborés lors d'un projet logiciel. Aucune méthodologie n'est imposée. L'accent a surtout été mis sur la possibilité de vérification de cohérence, de complétude et de "traçabilité" d'un document. Le modèle de données sous-jacent est du type "orienté objet".

I.5. Fil conducteur

Ce travail est subdivisé en trois parties. Dans la première partie, nous allons d'abord étudier de manière détaillée l'environnement de programmation NOMADE (NOyau de MAintenance et de DEveloppement). Ensuite, nous prendrons un peu de recul pour procéder à une étude critique de NOMADE.

La seconde partie sera consacrée à l'élaboration d'un système de sécurité pour NOMADE.

Finalement, nous proposerons dans la troisième partie une extension du méta-modèle d'ALMA (Atelier Logiciel sur Machine Abstraite) pour couvrir les aspects dynamiques.

PARTIE I

ETUDE DETAILLEE ET CRITIQUE DE L'ENVIRONNEMENT DE PROGRAMMATION NOMADE

(NOyau de MAintenance et de DEveloppement)

1. PRESENTATION

1.1. Historique

Le projet **NOMADE** (NOyau de MAintenance et de DEveloppement) vise à concevoir et développer un noyau d'atelier logiciel destiné à l'aide au développement et à la maintenance de gros logiciels (i.e. nécessitant plusieurs dizaines d'années-homme).

Démarré en 1987 par l'équipe de génie logiciel du Laboratoire de Génie Informatique de l'Institut de Mathématiques Appliquées de Grenoble à l'Université de Grenoble, ce projet entre actuellement dans sa phase de développement.

La phase de conception a largement bénéficié de l'expérience accumulée dans le projet **ADELE** (Atelier de DEveloppement de LogiciEls) ([Estublier 84], [Belkhatir 86a,86b,87]), qui est actuellement utilisé à un échelon industriel (AIRBUS, BULL, ...). A long terme, NOMADE remplacera ADELE, car il en constitue une extension et une généralisation.

1.2. Caractéristiques de NOMADE

Comparativement aux critères d'un atelier logiciel idéal définis dans l'introduction, voici la situation actuelle de l'atelier NOMADE :

NOMADE ne vise pas la totalité du cycle de vie d'un projet logiciel; il couvre les **phases de programmation et de maintenance**; il s'agit donc plutôt d'un environnement de programmation.

La **base de programmes** de NOMADE est bâtie selon des mécanismes empruntés à la fois aux modèles ERA (entité-relation-attribut) et aux types abstraits. Chaque objet est caractérisé par des attributs, associé à d'autres objets par des relations et est manipulable par des actions (outils). De plus, les utilisateurs de NOMADE (qui sont aussi considérés comme des objets) ne peuvent manipuler que les objets auxquels ils ont le droit d'accéder. Bien entendu, des mécanismes de partage des données entre plusieurs utilisateurs et des mécanismes de recouvrement en cas d'erreur sont prévus.

Visant à être un noyau d'atelier de programmation, NOMADE est **extensible**; il permet l'ajout de nouveaux outils, tout comme la redéfinition d'outils prédéfinis.

NOMADE est **portable** dans l'univers Unix : l'encapsulation des commandes Unix (compilation, édition des liens, ...) dans des actions rend Unix transparent aux utilisateurs.

En matière de gestion d'effets de bord et de protection de la base, NOMADE est **programmable** en ce sens que l'utilisateur peut lui-même définir ses propres stratégies. NOMADE garantit ensuite que ces décisions seront respectées pendant toute la durée du projet.

NOMADE est à **usage uniforme** car les interactions avec les usagers se font via un jeu prédéfini de primitives de communication.

La base de programmes est **active** en ce sens que des événements déclenchent des actions.

NOMADE permet la **distribution transparente** des travaux sur un réseau local de stations de travail hétérogènes.

1.3. Fil conducteur

Dans la suite, nous procéderons à une présentation détaillée de NOMADE. Les concepts sont peu nombreux, mais hautement interconnectés. Le système n'est basé que sur trois notions: les objets, leurs attributs et leurs relations; tout le reste étant construit sur ce fondement.

Ainsi, les sections suivantes seront organisées comme suit : partant de l'énumération des types d'objet et de leurs éléments (attributs de type texte) qui composent la base de programmes de NOMADE (section 2), nous détaillerons les concepts d'attribut (section 3), de relation (section 5), d'événement & action (section 6) et de droit usager (section 7) qui caractérisent les objets. La section 4 est consacrée à l'écriture des expressions-NOMADE, qui permettent la désignation d'ensembles d'objets et éléments. La notion de partition (entité conceptuelle regroupant des objets à caractéristiques communes) sera exposée dans la section 8 avant de passer aux différents mécanismes d'héritage (permettant le partage de propriétés communes) à la section 9. Dans la section 10, nous décrirons comment construire des configurations (agrégats de composants inter-reliés satisfaisant des contraintes de sélection) et la section 11 abordera quelques aspects plus techniques.

La section 12 confrontera l'approche NOMADE avec les paradigmes de la programmation orientée objets. La section 13 comparera le modèle NOMADE avec les modèles ERA (entité-relation-attribut).

Jusque là, la présentation de NOMADE sera neutre, c'est-à-dire que nous n'y discuterons pas de ses avantages et inconvénients eu égard à l'état de l'art actuel en matière d'ateliers logiciels. Ainsi, nous soulignerons dans la section 14 les principaux points forts de NOMADE et la section 15 sera consacrée à une étude critique pour en déduire des améliorations possibles.

2. TYPES D'OBJET ET TYPES D'ELEMENT

Pour réaliser un atelier logiciel articulé autour d'une base de données, il existe deux approches différentes.

La *première approche* consiste à utiliser purement et simplement un SGBD existant. Les inconvénients de cette approche résultent du fait que les SGBD classiques ne sont généralement pas "parés" aux spécificités propres aux ateliers ([Osterweil 81],[Bernstein 87],...) : taille importante des objets, objets structurés, longue durée des transactions, versions multiples, concepts de vues de plus haut niveau, etc.

La *seconde approche* consiste donc à définir un modèle de données ainsi que des mécanismes sous-jacents qui sont mieux adaptés aux spécificités des ateliers. NOMADE, développé selon cette approche, est axé sur un modèle orienté-objet.

Cette section définit les briques de base de NOMADE : les types d'objet et les types d'élément, premiers concepts nécessaires à la structuration du logiciel.

2.1. Types d'objet

Dans les langages de programmation modulaire tels que Ada ou Modula-2, la notion de module correspond à l'association entre une *interface* , décrivant les "ressources" (procédures, fonctions, types, etc.) fournies par le module, et une *réalisation* , implémentant les ressources de l'interface. Cette séparation entre interface et réalisation offre la possibilité de compilation séparée et facilite le développement multi-programmeur.

Sous NOMADE, la notion de *famille* est une généralisation de cette notion de module.

NOMADE gère aussi des informations relatives à ses *usagers* .

Famille, interface, réalisation et usager sont les seuls **types d'objet** de NOMADE.

2.1.1. Réalisation

On appelle **réalisation** une implémentation des ressources déclarées au niveau de l'interface.

2.1.2. Interface

On appelle **interface** la déclaration d'un ensemble de ressources : procédures, fonctions, types, variables globales, constantes, etc.

Pour une interface donnée, il existe plusieurs implémentations permettant de la réaliser. Une interface correspond donc à un ensemble de **versions de réalisations**. Remarquons qu'une réalisation peut **utiliser** d'autres interfaces. La relation "utilise" est définie comme suit : si A utilise B, alors le fonctionnement correct de A suppose l'existence d'une version correcte de B.

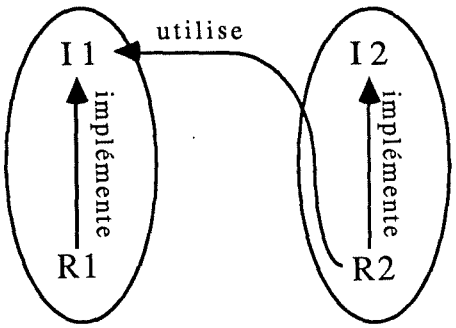


Figure 2.1. - Relations "implémente" et "utilise"

Une interface peut ne pas être implémentée par des ressources si elle se limite à des définitions de types, de constantes, etc.

2.1.3. Famille

Un logiciel doit pouvoir fonctionner sur différentes machines et sous différents systèmes d'exploitation. De plus, durant la phase d'implémentation du logiciel, des modifications dans la conception peuvent survenir. Ces raisons expliquent pourquoi, dans NOMADE, plusieurs **versions d'interface** coexistent au sein d'une même famille. Comme ces interfaces remplissent toutes la même fonctionnalité logique, il apparaît compréhensible de les regrouper dans une même entité.

C'est la raison pour laquelle, dans NOMADE, on appelle **famille** l'ensemble de toutes les versions d'une interface et de leurs versions de réalisations.

Famille, interface et réalisation définissent ainsi une hiérarchie à trois niveaux : une famille (niveau 1) est constituée d'un ensemble de versions d'interface (niveau 2), chacune étant implémentée par un ensemble de versions de réalisation (niveau 3).

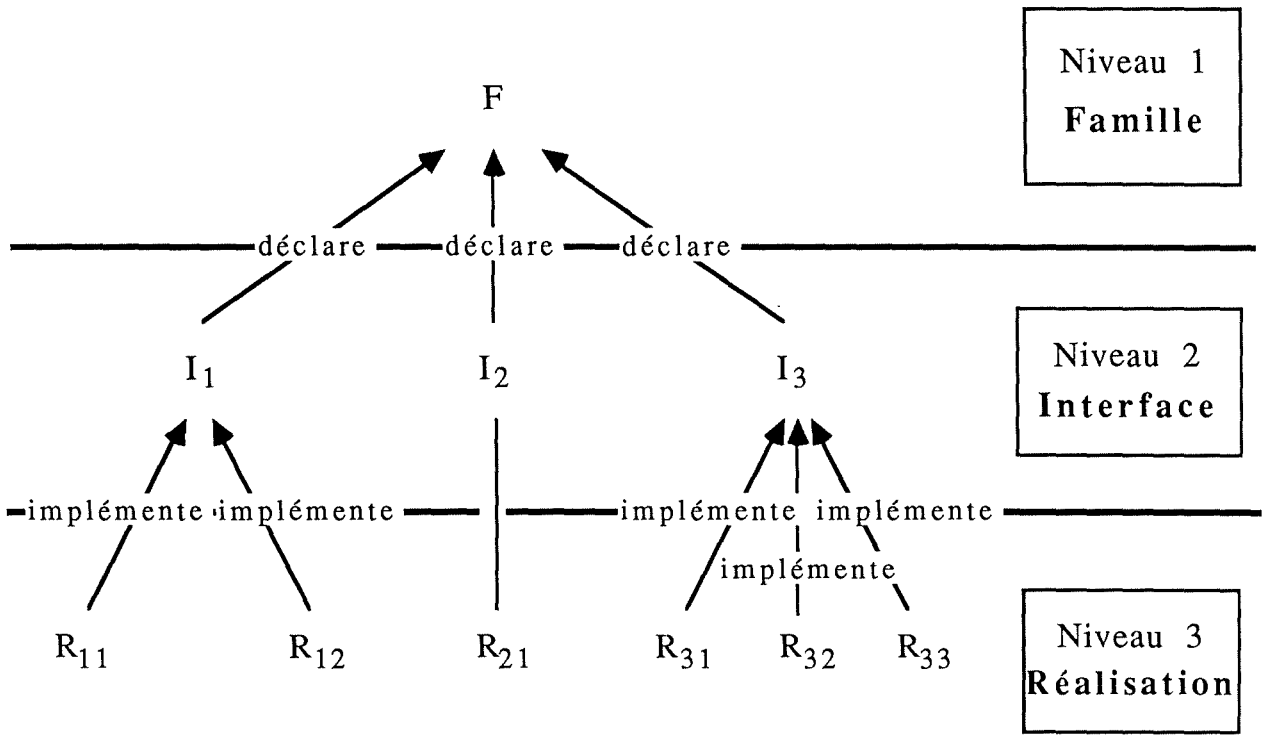


Figure 2.2. - Hiérarchie à trois niveaux

Exemple :

Supposons une famille F remplissant la fonctionnalité de tri. A chaque structure de données possible correspondra une interface. Soit l'interface I_1 pour laquelle les données à trier se présentent sous forme de vecteur, et l'interface I_2 pour laquelle les données sont représentées sous forme de liste chaînée.

Pour la représentation sous forme vectorielle, soit l'interface I_1, nous aurons plusieurs possiblilités d'implémenter les ressources déclarées dans I_1. Chaque algorithme (Quicksort, Heapsort, Shell) constituera une réalisation de I_1. D'où :

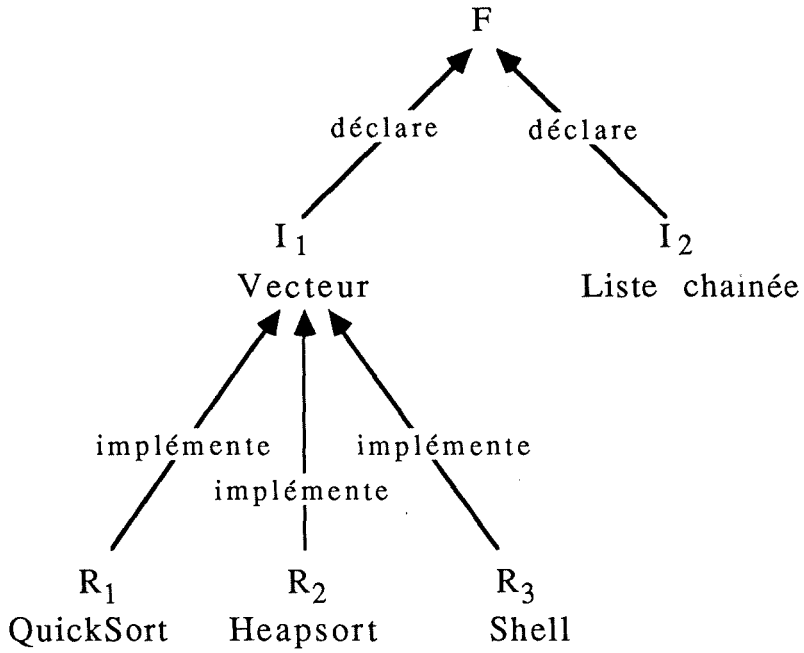


Figure 2.3. - Fonctionnalité de tri ♦

Cas particulier des familles bibliothèques - Définition

Une réalisation peut utiliser plusieurs interfaces d'une même famille si cette famille est une famille **bibliothèque.** ♦

Exemple :

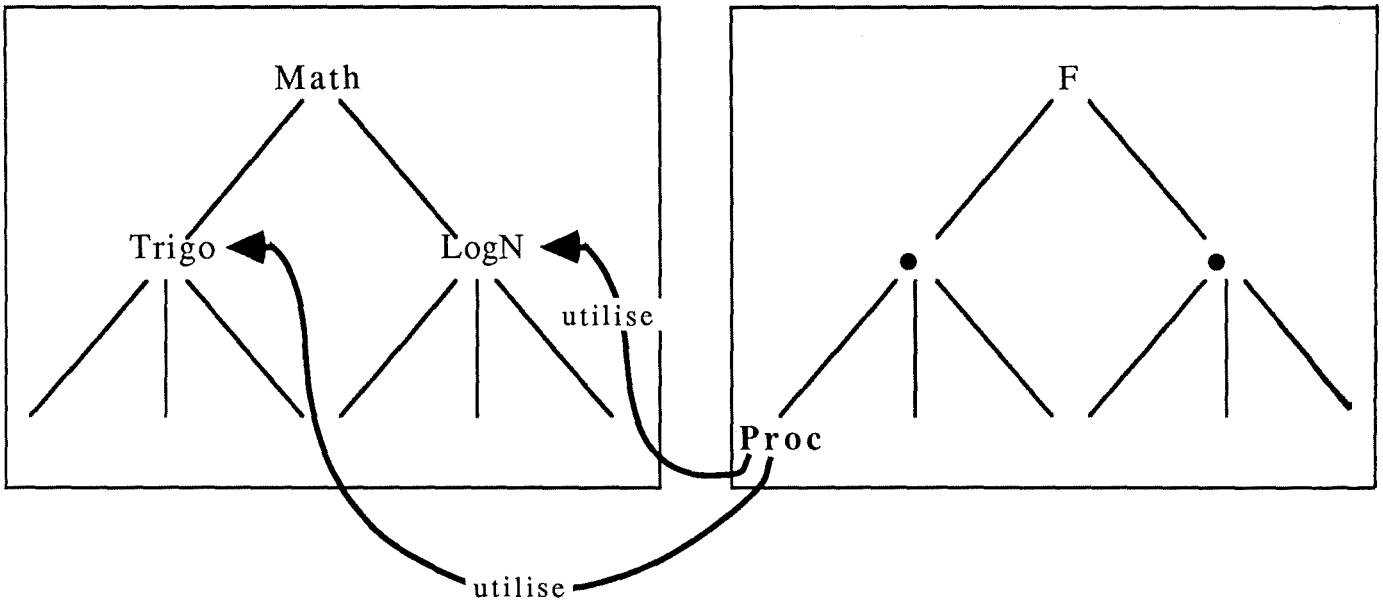


Figure 2.4. - Famille bibliothèque

Supposons deux interfaces Trigo (déclaration de fonctions trigonométriques) et LogN (déclaration de fonctions logarithmiques) d'une même famille Math. Pour que la procédure Proc d'une famille F puisse utiliser les interfaces Trigo **et** LogN de la famille Math, il faudra définir Math comme une famille bibliothèque. ♦

2.1.4. Usager

NOMADE permet non seulement de structurer les programmes mais aussi de tenir compte des **usagers** du système. C'est la raison d'être du type d'objet particulier "usager".

2.2. Désignation des objets

La structuration arborescente des objets conduit tout naturellement à la désignation suivante des objets :

- si l'objet est une **famille** (resp. un **usager**), alors il suffit de préciser son nom, car celui-ci est **unique** dans un projet; F est ainsi un exemple de désignation de famille;

- si l'objet est une **interface**, alors on la désigne par son nom préfixé par celui de sa famille à l'aide du séparateur (":"). Il va de soi que toutes les interfaces d'une même famille ont des noms différents. On aura ainsi l'interface $F:I_1$

- si l'objet est une **réalisation**, alors on la désigne par son nom préfixé par celui de son interface à l'aide du séparateur (":"). Il va de soi que toutes les réalisations d'une même interface ont des noms différents. On aura ainsi la réalisation $F:I_1:R_{11}$

Chacun des trois identifiants de famille, interface ou réalisation peut être la chaîne vide; des **règles par défaut** permettent alors de définir des conventions univoques de désignation (cf. annexe B).

2.3. Types d'élément

Tout objet de type famille, interface, réalisation ou usager est composé d'une série d'éléments. Il existe six **types d'élément** prédéfinis : texte source, code objet, document, document de révision, manuel et historique. Contrairement aux quatre autres types d'élément, les types d'élément historique et manuel ont une structure prédéfinie car ils sont gérés par NOMADE.

2.3.1. Texte Source (SourceText)

Un texte source peut être défini au niveau interface ou au niveau réalisation. Nous parlerons alors d'une *vue d'interface* ou d'une *révision de réalisation* .

2.3.1.1. Vue d'interface

Chaque version d'interface peut être composée d'un ensemble de textes source appelés **vues**. Les différentes vues d'une même interface sont équivalentes; elles correspondent à des façons différentes de déclarer les ressources de l'interface.

Par exemple, toutes les vues d'une même interface pourraient être écrites dans des langages de programmation différents. Un autre exemple serait que les différentes vues d'une même interface déclarent des sous-ensembles différents des ressources réellement implémentées par la réalisation correspondante. Des ressources critiques peuvent ainsi être protégées, si on restreint les accès aux vues.

2.3.1.2. Révision de réalisation

Chaque version de réalisation peut être composée d'un ensemble de textes source appelés **révisions.**

Les différences entre versions de réalisation correspondent à des modifications ayant une signification logique (passage d'une version de mise au point à une version définitive par exemple) : elles ont un caractère sémantique. Les différentes révisions, quant à elles, correspondent à des modifications plus mineures : elles résultent d'améliorations ou de corrections d'erreurs et sont donc plutôt d'ordre syntaxique. ([Tichy 82])

Les concepts de version, de révision et de vue sont suggérés par le schéma suivant :

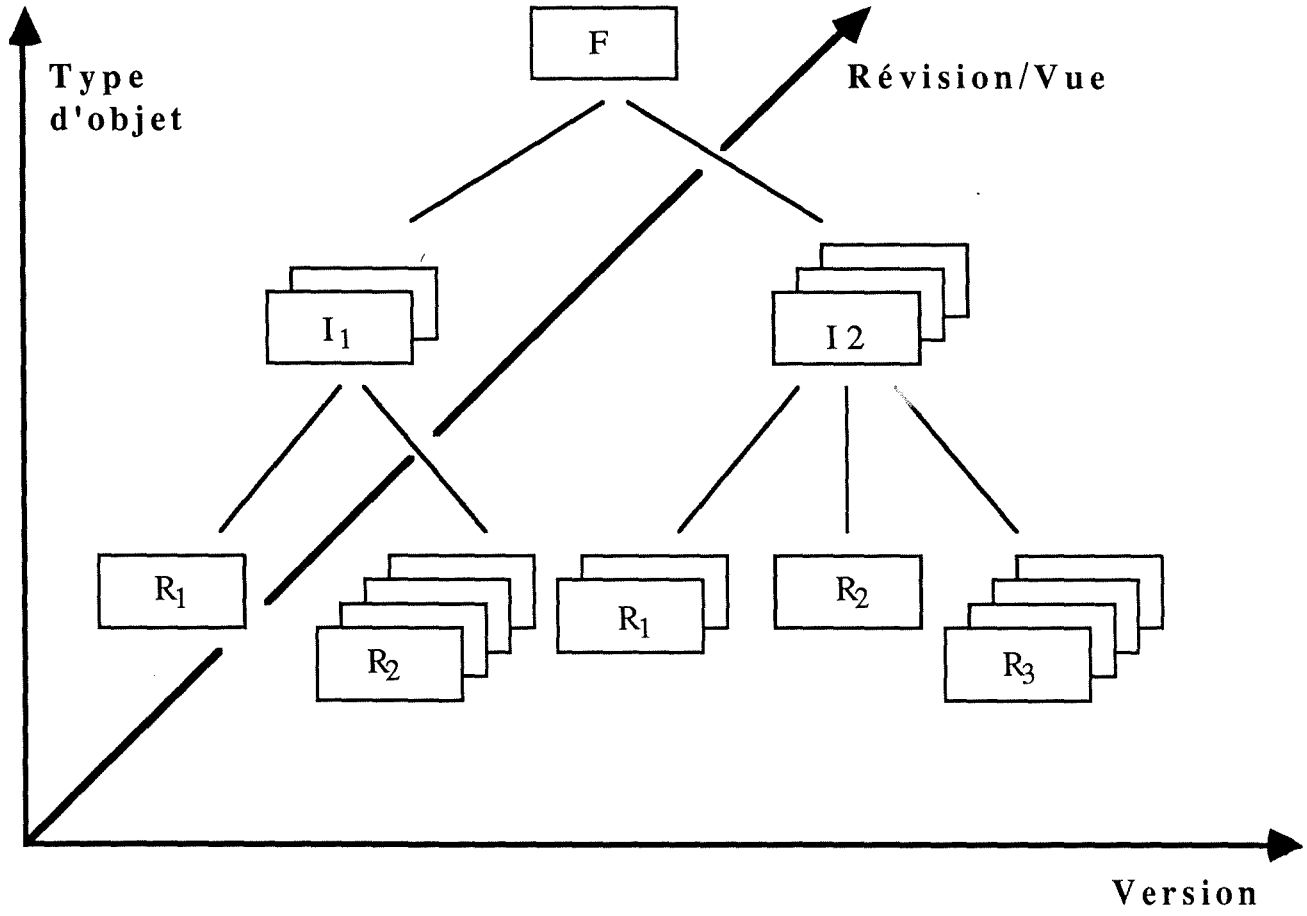


Figure 2.5. - Versions, révisions et vues

2.3.2. Code objet (ObjCode)

On peut associer à une réalisation autant de codes objet qu'elle possède de révisions.

2.3.3. Document (Document)

A chaque objet, l'utilisateur peut associer des documents, comme par exemple : jeu_de_test, mode_d'emploi, fichier_données, spécification, etc.

2.3.4. Document de révision (RevDoc)

Il s'agit d'un document de structure quelconque associé à une révision de version de réalisation.

2.3.5. Historique (History)

L'historique contient des renseignements concernant les évolutions d'une réalisation, ce qui offre à l'utilisateur une meilleure compréhension des changements que son logiciel subit. Lors de chaque création d'une nouvelle révision, l'historique mentionne si elle est construite à partir d'une révision existante. De plus, l'utilisateur peut fournir un commentaire qui sera stocké dans l'historique avec également son nom, la date et le type d'opération effectué.

NOMADE se réserve la gestion des historiques : l'utilisateur possède seulement un droit de consultation sur ces éléments.

2.3.6. Manuel (Manual)

Un manuel est un descripteur qui contient toutes les informations (attributs, relations, etc) relatives à une occurrence d'un type d'objet; il existe donc un manuel par objet. Les manuels sont structurés selon un **langage de documentation du projet et de définition de contraintes**. Ce langage sera explicité au cours des sections 3 , 5 , 6 et 7 : clauses **attributes**, **relations**, **actions** et **rights** respectivement.

Les manuels, comme les historiques, sont créés et gérés par NOMADE. Toutefois, l'utilisateur n'est pas limité à la seule consultation des manuels. Le logiciel étant en perpétuelle évolution, l'utilisateur modifiera le manuel de l'objet pour spécifier ses nouvelles caractéristiques.

2.4. Désignation des éléments

Comme chaque objet est constitué d'une série d'éléments, il faut étendre la désignation des objets pour permettre celle d'un élément d'un objet.

Un élément est désigné en ajoutant un suffixe à la désignation de l'objet correspondant, le séparateur retenu étant le point (".").

Un **manuel** est désigné par le suffixe ".man" (exemple : F.man); il y a aussi moyen d'identifier des parties de manuel, mais celles-ci ne seront définies que dans la suite; ainsi ".manattr" désigne la clause attributes d'un manuel (cf. section 3.), ".manrel" désigne la clause relations d'un manuel (cf. section 5.), ".manact" désigne la clause actions d'un manuel (cf. section 6.), ".manrights" désigne la clause rights d'un manuel (cf. section 7.), ".manpart" désigne les blocs partition d'un manuel (cf. section 9.), ".mansel" désigne les trois clauses de sélection d'un manuel (cf. section 10.) et ".manother" désigne le reste d'un manuel.

De même, un **historique** est désigné par le suffixe ".hist" : F:I3:R33.hist

Pour les objets de type interface, on désigne une **vue** par son nom : F:I_2.vue1

Pour les objets de type réalisation, on désigne une **révision** par son numéro d'ordre (maximum 3 chiffres) : F:I_2:R_{21}.15

Les noms de **documents** créés par les utilisateurs doivent seulement être différents des noms de documents créés par NOMADE : F:I_1:R_{11}.testdata Si un tel document est attaché à une révision particulière, alors on le suffixe par le numéro d'ordre de cette révision; le séparateur retenu étant toujours le point (".") : F:I_2:R_{21}.spécif.15

Définition : (provisoire)

Un **nom simple** est soit une désignation d'objet, soit une désignation d'un élément d'un objet; cette définition sera étendue à la section 8.6. ♦

2.5. Tableau de synthèse

	Source Text	ObjCode	Document	RevDoc	History	Manual
Famille			X			X
Interface	X		X			X
Realization	X	X	X	X	X	X
User			X			X

Tableau 2.1. - Tableau de synthèse des éléments possibles par type d'objet

3. ATTRIBUTS

La caractérisation des objets se décompose en deux parties. Une *partie statique* où l'on définit les attributs et les relations (cf. section 5.) des objets, et une *partie dynamique* où l'on assure la cohérence des attributs et relations en définissant des actions garantissant le respect des contraintes d'intégrité sur ces attributs et relations (cf. section 6.).

Cette section traite des différents types d'attribut et de leur sémantique.

3.1. Types d'attribut

A chaque objet/élément (cf. section 2.), on peut associer des attributs qui les caractérisent, et ce en les écrivant dans la clause attributes du manuel de l'objet correspondant. Les attributs peuvent être mono- ou multi-valués. Un attribut est mono-valué si une seule valeur lui est associée : attr = val. Si, au contraire, plusieurs valeurs sont associées à un même attribut, on parlera d'attribut multi-valué. Dans ce cas, les différentes valeurs sont définies en extension : attr = val_1, val_2, ..., val_n. Notons que les val_i peuvent contenir des méta-caractères (cf. annexe A) .

Il existe deux types d'attributs :

- les attributs prédéfinis qui sont créés et gérés par NOMADE;
- les attributs définis par l'utilisateur.

3.1.1. Attributs prédéfinis

Parmi ces attributs prédéfinis, on distingue ceux qui sont modifiables (**state**, **stateconf** et **language**) de ceux qui ne le sont pas.

- **state**	spécifie l'état d'un texte source : "experimental", "obsolete", "ok", etc.
- **stateconf**	spécifie l'état d'une configuration (cf. section 10.) : "incomplete", "inconsistent", "ok", etc.
- **language**	spécifie le langage d'un texte source;
- **date**	date de création de l'objet;
- **author**	nom de l'utilisateur ayant créé l'objet;
- **object**	type d'objet : Family, Interface, Realization ou User;
- **element**	type d'élément : SourceText, ObjCode, Document, RevDoc, History ou Manual;
- **family**	nom de la famille à laquelle l'objet ou l'élément appartient;
- **interface**	nom de l'interface de cet objet/élément;
- **realization**	nom de la réalisation de cet objet/élément;
- **name**	nom de l'élément.

Seuls ces attributs prédéfinis sont automatiquement associés aux objets pour lesquels ils sont pertinents.

3.1.2. Attributs définis par l'utilisateur

L'utilisateur peut, selon ses besoins, se définir ses propres attributs en étendant la clause **attributes**.

3.2. Clause attributes

La clause attributes présente dans tout manuel fait donc partie du langage de documentation du projet et de définition de contraintes. Suivant qu'elle apparaît dans le manuel d'une famille ou d'une interface, d'une réalisation, d'un usager, sa sémantique est différente. La syntaxe, elle, est identique.

3.2.1. Clause attributes d'un manuel de famille

Lorsqu'elle apparaît dans le manuel d'une famille, la clause attributes définit les valeurs *possibles* des attributs pour les objets/éléments de cette famille. La clause attributes d'un manuel de famille joue donc un **rôle définitionnel** (langage de définition de contraintes).

Si l'expression attr = $val_1 \ldots val_n$ (n>0) apparaît dans la clause attributes du manuel de la famille F, on dira que $val_1 \ldots val_n$ sont les valeurs possibles de l'attribut attr dans F.

La clause attributes d'un manuel de famille sert aussi à définir ce que l'on appelle un **modèle de famille**. Ce modèle exprime quels sont les éléments valides dans la famille et définit quels sont les attributs valides pour ses éléments.

De cette manière, la création d'un élément dans une famille sera refusée si cet élément n'est pas valide par rapport au modèle défini pour cette famille.

Exemple :

```
manual F
  family
    attributes
      def **                          (1)
        system = unix*;               (2)
        language = c, pascal;         (3)
      def :*.specif                   (4)
        element = document;           (5)
        editor = vi, emacs;           (6)
```

Ce fragment de manuel de la famille F définit que :

- pour tous les objets/éléments de F (1), les valeurs valides pour les attributs system et **language** sont respectivement unix* (2) (unix suivi de toute chaîne de caractères) et c ou pascal (3);

- pour les documents "specif" des interfaces de F (4), la valeur de l'attribut **element** doit être document (5) et ces documents doivent avoir été réalisés à l'aide de l'éditeur vi ou emacs (6).

Dans cet exemple, la création d'un objet/élément de cette famille avec l'attribut **language** = cobol serait refusée. ♦

3.2.2. Clause attributes d'un manuel d'interface, de réalisation ou d'usager

Dans un manuel d'interface, de réalisation ou d'usager, la clause attributes définit les valeurs *effectives* des attributs pour cet objet (et ses éléments). Ici, la clause attributes joue donc un **rôle factuel** (langage de documentation du projet).

Soit attr = $val_1 \ldots val_n$ (n>0) présent dans la clause attributes d'un tel manuel. On dira que $val_1 \ldots val_n$ sont les valeurs effectives de l'attribut attr pour l'objet/élément en question.

Exemple :

```
manual R
  realization
    attributes
      system = unix4.2;
      language = pascal;
```

Ce fragment de manuel de la réalisation R indique qu'elle est utilisable sous la version 4.2 de Unix et qu'elle est écrite en Pascal. ♦

4. QUALIFICATIONS, NOMS QUALIFIES ET EXPRESSIONS-NOMADE

4.1. Qualifications

Définition :

Une **qualification** est un triplet (attribut, opérateur relationnel, ensemble de valeurs). ♦

Nous avons déjà rencontré un cas particulier de qualifications, à savoir celles qui constituent les clauses attributes dans les manuels (sauf que là, l'opérateur relationnel est toujours l'opérateur d'égalité; cf. section 3.1.).

Nous pouvons à présent parler en toute généralité de *qualifications factuelles* (dont les valeurs d'attribut sont séparées implicitement par des **et** logiques) et de *qualifications définitionnelles* (dont les valeurs d'attribut sont séparées implicitement par des **ou** logiques).

Des exemples de qualifications factuelles sont celles des clauses attributes des manuels d'interface, de réalisation et d'usager. Des exemples de qualifications définitionnelles sont celles des clauses attributes des manuels de famille mais aussi celles des noms qualifiés :

4.2. Noms qualifiés

Définition :

Un **nom qualifié** est un nom simple (éventuellement nié) suivi d'une conjonction facultative de qualifications. ♦

Un tel nom qualifié constitue la brique de base pour la construction de ce que nous appelerons des expressions-NOMADE (décrites ci-dessous).

Exemple de nom qualifié :

F:*:* (**language** = pascal) désigne toutes les réalisations de la famille F qui sont écrites en Pascal. ♦

L'utilisation des attributs prédéfinis : **family, interface, realization** et **element** permet l'expression des noms simples sous forme de noms qualifiés.

Exemple :

F:I:R.E est équivalent à ** (**family** = F **and interface** = I **and realization** = R **and element** = E). ♦

Cependant, toute utilisation de certains de ces quatre attributs dans un nom qualifié résultera en un compactage automatique de ce dernier en un nom qualifié plus court.

Exemple :

** (**family** = F **and interface** = I **and realization** = R **and element** = E **and** A_1 = V_1 **and** ... **and** A_n= V_n)

sera transformé en :

F:I:R.E (A_1 = V_1 **and** ... **and** A_n= V_n) ♦

4.3. Expressions-NOMADE

Le concept d'expression-NOMADE sert à désigner tout un ensemble d'objets/éléments Ceci est déjà possible par le biais des méta-caractères dans un nom qualifié, mais le pouvoir d'expression reste cependant limité : la combinaison de noms qualifiés et de connecteurs logiques **or**, **and** et **not** s'impose donc.

Définition :

Une **expression-NOMADE** est une disjonction (éventuellement niée) de conjonctions (éventuellement niées) de noms qualifiés. ♦

Exemple d'expression-NOMADE :

(F:*:* (**author** = John,Jack **and language** = c)) **or** (G:* (**language** = h))

désigne l'ensemble de toutes les réalisations de la famille F qui sont écrites en c par John ou Jack, ainsi que toutes les interfaces de la famille G qui sont écrites en h. ♦

Une expression-NOMADE peut s'utiliser en tant que requête, renvoyant la valeur **true** s'il existe un objet/élément dans le projet courant qui la satisfait, et **false** sinon.

Remarque :

Malgré l'emploi de connecteurs logiques (**and**, **or**, **not**), les expressions-NOMADE sont des expressions ensemblistes plutôt que des prédicats logiques. Ainsi, il serait plus approprié de parler en termes d'intersections, réunions et compléments ensemblistes. Pour lever toute ambiguïté dans la suite, nous conviendrons que les symboles $\wedge$, v, $\sim$, et $\Rightarrow$ désignent les connecteurs logiques classiques, et que **and**, **or**, **not** et **implies** sont utilisés dans des expressions ensemblistes (expressions-NOMADE). Cette distinction est utile, car au niveau des ensembles désignés par des expressions-NOMADE, il y aura des intersections, réunions, compléments et inclusions ensemblistes.

5. RELATIONS

Les relations constituent le troisième (et dernier) concept de base de NOMADE. Elles constituent le deuxième volet de la description de la partie statique de la caractérisation des objets.

5.1. Types de relation

Seules les *relations binaires* sont autorisées. Nous traiterons séparément les relations réflexives, en raison de leur signification particulière. Afin de simplifier les notations et de pouvoir distinguer relations binaires et réflexives, nous appelerons relations binaires les relations binaires non réflexives.

5.1.1. Relations binaires

L'environnement NOMADE gère des **relations binaires** entre deux types d'objet/élément. Certaines relations seulement sont prédéfinies, comme par exemple la relation "depends_on" (notée **dep**) (ou : "uses", etc.). Cette relation de dépendance joue un rôle absolument fondamental dans NOMADE. Les autres relations doivent être définies par l'utilisateur.

L'**intérêt** des relations binaires est double :

- elles permettent de structurer les objets/éléments (aspect statique);
- elles sont à la base du mécanisme événement/action (aspect dynamique); ce mécanisme sera étudié en profondeur dans la section 6 qui lui est dédicacée.

5.1.2. Relations réflexives

Toute commande de l'atelier NOMADE est considérée comme une relation réflexive portant sur l'objet/élément qu'elle manipule (ce point sera justifié à la section 6.1.1.). Des occurrences d'une relation réflexive C existent implicitement sur tous les objets/éléments manipulables par cette commande C.

5.2. Clause relations

La clause relations présente dans tout manuel fait partie du langage de documentation du projet et de définition de contraintes. Suivant qu'elle apparaît dans le manuel d'une famille ou d'une interface, d'une réalisation, d'un usager, sa sémantique est différente. La syntaxe, elle, est identique.

5.2.1. Clause relations d'un manuel de famille

La clause relations d'un manuel de famille joue un **rôle définitionnel** (langage de définition de contraintes). Elle définit :

- quelles sont les relations (binaires ou réflexives) pouvant avoir un objet ou un élément d'un objet de F comme cible;

et pour chacune de ces relations R :

- son *co-domaine* , c'est à dire quels sont les objets ou éléments d'objets **de F** pouvant être cible d'une occurrence de R; la précision "de F" est importante, car elle est implicite. On appelle cela une insertion implicite de contexte (cf. les exemples ci-dessous).

- son *domaine* , c'est-à-dire quels sont les objets et éléments d'objets pouvant être source d'une occurrence de R. ♦

Les domaines et co-domaines sont écrits sous forme d'expressions-NOMADE.

Enonçons encore une propriété des relations qui ne sera exploitée que dans la section 6.1.2. :

Propriété P5.1 :

L'ordre textuel des relations dans une clause relations détermine leur ordre de **priorité** (notée p ci-dessous) décroissante. ♦

Exemple de clause relations d'une famille :

```
manual F
  family
    relations
      **.specif              specify   :*;
      **                     dep       :I1;
      (F4:*) or (F4:*:*)     dep       :I2;
      reflex                 compile   **(element = SourceText)
```

N'importe quel document specif peut spécifier une interface de F; la cible ":*" est donc implicitement vue comme "F:*".

N'importe quel objet/élément peut dépendre de l'interface $F:I_1$.

Seules les interfaces ou réalisations de la famille F_4 peuvent dépendre de l'interface $F:I_2$.

La commande (relation réflexive) "compile" s'applique à tous les textes source de F.

On a la hiérarchie de priorités suivante (selon P5.1) :

p(specify) > p(dep) > p(compile). ♦

5.2.2. Clause relations d'un manuel d'interface, de réalisation ou d'usager

La clause relations d'un manuel d'interface, de réalisation ou d'usager joue un **rôle factuel** (langage de documentation du projet).

Les *occurrences* de relations binaires entre objets (ou éléments associés à des objets) du type interface, réalisation ou usager sont consignées (par ordre de priorité décroissante) dans les clauses relations des manuels des objets *cibles* de ces relations.

On peut se demander pourquoi les occurrences d'une relation sont mémorisées dans le manuel de l'objet cible de cette relation. Comme on le verra à la section 6., les propagations d'effets de bord se font en "remontant" les relations; chaque objet doit donc "savoir" de qui il est la cible ! Si on a dès lors identifié une relation binaire pour laquelle les effets de bord se propagent en descendant le long de cette relation, il faudra plutôt utiliser sa relation réciproque.

Une **représentation graphique** assez immédiate des objets/éléments et relations s'impose : objets/éléments deviennent des noeuds et les relations se traduisent par des arcs étiquetés par le nom de la relation pour former un multigraphe orienté.

Exemple (de clause relations d'une interface) :

Soit la clause relations suivante d'une interface :

manual F1:I1
  interface
    relations
      $F_2:I_2:R_3$ dep $F_1:I_1$;
      $F_3:I_1:R_1$ dep $F_1:I_1$;
      U_5 writes $F_1:I_1$.specif;

Cette clause peut être représentée par le multigraphe suivant :

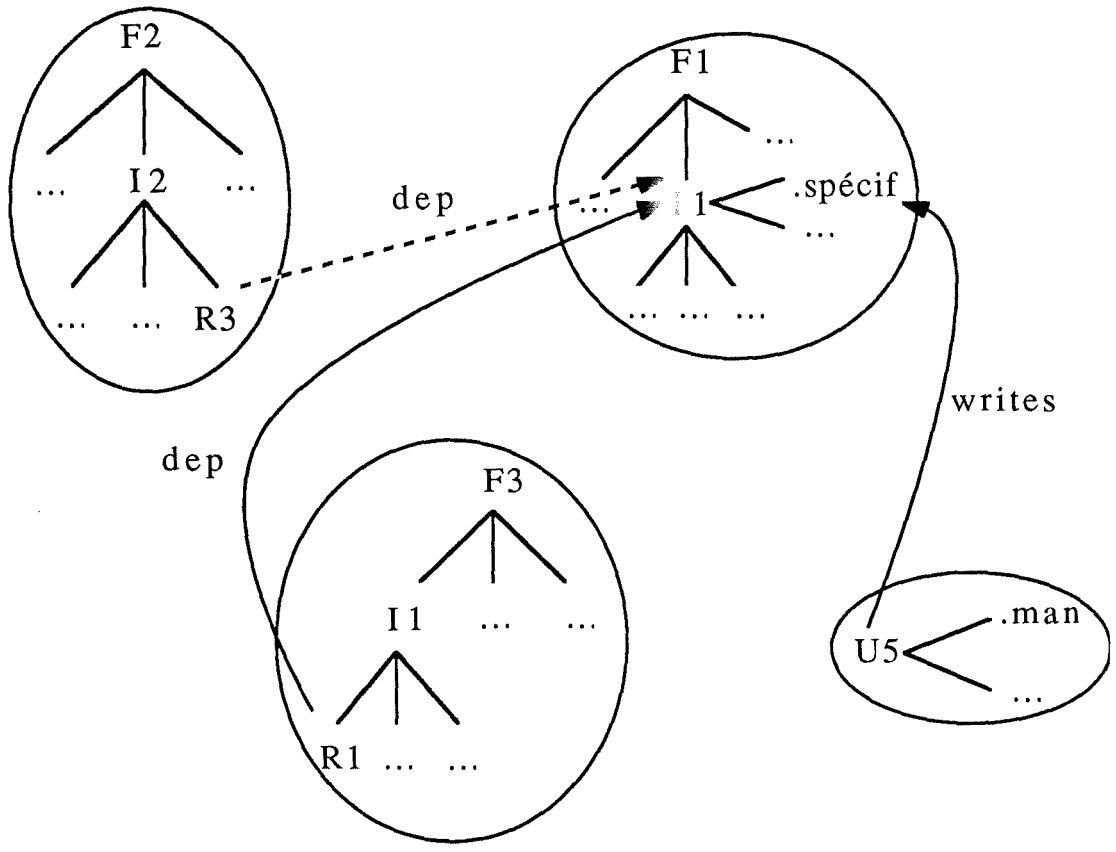


Figure 5.1. - Multigraphe ♦

6. EVENEMENTS ET ACTIONS

La définition des attributs et relations (cf. sections 3. et 5.) constitue la *partie statique* de la caractérisation des objets. Grâce aux notions d'événement et d'action, nous allons pouvoir exprimer les *aspects dynamiques* de cette caractérisation. Notons cependant que d'autres travaux ont déjà été effectués dans ce domaine ([Smith 77], [Brodie 81, 82]).

Vu l'évolution constante d'un logiciel, il s'avère impératif de pouvoir associer aux objets des procédures contrôlant la validité de leurs attributs et relations, garantissant à la base de programmes la cohérence nécessaire. Les notions d'*événement* et d'*action* sont basées sur l'existence de relations. Tout changement dans la base de programmes génère un événement auquel des actions peuvent être associées grâce à des clauses actions du langage de définition.

6.1. Types d'événement

Un **événement** correspond à la survenance instantanée d'un phénomène dans le monde réel et dénote sous quelles circonstances des actions doivent être déclenchées.

NOMADE distingue deux types d'événement : les événements associés aux émissions de commandes, ainsi que les effets de bord.

6.1.1. Evénements de type "command"

Ces événements sont générés lors de l'émission de commandes. Comme nous l'avons vu dans la section 5.1.2., toutes les commandes-NOMADE sont définies en tant que relations réflexives.

Par exemple, la déclaration de la commande "compile" portant sur tous les textes sources d'une famille se trouve dans la clause relations de cette famille :

relations
  reflex compile ** (**element** = SourceText);

En fait, NOMADE ne connaît que la commande (fictive) **command**, qui a deux arguments :

- un nom de relation réflexive (i.e. une commande);
- un nom simple (désignation d'objet/élément).

L'exécution d'une commande C sur un objet/élément O (à la date D par l'usager U) est interprétée comme l'exécution de **command** (C,O) et génère l'événement E(O,C,O,**command**,U,D). Nous verrons dans la section suivante que cette notation est en parfaite harmonie avec celle des effets de bord.

Pour tout événement de type **command**, les **actions** associées se trouvent dans des blocs **command** des clauses actions des manuels des objets (O) manipulables par la commande (C) en question :

```
manual O
  actions
    command
      for C do
        ...programme d'actions...
      done;
```

Exemple : définition de la commande "compile" pour les langages Pascal et C ($S désignant l'objet manipulé) :

```
actions
  command
    for compile do
      if $S (language = pascal)
        then pc $S.p;
        else if $S (language = C);
                then cc $S.c;
                else echo 'Langage inconnu';
             fi;
      fi;
    done;
```

Suivant le langage dans lequel $S est écrit, il y a appel au compilateur correspondant (pc ou cc) ou affichage d'un message d'erreur si le langage est inconnu. ♦

En réalité, les commandes se classent en deux catégories : les *commandes externes* et les *commandes internes* (ou : *primitives*). Les utilisateurs ne peuvent invoquer que les commandes externes. Celles-ci sont implémentées en termes d'autres commandes externes et en termes de commandes internes.

Pour les distinguer en cas de conflit de nom, les commandes internes sont précédées de la lettre I majuscule. A la commande externe 'catal' (écriture d'un manuel de l'espace de travail utilisateur vers la base de programmes) correspond la commande interne Icatal.

6.1.2. Effets de bord

Considérons la figure suivante :

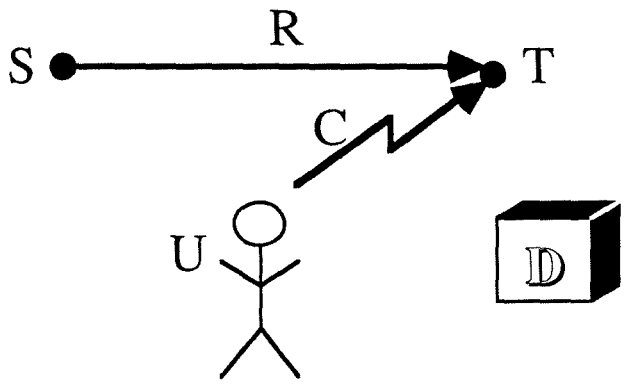


Figure 6.1. - Modification de T par U

Soit l'usager U qui modifie T via la commande C à la date D. Le programme d'actions attaché à C peut alors générer (primitive **propagate** - cf. section 6.2.) un événement de type effet de bord, car S (relié à T via la relation R) peut être devenu incohérent. Cet événement, noté E(S,R,T,C,U,D), déclenchera des actions qui se trouvent dans la clause actions du manuel de l'objet source (S) :

```
manual S
  actions
    for R(C) do
      ...programme d'actions...
    done;
```

Un programme d'actions peut utiliser les variables $S, $R, $T, $C, $U et $D, qui seront substituées dynamiquement par les valeurs correspondantes de l'événement déclencheur. Des variables de travail ($0, ..., $9) sont également disponibles. Les commandes reconnues par le système d'exploitation sous-jacent peuvent être utilisées.

Remarque : au cas où une commande génère des effets de bord sur plusieurs relations différentes, il se pose la question de savoir dans quel ordre ils seront pris en charge. La réponse est qu'ils sont automatiquement ordonnancés par l'ordre de priorité décroissante défini sur les relations impliquées (propriété P5.1. section 5.2.1.). ♦

On distingue quatre moments de déclenchement des actions associées à un effet de bord : postaction, obligation, sur accès et sur demande.

6.1.2.1. Moment "postaction"

L'effet de bord est pris en charge *après* l'exécution des actions associées à une commande. De telles actions sont appelées des postactions et renvoient une valeur booléenne reflétant si la base de programmes se trouve dans un état satisfaisant (évaluation de post-conditions).

Si toutes les postactions déclenchées par les effets de bord d'une même commande C renvoient **true**, alors les mises à jour de la base de programmes effectuées par les actions associées à C sont validées (opération analogue au "commit-transaction" des SGBD).

Dès qu'une telle postaction renvoit **false**, les mises à jour effectuées sont défaites (opération analogue au "abort-transaction" des SGBD) et les événements de type effet de bord qui sont en attente de prise en charge sont supprimés.

Exemple :

Soit la relation 'stabilise' de l'environnement PCTE [Gallo 86] avec 'S stabilise T' signifiant que T ne peut pas être modifié tant que S existe. Pour obtenir l'effet de cette relation sous NOMADE, il suffit de mentionner dans le manuel de S :

```
manual S
  actions
    postaction
      for stabilise (*) do
        fail;
      done;
```

Toute commande C qui modifierait la cible T serait défaite (en supposant qu'elle génère l'effet de bord E(S,stabilise,T,C,U,D)), car la primitive **fail**, renvoyant toujours la valeur **false**, fait échouer C. ♦

6.1.2.2. Moment "obligation"

Si (et seulement si) les mises à jour d'une commande sont *validées* , alors il y a déclenchement automatique des actions correspondant aux effets de bord de moment "obligation". De telles actions sont appelées des obligations.

Exemple :

Soit la commande 'modify I' par laquelle l'utilisateur U modifie l'interface I. Supposons que le programme d'actions de 'modify' génère des effets de bord sur toutes les réalisations dépendant de I, par exemple : E(R,dep,I,modify,U,D). L'action associée pourrait alors être :

```
manual R
  actions
    obligation
      for dep (modify) do
        if not (compile $S)
          then    record ();
                  state ($S, interface_modifiée);
        fi;
      done;
```

Si la compilation de $S échoue, l'événement E est enregistré (commande **record** - cf. section 6.2.) et l'état (attribut **state**) de $S devient "interface_modifiée" (grâce à la commande **state**). ♦

6.1.2.3. Moment "access"

Lors de l'accès à un objet, les actions associées aux effets de bord enregistrés dans son manuel par un programme d'actions quelconque (grâce à la commande **record**) sont exécutées. Ceci permet d'éviter l'utilisation d'un objet incohérent.

Exemple :

Soit une relation 'utilise' définie entre la source S et la cible T. Supposons que (suite à 'modify T') l'attribut **state** de S a reçu la valeur 'périmé' et que l'événement E(S,utilise,T,modify,U,D) a été enregistré dans le manuel de S.

Lors du prochain accès à S, il faudra le mettre à jour et positionner son attribut **state** à la valeur 'à_jour'. Nous écrirons donc dans le manuel de S :

```
actions
  access
    for modify (utilise) do
      ...séquence de mises à jour de $S...
      state ( $S , à_jour );
    done;
```

♦

6.1.2.4. Moment "demand"

Les actions associées aux effets de bord enregistrés dans le manuel d'un objet peuvent aussi être exécutées à la demande de l'utilisateur (commande **make** - cf. section 6.2.). C'est une variante et une extension du Make de Unix ([Feldman 79]) : lors de l'exécution du Makefile, les effets de bord sont détectés en se basant sur les dates de dernière modification gérées par le système d'exploitation. Sous NOMADE, la commande **make** déclenche les actions associées aux effets de bord *déjà* détectés et enregistrés.

Exemple :

soient E1 à E4 les événements enregistrés sur S :

E1 (S, R1, T1, C1, U1, D1)
E2 (S, R2, T2, C2, U2, D2)
E3 (S, R1, T1, C2, U3, D3)
E4 (S, R1, T4, C3, U4, D4)

Voici les événements que va réveiller la commande **make**, utilisée avec les paramètres source, relation et commande :

make (S)	réveille tous les événements E1 à E4;
make (S, R1)	réveille E1, E3 et E4;
make (S, *, C2)	réveille E2 et E3;
make (S, R2, C2)	réveille E2. ♦

6.2. Primitives du langage de description des actions

Nous présentons ici les quelques primitives du langage de description des actions qui sont nécessaires à l'élaboration de politiques de gestion des événements/actions.

* **propagate (S, R, T, C, U, D)** : génère l'événement E(S,R,T,C,U,D). Cette commande est intéressante pour propager un effet de bord constaté sur un objet S.

* **record (S, R, T, C, U, D)** : cette commande enregistre l'événement correspondant dans le manuel de la source S. record () enregistre l'événement courant.

* **clear (S, R, T, C, U, D)** : cette commande retire l'événement correspondant de la file d'attente des effets de bord en attente de prise en charge.

* **make (S, R, C)** exécute les actions relatives aux événements enregistrés dans le manuel de l'objet S, relatifs à la relation R et à la commande C. Les valeurs par défaut sont R = * et C = * de sorte que make (S) exécute toutes les actions associées aux événements présents dans le manuel de l'objet S.

* **fail** : renvoie toujours **false**.

7. DROITS USAGERS

Remarque préliminaire :

Dans cette section, nous ne donnerons qu'un bref aperçu de l'aspect droits usagers; nous y reviendrons en détail dans la partie II de ce travail. ♦

7.1. Expression des droits usagers : clause rights

Par droits usagers, on entend classiquement un ensemble de triplets (U, C, O) exprimant que l'usager U peut manipuler l'objet O par la commande C. Le système de protection présenté ici dévie de ce schéma par deux détails :

- O n'est pas nécessairement un objet unique, ni même une énumération d'objets (définition en extension), mais en toute généralité un prédicat désignant tout un ensemble d'objets (définition en compréhension), c'est-à-dire une expression-NOMADE; ceci permet une expression plus compacte et élégante des droits usagers;

- il n'est pas rare que des commandes différentes se trouvent reliées, pour un même groupe d'utilisateurs, aux mêmes objets (par exemple parce qu'elles correspondent à un même type d'accès : lecture, écriture, ...); ces commandes sont alors regroupées sous ce que nous appelons une étiquette.

Dans notre contexte, nous aurons donc plutôt des triplets (U, E, P) exprimant que l'utilisateur U peut, pour toutes les commandes liées à l'étiquette E, manipuler les objets/éléments satisfaisant le prédicat P. Pour chaque utilisateur U, on trouvera dans son manuel une **clause rights** contenant la liste des couples (E, P) tels que (U, E, P) ait lieu.

7.2. Vérification dynamique des droits usagers

Pendant toute une session de travail sous NOMADE, les droits d'un usager sont considérés statiques (c'est-à-dire comme étant ceux qui étaient en vigueur au moment du "login"), même s'ils subissent des modifications (cf. section 7.3.)

Nous avons vu (cf. sections 5.1.2. et 6.1.1.) que chaque commande correspond en fait à une relation réflexive et que son implémentation est définie quelque part dans une clause **actions**. Le programme d'actions correspondant à une commande C peut dès lors faire appel à une autre commande prédéfinie : **ChkRight** (E,O), dont la mission est de vérifier si l'usager courant U peut ou non exécuter la commande C sur l'objet O, et ce en vérifiant si l'expression-NOMADE associée à l'étiquette E dans les droits de U désigne ou non l'objet/élément O.

7.3. Modification des droits usagers

Les droits usagers ne sont pas fixés une fois pour toutes. Au contraire, ils peuvent évoluer de manière dynamique : un usager doit pouvoir ajouter, retirer ou changer les droits d' (à) une clause rights d'un autre usager, pourvu qu'il ait le droit de le faire. Les commandes **AddRight**(...), **DelRight**(...) et **ChgRight**(...) ont été conçues à cet effet.

8. PARTITIONS

Remarque préliminaire :

Cette section introduit une série de concepts dont l'utilité n'apparaîtra qu'au cours de la section 9. Ceci est dû au fait que les notions de partition et d'héritage sont intimement liées. ♦

Jusqu'ici, nous avons vu les différents objets qui constituent la base de programmes, leur désignation ainsi que la structure des manuels. Ces derniers sont donc une espèce de descripteur des propriétés des objets : attributs, relations inter-objets, actions à exécuter en réponse aux événements générés et expression des droits usagers.

Les manuels de famille servent donc à énumérer (rôle définitionnel) :

- les attributs permis pour les interfaces et réalisations de cette famille; et pour chaque attribut, ses valeurs possibles;
- les relations permises pour les interfaces et réalisations de cette famille; et pour chaque relation, son domaine et co-domaine;
- les événements reconnus par les interfaces et réalisations de cette famille; et pour chaque événement, son programme d'actions en réponse.

Les manuels d'interface, de réalisation et d'usager servent donc à préciser (rôle factuel) :

- les qualifications effectives de cet objet;
- les relations effectives ayant cet objet pour cible;
- les événements reconnus par cet objet; et pour chaque événement, son programme d'actions en réponse (on ne peut parler d'événements/actions effectifs, parce que ce ne sont pas des propriétés instanciables);
- les droits propres (s'il s'agit d'un objet de type usager).

On constate deux **problèmes** :

(1) pour les familles, il serait fastidieux de répéter les mêmes énumérations d'attributs & relations permis et événements/actions reconnus au cas où plusieurs familles voudraient imposer les mêmes possibilités, voire des possibilités très semblables : une factorisation s'impose donc;

(2) pour un usager, on ne trouve nulle part une énumération de ses attributs & relations permis et de ses droits par défaut. Il faudrait quelque chose qui joue pour les usagers le rôle définitionnel que joue une famille pour ses interfaces et ses réalisations.

Ces considérations nous amènent alors à la *motivation* du concept de **partition de familles** : le *regroupement* de plusieurs familles ayant une raison d'être commune dans un ensemble (appelé donc partition) possèdant aussi une énumération des attributs & relations permis et événements/actions reconnus : énumération des possibilités maximales, par défaut, de chaque famille membre (solution au problème (1)).

L'intérêt est de pouvoir considérer les partitions en tant qu'entités conceptuelles, ce qui enrichit la désignation des objets. Ceci faciliterait notamment l'écriture des domaines et co-domaines dans les clauses relations définitionnelles, qui décrivent alors les relations intra- et inter-partitions valides.

Le même raisonnement vaut aussi pour des **partitions d'usagers**, sauf qu'ici, la description des attributs & relations permis par la partition joue par rapport à chaque usager membre un rôle définitionnel (solution au problème (2).

Rappelons que ces objectifs ne s'éclairciront totalement qu'au cours de la section 9 (mécanismes d'héritage).

8.1. Définitions

Intuitivement, une **partition** est un ensemble de familles (ou d'usagers) reliées par une relation.

Une relation entre usagers est *explicite* : la clause relations d'un usager indique les occurrences de relation dont cet usager est cible. Mais une relation entre familles est *implicite* : il n'existe pas d'occurrences de relations pour les familles et la clause relations d'un manuel de famille F sert à déclarer les relations pouvant avoir des objets/éléments de F comme cible :

soient E1 ∈ F1 et E2 ∈ F2,
si on a : E1 R E2,
alors on a implicitement : F1 R F2.

Exemple :

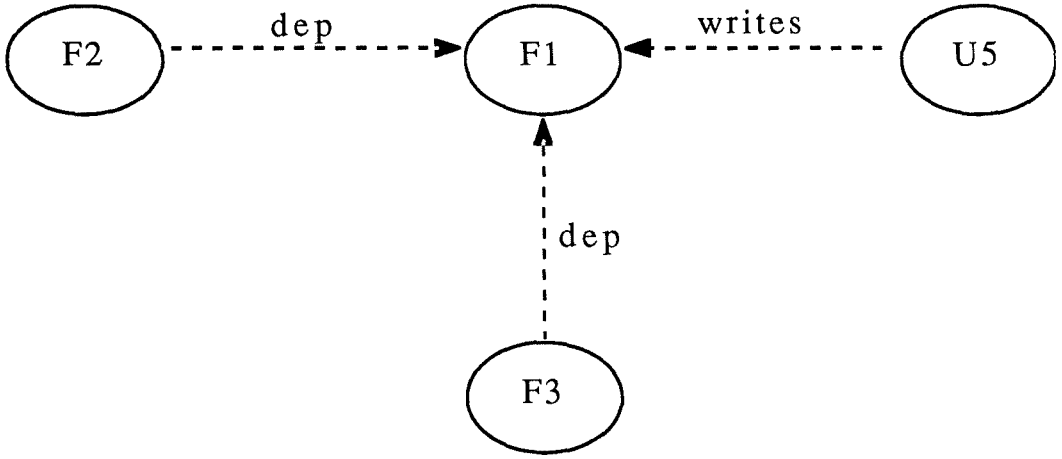


Figure 8.1. - Multigraphe implicite dérivé de la figure 5.1. ♦

Une définition plus *formelle* s'impose :

Définitions :

Une **partition de familles**, notée /R.A/, est l'ensemble des familles du sous-graphe implicite (et obligatoirement acyclique) sur la relation R issu de la famille ancêtre A et tel que tout chemin du graphe entier sur R entre une famille n'appartenant pas à /R.A/ et une famille appartenant à /R.A/ passe obligatoirement par A. ♦

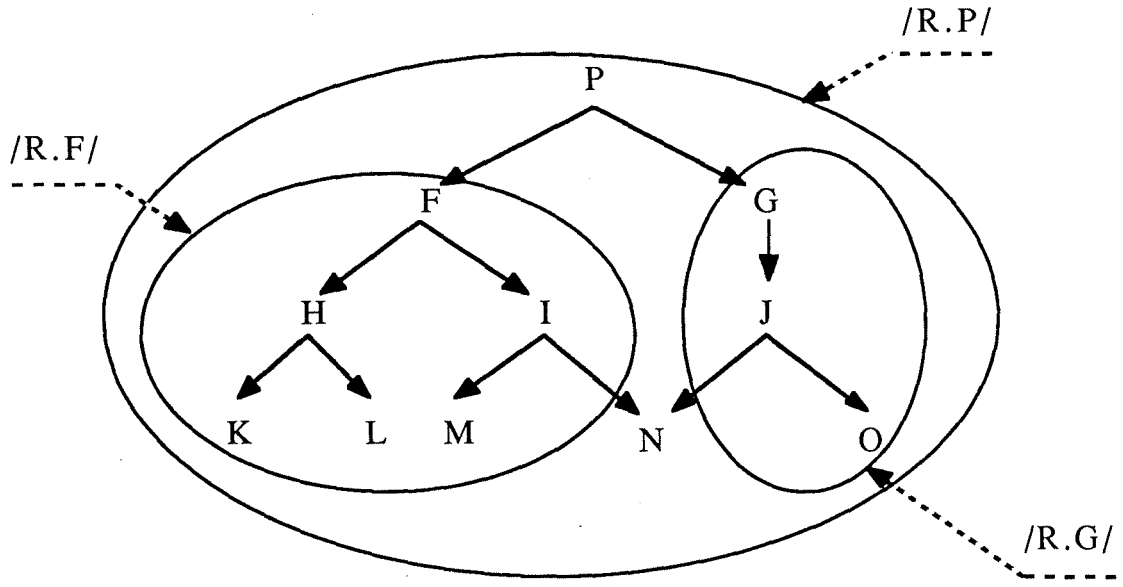
Une **partition d'usagers**, notée /R.A/, est l'ensemble des usagers du sous-graphe explicite (et obligatoirement acyclique) sur la relation R issu de l'usager ancêtre A et tel que tout chemin du graphe entier sur R entre un usager n'appartenant pas à /R.A/ et un usager appartenant à /R.A/ passe obligatoirement par A. ♦

Ces définitions permettent de dériver l'importante propriété suivante, dont l'utilité ne sera élucidée qu'à la section 9.2.2. :

Propriété P8.1 (provisoire) :

Deux partitions sur la même relation structurante ne peuvent se recouvrir. ♦

Exemple de partitions :



Légende : F, G, ... , P : familles (usagers)

⟶ : relation R implicite (explicite)

Figure 8.2. - Partitions

N ∉ /R.F/ car si N ∈ /R.F/, alors tout chemin de J (par exemple) vers N devrait passer par F;

N ∉ /R.G/ car si N ∈ /R.G/, alors tout chemin de I (par exemple) vers N devrait passer par G;

mais N ∈ /R.P/

Si on crée maintenant une occurrence de relation R entre un objet/élément de G et un objet/élément de I, alors il y aura disparition de I et de M de la partition /R.F/ "au profit" de /R.P/;

Inversément, la suppression de la dernière relation entre un objet/élément de J et un objet/élément de N fera apparaître N dans /R.F/, et ce "aux dépens" de /R.P/. ♦

8.2. Déclaration d'une partition : bloc partition

La déclaration d'une partition /R.A/ est effectuée au niveau du manuel de la famille (de l'usager) A par l'ajout d'un **bloc partition** sur la relation R; un tel bloc est composé des 3 (ou 4) clauses déjà vues (attributes, relations, actions et rights) afin de décrire les possibilités que /R.A/ offre à ses familles (usagers) membres.

La relation R est dite **relation structurante** de la partition ainsi définie; de plus, les relations autres que R apparaissant dans la clause relations d'un bloc partition sont dites **relations associées** à la relation structurante R.

8.3. Famille projet

Le multigraphe est issu d'une seule et même famille : la **famille projet**. Cette famille projet joue un rôle très important : par ses blocs partition, elle déclare implicitement les seules relations structurantes permises à travers tout le projet et devient ainsi ancêtre d'une partition pour chacune de ces relations structurantes. De plus, chaque bloc partition de la famille projet déclare implicitement ses seules relations associées permises.

8.4. Partition d'appartenance et partitions emboîtées

Pour des raisons de contrôle des relations et pour des raisons d'héritage (cf. section 9), on convient que, pour une relation structurante donnée R, toute famille (usager) A (du graphe sur R) **appartient** à une seule partition, à savoir celle dont l'ancêtre est le plus "proche" de A (en affectant un poids unitaire à chaque occurrence de R et en considérant tous les chemins remontant les occurrences de R).

D'autre part, sur la figure 8.2., nous découvrons *intuitivement* le concept de **partitions emboîtées**; par exemple /R.F/ et /R.G/ sont chacune englobées par /R.P/ (notation : /R.F/ $\subseteq$ /R.P/ , /R.G/ $\subseteq$ /R.P/).

De manière plus *formelle* :

Définition :

La **partition englobante** d'une partition /R.F/ (avec F différent de la famille projet) est /R.E/ tel que E est la famille la plus "proche" de F (en affectant un poids unitaire à chaque occurrence de R et en considérant tous les chemins sur la relation R menant de F vers la famille projet) qui déclare un bloc partition sur R. ♦

Ainsi, la propriété P8.1. peut être précisée de la manière suivante :

Propriété P8.1 : (définitive)

Deux partitions sur la même relation structurante ne peuvent se recouvrir; soit elles sont disjointes, soit l'une d'entre elles est emboîtée dans l'autre. ♦

Autrement dit, pour une relation structurante donnée, le graphe d'emboîtement de ses partitions est un arbre :

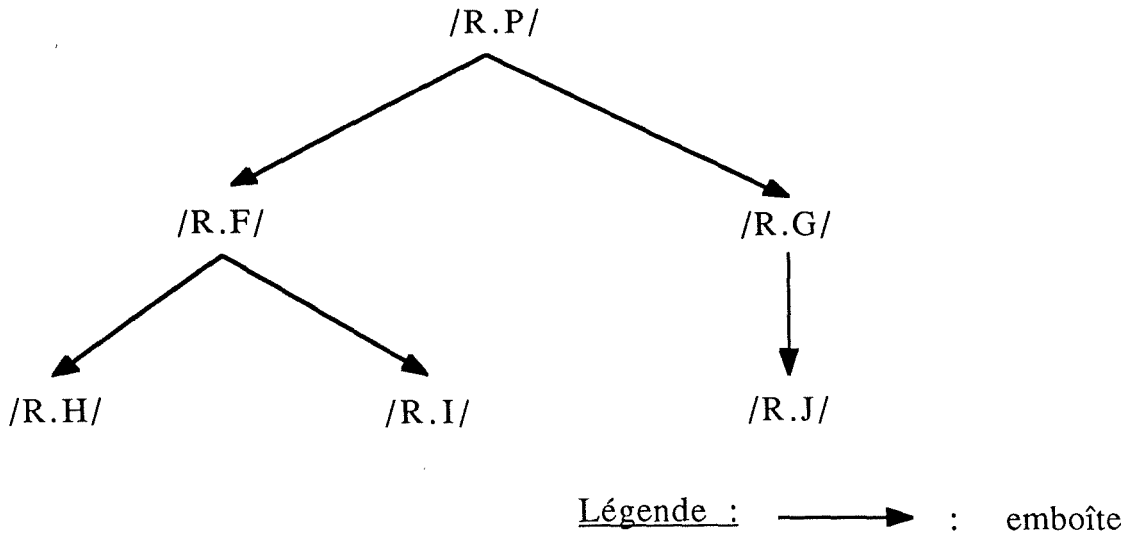


Figure 8.3. - Graphe d'emboîtement des partitions de la figure 8.2.

8.5. Retour à la notion de priorité des relations

Nous pouvons maintenant définitivement régler le problème de la définition de la **priorité** des relations. Comme nous venons de le voir, la famille projet déclare implicitement par ses blocs partition toutes les relations structurantes permises dans un projet, et ce par priorités décroissantes. De même, chaque bloc partition au niveau de la famille projet déclare implicitement par sa clause relations toutes ses relations associées (et ce également par priorités décroissantes : cf. propriété P5.1. section 5.2.1.). D'autre part, nous pouvons énoncer la contrainte suivante :

Contrainte C8.1. :

Toute relation associée ne peut être relation associée que d'une seule relation structurante.♦

Cette contrainte absolument fondamentale sera justifiée en 9.2.2.

La conjonction de ces considérations permet de définir de manière non ambiguë les rapports de priorité entre deux relations quelconques : l'ordre de priorité décroissante de toutes les relations (structurantes ou associées) est l'ordre textuel d'apparition des définitions de ces relations dans les clauses relations des blocs partition de la famille projet.

8.6. Retour aux noms simples

Nous pouvons à présent revenir sur la définition d'un **nom simple**, défini jusqu'à présent (cf. 2.4.) comme étant soit une désignation d'objet, soit une désignation d'élément.

Afin de permettre la désignation d'objet(s) ou d'élément(s) d'objet(s) d'une (de) partition(s), nous préfixons la construction actuelle par le nom d'une partition.

Cependant, un nom de partition /R.A/ peut former tout seul un nom simple : il sera alors interprété comme /R.A/** , i.e. tous les objets et éléments de /R.A/.

Exemple : (cf. figure 8.2.)

/R.F/[G-I]:* désigne toutes les interfaces de H et de I. ♦

9. MECANISMES D'HERITAGE DE CLAUSES

Dans la section précédente, nous avons introduit le concept de partition afin de regrouper des familles à caractéristiques communes et d'éviter ainsi toute ré-écriture fastidieuse de possibilités (clauses attributes, relations et actions) au niveau de ces familles : les possibilités par défaut d'une famille sont celles de sa partition. Derrière cette technique, on devine évidemment un *mécanisme d'héritage* entre la famille et sa partition. Les possibilités héritées par défaut peuvent bien sûr être affinées par la famille, mais sans dépasser les possibilités initialement héritées.

Le même raisonnement vaut pour un héritage de clauses actions et rights entre un usager et sa partition.

Ce type d'héritage a été baptisé **héritage d'instance** et a donc lieu entre une famille (usager) et sa partition (**héritage d'instance simple**), s'il n'y en a qu'une seule, ou ses partitions (**héritage d'instance multiple**) s'il y en a plusieurs (sur des relations structurantes différentes). Notons que le mot "partage" serait plus approprié que héritage d'instance, car une famille (usager) *appartient* à une partition, plutôt que d'y être *incluse* .

D'autre part, nous avons vu la notion de partitions emboîtées. Ici aussi, on pourrait imaginer une espèce d'héritage de possibilités entre une partition et sa partition englobante. Ce type d'héritage a été baptisé **héritage linéaire**.

9.1. Héritage linéaire

Chaque partition /R.F/ (telle que F n'est pas la famille projet) a une et une seule partition englobante /R.E/ au sens de la définition de 8.4. (les autres partitions que l'on pourrait croire englobantes sont en fait partition englobante de /R.E/, partition englobante de la partition englobante de /R.E/, etc.).

Dans la suite, nous entendrons par **clause effective** la clause résultante de l'application du mécanisme d'héritage linéaire à une clause donnée. Comme il n'y a pas de partition englobante pour toute partition issue de la famille projet, il n'y a pas d'héritage linéaire non plus et par conséquent, clause effective et clause héritante y sont identiques.

Définition :

Pour chaque couple (/R.F/, /R.E/) tel que /R.F/ est englobée par /R.E/, on définit l'**héritage linéaire** (qui est donc toujours un héritage simple) comme suit :

> pour chaque clause (**attributes**, **relations**, **actions** et **rights**) de /R.F/, la clause effective est la *réunion* de cette clause et de la clause effective correspondante de /R.E/. ♦

La sémantique de l'opérateur de réunion est définie dans l'annexe C. Un mécanisme d'*inhibition d'héritage* est offert pour empêcher qu'une partition hérite d'une clause (ou partie de clause) de sa partition englobante.

9.2. Héritage d'instance

Selon qu'une famille (un usager) appartient à une seule partition ou à plusieurs partitions (définies alors sur des relations structurantes différentes), nous sommes en présence d'un héritage d'instance simple ou d'un héritage d'instance multiple entre cette famille (cet usager) et cette (ces) partition(s). Nous étendons la notion de clause effective à tous les mécanismes d'héritage.

9.2.1. Héritage d'instance simple

Supposons d'abord le cas d'une famille (d'un usager) F appartenant à une seule partition /R.E/.

Définition :

L'**héritage d'instance simple** entre F et /R.E/ se définit comme suit:

- si F est une famille :
  - * pour chaque clause (**attributes**, **relations** ou **actions**) de F, la clause effective est l'*intersection* de cette clause et de la clause effective correspondante de /R.E/;
- si F est un usager :
  - * pour les clauses **actions** et **rights** de F, la clause effective est l'*intersection* de cette clause et de la clause effective correspondante de /R.E/;
  - * pour les clauses **attributes** et **relations**, il n'y a pas d'héritage d'instance simple (car au niveau d'une partition, ces clauses expriment des contraintes et au niveau d'un usager, elle expriment des instanciations). ♦

La sémantique de l'opérateur d'intersection est donnée dans l'annexe C.

Cet opérateur se justifie par le fait qu'une famille (usager) doit restreindre et spécialiser les possibilités offertes par sa partition. Un des objectifs a été qu'une clause de famille (usager) offre, par défaut, exactement les possibilités de sa clause héritée. Il faut donc qu'une clause héritante soit implicitement (ou explicitement) initialisée à :

* = * , s'il s'agit d'une clause attributes;

** * ** , s'il s'agit d'une clause relations;

* : ** , s'il s'agit d'une clause rights.

9.2.2. Héritage d'instance multiple

Supposons à présent que la famille (l'usager) F appartienne à plusieurs partitions $/R_1.E_1/$, $/R_2.E_2/$, ..., $/R_n.E_n/$ avec R_i tous distincts.

Définition :

L'**héritage d'instance multiple** entre F et les $/R_i.E_i/$ se définit comme suit :

- si F est une famille :

  * pour la clause **attributes** de F, la clause effective est l'*intersection* de cette clause et de la *réunion* des clauses effectives correspondantes des /$R_i.E_i$/;

  * pour les clauses **relations** et **actions** de F, il y a toujours héritage d'instance simple (la justification en sera donnée ci-dessous);

- si F est un usager :

  * pour la clause **rights** de F, la clause effective est l'*intersection* de cette clause et de la *réunion* des clauses effectives correspondantes des /$R_i.E_i$/;

  * pour la clause **actions** de F, il y a toujours héritage d'instance simple (la justification en sera donnée ci-dessous);

  * pour les clauses **attributes** et **relations** de F, il n'a pas d'héritage d'instance (même explication qu'en 9.2.1). ♦

La technique d'intersection avec la réunion se justifie comme suit : comme F appartient à plusieurs partitions, il est logique que F bénéficie de toutes les possibilités offertes par ces partitions (réunion). Ensuite, F doit restreindre et spécialiser (intersection) ces possibilités.

La notion de conflit de nom de l'héritage multiple classique est impossible ici en vertu du fonctionnement des opérateurs de réunion et d'intersection. Ceux-ci sont détaillés dans l'annexe C. Notons seulement que l'opérateur de réunion est appliqué ici entre clauses effectives de partitions (sur des relations différentes), alors qu'il était appliqué entre une clause de partition et une clause effective de la partition englobante dans l'héritage linéaire.

Toute inhibition d'héritage linéaire rencontrée lors du calcul de la clause effective de /$R_i.E_i$/ est annulée pour le calcul de la clause effective de /$R_k.E_k$/ ($R_i \neq R_k$).

Il est clair que l'héritage d'instance simple n'est jamais qu'un cas particulier de l'héritage d'instance multiple.

Expliquons maintenant pourquoi il n'y a pas d'héritage d'instance multiple au niveau des clauses relations et actions :

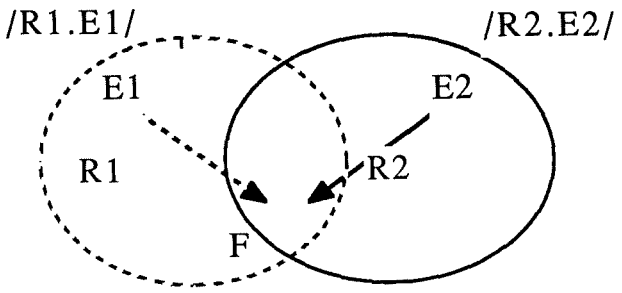
- pour une relation structurante donnée, la propriété P8.1. et la convention relative à la partition d'appartenance (cf. section 8.4) d'une famille (usager) excluent que celle-ci (celui-ci) appartienne à plusieurs partitions sur cette relation;

- d'autre part, la clause relations d'une partition /$R_i.E_i$/ ne peut contrôler que la relation structurante R_i et ses relations associées r_{ij}, qui ne peuvent être relations associées à une autre relation structurante R_k (contrainte C8.1.).

Ainsi, il ne peut y avoir ambiguïté pour déterminer de quelle partition /$R_i.E_i$/ une définition d'une clause **relations** ou **actions** va hériter.

Exemple :

supposons le graphe suivant (tel que p(R1) > p(R2)) :



Considérons les manuels suivants :

```
manual  E1
   family
      partition R1
         relations
            S1  R1   T1 ;
            S2  r11  T2 ;
```

```
manual  E2
   family
      partition R2
         relations
            S3  R2   T3 ;
            S4  r21  T4 ;
```

```
manual  F
   family
      relations
         S5  R1   T5 ;
         S7  r11  T7 ;
         S6  R2   T6 ;
         S8  r21  T8 ;
```

F, E_1, E_2 : familles
R_i : relations structurantes
r_{ij} : relations associées à R_i

Figure 9.1. - Absence d'héritage multiple pour les relations

et supposons que les 2 blocs partition ci-dessus soient les blocs effectifs (après héritage linéaire), alors la clause relations effective (après héritage d'instance simple) est :

```
S5 and S1  R1    T5 and T1 ;
S7 and S2  r11   T7 and T2 ;
S6 and S3  R2    T6 and T3 ;
S8 and S4  r21   T8 and T4 ;
```

Figure 9.2. - Clause relations effective de la famille F

Le même raisonnement permet d'exclure l'héritage d'instance multiple des actions.

9.3. Retour aux clauses

Connaissant la syntaxe des quatre clauses et les mécanismes d'héritage inter-clauses, nous pouvons maintenant expliquer le mécanisme de validation d'une clause attributes ou relations factuelle, l'interprétation d'une clause rights, ainsi que le fonctionnement d'une clause actions.

9.3.1. Validité d'une clause attributes factuelle

La qualification factuelle "A=V" est valide dans une clause attributes d'un manuel d'interface ou de réalisation si la valeur V pour l'attribut A est permise par la clause attributes *effective* de la famille contenant cette interface ou réalisation (c'est-à-dire de la clause attributes de cette famille *et* d'*au moins une* clause attributes effective relative à une des partitions contenant cette famille). Ainsi, par le mécanisme d'héritage d'instance (intersection !), même si la clause attributes d'une famille permet plus de possibilités que la réunion des clauses attributes effectives de ses partitions, elle ne saurait que restreindre cette réunion, ce qui a été un des objectifs.

La qualification factuelle "A=V" est valide dans une clause attributes d'un manuel d'usager si la valeur V pour l'attribut est permise par au moins une clause attributes effective des partitions contenant cet usager.

9.3.2. Validité d'une clause relations factuelle et interprétation d'une clause rights

Un raisonnement analogue vaut pour les clauses relations et rights.

9.3.3. Fonctionnement d'une clause actions

Une clause actions sert à associer des programmes d'actions aux événements reconnus par un objet. Nous avons vu comment fonctionnent l'héritage linéaire et l'héritage d'instance entre clauses actions, mais aussi que la clause actions est difficilement comparable aux trois autres clauses (car redéfinition implicite, même opérateur pour les deux types d'héritage, etc.).

Cette différence s'accentue encore si on considère qu'une clause actions joue toujours le même rôle, quel que soit son emplacement, alors qu'une clause attributes par exemple joue tantôt un rôle définitionnel, tantôt un rôle factuel. Ainsi, nous devons étendre la notion d'héritage d'instance pour qu'il ait lieu entre la clause actions d'une interface (réalisation) et celle de la partition d'appartenance de la famille contenant cette interface (réalisation).

Si un événement E(S, R, T, C, U, D) est généré sur l'objet/élément S, alors la démarche à suivre lors de la prise en charge de E pour trouver le programme d'actions à exécuter est la suivante :

- chercher " **for** R(C) **do** ... **done;** " dans la clause actions associée à l'objet S, (resp. à l'objet contenant l'élément S);
- si absent, alors poursuivre la recherche dans la clause actions de la partition d'appartenance P (contrôlant R) de S (si S est une famille/un usager ou un élément de famille/usager), (resp. de la famille de S, si S est une interface/réalisation ou un élément d'interface/réalisation);
- si absent, alors poursuivre la recherche dans la clause actions de la partition englobante de P, etc.

Cette recherche se terminera toujours au plus tard au niveau d'une partition issue de la famille projet, mais elle peut ne pas être fructueuse : NOMADE étant un noyau, la responsabilité incombe à ses utilisateurs de prévoir des programmes d'actions pour tous les types d'événements susceptibles d'être produits.

9.4. Synthèse

Résumons les différents types d'héritage subis par chaque clause en fonction de son contexte :

contexte/clause	attributes	relations	actions	rights
partition de familles	HL	HL	HL	/
partition d'usagers	HL	HL	HL	HL
famille	HIS / HIM	HIS	HIS	/
interface			HIS	/
réalisation			HIS	/
usager			HIS	HIS/HIM

HL = héritage linéaire
HIS = héritage d'instance simple
HIM = héritage d'instance multiple

Tableau 9.1. - Types d'héritage par clause et contexte

Résumons aussi les significations des clauses effectives en fonction de leur contexte :

contexte/clause	attributes	relations	actions	rights
partition de familles	définition des qualifications ($A_i = V_i$) permises dans les clauses attributes des familles de cette partition	définition des relations (S_i R_i T_i) permises dans les clauses relations des familles de cette partition	définition des programmes d'actions à exécuter en réponse à des événements sur les objets de cette partition	/
partition d'usagers	définition des qualifications ($A_i = V_i$) permises dans les clauses attributes des usagers de cette partition	définition des relations (S_i R_i T_i) permises dans les clauses relations des usagers de cette partition	définition des programmes d'actions à exécuter en réponse à des événements sur les usagers de cette partition	définition des objets/éléments manipulables pour chaque étiquette par les usagers de cette partition
famille	définition des qualifications ($A_i = V_i$) permises dans les clauses attributes des interfaces et réalisations de cette famille; définition des éléments permis	définition des relations (S_i R_i T_i) permises dans les clauses relations des interfaces et réalisations de cette famille	définition des programmes d'actions à exécuter en réponse à des événements sur les interfaces et réalisations de cette famille	/
interface	liste des qualifications ($A_i = V_i$) réelles de cette interface et de ses éléments	liste des relations (S_i R_i T_i) réelles ayant cette interface ou un de ses éléments comme cible	définition des programmes d'actions à exécuter en réponse à des événements sur cette interface ou ses éléments	/
réalisation	liste des qualifications ($A_i = V_i$) réelles de cette réalisation et de ses éléments	liste des relations (S_i R_i T_i) réelles ayant cette réalisation ou un de ses éléments comme cible	définition des programmes d'actions à exécuter en réponse à des événements sur cette réalisation ou ses éléments	/
usager	liste des qualifications ($A_i = V_i$) réelles de cet usager et de ses éléments	liste des relations (S_i R_i T_i) réelles ayant cet usager ou un de ses éléments comme cible	définition des programmes d'actions à exécuter en réponse à des événements sur cet usager ou ses éléments	définition des objets/éléments manipulables pour chaque étiquette par cet usager

Tableau 9.2. - Significations des clauses effectives

9.5. Intérêt de la notion de partition

Il est maintenant temps de rediscuter de l'intérêt conceptuel de la notion de partition. Outre l'enrichissement de la désignation des objets/éléments dans les expressions-NOMADE, la notion de partition permet surtout de définir et de contrôler des relations inter- et intra-partitions.

Du point de vue de la **structuration des logiciels** (architecture logique), il existe notamment la structuration hiérarchique (i.e. organisation en niveaux différents et ordonnés via une restriction des relations permises entre les composants du logiciel). Une hiérarchie "utilise" (ou "dépend_de") permet par exemple de structurer un logiciel par niveaux d'abstraction. Un exemple connu est le modèle ISO (Interconnexion de Systèmes Ouverts) de l'OSI dans le domaine des télécommunications. L'intérêt d'une telle hiérarchie "utilise" réside surtout dans une factorisation du travail (et ce dans toutes les phases du cycle de vie d'un logiciel) et une simplification des composants par élimination de redondances fonctionnelles.

Voyons à présent comment on peut définir sous NOMADE une telle structuration par niveaux (ou couches) d'abstraction.

Exemple :

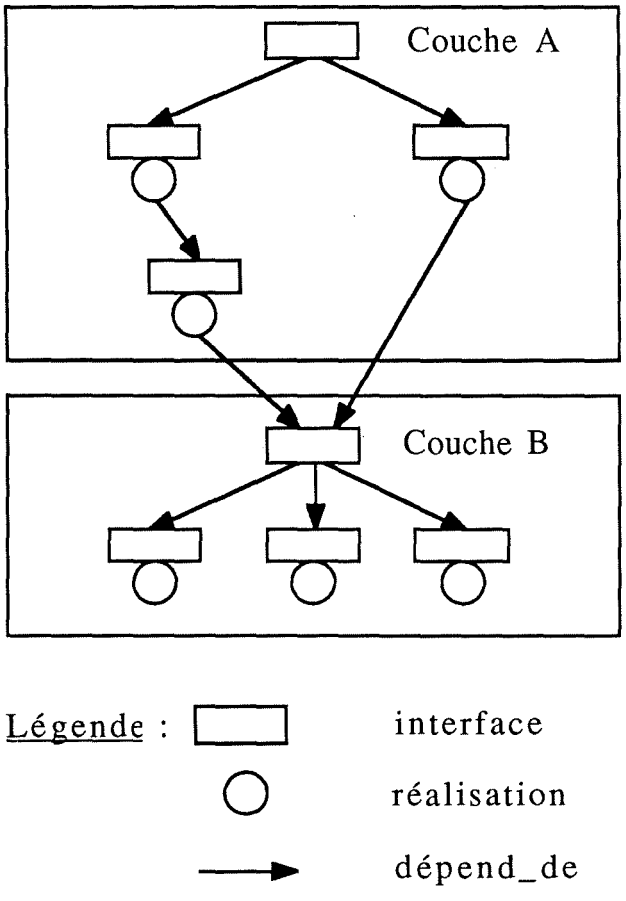


Figure 9.3. - Structuration en couches

A l'intérieur d'une couche, des relations de dépendance quelconques sont autorisées. Cependant, la couche A se trouvant à un niveau d'abstraction plus élevé que la couche B, aucune réalisation de B ne peut dépendre d'une interface de A, mais l'inverse est vrai. De plus, on souhaite que les réalisations de la couche A ne dépendent que des interfaces de la couche A ou d'une interface spéciale de la couche B, interface qui contient des déclarations de toutes les ressources offertes par la couche B à la couche A.

Une telle structuration peut être simulée par des partitions sur la relation de dépendance, mais telles qu'il n'y ait pas d'héritage linéaire entre ces partitions. La figure 9.3. devient alors :

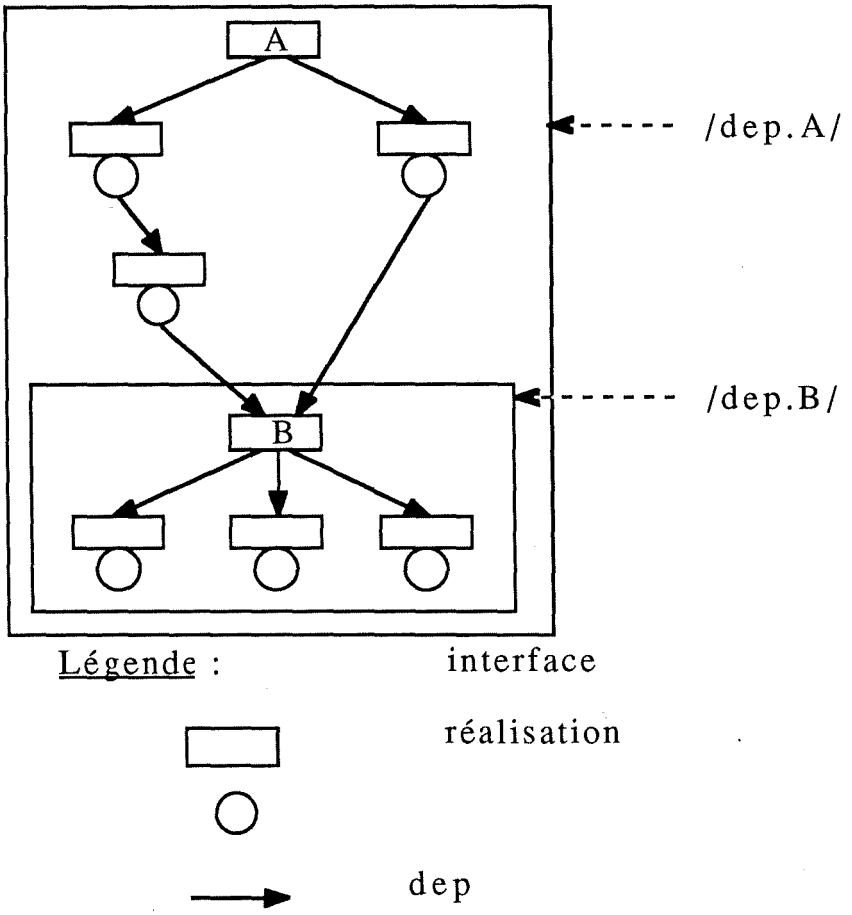


Figure 9.4. - Simulation sous NOMADE de la structuration en couches

Les manuels auront la structure suivante :

```
manual A                              (1)
   family
     :
      partition dep                   (2)
        :
         relations
            /dep.A/  dep  ** ;        (3)
        :
   :
```

```
manual B                              (4)
   family
     :
      partition dep                   (5)
        :
         relations
            /dep.B/  dep  ** ;        (6)
            **  dep  stopinh ;        (7)
        :
```

Figure 9.5. - Manuels des familles A et B

Explications :

(3) doit être lu comme : "/dep.A/** dep /dep.A/** ;", car il y a insertion implicite du contexte défini en (1) et (2); ceci ne permet donc des occurrences de dep qu'à l'intérieur de la partition /dep.A/;

(6) doit être lu comme : "/dep.B/** dep /dep.B/** ;", car il y a insertion implicite du contexte défini en (4) et (5); ceci ne permet donc des occurrences de dep qu'à l'intérieur de la partition /dep.B/;

(7) empêche l'héritage linéaire pour dep entre /dep.B/ et /dep.A/, car sinon la clause relations effective de /dep.B/ serait (sans aucun apport d'information) :

```
relations
    /dep.B/    dep    /dep.B/ ;
    /dep.A/    dep    /dep.A/ ;
```

Figure 9.6. - Clause relations effective de /dep.B/ si pas de stopinh

Cependant, il n'y a toujours rien qui permette que tout objet/élément de /dep.A/ puisse dépendre de l'interface de la famille racine de /dep.B/.

Un cas particulier s'impose donc : on convient qu'une famille ancêtre d'un partition P appartient aussi à la partition englobante de P.

Voilà pourquoi les extraits de manuel de la figure 9.5. sont suffisants tels quels. En effet B ∈ /dep.B/ et B ∈ /dep.A/ , c'est-à-dire (3) permet aussi à tout objet/élément de /dep.A/ de dépendre de l'interface de la famille racine de /dep.B/. ♦

En supprimant l'inhibition d'héritage linéaire, on peut évidemment gérer des structurations moins hiérarchisées, car des relations inter-partitions à priori quelconques sont alors possibles.

Bien entendu, dans un même projet peuvent coexister de multiples structurations, selon des relations structurantes différentes. Par exemple, structuration des programmes en niveaux d'abstraction (relation dep), structuration des usagers en classes (relation affectation), structuration des documents en classes (relation documente), etc.

10. CONFIGURATIONS

Remarques préliminaires :

- la notion de configuration, bien que très importante dans NOMADE, n'est pas essentielle à la compréhension de ce travail; c'est pourquoi cette section se limitera à une présentation assez sommaire. Pour plus de détails, nous renvoyons le lecteur à l'article [Estublier 88a] spécialement consacré aux configurations sous ADELE et NOMADE;

- il existe d'autres travaux largement reconnus dans ce domaine. Citons par exemple les gestionnaires de configuration de première génération - Make ([Feldman 79]), RCS ([Tichy 82]), SCCS ([Rochkind 75]) - et ceux de la seconde génération - DSEE ([Leblang 84]) et SIO ([Bernard 87]). ♦

Un utilisateur doit pouvoir se fier aux spécifications d'une interface en ignorant les détails de son implémentation, qu'elle soit constituée d'un seul code source ou d'un ensemble de codes source dont dépend (transitivement) le code source implémentant l'interface. C'est dans cette optique que la notion de configuration a été définie, facilitant la gestion et la maintenance des grands produits logiciels.

Définition : une configuration est un ensemble cohérent d'objets choisis à raison de un objet par famille et satisfaisant une série de contraintes. ♦

Nous pouvons à présent présenter le type d'objet **realization** comme une espèce de généralisation des notions de **corps** (c'est-à-dire un code source) et de **configuration** (c'est-à-dire un ensemble de codes source) :

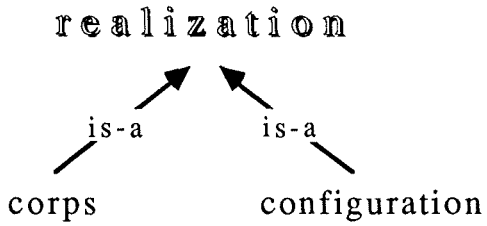


Figure 10.1. - Type d'objet realization

En général, les gestionnaires de configuration ignorent la sémantique des objets qu'ils manipulent. Du moins sont ils souvent limités à l'attribut date_de_dernière_modification, à la relation de dépendance ... La puissance de NOMADE pour la construction d'une configuration réside dans le fait que les composants ne doivent pas être désignés explicitement, mais qu'ils peuvent être choisis sur base de prédicats sur leurs attributs.

Ainsi, dans un système complexe, la définition d'une configuration peut s'exprimer par quelques lignes décrivant la manière de choisir les différents composants : nous parlerons des *contraintes de sélection* des composants d'une configuration.

Remarque : il est important de signaler que la notion de configuration ne se limite pas à des ensembles de codes source. Bien que cela apparaisse moins naturel à première vue, il est possible de construire des configurations de documents par exemple. ♦

10.1. Construction d'une configuration

Une configuration d'une interface ou ensemble des implémentations nécessaires pour sa réalisation (liste de ses composants) est construite de la manière suivante : à partir du graphe de dépendance issu du corps de la réalisation associée à l'interface considérée, la configuration est établie en sélectionnant un corps par famille.

Cette sélection s'avère d'autant plus difficile que le nombre d'objets est grand. Il s'agit de parcourir les familles et de choisir une révision de version de réalisation par famille pour réaliser la configuration.

Pour cela, NOMADE doit pouvoir :

1° déterminer les relations de dépendance et

2° sélectionner un composant.

10.1.1. Détermination des relations de dépendance

Sous NOMADE, il y a deux possibilités de détermination des relations de dépendance. Soit l'utilisateur les fournit explicitement, soit NOMADE les extrait automatiquement du code-source. Cette extraction s'opère par un parcours du code-source et recherche des mots clés indiquant les fichiers importés (ou inclus, ...). Cette recherche s'effectue indépendamment du langage de programmation.

10.1.2. Modalités de sélection des composants

La relation de dépendance mentionnant uniquement une version d'interface, une version et une révision de réalisation doivent être choisies par NOMADE. La sélection s'opère via des **contraintes de sélection** basées sur des attributs.

NOMADE propose trois types de contraintes :

- les contraintes de sélection impérative (**selimp**) : elles doivent être satisfaites par tous les composants;
- les contraintes de sélection conditionnelle (**selcond**) : un composant doit satisfaire une telle contrainte s'il possède les attributs présents dans la contrainte;
- les contraintes de sélection par défaut (**seldef**) : elles permettent d'exprimer des préférences.

Exemple :

```
configuration F:I:C
    selimp
        ** (language = pascal)                    (1)
    selcond
        (** (system = unix4.2, unix5.1))          (2)
    seldef
        ** (type = debug)                         (3)
```

Les spécifications de la configuration F-I-C imposent que :

(1) tous les composants soient écrits en pascal;

(2) si l'attribut système est présent, il doit avoir pour valeur unix4.2 ou unix5.1;

(3) on choisira de préférence des composants en cours de mise au point. ♦

10.1.3. Algorithme de construction

A partir des spécifications (i.e. contraintes de sélection) d'une configuration, l'algorithme construit la configuration qui implémente l'interface en question. L'algorithme procède en largeur d'abord en consultant la fermeture transitive sur la relation de dépendance. Le choix d'une (révision de version de) réalisation par interface s'effectue en deux étapes.

Etape 1 : traitement des contraintes s'appliquant à l'interface; celles-ci se trouvent soit dans les spécifications de la configuration (**contraintes globales**), soit dans le manuel des réalisations choisies (**contraintes locales**). Ce traitement s'effectue en deux parties :

- traitement des contraintes conditionnelles (**selcond**) : si l'attribut de la contrainte est présent dans un corps, la contrainte conditionnelle s'ajoute à la liste des contraintes impératives;

- traitement des contraintes impératives (**selimp**) : construction de la liste de tous les composants qui satisfont les contraintes impératives.

Les contraintes qui entrent en conflit sont détectées et marquées.

Etape 2 : parcours de la liste construite à l'étape 1.

- si un conflit de contraintes a été détecté, l'utilisateur reçoit le message 'configuration inconsistante';

- si la liste ne contient qu'un seul composant, il est choisi; si la liste en contient plusieurs, on applique les règles par défaut : révision la plus récente ou contrainte par défaut (**seldef**) définie par l'utilisateur;

- si la liste est vide, l'utilisateur reçoit le message 'configuration incomplète'.

Exemple : supposons que l'on désire construire la configuration issue de la réalisation $F_1{:}I_1{:}R_1$. Nous associons à cette configuration un certain nombre de contraintes (impératives, conditionnelles et par défaut) que nous avons appelées contraintes globales. Pour simplifier, notre exemple contient des réalisations qui dépendent d'*au plus* une interface.

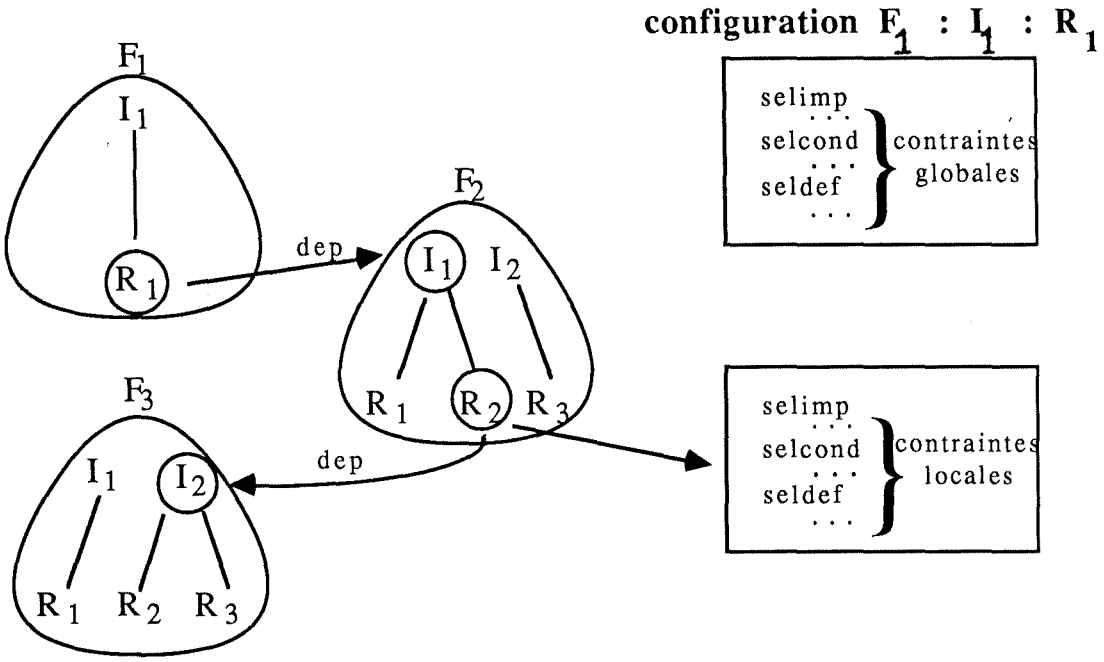


Figure 10.2. - Construction d'une configuration

Posons F_2:I_1, l'interface dont dépend la réalisation F_1:I_1:R_1. Les contraintes globales de la configuration vont servir à choisir une réalisation parmi toutes les réalisations de l'interface F_2:I_1. Lors de l'étape 1, il y a création de la liste des réalisations qui satisfont les contraintes globales impératives (**selimp**). De plus, les contraintes conditionnelles doivent être satisfaites par les réalisations possédant les attributs présents dans ces contraintes globales conditionnelles (**selcond**).

Ensuite (étape 2), trois cas peuvent se présenter. Soit la liste est vide en raison de contraintes conflictuelles ou parce qu'aucun objet ne satisfaisait les contraintes; soit la liste contient une réalisation et celle-ci est choisie; soit la liste en contient plusieurs et on applique les contraintes globales par défaut (**seldef**).

Supposons dans notre cas que la réalisation F_2:I_1:R_2 ait été choisie. Dès ce moment, les contraintes associées à la réalisation F_2:I_1:R_2, que nous appellerons contraintes locales, s'ajoutent aux contraintes globales pour la sélection du prochain composant. Et l'algorithme se poursuit de la même manière ...

Remarque : nous avions fait l'hypothèse qu'il n'y avait qu'une seule relation de dépendance par réalisation. En réalité, une réalisation peut dépendre de plusieurs interfaces de familles différentes. Dans ce cas, l'algorithme procède en largeur d'abord en épuisant toutes les relations de dépendance d'une réalisation avant de traiter les réalisations du niveau suivant. ♦

10.2. Transparence entre configuration et corps

Dans la construction d'une configuration, ce seront tout aussi bien des corps que des configurations qui seront choisis pour former la configuration. Le choix est en réalité transparent à l'utilisateur. Pour lui, NOMADE sélectionne des réalisations, peu importe que ce soit un corps ou une configuration qui se cache derrière cette réalisation. Toutefois, les corps seront plus prioritaires que les configurations dans la sélection des composants d'une configuration.

Ainsi, les configurations sont invisibles à l'utilisateur, leur réutilisabilité est transparente et elles réalisent un bon niveau d'abstraction.

11. ASPECTS TECHNIQUES

Dans cette section, nous aborderons une série d'aspects techniques indépendants les uns des autres.

11.1. Implémentation des éléments de type révision

Vu le nombre parfois élevé de révisions pour une même version de réalisation, ce serait un énorme gaspillage de place mémoire que de maintenir autant de fichiers à contenu semblable.

Voilà pourquoi une technique de *deltas inverses* entre révisions successives d'une même version de réalisation (analogue à RCS [Tichy 82]) est incorporée afin d'empêcher une telle co-existence. Cette technique est cependant entièrement transparente pour les utilisateurs (sauf que le temps de chargement d'une révision semblera un peu plus long que d'habitude). L'expérience avec ADELE (précurseur de NOMADE) a montré que moyennant cette technique, 20 révisions n'occuperont plus que la place de 2 !

11.2. Recouvrement d'erreurs, concurrence d'accès et temps d'accès

Comme tout bon SGD, NOMADE possède des *mécanismes de recouvrement d'erreurs* (afin de rétablir l'intégrité de la base de programmes après un incident).

De même, NOMADE permet le *verrouillage exclusif* d'un objet (et de tout ce qui en dépend) afin d'éviter tous les problèmes dûs à la concurrence d'accès à la base de programmes.

Il est évident (et la partie III l'illustrera parfaitement) que les données les plus sollicitées de la base de programmes sont les manuels. Il est donc impératif d'optimiser les temps d'accès à ces descripteurs d'objet. NOMADE gère à cet effet une espèce de mémoire-cache de manuels : un algorithme du type LRU ("least recently used") assure la disponibilité en mémoire centrale des manuels les plus récemment utilisés.

11.3. Noms externes

Les fichiers constituant la base de programmes sont évidemment protégés par les mécanismes de protection du système d'exploitation sous-jacent de manière à ce que, idéalement, seul le logiciel NOMADE puisse y accéder. Vu la granularité élevée (fichiers !) des composants de cette base de programmes, et vu que leur modification peut durer longtemps et ne peut se faire que via un éditeur (quelconque, c'est à l'usager d'en choisir un), il est préférable que de tels changements se fassent d'abord en dehors de cette base.

Il existe donc deux commandes ("read" et "catal") assurant les transferts de la base vers le répertoire courant (ou mieux : *espace de travail*) d'un usager et vice-versa. Pour ne pas imposer une nomenclature fixe pour les fichiers créés à la suite d'une commande read, NOMADE permet de préciser lors d'un read quel sera le nom du fichier à créer. Ce nom est appelé **nom externe** par opposition aux **noms internes** (ou noms simples). Ceci est particulièrement important car tout usager aura une manière différente de désigner des interfaces, des révisions, etc. Il faut donc qu'il se retrouve facilement dans son propre espace de travail, sans procéder à des changements de nom.

Toutefois, pour des raisons évidentes, toute référence dans un texte source à un élément/objet doit être le nom-NOMADE de ce dernier, car tous les outils de NOMADE travaillent directement dans la base de programmes.

12. COMPARAISON AVEC LES PARADIGMES DE LA PROGRAMMATION ORIENTEE OBJETS

Certaines analogies existent entre l'approche NOMADE et les paradigmes de la programmation orientée objets (POO) (Eiffel : [Meyer 85b,87a,87b,87c]; Objective-C : [Cox 86]; Smalltalk-80 : [Goldberg 83], [Ingalls 81], [LRG 81], [Robson 81]; ...). Il faut cependant clarifier que NOMADE n'impose nullement la programmation en un langage dit "orienté objets", mais que le parallélisme se situe plutôt au niveau de la structuration des informations contenues dans les clauses attributes, relations, actions et rights.

12.1. Classes, objets et héritage

Le mécanisme d'**héritage linéaire** permet de rapprocher les notions de **classe** (POO) et de **partition** (NOMADE), car entre deux partitions /R.F/ et /R.E/ telles que /R.E/ emboîte /R.F/, on a implicitement une relation "is_a" (ou : "est_une_sous_classe_de", ou : "est_une_spécialisation_de") :

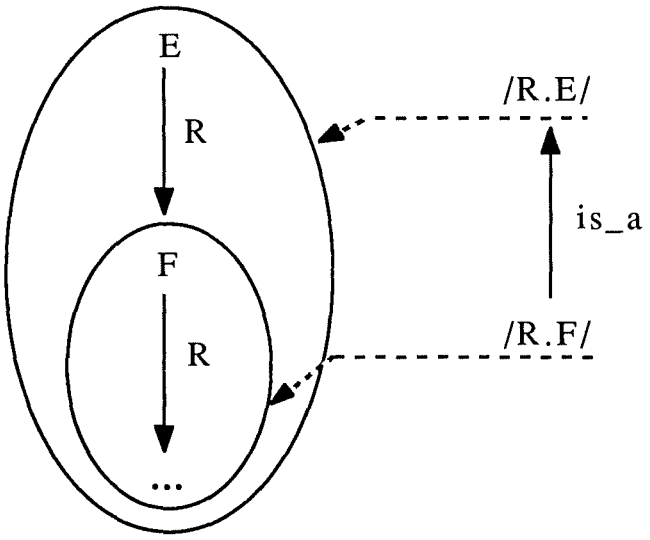


Figure 12.1. - Partitions emboîtées et héritage linéaire

L'héritage linéaire ajoute les caractéristiques de /R.E/ à celles définies explicitement chez /R.F/, mais /R.F/ peut inhiber (partiellement) cet héritage.

Cet héritage linéaire est toujours simple, car des précautions ont été prises pour qu'une partition (non issue de la famille projet) ait exactement une partition englobante, c'est-à-dire qu'elle soit la spécialisation d'une et une seule autre partition définie sur la même relation structurante.

Cependant, la technique d'**héritage d'instance** n'a pas d'équivalent en POO. C'est logique en ce sens qu'il ne s'agit pas d'un véritable héritage (toujours basé sur une relation d'**inclusion**), mais plutôt d'un "partage" de caractéristiques, étant donné qu'une famille (usager) **appartient** à une partition. La mise en oeuvre (intersection !) revient en fait à sélectionner pour la famille un sous-ensemble des caractéristiques offertes par sa (ses) partition(s).

On peut considérer les interfaces, réalisations et usagers de NOMADE comme la contrepartie du concept d'**objet** (d'instance) de la POO : une interface/réalisation appartient à une famille; un usager appartient à une partition d'usagers. Ceci est confirmé par le fait que interfaces, réalisations et usagers ne subissent aucun héritage pour les attributs et relations, mais héritent pour les actions et les droits).

12.2. Attributs

La clause attributes d'une partition et d'une famille est comparable à la définition de la structure des données (variables d'instance, attributs, etc.) d'une classe. NOMADE ne connaît cependant que le type chaîne de caractères, mais ce dernier permet (en utilisant des méta-caractères) la création de sous-types (simples) quelconques.

La clause attributes d'une interface, réalisation ou d'un usager est comparable à l'initialisation de cette structure de données. En d'autres termes, cela correspond au remplissage du "template" défini par la clause attributes effective de la famille, s'il s'agit d'une interface/réalisation, ou de la partition, s'il s'agit d'un usager.

12.3. Relations

La clause relations d'une partition ou d'une famille n'a pas d'équivalent en POO. Cependant, les clauses relations des interfaces, réalisations et usagers en collaboration avec les phrases "partition R" des blocs partition des familles et usagers permettent le calcul du multigraphe implicite entre les familles et usagers du projet, ainsi que le calcul des partitions de ce multigraphe. C'est ainsi que se détermine qui hérite de qui (c'est-à-dire dans le cas de l'héritage linéaire : quelle partition est englobée par quelle partition). Un tel calcul (dynamique) n'existe pas en POO, où chaque classe définit explicitement (statiquement) sa (ses) superclasse(s).

12.4. Evénements et actions

La génération d'un événement E(S, R, T, C, U, D) sous NOMADE correspond à l'envoi d'un message (au sens Smalltalk) à S. R et C permettent à S de sélectionner l'action correspondante. Les paramètres réels (mais implicites) sont ceux de l'événement.

Les clauses actions de partitions, familles, interfaces, réalisations ou usagers sont la réplique exacte des définitions de routines (méthodes, etc.) par les classes en POO. Dans une expression du type "**for** R(C) **do** ... **done** ;", c'est en fait le couple (R, C) qui correspond au nom de la routine.

Dans NOMADE, il n'y a pas de sélecteurs, car le passage de paramètres y est implicite et ces paramètres sont toujours les mêmes : les paramètres formels sont $S (équivalent de **self** en plusieurs langages orientés objets), $R, $T, $C, $U et $D. Toutes les actions sont des routines du type procédure et il n'y a pas de valeurs en retour (sauf une information quant à la réussite ou l'échec de l'action).

13. COMPARAISON AVEC L' APPROCHE ERA

La base de programmes gérée par NOMADE ne s'appuie pas sur un SGBD existant. Ainsi, il peut être intéressant de comparer les principes du modèle de données de NOMADE avec ceux d'un outil conceptuel comme les modèles entité-relation-attribut (ERA) ([Bodart 83],...) ou le modèle d'accès généralisé ([Hainaut 86]).

13.1. Types d'objet et types d'élément

Les *types d'objet* (TO) de NOMADE (**family**, **interface**, **realization** et **user**) seraient des *types d'entité* (TE) dans un modèle ERA. Cependant, NOMADE offre à l'utilisateur une gamme beaucoup plus large de niveaux de granularité pour la désignation des objets. Le concept de famille par exemple, regroupe en tant qu'entité conceptuelle, un ensemble d'interfaces et de réalisations. De plus, la notion de partition permet la désignation en tant qu'entité conceptuelle d'un ensemble de familles visées par les mêmes caractéristiques.

Ainsi, grâce aux concepts de famille et de partition, la désignation d'objets ne se limite pas à une occurrence d'un TE, mais s'étend à des sous-ensembles cohérents de TE, ce qui est impossible avec les modèles ERA classiques.

Les TO de NOMADE sont caractérisés par des *attributs* , tout comme les TE. Tous les attributs sont *élémentaires* (*non décomposables*) et leurs valeurs sont des chaînes de caractères de *longueur variable* , mais inférieure à 64 caractères.

Les attributs des interfaces, réalisations et usagers sont *répétitifs* . Seuls les attributs prédéfinis sont *obligatoires* pour ces TO : leurs valeurs sont calculées automatiquement par NOMADE. Les autres attributs sont *facultatifs* , car définis et gérés par les usagers; c'est-à-dire deux occurrences d'un même TO n'ont pas forcément les mêmes attributs facultatifs. Les valeurs de ces attributs ne peuvent contenir de méta-caractères.

Les attributs (après héritage d'instance) d'une famille sont aussi *répétitifs* et peuvent même contenir des méta-caractères. Le rôle de ces attributs n'est pas de caractériser la famille, mais plutôt de donner la liste des attributs possibles pour ses interfaces et réalisations (qu'ils soient *obligatoires* ou *facultatifs*) et pour chacun de ces attributs, de définir son *domaine* de valeurs.

Les *éléments* d'un TO peuvent aussi être considérés comme des attributs, qui sont du type texte alors. De tels attributs sont absolument primordiaux pour une BD de génie logiciel, mais ils ne sont pas directement pris en compte par les SGBD classiques. Les éléments peuvent à leur tour être caractérisés par des attributs, mais ceux-ci ne peuvent plus être du type texte. Le seul élément obligatoire pour tous les TO, à savoir le manuel, ne peut pas avoir d'attributs, car c'est lui qui contient les attributs et les valeurs d'attribut. En plus, le manuel d'une famille définit les éléments facultatifs possibles pour les interfaces et réalisations de cette famille.

13.2. Relations

La notion de *relation* (ou : *association*) de l'approche ERA est aussi présente dans NOMADE, sauf qu'ici, il y a non seulement des relations entre TO, mais aussi des relations entre éléments de TO. Les relations NOMADE sont toujours des *relations binaires sans attributs* (ignorons pour le moment les relations réflexives). D'autre part, la notion de *connectivité* est absente ici, car elle vaut toujours 0 - N.

Il n'y a pas de relations entre familles, car les relations consignées dans le manuel d'une famille ne font qu'indiquer les relations pouvant avoir des (éléments de) réalisations ou interfaces de cette famille comme cible. Seuls les manuels ne peuvent pas participer à des occurrences de relations, car c'est dans le manuel d'un objet que l'on trouve les occurrences de relations.

Curieusement, même si ces relations binaires sont représentées graphiquement par des arcs orientés pour distinguer source et cible, NOMADE ne prévoit que des *chemins d'accès* qui remontent ces arcs, et ce en stockant les occurrences de relations dans le manuel de l'objet/élément cible.

13.3. Divers

Il est intéressant de noter que dans la base de programmes de NOMADE se côtoient le *schéma* (clauses **attributes** et **relations** des familles et partitions) et l'*extension* de ce schéma (clauses **attributes, relations, actions** et **rights** des interfaces, réalisations et usagers).

NOMADE considère ses usagers comme des objets, au même titre que les familles, interfaces et réalisations. Le mécanisme des *droits usagers* est assimilable à celui des *vues d'utilisateur*.

Une BD de génie logiciel ne peut pas se limiter à être une image fidèle du monde réel (aspect passif), mais elle doit réagir de manière autonome dans certains cas (aspect actif). La base de programmes gérée par NOMADE est *active* par la combinaison des notions d'*événement*, d'*action* et de *propagation des effets de bord*.

13.4. Schéma E/A étendu du modèle de données de NOMADE

La figure 13.1. représente le modèle de données de NOMADE sous la forme d'un schéma ERA étendu : types d'entité, relations n-aires, relations *is-a*, rôles multi-domaines, attributs répétitifs et décomposables. Le résultat obtenu est un modèle hybride car parmi les relations se trouve une méta-relation.

Le concept fondamental d'objet dans NOMADE donne logiquement lieu au squelette de ce schéma : une hiérarchie *is-a* (en pointillé sur le schéma) issue du TE OBJECT. Comme sous-types du TE OBJECT, on retrouve bien évidemment les quatre types d'objet de NOMADE : FAMILY, INTERFACE, REALIZATION et USER. De plus, PARTITION apparait également comme un TE : nous justifierons ce choix à la section 15.

Comme nous l'avions mentionné à la section 10., le TE REALIZATION constitue un sur-type de BODY et CONFIGURATION. Le TE PARTITION se décompose aussi en deux sous-types : USER-PART et FAM-PART, puisque les partitions sont soit des partitions d'usagers, soit des partitions de familles.

Un objet étant composé d'une série d'éléments, ELEMENT aurait donc pu apparaître comme attribut multi-valué du TE OBJECT. L'existence de relations auxquelles participe au moins un élément nous a obligé à modéliser ELEMENT en TE. Derrière le TE ELEMENT se cache une hiérarchie *is-a* déduite de la section 2 (types d'élément) représentée par la figure 13.2. Nous y avons cependant supprimé le sous-type MANUAL, car il ne participe jamais à des relations et parce que son contenu a été transféré dans le TE OBJECT.

Les attributs obligatoires du TE OBJECT sont ObjName, AttrClause , et ActClause.

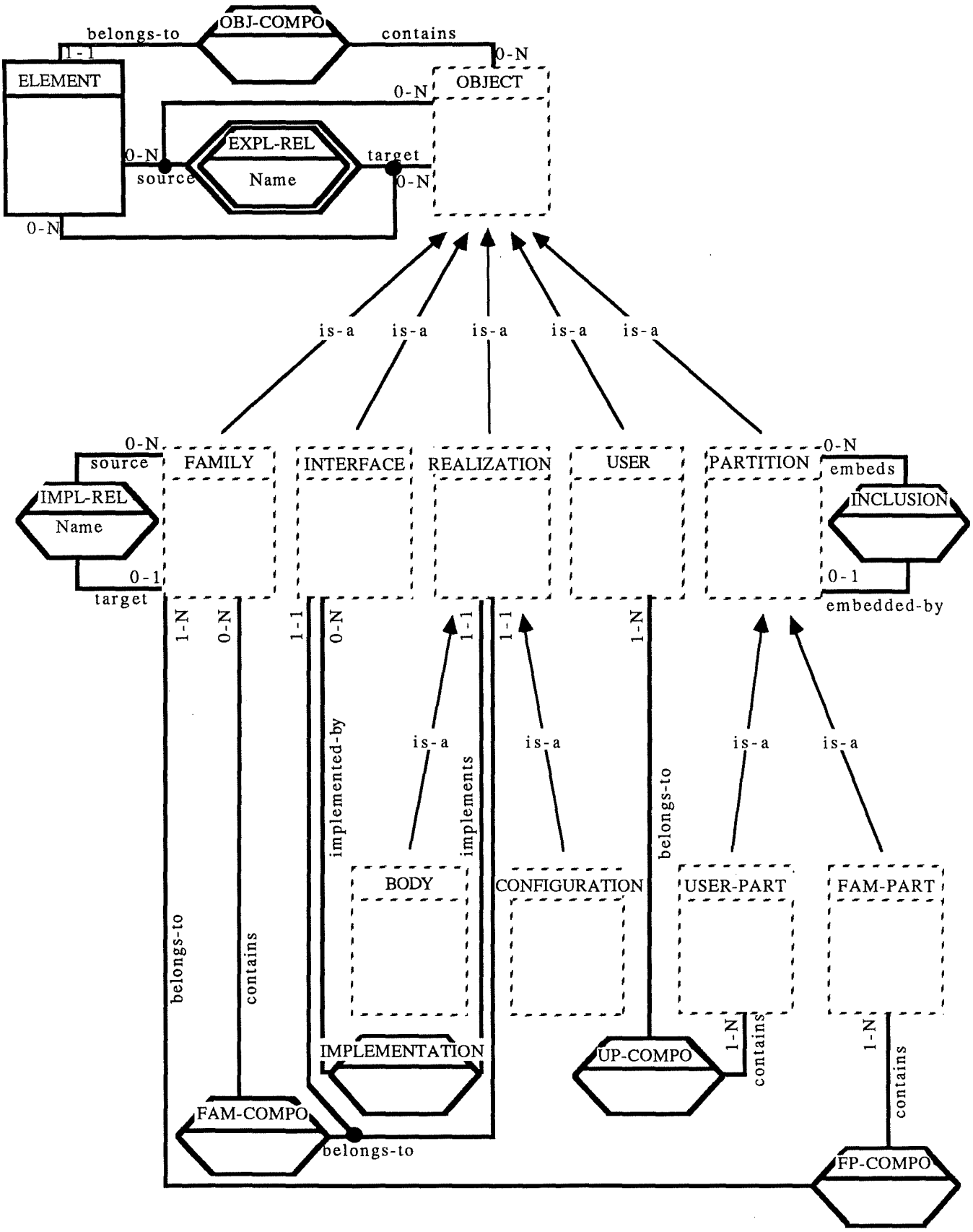


Figure 13.1. - Schéma ERA étendu du modèle de données de NOMADE

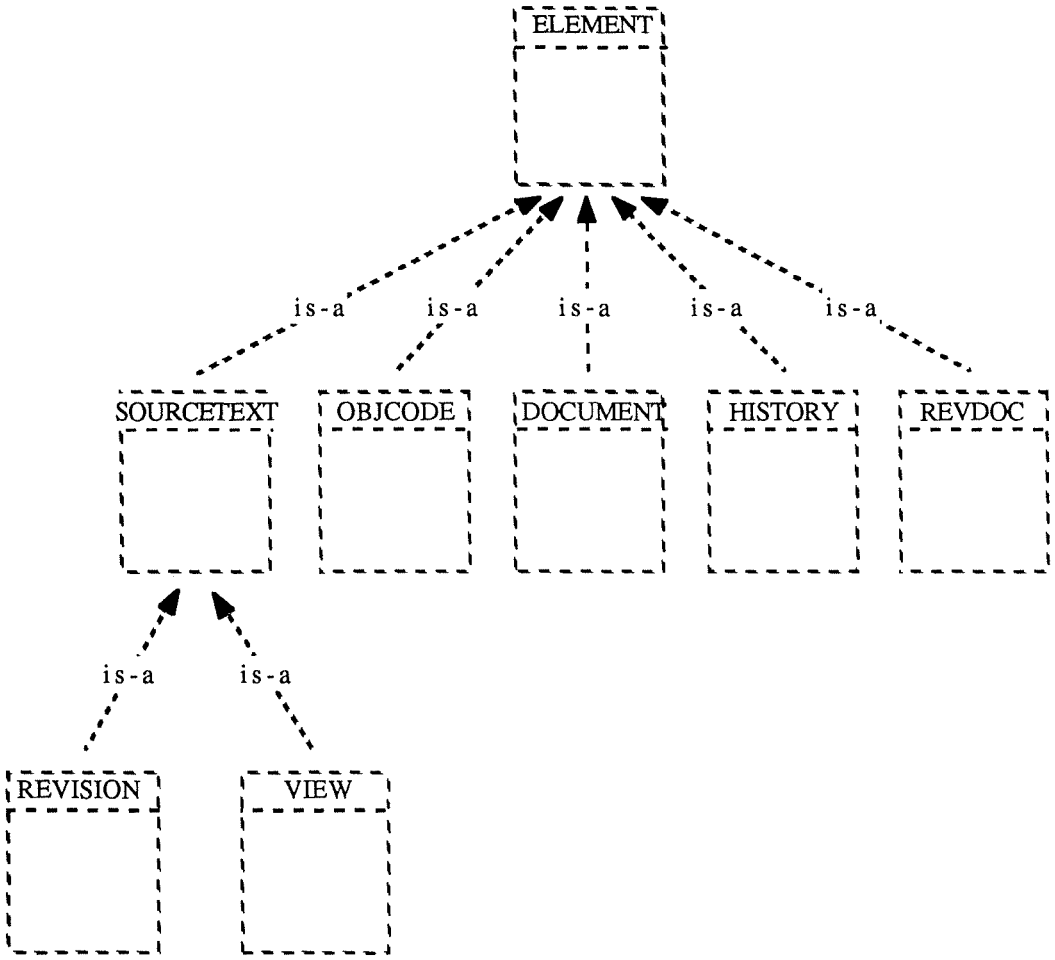


Figure 13.2. - Hiérarchie *is-a* issue du TE ELEMENT

Les attributs suivants sont facultatifs; leur héritage est contrôlé comme suit :

- [RelClause] : hérité par les sous-types FAMILY et PARTITION;
- [RightsClause] : hérité par les sous-types USER et USER-PART;
- [SelClause] : hérité par les sous-types FAMILY, INTERFACE et REALIZATION.

Les attributs du TE ELEMENT sont ElemName et Contents (texte libre). LsDelta est un attribut du sous-type REVISION; il contient les deltas (cf. section 11.1.) des révisions antérieures de Contents.

Analysons avant tout la méta-relation EXPL-REL qui confère au schéma son caractère hybride.

EXPL-REL : en réalité, tout acquéreur du logiciel NOMADE a la possibilité de définir dans les clauses **relations** des blocs **partition** de la famille projet les relations offertes à l'intérieur de son propre modèle de cycle de vie. C'est pourquoi il n'était pas possible de représenter ces relations NOMADE sous forme d'une série de relations entre objets/éléments puisqu'elles sont propres à un modèle de cycle de vie particulier. La présence de la méta-relation EXPL-REL résoud ce problème, exprimant l'ouverture de l'atelier NOMADE au niveau des relations possibles entre objets/éléments. Comme une relation NOMADE peut exister entre objets, entre éléments ou entre objets et éléments, les rôles de EXPL-REL sont multi-domaines.

Examinons à présent les sept autres relations apparaissant sur le schéma :

OBJ-COMPO : un OBJECT contient une série d'ELEMENT (0-N) et un ELEMENT appartient à un et un seul OBJECT (1-1).

IMPL-REL : c'est une synthèse de la relation EXPL-REL puisqu'une occurrence de IMPL-REL existera dès qu'une occurrence de EXPL-REL portera sur des objets/éléments de deux familles différentes;

INCLUSION : modélise les relations d'emboîtement entre partitions. Une partition a au plus une partition englobante (0-1) mais elle emboîte un nombre quelconque de partitions (0-N);

IMPLEMENTATION : cette relation représente le lien entre les TE INTERFACE et REALIZATION. Une réalisation implémente une et une seule interface (1-1); une interface est implémentée par un ensemble de réalisations (0-N);

FAM-COMPO : une famille est composée d'un ensemble d'interfaces et de réalisations (0-N); d'où le rôle BELONGS-TO multi-domaine entre les TE INTERFACE et REALIZATION. Une interface ou une réalisation appartient à une et une seule famille (1-1);

UP-COMPO : une partition d'usagers contient au moins un usager (1-N) et un usager appartient à au moins une partition (1-N);

FP-COMPO : de même, une partition de familles contient au moins une famille (1-N) et une famille appartient à au moins une partition (1-N).

Malgré les différentes extensions au modèle ERA utilisées dans cette modélisation, toutes les finesses et particularités (comme par exemple la complexité des mécanismes d'héritage) de l'atelier NOMADE n'ont pu prendre place sur le schéma. Ces particularités, inexprimables graphiquement, ont donc engendré un grand nombre de contraintes d'intégrité. Une autre série de contraintes d'intégrité permet de gérer la redondance du schéma. Vu l'inutilité d'une énumération exhaustive de ces contraintes, nous avons préféré nous limiter à deux exemples représentatifs.

Exemples :

- les occurrences de EXPL-REL ont pour source et target uniquement des interfaces, réalisations, usagers ou éléments (non-exprimable en ERA).
- les occurrences de IMPL-REL résument les occurrences de EXPL-REL (redondance contrôlée); ♦

14. POINTS FORTS DE NOMADE

14.1. Mécanismes d'activation

Contrairement à certains ateliers (ALMA [van Lamsweerde 86, 88b], PMDB [Penedo 85, 87], SODOS [Horowitz 86], ...), et comme (SKB [Meyer 85a], [Feiler 87]), la base de programmes NOMADE est **active** : à partir d'un changement effectué, des effets de bord (**événements**) peuvent être détectés, et les **actions** déclenchées garantissent en quelque sorte la cohérence des données.

Le mécanisme d'événements/actions est intéressant tant par sa puissance que par sa souplesse. Puissance, car il permet d'exprimer les politiques les plus variées de gestion des effets de bord. Souplesse, car rien n'est figé : le responsable d'un projet peut définir et assurer le maintien des contraintes d'intégrité dynamiques selon ses propres besoins.

14.2. Configurations

La désignation des objets par expressions-NOMADE constitue un autre point fort de NOMADE. Elle est primordiale pour la construction des configurations. Sur base de contraintes de sélection exprimées à l'aide de ces expressions, NOMADE construit la configuration demandée.

C'est ici que se situe la principale contribution des concepteurs d'ADELE et de NOMADE : contrairement aux outils dans ce domaine (Make, SCCS, RCS, DSEE, ...) les configurations ne sont plus caractérisées par des "patterns" syntaxiques de noms d'objets.

14.3. Bonnes performances

Etant donné que NOMADE (tout comme son précurseur ADELE) n'est pas basé sur un SGBD existant pour gérer les accès aux informations de la base de programmes, on peut s'attendre à un maintien des bonnes performances en temps d'accès.

15. AMELIORATIONS PROPOSEES DE NOMADE

Dans cette section, nous aborderons de manière critique quelques aspects de NOMADE qui nous ont gêné au cours de la rédaction de cette introduction à NOMADE. Etant donné le caractère encore fort expérimental des spécifications actuelles de NOMADE, nous ne critiquerons que certains points de vue inhérents à la philosophie de NOMADE. Pour éliminer toute ambiguïté, remarquons que cette critique se base sur les spécifications datant de janvier 1988.

15.1. Familles et partitions

La présentation "classique" de la notion de famille (et c'est celle qui a été reprise jusqu'ici) consiste à définir une famille comme la racine d'un arbre d'interfaces et de réalisations (arbre de hauteur 3), ainsi qu'à dire que famille et usager sont deux types d'objet somme toute assez similaires.

Nous avons trouvé que cette démarche mène à quelques lourdeurs de présentation dues essentiellement aux constatations suivantes :

- une famille (tout comme une partition) joue un rôle définitionnel, alors qu'interface, réalisation et usager jouent un rôle factuel;
- le mot famille a acquis (implicitement et involontairement) une deuxième signification : souvent, il désigne tout l'arbre, plutôt que rien que sa racine;
- famille et usager n'ont finalement pas grand chose en commun.

En vue de remédier à cette situation un peu floue, nous proposons les modifications suivantes :

- abandon de la définition classique d'une famille et adoption de sa définition implicite via une légère adaptation : plutôt que de chercher un nouveau terme pour la racine de l'arbre, nous supprimons cette racine ! Ainsi une famille devient une forêt d'arbres de hauteur 2 (racines = interfaces; feuilles = réalisations) :

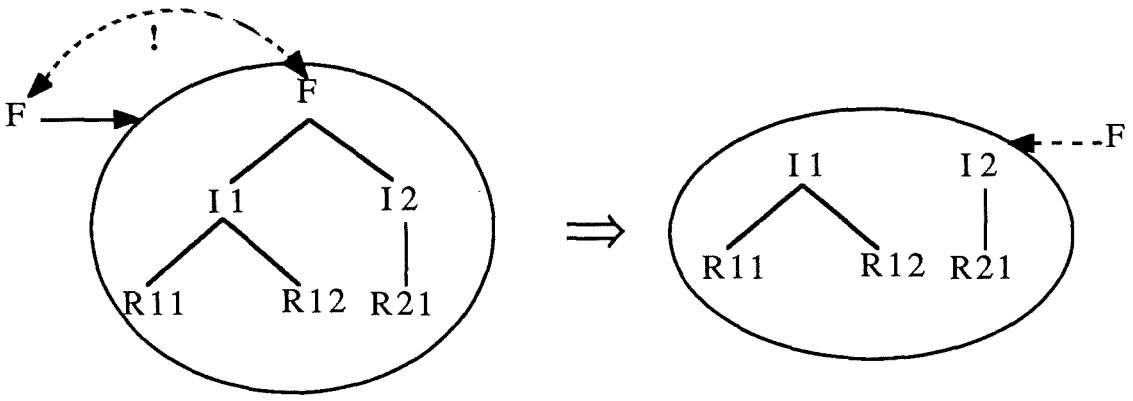


Figure 15.1. - Nouvelle définition de la notion de famille

- la notion de partition devient un type d'objet, dont chaque occurrence dispose d'un manuel propre; i.e. disparition des blocs partition dans les manuels de familles et d'usagers, ainsi que création de l'alternative supplémentaire "Partition" pour ObjType :

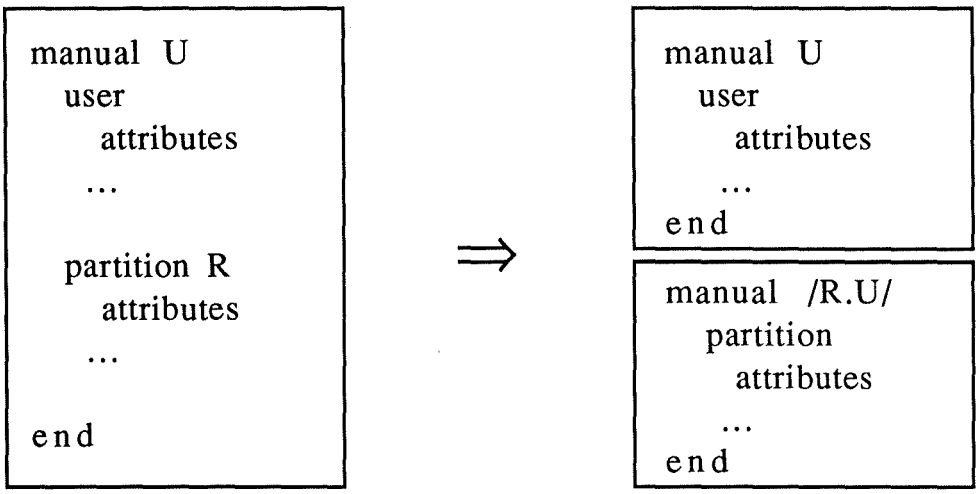


Figure 15.2. - Nouvelle définition de la notion de partition

Ces redéfinitions nous semblent faciliter une présentation plus homogène des types d'objets de NOMADE. Nous avons donc deux types d'objet (famille et partition) dont les instances sont des collections d'objets, ainsi que trois types d'objet (interface, réalisation et usager) dont les instances sont des objets simples.

15.2. Limitation de la couverture du cycle de vie

NOMADE ne vise pas la couverture des phases du cycle de vie d'un logiciel situées en amont de la programmation et de la maintenance de code, à savoir : l'analyse des besoins, la spécification des besoins et la conception. Voilà pourquoi nous avons parlé d'une base de programmes, plutôt que d'une base de projet.

Cet état se reflète dans le nombre limité et fixé de types d'objets proposés par NOMADE. Il y en a quatre (ou cinq, si on tient compte de la section 15.1.) : famille, réalisation, interface, usager et partition, mais aucun mécanisme ne permet la création de nouveaux types d'objet.

De plus, les trois premiers types d'objet sont très fortement orientés vers la phase de codage (en un langage de programmation modulaire). Même si un nombre arbitraire d'éléments quelconques peut être attaché à chaque objet, cette situation oblige à démarrer l'utilisation de NOMADE seulement au moment où l'architecture physique est connue, c'est-à-dire fort tard dans le cycle de vie.

Ce **manque de généricité** du modèle de données de NOMADE induit alors l'impossibilité d'une gestion automatisée des documents produits lors des étapes en amont du codage et l'abandon d'une "traçabilité" entre produits logiciels intermédiaires.

La seule latitude réside dans la définition de relations binaires quelconques entre les types d'objet et types d'éléments prédéfinis (c'est-à-dire dans le choix des instances de la méta-relation EXPL-REL identifiée à la section 13.4.) et dans le choix des attributs des types d'objets.

15.3. Pauvreté du modèle de données

Mis à part la non généricité du modèle de données de NOMADE et le fait qu'il ne couvre pas totalement le cycle de vie d'un logiciel, ce modèle est caractérisé par les faiblesses suivantes:

- les attributs prédéfinis et obligatoires "object" et "element" dénotent de manière explicite la relation existant entre un type d'objet et une occurrence de ce type; dans une modélisation plus propre, une telle relation est implicite car chaque objet "sait" de quel type il est;

- de même, les attributs prédéfinis et obligatoires "family", "interface" et "realization" constituent un codage à bas niveau des relations FAM-COMPO et IMPLEMENTATION que nous avons rendues explicites sur la figure 13.1.;

- les attributs sont non décomposables;

- la distinction entre attributs (au sens classique) et éléments (attributs de type texte de longueur quelconque) est très artificielle; en effet, "document" par exemple est plutôt un type d'objet (pouvant avoir ses propres attributs et relations) qu'un attribut d'un type d'objet;

- les types d'éléments "document" et "revdoc" manquent d'orthogonalité : ils ne se complètent pas car ils n'ont pas de spécificité propre;

- les concepts de versions et révisions ne s'appliquent qu'à des codes source de réalisations : une généralisation à toutes les entités de type texte s'impose;

- seules les relations binaires sans attributs sont permises; une telle limitation donne souvent lieu à de mauvaises modélisations de la réalité, car l'utilité des relations avec attributs et des relations au moins ternaires n'est plus à démontrer;

- le manque de parallélisme entre les rôles d'une partition de familles et d'une partition d'usagers par rapport à leurs éléments (familles et usagers) dévoile le caractère non homogène du modèle de données, car il n'y a pas moyen d'expliquer de manière unifiée le fonctionnement des mécanismes d'héritage.

Nous pensons que tous ces problèmes ne sont pas résolubles en adaptant le modèle de données actuel. Nous suggérons par exemple le recours à une représentation basée sur un modèle ERA auquel on ajoute le concept de partition (c'est-à-dire le mécanisme d'abstraction d'association [Brodie 81]).

15.4. Inadéquation du terme héritage d'instance

En matière d'héritage, le vocabulaire de NOMADE induit malheureusement en erreur : si l'héritage linéaire correspond exactement à la notion classique d'héritage de propriétés/relations entre un type et ses sous-types, il n'en est pas de même pour l'**héritage d'instance**. En effet, ce dernier n'a rien à voir avec l'héritage classique qui a lieu selon une relation d'inclusion (souvent appelée "is_a"), puisque l'héritage d'instance a lieu selon une relation d'appartenance (souvent appelée "is_an_instance_of"). Il serait donc plus approprié de parler de **partage** de propriétés : par exemple, une famille appartient à une partition et jouit donc des propriétés communes offertes par cette dernière.

15.5. Non intégration d'éditeurs structurels

L'environnement NOMADE se présentant comme une couche supplémentaire au système d'exploitation utilisé, rien n'empêche l'utilisation d'éditeurs structurels pour la création et la maintenance de codes sources, documents, manuels, etc. Cependant, de tels éditeurs ne sont pas intégrés dans NOMADE, dont la base de programmes ne contient que des fichiers texte (suite de caractères), car les outils de NOMADE sont basés sur cette représentation interne.

Ainsi, tout texte formel (composition bien formée de structures syntaxiques) mémorisé sous forme d'arbre syntaxique abstrait doit être transformé en fichier texte préalablement à tout transfert de l'espace de travail d'un usager vers la base de programmes (commande "catal").

15.6. Langages

Nous avons vu que dans la base de programmes de NOMADE (et plus précisément dans les manuels) se côtoient la définition du modèle de cycle de vie (le schéma) ainsi que les informations du monde réel correspondant à ce schéma. Ceci implique alors que le **langage de définition du modèle** et le **langage de documentation du projet** ont la même syntaxe (à l'utilisation des méta-caractères près). Ainsi, l'interprétation à donner à une clause d'un manuel est dépendante du type de l'objet décrit par le manuel :

- une clause de famille/partition fait partie du langage de définition du modèle (rôle définitionnel);
- une clause d'interface, de réalisation ou d'usager fait partie du langage de documentation du projet (rôle factuel).

Les opinions sur cette double sémantique d'un langage mixte sont évidemment contraires : les uns la qualifieront d'une rare élégance, d'autres la condamneraient pour son caractère déroutant.

Malheureusement, NOMADE n'offre pas de véritable **langage de requêtes** pour extraire des données de la base de programmes : une commande ("attr" en l'occurrence) extrait les valeurs d'un attribut d'un objet. Les expressions-NOMADE permettent de désigner des sous-ensembles d'objet de la base de programmes, mais leur pouvoir d'expressions est insuffisant en ce sens qu'elles se limitent à contraindre des valeurs d'attribut et pas également des occurrences de relations.

15.7. Déclaration procédurale des priorités des relations

Etant donné que les priorités des relations sont décroissantes d'après leur ordre textuel d'apparition dans les clauses relations des blocs partition de la famille projet, il existe des cas de figure où une telle déclaration procédurale et implicite manque de puissance d'expression.

Soient par exemple deux relations structurantes R_1 et R_2 et leurs deux relations associées respectives r_{11} et r_{21}. Il est impossible de déclarer par cette méthode que : $p(R_1) > p(R_2) > p(r_{11}) > p(r_{21})$.

Voilà pourquoi nous préconisons le recours à une déclaration explicite des priorités de toutes les relations au niveau de la famille projet.

PARTIE II

ELABORATION D'UN SYSTEME DE SECURITE POUR L'ENVIRONNEMENT DE PROGRAMMATION NOMADE

1. INTRODUCTION

Dans tout système de gestion de données (SGD) émerge le besoin d'un **système de sécurité** (ou de protection) qui soit un compromis entre les deux extrêmes que sont l'*isolation* (sécurité maximale, mais aucun partage) et l'*anarchie* (partage maximal, mais sécurité nulle). La mission de ce système de sécurité est donc d'assurer un **partage contrôlé** des données, c'est-à-dire de les protéger contre toute lecture, modification ou destruction non autorisée (frauduleuse ou accidentelle). D'autre part, il est souhaitable qu'un tel système de sécurité (informatisé) soit une image la plus fidèle possible de la vie réelle.

1.1. Composants d'un système de sécurité

1.1.1. Identification et authentification

Classiquement, le premier composant d'un système de sécurité sert à l'**identification** de l'utilisateur du SGD et à l'**authentification** de cette identification. Cette double procédure consiste traditionnellement en la fourniture d'une identité, respectivement d'une preuve de cette identité (mot de passe, par exemple) et est effectuée une fois pour toutes lorsqu'un utilisateur manifeste le désir d'accéder au SGD.

1.1.2. Expression des règles d'autorisation

Le deuxième composant d'un système de sécurité sert alors à l'**expression des règles d'autorisation** attachées à chaque utilisateur. Ici, il existe de nombreuses approches, dont beaucoup de modèles à vocation militaire. Nous n'approfondirons pas les particularités propres aux systèmes de sécurité militaires, qui sont de toute façon incompatibles avec nos objectifs ([Landwehr 81]).

Le modèle le plus fréquemment utilisé pour exprimer des règles d'autorisation est celui de la *matrice d'autorisation* (ou matrice d'accès). Les lignes de cette matrice M correspondent à l'ensemble des sujets actifs (les utilisateurs) et les colonnes correspondent à l'ensemble des objets passifs (les données) manipulés par les sujets. Bien entendu, tous les sujets peuvent aussi être des objets manipulables. La valeur de M[s,o] désigne alors l'ensemble des règles d'autorisation s'appliquant au sujet s et relatives à l'objet o. La matrice M reflète ainsi l'état de protection assuré par le système de sécurité.

Si la granularité des objets répertoriés dans les colonnes de M est une bonne mesure du degré de sophistication du système de sécurité, il existe cependant une meilleure évaluation ([Date 82]) : le type de contrôle exercé par les entrées M[s,o]. Ainsi, on distingue :

- *contrôle indépendant des valeurs* : seul un type d'accès (lecture, écriture, exécution, ...) est mentionné;

- *contrôle dépendant des valeurs* : un prédicat d'accès est ajouté au type d'accès. Ce prédicat s'exprime en termes de valeurs d'attribut et restreint ainsi les données accessibles. Supposons par exemple que o soit le type d'articles "Employé", le prédicat d'accès attaché au type d'accès lecture pourrait alors limiter les accès en lecture aux employés du département "Ventes";

- *contrôle statistique* : des opérateurs statistiques (somme, moyenne, comptage, ...) sont attachés au type d'accès; ainsi, dans l'exemple précédent, on pourrait permettre seulement l'extraction de la moyenne des salaires des employés;

- *contrôle dépendant du contexte* : le prédicat d'accès contient aussi des références à des variables du SGD, plutôt qu'exclusivement des constantes comme "Ventes"; des exemples de telles variables seraient le nom de l'utilisateur, la date et l'heure, l'adresse du terminal, ...

Il est évident que cette matrice d'autorisation doit elle-même être protégée contre toute mise à jour frauduleuse.

Il faut cependant remarquer que, vu le nombre parfois impressionnant de sujets et d'objets, cette matrice risque d'être creuse et que ce serait donc un énorme gaspillage de place mémoire que de vouloir gérer une telle matrice. Face à ce problème, il existe 3 solutions qui permettent une classification primaire des systèmes de sécurité basés sur le modèle de la matrice d'autorisation :

- soit on implémente cette matrice *ligne par ligne* , c'est-à-dire que l'on associe à chaque sujet s la liste des couples (o,M[s,o]) qui le concernent; cette liste est parfois appelée "user profile" et l'approche est connue sous le nom "capability list approach"; des exemples connus sont le modèle "Take/Grant" ([Jones 76]), le modèle HRU ([Harrison 76]), ...;

- soit on implémente cette matrice *colonne par colonne*, c'est-à-dire que l'on associe à chaque objet o la liste des couples (s,M[s,o]) qui le concernent; cette liste est parfois appelée "object profile" et l'approche est connue sous le nom "access control list approach"; des exemples connus sont Unix, le modèle de Hartson ([Hartson 84]), System R ([Griffiths 76]), ...;

- soit on *combine* simultanément ces deux approches; exemple : Multics.

Quelle que soit l'approche choisie, il faut signaler un autre moyen de gagner en souplesse par rapport à la "rigidité" d'une matrice : plutôt que de procéder par énumération des objets manipulables (ou sujets manipulants), on préférera éventuellement les désigner par prédicat. L'avantage essentiel de cette technique est qu'il ne faut pas nécessairement modifier les "profiles" à chaque création/suppression d'objet/sujet.

Dans la suite, nous continuerons cependant à parler de la matrice d'autorisation qui est donc une abstraction utile de nombreux systèmes existants.

Le gros *inconvénient* du modèle de la matrice d'autorisation est qu'il ne permet pas vraiment de contrôler le flux des données (ce qui est l'objectif primaire de tout système de sécurité militaire). Ainsi, une copie de données confidentielles vers un fichier non confidentiel suffit pour anéantir toute intention de protection. Toutefois, ce modèle est réputé pour sa simplicité, généralité et flexibilité.

1.1.3. Vérification des règles d'autorisation

Le troisième composant d'un système de sécurité sert à assurer le **respect des règles d'autorisation** stockées dans la matrice d'accès. Appelé "reference monitor", "arbiter", "security enforcer", "enforcement process", ... dans la littérature, ce mécanisme a pour mission de vérifier si une demande d'accès D[s,o] est compatible ou non avec M[s,o]. Si le contrôle exercé par M[s,o] est dépendant des valeurs des données, alors ce mécanisme de vérification ne peut donner de décision d'accès (accepté/refusé) sans accéder lui-même aux données relatives à o; il en découle que ce processus doit avoir le privilège de lecture de toutes les données. D'autre part, il est clair que le prix à payer pour cette vérification est une certaine dégradation des performances au niveau des temps d'accès.

Il se pose aussi la question du moment où cette vérification a lieu. Idéalement on voudrait qu'elle se fasse le plus tôt possible, par exemple au temps de compilation d'une application (si le contrôle est indépendant des valeurs, bien sûr). Mais ce n'est pas toujours réalisable car plusieurs personnes pourraient se servir de cette application et la matrice d'autorisation est susceptible de changer au cours du temps. En toute généralité, et ce surtout si les opérations se font de manière interactive, donc imprévisible, il faut alors recourir à la vérification dynamique (à chaque demande d'accès).

Que faire maintenant si une demande d'accès est refusée? Tout dépend évidemment de la sensibilité des données, du type d'accès, de l'origine de la demande (application "batch" ou utilisateur interactif), ... Les réactions possibles s'échelonnent d'un simple code retour négatif jusqu'au blocage du terminal, en passant par l'arrêt du programme.

1.1.4. Commandes d'autorisation

Comme nous venons de le remarquer, il est clair que la matrice d'autorisation n'est pas une constante : d'une part, les créations et suppressions de sujets/objets provoquent des créations et suppressions des lignes/colonnes correspondantes, et d'autre part, l'ensemble des règles d'autorisation d'une case M[s,o] peut changer dans le temps (considérons création et destruction de ces règles comme un cas particulier de modification). D'où la nécessité d'un quatrième composant du système de sécurité, à savoir des **commandes d'autorisation** modifiant la matrice d'accès et un **processus d'autorisation** ayant pour mission de vérifier préalablement s'il s'agit d'une transformation "légale" de la matrice, la signification de l'adjectif "légal" étant propre à chaque système de sécurité.

La flexibilité de ce composant peut se mesurer au nombre de personnes pouvant effectuer de telles commandes d'autorisation. Il s'agit de choisir entre la *centralisation* (toute autorisation émane d'une seule personne, appelée administrateur central) et la *décentralisation* (n'importe qui peut, dans la mesure de ses possibilités, modifier la matrice d'accès).

1.1.5. Divers

D'autres aspects touchant à la sécurité sont le *problème d'inférence* (déduction d'informations ponctuelles contenues dans une BD statistique par exemple), le recours à la *cryptographie* pour empêcher des tentatives d'accès aux données sans passer par le SGD, ...

1.2. Fil conducteur

La suite de cette partie sera consacrée à une étude approfondie du système de sécurité de l'atelier logiciel NOMADE.

Comme nous l'avons déjà signalé dans la première partie, NOMADE est une couche supplémentaire au-dessus du système d'exploitation. Il en découle que l'étude qui suit ne sera pas concernée par le premier composant d'un système de sécurité (identification et authentification), car celui-ci est normalement déjà offert par le système d'expoitation.

D'autre part, nous supposerons avoir affaire à des utilisateurs qui n'ont ni emprunté, ni volé, ni forgé l'identification d'un autre utilisateur et qui soient dignes de confiance face au problème du flux des informations (c'est-à-dire qui ne copient pas des données confidentielles dans des fichiers non protégés).

Finalement, nous devons aussi supposer que personne n'essaie d'accéder à la base de programmes (entièrement implémentée sous forme de fichiers texte) sans passer par NOMADE, c'est-à-dire en cassant les protections mises en place par le système d'exploitation. Un remède efficace à de telles infiltrations serait bien entendu la cryptographie, mais c'est là une solution à pondérer soigneusement !

Ainsi, dans la section 2, nous exposerons l'expression des règles d'autorisation (appelées droits usagers sous NOMADE). Ensuite (section 3), nous détaillerons le mécanisme de vérification de ces droits avant d'aborder (dans la section 4) les commandes de modification de ces droits. En conclusion (section 5), nous essayerons d'évaluer ce système de sécurité.

Remarque : les spécifications de NOMADE étant encore fort expérimentales, les considérations suivantes jouent un rôle plutôt exploratoire. Voilà pourquoi nous envisagerons souvent un maximum d'alternatives, afin de donner une base aux expérimentations futures avec des variantes du système de protection proposé. ♦

2. EXPRESSION DES DROITS USAGERS

Il découle de l'introduction qu'il n'existe pas une manière unique et idéale d'expression des droits usagers. Chacune présente avantages et inconvénients, l'important étant qu'elle s'intègre adéquatement au système dans lequel on l'utilise.

Dans cette section, nous examinerons la représentation adoptée sous NOMADE et rappellerons les différents mécanismes d'héritage. Afin de simplifier les notations et pour éviter la rigueur d'une grammaire, nous introduirons une structure de données abstraite accueillant la représentation des droits usagers. Cette structure servira tout au long de cette seconde partie pour la construction des algorithmes de gestion des droits usagers.

2.1. Clause rights

Après avoir explicité l'approche adoptée, nous mettrons en évidence les particularités propres au système de protection de NOMADE.

2.1.1. Approche choisie

L'approche choisie est la "capability list approach", implémentation par usager de la matrice d'autorisation. A ce choix, nous avançons deux justifications. Une justification technique (liée aux performances de NOMADE) que nous donnerons à la section 2.1.3. Une seconde justification, plus rationnelle, serait d'invoquer le fait que les mécanismes d'autorisation reflètent le plus possible la vie courante. De ce point de vue, il semble plus naturel d'indiquer chez les usagers ce qu'ils peuvent manipuler plutôt que de mémoriser dans les objets par qui ils peuvent être manipulés.

La **clause rights** du manuel d'un usager précise quelles commandes il a le droit d'exécuter sur quels objets. La technique la plus rudimentaire consisterait à placer dans le manuel de chaque usager U, un ensemble de couples (C,O) précisant que l'usager U a le droit d'exécuter la commande C sur l'objet O.

Etant donné la taille des projets logiciels que NOMADE est sensé gérer, procéder d'une telle manière rendrait les clauses rights des manuels inexploitables pour NOMADE et incompréhensibles pour l'utilisateur en raison de l'énorme quantité des objets. Nous proposons une notation plus compacte et plus élégante des droits par le passage des couples (C,O) à des couples (E,P) : une **étiquette** (E) regroupe plusieurs commandes, et un **prédicat** (P) désigne un ensemble d'objets.

2.1.2. Notion d'étiquette

Il existe plusieurs critères selon lesquels classer les commandes. Par exemple, on peut décider de répertorier les commandes selon le type d'accès qu'elles nécessitent (lecture, écriture, ...). C'est le cas dans le système Unix où, utilisant l'approche de l'"access control list", on associe des droits aux objets (fichiers) selon le type d'accès nécessité. L'exécution d'une commande sur un fichier sera autorisée si le type d'accès qu'elle exige est permis sur ce fichier. On pourrait également classer les commandes selon le degré de confidentialité des données qu'elles manipulent.

Les deux critères de classement des commandes que nous venons d'évoquer n'ont pas été choisis arbitrairement. En classant les commandes, nous permettons de les regrouper par catégories en fonction du niveau de protection que leur utilisation exige. C'est dans cette optique qu'a été définie la notion d'**étiquette**. Sous NOMADE, les droits ne se rapportent pas à une commande mais plutôt à une étiquette. L'association entre commandes et étiquettes se fait grâce aux programmes d'actions des commandes (cf. section 3.2.1.). Ainsi, à une même étiquette, on pourra associer une ou plusieurs commandes. La notion d'étiquette rend l'expression des droits usagers plus synthétique.

Voici par exemple comment on établirait les droits usagers selon le type d'accès des commandes, comme dans le système Unix :

```
manual U
  user
    rights
      read  : objet1;
      write : objet2;
```

Ce fragment de manuel exprime que l'usager U a le droit d'exécuter les commandes regroupées sous l'étiquette read sur l'objet1 et qu'il peut également exécuter les commandes regroupées sous l'étiquette write sur l'objet2. Cet exemple suppose que les deux associations :

- commandes nécessitant un accès en lecture / étiquette read,
- commandes nécessitant un accès en écriture / étiquette write,

ont été réalisées dans des programmes d'actions. ♦

Remarques :

1° Une réflexion s'impose quant aux *commandes de création* . D'une part, elles ne sont pas spécialement "nuisibles"; on peut donc se demander la raison pour laquelle on pourrait refuser la création d'un objet. D'autre part, comme les objets ne sont pas initialisés lors de leur création, il est impossible qu'ils respectent certaines contraintes sur les attributs. Pour ces deux raisons, il serait inutile d'associer des droits aux commandes de création.

2° On peut se demander ce qu'il en est des *commandes Unix* . Ces commandes seront exécutées si elles sont autorisées par le système de protection de Unix. De plus, les usagers peuvent rajouter des droits sous NOMADE si cela s'avère nécessaire.

3° Nous avons réalisé, tout au long de la première partie, que les *méta-caractères* étaient souvent utilisés dans NOMADE. Les étiquettes sont un exemple où les méta-caractères sont interdits. En effet, la signification d'une étiquette E contenant des méta-caractères pourrait être involontairement étendue lors de la création d'une nouvelle étiquette F qui "matche" E, alors que les commandes associées à F devraient être protégées autrement que celles qui sont déjà associées aux étiquettes qui "matchaient" E. D'autre part, étant donné que les étiquettes désignent déjà des ensembles de commandes, il serait de toute façon inutile de regrouper encore des étiquettes sous des noms d'étiquette avec méta-caractères

Nous avons supposé jusqu'ici que les objets étaient désignés univoquement dans les droits usagers (par exemple objet1 et objet2 ci-dessus), nous allons voir qu'en réalité, la désignation offerte est beaucoup plus générale.

2.1.3. Désignation des objets

Spécifier des droits usagers par énumération d'objets dans de grands projets logiciels rendrait ces systèmes inutilisables. Pour des raisons quelque peu semblables aux justifications des étiquettes, il apparaît assez naturel de placer des mesures de protection par catégorie d'objets, regroupés suivant certaines caractéristiques. Au lieu de devoir énumérer la liste des objets visés par un droit, l'utilisateur dispose des *expressions-NOMADE* , qui permettent d'exprimer en intension des ensembles d'objets.

Exemple :

```
manual U
  user
    rights
      E1 : F1:**;
      E2 : /R.F2/** (date > 01_01_1988);
```

L'usager U a le droit d'exécuter :

- toutes les commandes regroupées sous l'étiquette E1 sur tous les objets/éléments de la famille F1;

- toutes les commandes regroupées sous l'étiquette E2 sur tous les objets/éléments de la partition /R.F2/ s'ils sont plus récents que le 1° janvier 1988. ♦

Remarques :

1° Certains modèles, tel que celui de [Hartson 84], permettent d'accepter ou de refuser l'accès aux informations en fonction du terminal sur lequel l'utilisateur est connecté. Une telle protection est impossible dans NOMADE, car on ne peut attacher aux objets des attributs tels que le terminal. Toutefois, grâce à l'attribut author, un tel *contrôle dépendant du contexte* est possible en utilisant la variable $U référençant l'utilisateur courant.

Exemple : read : ** (author = $U) ♦

2° Nous avions promis à la section 2.1.1. une *justification technique* quant au choix de l'approche "capability list". Supposons que l'utilisateur U exécute une commande s'appliquant à un ensemble d'objets. Si les droits étaient stockés dans les objets (donc dans les manuels de ces objets: "access control list"), pour chaque exécution d'une commande sur un objet, il aurait fallu charger en mémoire le manuel de l'objet en plus de l'objet lui-même. Le chargement de manuel étant une des opérations les plus fréquentes sous NOMADE, il était préférable, de ce point de vue technique, de stocker les droits usagers dans les manuels des usagers ("capability list approach").

3° Grâce à la désignation des objets par des expressions-NOMADE, nous pouvons exprimer à la fois des objets existants, mais tout aussi bien des objets qui *existeront* par la suite.

Exemple : l'expression F1:** désigne tous les objets et éléments de la famille F1. Lorsque de nouveaux objets/éléments seront créés dans cette famille, la signification de l'expression F1:** sera étendue à ces nouveaux objets/éléments sans modification de l'expression. Les portées des expressions-NOMADE sont donc dynamiques : elles varient en fonction de l'évolution des ensembles d'objets/éléments qu'elles désignent. ♦

2.1.4. Droits propres, hérités et effectifs

Expliquons brièvement les diverses nomenclatures (droits propres, hérités et effectifs), ainsi que les mécanismes d'héritage de la clause rights. Précisons qu'une clause rights ne peut se trouver que dans un manuel d'usager ou de partition d'usagers.

2.1.4.1. Droits propres

Le manuel d'un usager précise ses droits propres, ce qui ne correspond pas à ses possibilités réelles. On parlera également des droits propres d'une partition, c'est-à-dire des droits apparaissant dans son manuel (si on convient qu'une partition est un objet).

2.1.4.2. Droits hérités

Distinguons le cas d'un usager et celui d'une partition.

- les droits hérités d'un **usager** sont les droits résultant de l'application du mécanisme d'héritage linéaire. Ils s'obtiennent de la manière suivante : en partant de la (des) partition(s) à laquelle (auxquelles) *appartient* cet usager et en remontant selon les partitions englobantes jusqu'à la partition issue de la famille projet (héritage linéaire), on effectue la *réunion* de toutes les clauses rights.

- les droits hérités d'une **partition** sont les droits résultant de l'application du mécanisme d'héritage linéaire. Il s'obtiennent comme suit : en partant de la partition *englobante* et en remontant selon les partitions englobantes jusqu'à la partition issue de la famille projet, on effectue la *réunion* de toutes les clauses rights.

2.1.4.3. Droits effectifs

Distinguons à nouveau le cas d'un usager et celui d'une partition.

- les droits effectifs d'un **usager** correspondent à l'*intersection* de ses droits propres et de ses droits hérités. La clause rights d'un manuel d'usager U est implicitement initialisée à '*:**;', ce qui signifie que, par défaut, les droits effectifs de U sont égaux à ses droits hérités. Toute modification de '*:**' restreint les possibilités de U.

- les droits effectifs d'une **partition** correspondent à la *réunion* de ses droits propres et de ses droits hérités.

2.2. Structure de données abstraite de la base de programmes

Se baser sur la grammaire de NOMADE (cf. annexe A) pour l'élaboration des algorithmes (cf. annexe E) de gestion des droits usagers et de leurs spécifications aurait entraîné une certaine "lourdeur" dans les notations, tout en imposant au lecteur un exercice de maîtrise inutile. Nous proposons donc une *structure de données abstraite* reflétant la structure de la base de programmes.

2.2.1. Définition des types

Définissons avant tout un premier type, le type Object :

type Object = ObjName : **string**
Man : Manual
...

Il servira à représenter les différents objets de NOMADE. Dans le cadre des règles d'autorisation, Object sera une famille, une interface, une réalisation ou un usager, ou encore une partition, que nous considérons ici comme un objet pour homogénéiser les notations (cf. partie I - section 15.1.). Un objet est constitué entre autres de son nom (ObjName) et de son manuel (Man), le reste ne nous intéressant pas ici.

Précisons ensuite le type Manual :

type Manual = ...
LsRight : **list of** Right
...

Les manuels, comme nous l'avons vu tout au long de la première partie, décrivent les objets. Seule la clause rights d'un manuel nous intéresse ici. Elle est symbolisée par une liste (LsRight) de droits (Right).

Définissons à présent ce que l'on entend par Right :

type Right = Label : **string**
LsNe : **list of** Ne

Un droit (Right) se compose d'une étiquette (Label, notion expliquée à la section 2.1.2.) et d'une disjonction (LsNe) d'expressions-NOMADE (Ne).

type Ne = Sign : {yes,no}
LsLsQN : **list of** LsQN

Une expression-NOMADE (Ne) est une disjonction (LsLsQN) positive (Sign = yes) ou négative (Sign = no) de LsQN.

type LsQN = Sign : {yes,no}
LsQN : **list of** QN

Une LsQN est une conjonction positive (Sign = yes) ou négative (Sign = no) de noms qualifiés (QN).

type QN = Sign : {yes,no}
N : nom simple
LsQ : **list of** Q

Un nom qualifié (QN) est un nom simple (N) positif (Sign = yes) ou négatif (Sign = no) caractérisé par une conjonction (LsQ) de qualifications (Q).

type Q = Attribute : **string**
RelOp : {=,≠,<,≤,>,≥}
LsV : **list of string**

Une qualification (Q) est formée d'un attribut (Attribute), d'un opérateur relationnel (RelOp) et d'une liste (LsV) de valeurs.

Finalement, la variable BP définissant la base de programmes aura la structure :

var BP = **list of** Object

Remarque :

Contrairement à la grammaire des expressions-NOMADE (qui n'utilise que des parenthèses; cf. annexe A), nous employerons les crochets "[" et "]" pour délimiter les expressions-NOMADE, ainsi que les parenthèses "(" et ")" pour délimiter les conjonctions de noms qualifiés. Ceci facilitera la lecture et la compréhension des exemples répartis dans le restant de cette section. ♦

Exemple :

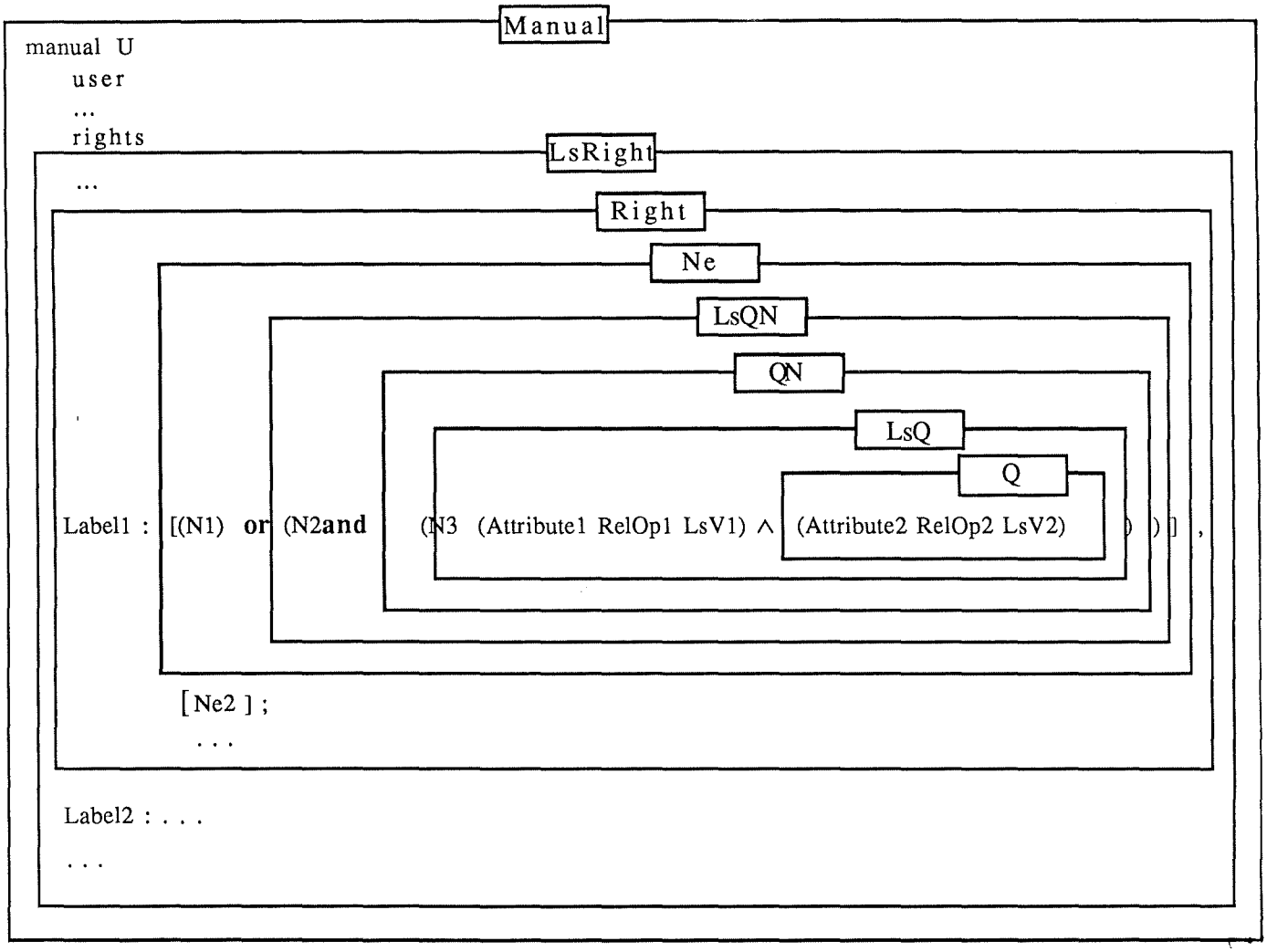


Figure 2.1. - Structure abstraite d'un manuel

La figure représente un fragment du manuel de l'usager U et plus particulièrement la clause rights, qui nous intéresse ici. ♦

Exemple :

Partant d'une expression de droit concrète, nous pouvons représenter les différents types que nous venons de définir. Supposons le droit suivant présent dans le manuel de l'usager U :

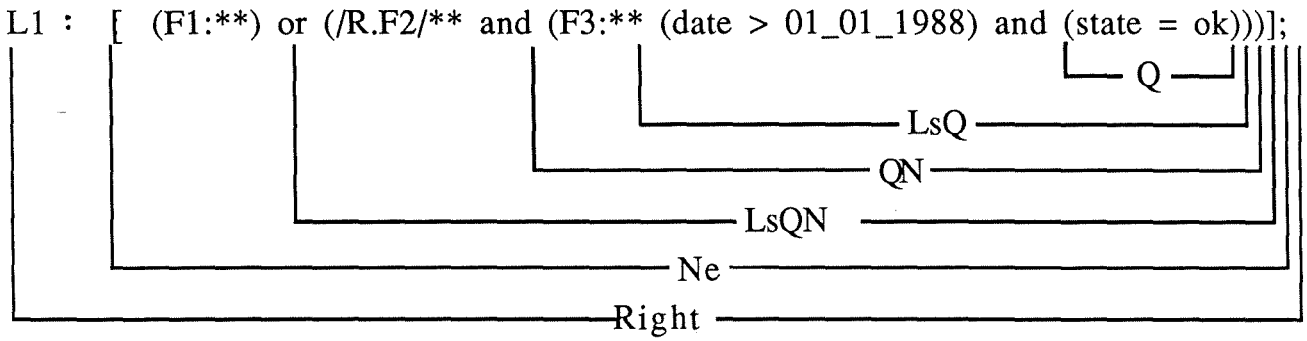


Figure 2.2. - Les différents types sur base d'un exemple concret

Cette expression se traduira en langage courant : l'usager U a le droit d'exécuter les commandes regroupées sous l'étiquette L1 sur :

- tous les objets/éléments de la famille F1;
- tous les objets/éléments qui appartiennent à la fois :
  * à la partition /R.F2/ ;
  * à la famille F3 à condition qu'ils soient plus récents que le 1° janvier 1988 et qu'ils se trouvent dans l'état "ok". ♦

2.2.2. Définition des opérateurs

2.2.2.1. Projection

Soit le type T défini comme suit :

type T = A : . . .
B : . . .
C : . . .

var X : **list of** T

La *projection* de X sur les champs B et C se note X[B,C] et se définit comme suit :

X [B, C] = { (b, c) | ∃ a : (a, b, c) ∈ X }

Exemple : BP [ObjName]

Une projection sur le nom d'objet désigne l'ensemble des noms de tous les objets de la base de programmes (BP). ♦

2.2.2.2. Généralisation de la projection

La projection généralisée s'obtient à l'aide d'une projection et d'une sélection car certains champs doivent prendre leurs valeurs dans des ensembles définis de valeurs.

Reprenant la variable X définie ci-dessus, la *projection généralisée* sur le champ B tel que C prenne ses valeurs dans V se note X [B, (C ∈ V)] et se définit comme suit :

$$X\,[B, (C \in V)] = \{\, b \mid \exists\, a,c : (c \in V) \wedge (a, b, c) \in X \,\}$$

Exemple : BP [(ObjName = U), Man]

Une projection généralisée sur le nom d'objet et le manuel désigne le manuel de U, puisque ObjName est supposé identifiant parmi les champs du type Object. ♦

2.2.2.3. Fonctions utilitaires

Nous utiliserons les fonctions **size** (L) et **first** (L) renvoyant respectivement la taille et le premier élément d'une liste L.

2.2.2.4. Divers

Les opérateurs $\cup=$ et $\setminus=$ simplifieront les notations. Si A et B sont deux listes ou deux éléments de liste, A $\cup=$ B signifiera A := A $\cup$ B et A $\setminus=$ B signifiera A := A $\setminus$ B. De même, nous utiliserons les opérateurs logiques **or=** et **and=**.

Nous nous servirons également des opérateurs d'appartenance ($\in$ et $\notin$) à des collections.

Le symbole ≡ entre deux expressions régulières E_1 et E_2 se lira E_1 "matche" E_2. Le symbole ≈ entre deux expressions régulières E_1 et E_2 se lira E_1 ne "matche" pas E_2.

3. ETABLISSEMENT ET VERIFICATION DES DROITS USAGERS

Nous venons de présenter l'expression des droits usagers. Nous connaissons donc leur structure sous-jacente. Il faut maintenant détailler le mode de vérification de ces droits. Auparavant, montrons le mécanisme d'établissement des droits effectifs, puisque nous venons de voir qu'ils résultent d'opérations sur les droits propres et les droits hérités.

3.1. Etablissement des droits effectifs

Nous présentons ici les deux politiques possibles pour l'établissement des droits effectifs. Nous expliciterons ensuite les raisons du choix effectué.

3.1.1. Politiques d'établissement

3.1.1.1. Etablissement statique

Par *établissement statique* , nous entendons que les droits effectifs sont calculés périodiquement. Entre deux établissements de droits, les droits effectifs de l'usager sont figés, c'est-à-dire qu'ils ne tiennent pas compte des modifications qui interviendraient.

Cette politique présente l'*avantage* d'exiger peu de calculs, du fait que l'établissement est peu fréquent.

Par contre, cette politique a l'*inconvénient* de se baser sur des droits effectifs qui ne reflètent plus la réalité, dès le moment où les droits subissent des modifications.

3.1.1.2. Etablissement dynamique

Dans ce cas, le calcul des droits effectifs doit se faire chaque fois que les droits d'un usager sont modifiés. Cette politique présente des caractéristiques inverses par rapport à la précédente. En effet, dans toutes les situations, elle a l'*avantage* de se baser sur des droits effectifs exacts.

Son *inconvénient* est de devoir effectuer le calcul des droits effectifs aussi souvent que les droits propres ou hérités sont modifiés.

3.1.1.3. Politique adoptée

Il fallait donc choisir entre un minimum de calculs et un meilleur reflet de la réalité. Voici les deux principales raisons qui ont orienté le choix :

- dans un système tel que NOMADE, on peut considérer que les droits usagers évolueront peu une fois qu'ils auront été définis pour un usager;

- le calcul des droits effectifs, comme on le constatera à la section 3.1.2., est une opération coûteuse en temps. Il n'était donc pas pensable d'exécuter ces opérations après chaque commande interactive de modification des droits.

C'est donc la première approche (établissement statique) qui a été choisie. Il est évident que les droits ne reflètent plus la réalité dès qu'intervient une modification, mais le fait que les droits évolueront peu pallie à cet inconvénient. Il reste à fixer le moment où l'établissement s'effectue.

Il apparaît assez clairement que le moment où l'usager se connecte ("login" sous NOMADE) soit le plus propice à ce genre d'opération. L'opération de "login" est déjà assez coûteuse en temps, peu importe qu'elle soit pénalisée par l'établissement des droits effectifs. C'est une des rares opérations qui peut se permettre un temps de réponse moins performant car on l'effectue une seule fois par session.

3.1.2. Algorithme d'établissement des droits effectifs

3.1.2.1. Etude d'un exemple

A l'aide d'un exemple, nous allons suggérer intuitivement le mécanisme d'établissement des droits propres et des droits hérités (dont nous pourrons déduire les droits effectifs), ce qui mettra en évidence les particularités algorithmiques. Soit le schéma suivant où l'on a représenté la clause rights de l'usager U et celles de toutes les partitions englobantes de ses partitions d'appartenance (c'est-à-dire toutes les partitions intervenant dans l'application des mécanismes d'héritage pour l'usager U).

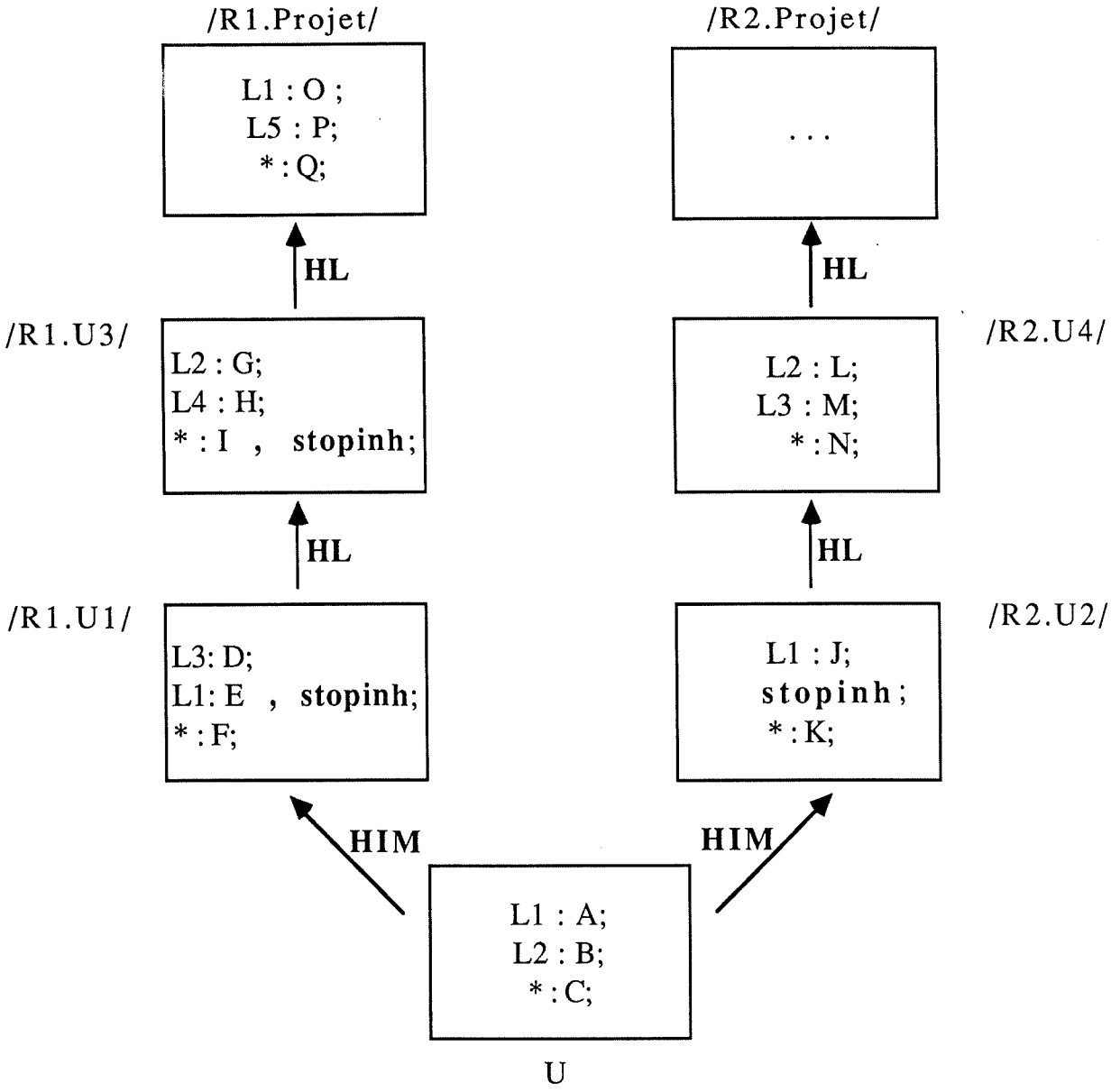


Figure 3.1. - Clauses rights de U et de ses partitions englobantes

Les étiquettes sont représentées par un L suivi d'un chiffre. Les autres majuscules symbolisent des disjonctions d'expressions-NOMADE de complexité quelconque. Supposons donc que l'usager U se connecte sous NOMADE. Nous allons voir comment s'effectue le parcours des divers chemins d'héritage des clauses rights pour déterminer les droits propres et les droits hérités de U. Pour bien marquer les ajouts d'une étape, nous les entourerons de cercles.

Pour commencer, on part de l'usager (ici U), et on copie ses droits propres. Nous obtenons :

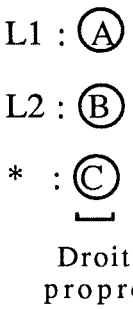


Figure 3.2. - Etablissement des droits de U - Etape 1

Ensuite, à partir de toutes les partitions auxquelles U *appartient* (héritage d'instance multiple), on parcourt le chemin d'héritage en remontant de partition englobante en partition englobante et ce jusqu'à la famille projet (nommée Projet).

Commençons par la partition de relation structurante R1. Dans la clause rights de la partition /R1.U1/, on trouve :

- l'étiquette L3 que nous n'avons pas encore rencontrée. Donc, comme droits propres pour L3, il faut placer l'expression se rapportant à l'étiquette "*" chez U; D se place dans les droits hérités pour L3;

- l'étiquette L1 mentionnant E, **stopInh**. E passe dans les droits hérités pour L1 et **stopInh** indique que l'héritage est terminé sur ce chemin pour cette étiquette;

- l'étiquette "*" : F passe dans les droits hérités pour "*" et pour les étiquettes non traitées, c'est-à-dire également pour L2.

Les droits propres et hérités deviennent :

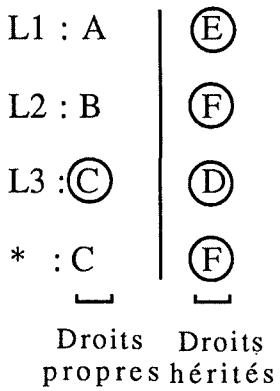


Figure 3.3. - Etablissement des droits de U - Etape 2

Passons à la partition *englobante* /R1.U3/ (héritage linéaire). Nous y trouvons :

- l'étiquette L2 ; G s'ajoute aux droits hérités;

- l'étiquette L4 que nous n'avons pas encore rencontrée. Nous initialisons les droits propres pour L4 avec C ("*" chez U) et les droits hérités pour L4 avec F ("*" chez /R1.U1/). H s'ajoute à ces droits hérités;

- l'étiquette "*" ; I s'ajoute aux droits hérités pour "*" ainsi que pour toutes les étiquettes autres que L2 et L4. I ne s'ajoute pas aux droits hérités pour L1 puisque l'héritage a déjà été arrêté en /R1.U1/ pour cette étiquette. '**stopInh**' indique que l'héritage est arrêté pour toutes les autres étiquettes, soit pour toutes les étiquettes différentes de L2 et L4.

Les droits propres et hérités deviennent :

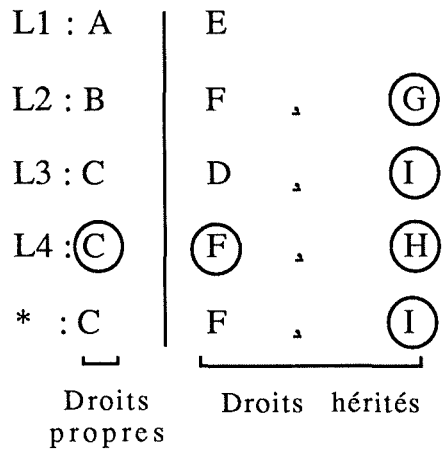


Figure 3.4. - Etablissement des droits de U - Etape 3

Passons à la dernière partition sur ce chemin, celle issue de la famille projet. On y trouve:

- l'étiquette L1 pour laquelle l'héritage a déjà été arrêté en /R1.U1/;
- l'étiquette L5; or l'héritage a déjà été arrêté en /R1.U3/ pour toutes les étiquettes autres que L2 et L4;
- l'étiquette "*"; Q s'ajoute donc aux étiquettes pour lesquelles l'héritage n'a pas été arrêté : soit L2 et L4.

Les droits propres et hérités deviennent :

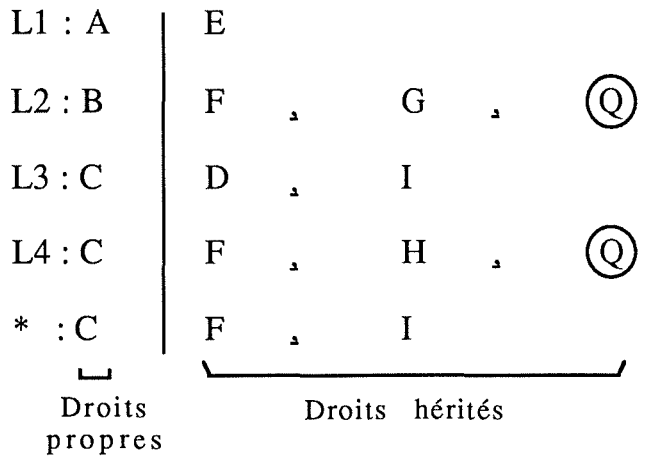


Figure 3.5. - Etablissement des droits de U - Etape 4

Ce chemin d'héritage étant terminé, il faut parcourir les autres chemins. Dans notre exemple, il reste le chemin des partitions basées sur la relation R2. Dès ce moment, comme nous passons à un autre chemin d'héritage, les inhibitions d'héritage linéaire rencontrées pour certaines étiquettes n'ont plus d'effet. Tout est en quelque sorte réinitialisé et l'on poursuit le calcul normalement. Nous passons donc dans /R2.U2/ où l'on trouve :

- l'étiquette L1; J s'ajoute aux droits hérités pour L1;
- le mot clé **stopInh** indique que l'héritage est terminé; pour toutes les partitions englobantes, il n'y a donc plus héritage;
- l'étiquette "*"; K s'ajoute aux droits hérités pour les étiquettes différentes de L1.

Les droits propres et hérités deviennent :

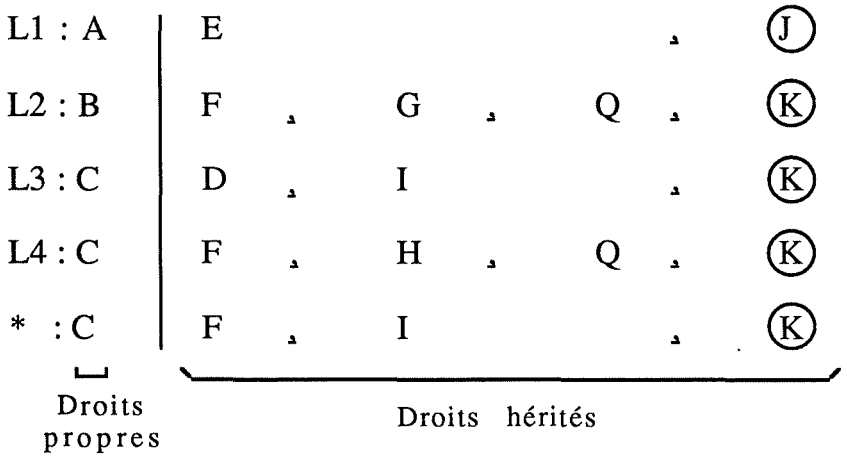


Figure 3.6. - Etablissement des droits de U - Etape 5

Puisque l'on a rencontré le mot clé **stopInh**, il n'y a plus d'héritage sur ce chemin; comme il n'y a pas d'autre partition d'appartenance, les droits propres et les droits hérités obtenus sont définitifs. Leur intersection donne les droits effectifs.

Remarque : en réalité et uniquement pour une question de syntaxe-NOMADE, il n'est pas possible de stocker ces droits effectifs. La vérification (cf. section 3.2.) par rapport aux droits effectifs, vérifiera donc par rapport aux droits propres **et** (et logique !) par rapport aux droits hérités.

3.1.2.2. Spécification de "InitRights"

Lors du "login" d'un usager, InitRights est invoqué. Il a pour but de déterminer les droits propres et hérités (et donc les droits effectifs) de l'usager. Nous allons d'abord définir la structure de données nécessaire à ce calcul.

Les droits s'expriment par rapport à des *étiquettes* . Pour chaque étiquette, on calculera les *droits propres* et *hérités* correspondants. Nous pouvons maintenant ajouter un nouveau type à la structure de données abstraite définie en 2.2.

```
type LoginRight = Label    : string                    {première approximation}
                  LsOwnNe  : list of NOMADEexpression
                  LsInhNe  : list of NOMADEexpression
```

pour déclarer ensuite la variable LoginRights :

var LoginRights : **list of** LoginRight

L'exemple traité à la section 3.1.2.1. ci-dessus suggère que deux informations manquent encore. Lors du parcours des différents chemins d'héritage, nous avons vu que la rencontre du mot clé "**stopInh**" arrête totalement l'héritage sur un chemin, ou partiellement s'il porte sur une étiquette particulière. Dans ce cas, il faut mémoriser cette information; nous ajoutons le champ booléen *InhPoss* (indiquant si l'héritage est encore possible pour cette étiquette) à la structure de données du type LoginRight.

Deuxièmement, la signification de l'étiquette "*" étant "toutes les étiquettes autres que celles apparaissant dans la clause rights de l'usager que l'on traite", il s'agit de marquer les étiquettes déjà traitées. Par étiquette, nous ajoutons le champ booléen *Already* mentionnant si l'étiquette a déjà été traitée (car alors, elle ne sera pas comprise dans la signification de l'étiquette '*'). La structure de données LoginRight devient :

```
type LoginRight  = Label      : string                          {version définitive}
                   LsOwnNe    : list of NOMADEexpression
                   LsInhNe    : list of NOMADEexpression
                   InhPoss    : boolean
                   Already    : boolean
```

D'où la spécification suivante de InitRights :

function *InitRights* (User) : **list of** LoginRight

paramètre : User ::= usager

effet : création de la structure LoginRights pour cet usager, comprenant ses droits propres et ses droits hérités. ♦

3.2. Vérification des droits usagers

La technique la plus rudimentaire aurait été de figer le système de protection en "câblant" les procédures de vérification : chaque exécution d'une commande aurait déclenché l'exécution d'une primitive de vérification. Cette technique allait à l'encontre des caractéristiques d'un bon système de sécurité, qui se doit d'être le plus souple possible. Dans NOMADE, le système de protection n'est pas figé, mais il est adaptable. Toutes les politiques de protection peuvent être définies, partant d'un système strict à un système très "libre".

3.2.1. Mécanisme d'association d'une commande à une étiquette

La technique consiste à utiliser le mécanisme des événements et actions (cf. partie I - section 6) pour effectuer la vérification. En effet, grâce aux actions associées aux commandes, nous pouvons aisément déclencher les vérifications souhaitées lors de l'exécution d'une commande. La primitive IChkRight (L, O) vérifie (par rapport aux droits effectifs de l'usager courant) si l'objet O est désigné par la portée de l'étiquette L.

Exemple :

```
actions
  command
    for lire do
      if IChkRight (LabelX, $S)
        then Ilire ($S)
        else echo 'message'
      fi;
    done;
```

L'association entre commandes et étiquettes s'effectue dans le bloc command de la clause actions d'un manuel. Dans notre exemple, nous associons la commande "lire" à l'étiquette LabelX. Lors de l'exécution de cette commande, le programme d'actions exprime qu'il faut vérifier si l'usager a le droit de manipuler l'objet $S : l'objet $S est-il compris dans les droits effectifs de l'étiquette LabelX ? Si oui, le programme d'actions fait appel à la commande interne Ilire; sinon, l'usager reçoit un message d'erreur. ♦

Pour définir une politique de protection particulière, il suffira donc de l'exprimer à l'aide du langage des actions. Ces spécifications se trouveront par exemple dans le manuel de la famille projet et seront héritées par les autres objets manipulables par la même commande. De plus, on pourra de protéger la politique définie en autorisant la modification du manuel de la famille projet uniquement à une espèce de super-utilisateur.

La technique est très *souple* puisqu'elle permet d'effectuer toutes les vérifications imaginables. On pourrait par exemple faire l'hypothèse que pour manipuler un objet, un utilisateur doit posséder des droits sur toute la famille de cet objet. A l'extrême, l'exécution d'une commande sur un objet pourrait entraîner des vérifications par rapport à un autre objet. Cet exemple, plus théorique que réaliste, illustre bien la souplesse d'utilisation offerte par le système de protection.

Maintenant que nous avons vu la manière d'utiliser le mécanisme des actions pour définir une politique de protection, il reste à établir le fonctionnement de la primitive de vérification.

3.2.2. Spécification de "IChkRight"

Deux principes vont nous aider à élaborer la technique de vérification. On vérifie assez facilement, que :

$$(1) \quad \forall x, \forall y, \forall z : [\, x \Rightarrow y \vee z \,] \Leftrightarrow [\, (x \Rightarrow y) \vee (x \Rightarrow z) \,]$$

$$(2) \quad \forall x, \forall y, \forall z : [\, x \Rightarrow y \wedge z \,] \Leftrightarrow [\, (x \Rightarrow y) \wedge (x \Rightarrow z) \,]$$

La double vérification par rapport aux droits propres et aux droits hérités est équivalente à la vérification par rapport aux droits effectifs. La primitive IChkRight effectue la vérification suivante : à partir d'un nom qualifié N et d'une étiquette E, elle renvoit la valeur **true** si N est compris dans les droits propres **et** dans les droits hérités (pour E) de l'usager courant, et **false** sinon.

La primitive de vérification est alors basée sur (2) et est spécifiée de la façon suivante :

function *IChkRight* (Lbl, QN) : **boolean**

paramètres : QN ::= objet/élément qualifié (OE LsQ)
Lbl ::= étiquette

effet : retourne (QN **implies first** (LoginRights[(Label ≡ Lbl), LsOwnNe])
∧ QN **implies first** (LoginRights[(Label ≡ Lbl), LsInhNe]))

♦

LsNe étant une disjonction de Ne; nous pouvons appliquer (1) :

function *QNImpliesLsNe* (QN, LsNe) : **boolean**

paramètres : QN ::= OE LsQ
LsNe ::= $\mathbf{or}_{i\in I}$ Ne_i (#(I) ≥ 1)

effet : retourne (QN **implies** LsNe),
c'est-à-dire selon (1) : $\vee_{i\in I}$ QNImpliesNe (QN, Ne_i) ♦

Ceci nécessite donc une nouvelle fonction : QNImpliesNe. Pour des raisons que nous expliquerons à la section 4.2.3., il est possible de s'arranger pour que l'expression-NOMADE Ne ne soit pas niée (Sign = yes). Les expressions-NOMADE sont donc des disjonctions non niées de listes de noms qualifiés; nous appliquons le principe (1), d'où :

function *QNImpliesNe* (QN, Ne) : **boolean**

paramètres : QN ::= OE LsQ
Ne ::= $\mathbf{or}_{i\in I}$ $LsQN_i$ (#(I) ≥ 1)

effet : retourne (QN **implies** Ne),
c'est-à-dire selon (1) : $\vee_{i\in I}$ QNImpliesLsQN (QN, $LsQN_i$) ♦

De même, cette fonction requiert à son tour une autre fonction : QNImpliesLsQN. Encore une fois, il est possible de s'arranger pour que la conjonction LsQN ne soit pas niée (cf. section 4.2.3.). Nous sommes donc en présence d'une conjonction non niée de noms qualifiés et en appliquant le principe (2), on obtient :

function *QNImpliesLsQN* (QN_A, $LsQN_B$) : **boolean**

paramètres : QN_A ::= OE_A LsQ_A
$LsQN_B$ ::= $\wedge_{j\in J}$ QN_j (#(J) ≥ 1)

effet : retourne (QN_A **implies** $LsQN_B$)
c'est-à-dire selon (2) : $\wedge_{j\in J}$ QNImpliesQN (QN_A, QN_j) ♦

Nous devons donc pouvoir effectuer la vérification par rapport aux éléments de cette conjonction qui sont des noms qualifiés : c'est la fonction QNImpliesQN.

function *QNImpliesQN* (QN_A, QN_B) : **boolean**

paramètres : QN_A ::= OE_A LsQ_A
QN_B ::= [**not**] N_B LsQ_B

effet : retourne (QN_A **implies** QN_B) c'est-à-dire :
si QN_B non nié

alors OEImpliesN (OE_A, N_B) $\wedge$ LsQImpliesLsQ (LsQ_A, LsQ_B)

sinon OEImpliesNotN (OE_A, N_B) v LsQImpliesNotLsQ (LsQ_A, LsQ_B) ♦

Ici, il s'agit de vérifier si un nom qualifié implique un autre nom qualifié éventuellement nié. Il faut avant tout savoir si OE_A implique N_B : nous utiliserons la fonction OEImpliesNotN ou OEImpliesN suivant que le nom qualifié QN_B est nié ou pas. Ensuite, il reste à vérifier si l'implication est toujours vraie en tenant compte des qualifications. Pour cela, nous utiliserons la fonction LsQImpliesNotLsQ (resp. LsQImpliesLsQ si QN_B n'est pas nié). Comme ces deux fonctions seront aussi utilisées dans la technique de vérification des modifications des droits usagers, nous les spécifierons dans la section 4.2.3.

Spécifions la fonction OEImpliesN :

function *OEImpliesN* (OE_A, N_B) : **boolean**

paramètres : OE_A ::= objet/élément

N_B ::= [/R_B.A_B/] [OE_B]

effet : retourne (OE_A **implies** N_B) ♦

Examinons les cas possibles. Etant donné la représentation d'un nom simple, cinq cas peuvent se présenter. Soit un objet/élément sans/avec méta-caractères (cas 1 et 2), soit un objet/élément sans/avec méta-caractères préfixé d'un nom de partition (cas 3 et 4), soit uniquement un nom de partition (cas 5). D'où :

N_A \ N_B	OE_B (sans)	OE_B (avec)	/R_B.A_B/OE_B (sans)	/R_B.A_B/OE_B (avec)	/R_B.A_B/ (sans)
OE_A (sans méta-caractères)	1	2	3	4	5

Tableau 3.1. - Evaluation de OE_A **implies** [**not**] N_B : cas possibles

Traitement des cas : évaluation de OE_A **implies** N_B

1° Comparaison entre deux objets/éléments. Ni OE_A, ni OE_B ne contient de méta-caractères; il suffit de tester l'égalité syntaxique entre les deux objets/éléments : $OE_A = OE_B$.

2° Comparaison entre deux objets/éléments. Pour savoir s'il y a compatibilité, étant donné que OE_B peut contenir des méta-caractères, il faut appliquer la fonction "match" (notée $\equiv$) qui compare un string de caractères à une expression régulière. Nous écrivons : $OE_A \equiv OE_B$.

3° Le nom N_B désigne un objet/élément précédé d'un nom de partition. Outre le fait que $OE_A = OE_B$ (puisque OE_B ne contient pas de méta-caractères), il faut de plus que OE_A appartienne à la partition /R_B.A_B/ car il se pourrait que $OE_A = OE_B$ et $OE_A \notin$ /R_B.A_B/, obligeant OEImpliesN à renvoyer la valeur **false**. Donc $OE_A \in$ /R_B.A_B/ $\wedge$ $OE_A = OE_B$.

4° Ce cas est identique au cas 3° excepté le fait que OE_B peut contenir des méta-caractères. L'égalité est remplacée par un "match", soit $OE_A \in /R_B.A_B/ \wedge OE_A \equiv OE_B$.

5° Le nom N_B est uniquement un nom de partition. Dans ce cas, l'objet/élément OE_A doit appartenir à cette partition. Soit $OE_A \in /R_B.F_B/$.

D'où le tableau de synthèse suivant :

N_A \ N_B	OE_B (sans)	OE_B (avec)	$/R_B.A_B/OE_B$ (sans)	$/R_B.A_B/OE_B$ (avec)	$/R_B.A_B/$ (sans)
OE_A (sans méta-caractères)	$OE_A = OE_B$	$OE_A \equiv OE_B$	$OE_A \in /R_B.A_B/ \wedge OE_A = OE_B$	$OE_A \in /R_B.A_B/ \wedge OE_A \equiv OE_B$	$OE_A \in /R_B.A_B/$

Tableau 3.2. - Evaluation de OE_A **implies** N_B

Spécifions la fonction OEImpliesNotN :

function *OEImpliesNotN* (OE_A, N_B) : **boolean**

paramètres : OE_A ≡ objet/élément
N_B ≡ **not** $[/R_B.A_B/]$ $[OE_B]$

effet : retourne (OE_A **implies not** N_B) ♦

En appliquant un raisonnement analogue à l'élaboration du tableau 3.2., on obtient (en notant ≈ la fonction "ne matche pas") :

N_A \ N_B	OE_B (sans)	OE_B (avec)	$/R_B.A_B/OE_B$ (sans)	$/R_B.A_B/OE_B$ (avec)	$/R_B.A_B/$ (sans)
OE_A (sans méta-caractères)	$OE_A \neq OE_B$	$OE_A \approx OE_B$	$OE_A \notin /R_B.A_B/$ v $OE_A \neq OE_B$	$OE_A \notin /R_B.A_B/$ v $OE_A \approx OE_B$	$OE_A \notin /R_B.A_B/$

Tableau 3.3. - Evaluation de OE_A **implies not** N_B

3.2.3. Optimisation

Il est intéressant de s'attarder quelque peu sur la primitive Chkright. Supposons LsOwnNe composée des expressions Ne_A et Ne_B. Les expressions-NOMADE étant liées par l'opérateur **or**, la vérification devra, pour réussir, renvoyer **true** pour une de ces deux expressions. Or, si Ne_B **implies** Ne_A, il est inutile de stocker Ne_B puisque si la vérification échoue par rapport à Ne_A, elle échouera forcément par rapport à Ne_B. Nous pouvons donc compacter les variables LoginRights lors de leur construction de façon à éviter tout stockage inutile et optimiser les temps de vérification.

4. MODIFICATION DES DROITS USAGERS

Par modification des droits usagers, nous entendons toute action changeant les droits d'un usager (ou d'une partition d'usagers), que ce soit en addition, en suppression ou en mise à jour. Dans cette section, nous décrirons la solution actuellement adoptée par NOMADE, ainsi que plusieurs variantes envisageables.

4.1. Primitives de modification des droits usagers

Avant de plonger dans les détails du "comment ?", énonçons les principes à la base de toutes les solutions afin d'en déduire des spécifications des primitives de modification des droits. D'autre part, nous en délimitons clairement les objectifs en répondant à deux questions préliminaires.

4.1.1. Principe de base

Dans la vie quotidienne, il est courant qu'on ne puisse donner que ce qu'on a. Ici, il en est de même : on ne peut déléguer (donner) que des droits qu'on possède soi-même (sauf qu'ici, le fait de donner un droit à quelqu'un d'autre n'implique évidemment pas la perte de ce droit : on en donne plutôt une copie).

Si on continue ce raisonnement, il semble logique d'adopter cette même politique si on enlève un droit à quelqu'un : il faut qu'on dispose soi-même de ce droit pour pouvoir le retirer (ici il n'y a évidemment pas d'analogie avec la vie courante).

Finalement, si on modifie les droits de quelqu'un, cela revient à lui ajouter/supprimer des droits et le même principe reste en vigueur.

En résumé, voici le principe de base du système de modification des droits usagers, principe définissant ce que nous entendons par modification "légale" :

"Un usager ne peut donner/retirer que des droits qu'il possède lui-même".

Il reste cependant à clarifier quelques petits détails :

- premièrement, qu'est-ce qu'on entend par "droits qu'il possède lui-même" dans le contexte NOMADE ? Il est clair que les droits propres d'un usager (c'est-à-dire les droits explicités dans la clause rights de son manuel) ne conviennent pas parce qu'ils subissent un héritage d'instance multiple, qui consiste à restreindre ces droits propres à leur intersection avec l'union des droits effectifs des partitions auxquelles cet usager appartient. Le principe ci-dessus peut donc être précisé dans la terminologie NOMADE :

  "Un usager ne peut donner/retirer que ses droits effectifs";

- deuxièmement, il est souhaitable que les primitives de modification des droits soient elles-mêmes protégées par des droits (ChkRight);

- troisièmement, même si la notion d'auteur d'un objet/élément existe (attribut "author"), elle ne sera pas utilisée lors d'une modification de droits : on peut donner/retirer des droits portant sur des objets/éléments sans en être l'auteur ou avoir reçu explicitement ces droits par leurs auteurs. Ceci est dû au fait que les droits s'expriment par des expressions-NOMADE et se transmettent aussi sous cette forme, plutôt que par énumération d'objets/éléments.

4.1.2. Identification et spécification des primitives de modification des droits usagers

4.1.2.1. Identification

De ce qui précède, on peut déduire la nécessité d'au moins deux primitives : l'une (appelée "IAddRight" dans la suite) pour ajouter des droits à quelqu'un, l'autre (appelée "IDelRight" dans la suite) pour retirer des droits à quelqu'un.

Une troisième commande (appelée "IChgRight" dans la suite) est envisageable afin de permettre des ajouts/suppressions simultanées aux/des droits de/à quelqu'un, mais le besoin d'une telle commande semble moins urgent (surtout qu'une version simplifiée consisterait à enchaîner un "IDelRight" et un "IAddRight").

Mais qui est ce "quelqu'un" dans les spécifications informelles ci-dessus? Cela peut être un autre usager ou une partition d'usagers (car une telle partition possède aussi une clause rights et est donc susceptible de subir des modifications de ses droits). Bien entendu, le modifieur de droits sera toujours un usager, car une partition d'usagers n'est pas une entité active.

Il est aussi primordial que ces 2 (ou 3) primitives soient les *seules* possibilités de modification d'une clause rights; cette clause doit être accessible en lecture seulement pour toutes les autres commandes.

Avant de passer à des spécifications plus formelles, rappelons qu'un droit est une étiquette suivie d'une liste d'expressions-NOMADE (séparées implicitement par des **or**). Cependant, dans le cas de la modification des droits, nous ne considérerons (pour des besoins de paramétrage des commandes) que le cas particulier d'un droit où une seule expression-NOMADE suit l'étiquette. Ceci n'est nullement restrictif pour la puissance des commandes, elles n'en perdent qu'un peu en souplesse.

4.1.2.2. Spécification de "IAddRight"

Soit U' l'usager exécutant cette commande (c'est-à-dire que la variable LoginRights est relative à U') et soit Target la désignation de l'objet (usager ou partition d'usagers) visé par l'addition de droit. Soit TargetMan le manuel de Target (i.e. TargetMan = BP [(ObjName = Target) , Man]).

Si U' possède lui-même le droit qu'il veut donner, alors soit IAddRight ajoutera Ne à la liste des expressions-NOMADE associées à l'étiquette en question L (si L existe déjà dans la clause rights de Target), soit IAddRight ajoutera le droit "L : Ne" à la clause rights de Target (sinon). D'où la spécification :

procedure *IAddRight* (Target , L , Ne)

paramètres : Target ::= nom simple désignant soit un usager (U), soit une partition d'usagers (/R.U/) ;
L ::= étiquette ;
Ne ::= expression-NOMADE.

effet : **si** (1) Target ∈ BP [ObjName]
∧ (2) L est une étiquette valide
et Ne est une expression-NOMADE valide

∧ (3) Ne désigne un ensemble d'objets/éléments inclus dans l'ensemble désigné par les droits effectifs relatifs à l'étiquette L de LoginRights

alors TargetMan.LsRight [LsNe , (Label = L)] ∪= Ne
si L ∈ TargetMan.LsRight [Label]
TargetMan.LsRight ∪= (L , (Ne))
sinon

sinon message d'erreur. ♦

Il est primordial de noter que les ajouts de droits se font dans les droits propres de Target, et non dans ses droits effectifs.

Plusieurs remarques s'imposent :

Remarque 1 :

Aucune vérification n'est faite pour voir (au cas où L existe déjà) si Ne figure déjà dans la liste des expressions-NOMADE associées à L.

Une telle vérification semble fastidieuse à effectuer (surtout qu'il se peut que l'ensemble des objets/éléments désignés par Ne soit un sous-ensemble de l'ensemble des objets/éléments désignés par la liste des expressions-NOMADE déjà associées à L). Mais nous verrons en 4.2. une technique qui réalise justement ce genre de vérification dans un autre contexte et qui deviendrait réutilisable ici. Le prix à payer serait évidemment un temps d'exécution plus long pour IAddRight, mais il ne faut pas oublier que cette commande est utilisée rarement et que c'est IChkRight qui devrait être optimisé.

On peut donc éventuellement ajouter une quatrième condition :

(4) Ne désigne un ensemble d'objets/éléments non inclus dans l'ensemble désigné par les droits propres relatifs à l'étiquette L de la clause rights de TargetMan.

A l'inverse , on pourrait préconiser une politique d'addition de droits qui n'essaie pas un tel compactage de la clause rights cible. Dans ce cas, on pourrait alors réutiliser la technique exposée en 4.2. pour compacter la variable LoginRights lors de sa construction, plutôt que la clause rights lors d'une mise à jour. Ceci ralentit le processus de "login", ne pénalise pas les IAddRight et accélère les IChkRight. Cette politique tolère donc les redondances dans les clauses rights : un même droit y est stocké autant de fois qu'il a été donné. Toutefois, le principe de modification ne garantit pas qu'un donneur retire un droit qu'il a donné, car ce principe n'impose pas de garder trace du nom d'un donneur de droit. Une telle mémorisation nous semble inutile, car il s'agit là essentiellement d'un problème de coordination entre les responsables de la modification de droits.

Les deux alternatives étant justifiables, nous n'en imposons aucune.

Remarque 2 :

La condition (3) correspond à la vérification de "légalité" de la modification.

Remarque 3 :

Notre objectif n'est pas de trouver les paramètres qui facilitent le plus possible la tâche du donneur de droits. On peut imaginer divers procédés qui rendent plus aisée la fourniture du paramètre Ne, mais ultimement, toutes ces variantes se ramènent au paramétrage ci-dessus.

Remarque 4 :

D'autres scenarios de modification sont envisageables (cf. section 4.3.).

Remarque 5 :

Un droit obtenu peut sans aucune contrainte être donné à un autre usager (ceci reviendrait à l'option "with grant option" dans le System R [Griffiths 76]), à condition de pouvoir modifier les droits de cet usager.

4.1.2.3. Spécification de "IDelRight"

Soit U' l'usager exécutant cette commande (c'est-à-dire que la variable LoginRights est relative à U') et soit Target la désignation de l'objet (usager ou partition d'usagers) visé par la suppression de droit. Soit TargetMan le manuel de Target (i.e. TargetMan = BP [(ObjName = Target) , Man]).

Si U' possède lui-même le droit qu'il veut retirer, alors soit IDelRight supprimera Ne de la liste des expressions-NOMADE associées à l'étiquette en qustion L (si cette liste a au moins deux éléments), soit IDelRight supprimera le droit "L : Ne" dans la clause rights de Target (sinon). D'où la spécification :

procedure *IDelRight* (Target , L , Ne)

paramètres : Target ::= nom simple désignant soit un usager (U), soit une partition d'usagers (/R.U/) ;
L ::= étiquette ;
Ne ::= expression-NOMADE.

effet : **si** (1) Target ∈ BP [ObjName]
∧ (3) Ne désigne un ensemble d'objets/éléments inclus dans l'ensemble désigné par les droits effectifs relatifs à l'étiquette L de LoginRights
∧ (5) L ∈ TargetMan.LsRight [Label]
∧ (6) Ne ∈ TargetMan.LsRight [LsNe , (Label = L)]

alors TargetMan.LsRight \= (L , (Ne))
si size(TargetMan.LsRight [LsNe , (Label=L)]) = 1
TargetMan.LsRight [LsNe , (Label=L)] \= Ne
sinon

sinon message d'erreur. ♦

Les remarques 2, 3 et 4 de la section précédente s'appliquent ici aussi.

Remarquons aussi que nous n'avons pas implémenté un système de **révocation récursive** tel qu'il existe par exemple dans System R ([Griffiths 76]). Illustrons cette dernière technique par deux exemples :

Exemple1 : supposons que les opérations suivantes sont exécutées chronologiquement selon leur numérotation :

(1) U1 crée O
(2) U1 ajoute (with grant option) "lire O" à U2

(3) U2 ajoute (with grant option) "lire O" à U3
(4) U1 enlève "lire O" à U2

L'effet souhaité de l'opération (4) est que non seulement l'opération (2) soit défaite, mais aussi l'opération (3) (qui n'aurait pas pu avoir lieu sans l'opération (2) !). L'objectif est donc de restaurer pour le triplet (U2,U3,O) l'état de protection précédant l'opération (2). ♦

Cette technique n'est pas triviale; examinons le cas suivant :

Exemple 2 :

(1) U1 crée O
(2) U1 ajoute (with grant option) "lire O" à U2 et à U3
(3) U2 ajoute (with grant option) "lire O" à U4
(4) U3 ajoute (with grant option) "lire O" à U4
(5) U1 enlève "lire O" à U2

En supposant une élimination de redondance lors d'un ajout, on ne peut aveuglément défaire l'opération (3) suite à l'opération (5), car U4 peut entretemps aussi lire O à cause de l'opération (4) ! ♦

La solution consiste alors à effectuer un estampillage de chaque transmission de droits et de garder une trace pour savoir qui a reçu quel droit de qui. Comme indiqué à la section 4.1.2.2. (remarque 1), nous n'avons pas voulu suivre ce chemin.

4.1.2.4. Spécification de "IChgRight"

L'expression $\Delta(Ne_1,Ne_2)$ donne une liste d'expressions-NOMADE désignant l'ensemble de tous les objets/éléments qui sont désignés soit par Ne_1, soit par Ne_2, mais pas par les deux.

Soit U' l'usager exécutant un changement de droit (c'est-à-dire que la variable LoginRights est relative à U') et soit Target la désignation de l'objet (usager ou partition d'usagers) visé par le changement de droit. Soit TargetMan le manuel de Target (i.e. TargetMan = BP [(ObjName = Target) , Man]).

Si U' possède lui-même le droit qu'il veut retirer/ajouter (i.e. changer), alors IChgRight remplacera Ne_1 dans la liste des expressions-NOMADE associées à l'étiquette en question L par Ne_2. D'où la spécification:

procedure *IChgRight* (Target , L , Ne_1 , Ne_2)

paramètres : Target : nom simple désignant soit un usager (U), soit une partition d'usagers (/R.U/) ;
L : étiquette ;
Ne_1 : expression-NOMADE ;
Ne_2 : expression-NOMADE .

effet : **si** (1) Target ∈ BP [ObjName]
∧ (2) Ne_2 est une expression-NOMADE valide
∧ (5) L ∈ TargetMan.LsRight [Label]
∧ (6) Ne_1 ∈ TargetMan.LsRight [LsNe , (Label = L)]

$\wedge$ (7) $\Delta(Ne_1,Ne_2)$ désigne un ensemble d'objets/éléments inclus dans l'ensemble désigné par les droits effectifs relatifs à l'étiquette L de LoginRights

alors TargetMan.LsRight [LsNe , (Label = L)] \= Ne_1
TargetMan.LsRight [LsNe , (Label = L)] U= Ne_2

sinon message d'erreur. ♦

Quant à la condition (7), elle exprime la vérification de "légalité" de l'opération.

Remarquons que cette spécification est beaucoup plus correcte que si on implémentait IChgRight (Target , L , Ne_1 , Ne_2) comme suit :

IDelRight (Target , L , Ne_1);
IAddRight (Target , L , Ne_2) .

Dans ce cas, les conditions à satisfaire seraient :

(1) $\wedge$ (2) $\wedge$ (3) [$\wedge$ (4)] , puis (1) $\wedge$ (3) $\wedge$ (5) $\wedge$ (6),

c'est-à-dire que :

(1) serait vérifiée 2 fois (ce qui n'est pas grave);

(3) serait vérifiée 2 fois (mais pour des expressions-NOMADE différentes); cela revient à évaluer $\Delta(Ne_1,Ne_2)$ = Ne_1 **or** Ne_2, ce qui est une approximation extrême de Ne_1 **exclusive-or** Ne_2;

(4) serait éventuellement vérifiée, alors que ce n'est pas nécessaire.

Avant de discuter l'implémentation, il faut encore éclaircir deux questions :

4.1.3. Droit de modification

Comme nous l'avons remarqué dans la section 4.1.1., il faut idéalement aussi vérifier le "droit de modification", c'est-à-dire si le modifieur a le droit de modifier la clause rights de Target. La responsabilité d'une telle vérification incombe complètement aux utilisateurs de NOMADE ! Les trois commandes de modification des droits usagers sont bien sûr définies dans la clause actions du manuel de la famille projet (par exemple). C'est donc là qu'on peut prévoir un IChkRight adéquat.

Montrons comment écrire le programme d'actions relatif à la commande externe AddRight (par exemple) :

Exemple : ($S désigne U, $1 désigne L, $2 désigne Ne)

```
actions
    command
        for AddRight do
            if IChkright ( ModifDroit , $S )
                then IAddRight ( $S , $1 , $2 ) ;
                else IDisplay ("Vous ne pouvez pas modifier les
                                droits de" + $S ) ;
            fi ;
```

done ; ♦

Outre le IChkRight à placer dans les programmes d'actions relatifs aux commandes de modification, il faut aussi définir dans les clauses rights des usagers quelles sont leurs cibles potentielles pour ces commandes de modification (attachées à l'étiquette ModifDroit dans nos exemples).

Exemple :

Soit la clause rights du programmeur en chef Jean-Michel :

rights
ModifDroit : /Programmeur. Jean-Michel/ ** ;
...

Elle exprime que Jean-Michel ne peut manipuler que les droits propres des membres de la partition regroupant les programmeurs (dont il est d'ailleurs l'ancêtre). ♦

De cette manière, l'établissement d'une certaine hiérarchie entre les usagers est entièrement à charge des usagers eux-mêmes, car NOMADE ne favorise aucune politique de décision à ce niveau. Cela résoud aussi le problème de savoir qui peut modifier les droits d'une partition d'usagers. Le bon sens dit qu'idéalement, il faudrait en être l'ancêtre ou appartenir à une partition qui englobe (de manière éventuellement transitive) cette partition, mais on peut définir d'autres solutions.

4.1.4. Ajout/retrait intégral contre ajout/retrait partiel

Revenons maintenant aux conditions (3) et (7) qui, rappelons-le, servent à assurer le respect du principe de base). Deux alternatives sont possibles :

- par *ajout/retrait "intégral"* , nous entendons le fait que le modifieur U' donne/retire un droit "L : Ne" tel que Ne se trouve **textuellement** dans la liste des expressions-NOMADE associées à l'étiquette L dans LoginRights de U';

- par *ajout/retrait "partiel"* , nous entendons le fait que le modifieur U' donne/retire un droit "L : Ne" tel que l'ensemble des objets/éléments désignés par Ne soit un **sous-ensemble** de l'ensemble des objets/éléments désignés par la liste des expressions-NOMADE associées à l'étiquette L dans les LoginRights de U'.

La première solution est très facile à implémenter car la condition (3) devient une simple égalité syntaxique entre expressions-NOMADE. Cependant, cette alternative est extrêmement limitative dans ses possibilités, car il est par exemple souvent utile de ne donner qu'un sous-ensemble de ses propres droits.

Voilà pourquoi la seconde solution semble préférable, surtout que la première n'en est qu'un cas particulier. Toutefois, la vérification s'en trouve considérablement compliquée et il s'agit de trouver un moyen simple d'écrire une expression-NOMADE qui satisfasse aux contraintes de l'ajout/retrait partiel. La prochaine section (4.2.) est entièrement consacrée à l'étude de ce problème.

4.2. Technique de vérification d'implication entre listes d'expressions-NOMADE

4.2.1. Identification du problème et difficultés

En toute généralité, il faudrait donc une méthode pour déterminer si l'ensemble des objets/éléments désignés par une liste d'expressions-NOMADE est inclus ou non dans l'ensemble des objets/éléments désignés par une autre liste d'expressions-NOMADE.

L'utilité primordiale d'une telle méthode sera la vérification du principe de base par les trois primitives IAddRight, IDelRight et IChgRight (c'est-à-dire la vérification de leurs conditions (3) et (7)), et dans ce cas on en utilisera le cas particulier où la première liste d'expressions-NOMADE se limite à un seul élément.

Mais nous avons déjà trouvé quelques autres situations où une telle technique deviendrait réutilisable :

- compactage de la structure de données LoginRights en vue d'améliorer les performances du ChkRight (section 3.2.3.);
- élimination de redondance avec les droits propres ou effectifs lors d'un AddRight (section 4.1.);
- nous verrons (section 4.3.) une variante de scénario pour IAddRight et IDelRight qui nécessitera une telle technique dans toute sa généralité (liste d'expressions-NOMADE par rapport à une liste d'expressions-NOMADE).

Le problème n'est cependant pas trivial a priori. Considérons un peu l'exemple suivant :

Exemple :

Pour l'étiquette L, l'usager U' a pour droit propre:

L : A **or** B

et U' veut donner à U le droit de manipuler C via l'étiquette L (supposons que U' ait le droit d'effectuer AddRight (U , L , C)).

Si on note OE_{Ne} l'ensemble des objets/éléments désignés par l'expression-NOMADE Ne, alors il se peut qu'on ait (lors de la vérification par rapport aux droits propres de U') la situation suivante :

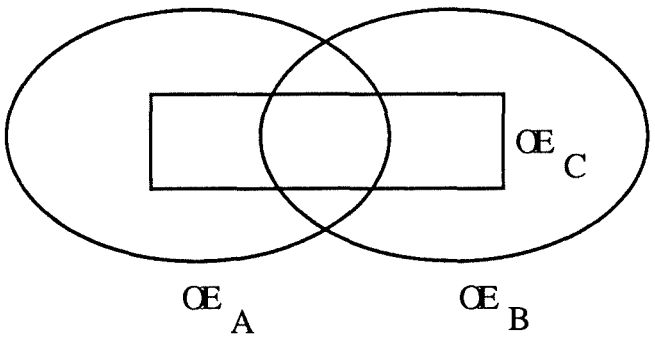


Figure 4.1. - Inclusion difficilement vérifiable : $OE_C \subseteq OE_A \cup OE_B$

Il est extrêmement difficile de vérifier cette inclusion; il est évidemment exclus de vouloir procéder comme suit : identifier tous les objets/éléments actuellement existants qui appartiennent à OE_C et vérifier s'ils appartiennent tous à OE_A ou à OE_B. Cette solution présente l'important avantage d'implémenter une condition *nécessaire et suffisante*. Elle doit cependant être écartée en raison de son coût prohibitif : l'énorme quantité potentielle des objets la rendrait inutilisable.

♦

Il faut donc trouver un autre moyen de déterminer l'inclusion ou la non inclusion d'un ensemble d'objets/éléments dans un autre ensemble d'objets/éléments. La solution exposée ci-dessous consiste en gros à se poser la question à l'envers. Elle se base sur le développement en trois étapes suivant :

> **Etape 1** : chercher un **procédé de construction** d'une liste d'expressions-NOMADE, telle que l'ensemble des objets/éléments qu'elle désigne soit un sous-ensemble de l'ensemble des objets/éléments désignés par une autre liste d'expressions-NOMADE, donnée au départ.
>
> **Etape 2** : en dériver une **condition nécessaire** (mais pas suffisante) d'inclusion.
>
> **Etape 3** : écrire un **algorithme de vérification d'inclusion** basé sur cette condition nécessaire. Cet algorithme déterminera si son premier argument a (ou non) été obtenu via le procédé de construction (identifié à l'étape 1) à partir de son second argument.

Montrons d'abord les bases théoriques de ce procédé.

4.2.2. Excursion dans la théorie des ensembles

Définissons d'abord les notations utilisées :

Notations :

soit E un ensemble (fini ou infini) :

- nous noterons E^- tout sous-ensemble (non nécessairement strict) de E, c'est-à-dire : $\varnothing \subseteq E^- \subseteq E$;
- nous noterons E^+ tout sur-ensemble (non nécessairement strict) de E, c'est-à-dire : $E \subseteq E^+ \subseteq E'$; (en supposant connu E', le plus grand sur-ensemble possible de E, E' pouvant être un ensemble infini);
- nous noterons C(E) l'ensemble complémentaire de E par rapport à E';
- nous noterons #(E) le nombre d'éléments de E. ♦

Dans la suite, on utilisera les ensembles d'indices suivants :

$I = \{ 1, \ldots, m \}$ avec $m \in \mathbb{N}^*$
$J = \{ 1, \ldots, n \}$ avec $n \in \mathbb{N}^*$

(nous ne considérerons dans la suite que des ensembles I^+ et J^+ finis).

Etape 1 : procédé de construction

On distingue les cas particuliers de construction de sous-ensembles et sur-ensembles suivants :

(i) Si $E = \bigcup_{i \in I} e_i$

alors on peut construire les E^- et E^+ selon les procédés génériques suivants :

$$E^- = \bigcup_{i \in I^-} e_i^- \qquad (1)$$
$$E^+ = \bigcup_{i \in I^+} e_i^+ \qquad (2)$$

(l'expression (2) utilise un abus de notation, car pour $i \in I^+ \setminus I$, e_i n'est pas défini, c'est-à-dire e_i^+ se définit par rapport à un ensemble inexistant; on remplacera donc mentalement ces e_i^+ par e_i, car l'expression :

$$E^+ = (\bigcup_{i \in I} e_i^+) \cup (\bigcup_{i \in I^+ \setminus I} e_i)$$

est un peu lourde).

(ii) Si $E = C(\bigcup_{i \in I} e_i)$

alors on peut construire les E^- et E^+ selon les procédés génériques suivants :

$$E^- = C(\bigcup_{i \in I^+} e_i^+) \qquad (3)$$
$$E^+ = C(\bigcup_{i \in I^-} e_i^-) \qquad (4)$$

(iii) Si $E = \bigcap_{i \in I} e_i$

alors on peut construire les E^- et E^+ selon les procédés génériques suivants :

$$E^- = \bigcap_{i \in I^+} e_i^- \qquad (5)$$
$$E^+ = \bigcap_{i \in I^-} e_i^+ \qquad (6)$$

(iv) Si $E = C(\bigcap_{i \in I} e_i)$

alors on peut construire les E^- et E^+ selon les procédés génériques suivants :

$$E^- = C(\bigcap_{i \in I^-} e_i^+) \qquad (7)$$
$$E^+ = C(\bigcap_{i \in I^+} e_i^-) \qquad (8)$$

Remarque : des démonstrations et des illustrations des formules (1) - (8) se trouvent dans l'annexe D.

Il est important de signaler que **ce ne sont évidemment pas les seuls procédés** de construction de sur-ensembles et sous-ensembles de E. La particularité de la technique exposée ici est qu'elle empêche la construction de sur-ensembles et sous-ensembles dont les composants ne se définissent pas par rapport aux composants de E, ce qui était justement la difficulté identifiée en 4.2.1.

Ce qui est intéressant dans ces procédés, c'est qu'ils peuvent bien sûr être appliqués de manière imbriquée, c'est-à-dire que chaque e_i ($\forall$ i $\in$ I) de E peut à son tour prendre n'importe quelle forme parmi les cas (i), (ii), (iii) et (iv) !

Etape 2 : Dérivation d'une condition nécessaire d'inclusion

Nous pouvons à présent déduire de ces procédés des conditions **nécessaires** (mais pas suffisantes) pour qu'une inclusion ait lieu. Ainsi :

(9) $\forall i \in I, \exists j \in J : e_i \subseteq f_j \Rightarrow \bigcup_{i\in I} e_i \subseteq \bigcup_{j\in J} f_j$ ♦

(10) $\forall j \in J, \exists i \in I : f_j \subseteq e_i \Rightarrow C(\bigcup_{i\in I} e_i) \subseteq C(\bigcup_{j\in J} f_j)$ ♦

(11) $\forall j \in J, \exists i \in I : e_i \subseteq f_j \Rightarrow \bigcap_{i\in I} e_i \subseteq \bigcap_{j\in J} f_j$ ♦

(12) $\forall i \in I, \exists j \in J : f_j \subseteq e_i \Rightarrow C(\bigcap_{i\in I} e_i) \subseteq C(\bigcap_{j\in J} f_j)$ ♦

En toute rigueur, ces quatre formules sont même des généralisations des cas (i) - (iv) ci-dessus. Examinons par exemple le cas (i) :

selon (1), E^- aura au plus m composantes, mais rien n'empêche de créer plusieurs e_i^- disjoints pour le même e_i, c'est-à-dire :

$e_i^- = \bigcup_{j\in N^*} e_{ij}$ tel que $e_{ij} \subseteq e_i$:

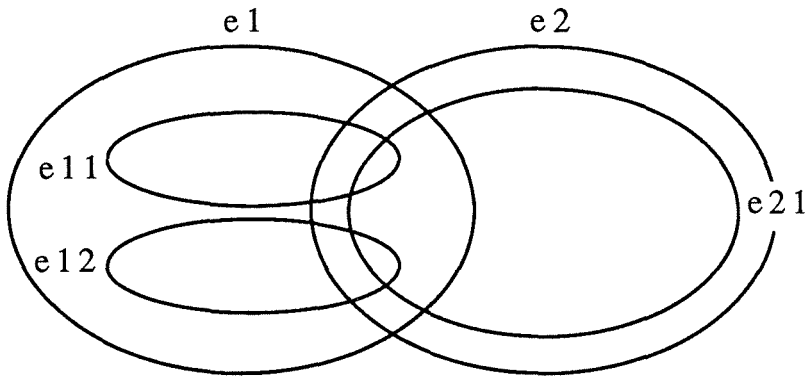


Figure 4.2. - Généralisation de (1) et (2) : $E = e_1 \cup e_2$; $E^- = (e_{11} \cup e_{12}) \cup e_{21}$

De même, selon (2), E^+ aura au moins m composantes, mais rien n'empêche de créer un e_i^+ qui soit sur-ensemble de e_i plusieurs différents (cas de e_1 dans la figure 4.2. ci-dessus).

Il faut clairement souligner les limites de la méthode : comme nous l'avons déjà indiqué plusieurs fois, elle ne fournit que des cas particuliers de vérifiabilité de l'inclusion. Ainsi, elle ne tient compte que des cas où les deux ensembles à comparer sont définis de la même manière (réunion, complément de réunion, intersection, complément d'intersection). Toutefois, il ne faut pas oublier que m ou n peuvent être égaux à 1 (ce qui permet le traitement de cas extrêmes), et que, bien entendu, on peut combiner les formules (9), (10), (11) et (12) à volonté.

Etape 3 : Algorithme de vérification d'inclusion

Afin de fournir une structure très modulaire et plusieurs points d'entrée à cet algorithme de vérification, éclatons les formules (9) à (12) :

(9) devient :

$$(\wedge_{i\in I}\,(e_i \subseteq (\cup_{j\in J}\, f_j))) \;\Rightarrow\; ((\cup_{i\in I}\, e_i) \subseteq (\cup_{j\in J}\, f_j)) \qquad (13)$$

$$(\vee_{j\in J}\,(e_i \subseteq f_j)) \;\Rightarrow\; (e_i \subseteq (\cup_{j\in J}\, f_j)) \qquad (14)$$

Cette décomposition en deux formules donne lieu à la possibilité d'appliquer tout de suite (14) au lieu de (13) d'abord (qui elle utilisera de toute façon (14)) au cas où #(I) = 1.

De même :

(10) devient :

$$(\wedge_{i\in I}\,(f_j \subseteq (\cup_{i\in I}\, e_i))) \;\Rightarrow\; (C(\cup_{i\in I}\, e_i) \subseteq C(\cup_{j\in J}\, f_j)) \qquad (15)$$

$$(\vee_{i\in I}\,(f_j \subseteq e_i)) \;\Rightarrow\; (f_j \subseteq (\cup_{i\in I}\, e_i)) \qquad (16)$$

(11) devient :

$$(\wedge_{j\in J}\,((\cap_{i\in I}\, e_i) \subseteq f_j)) \;\Rightarrow\; ((\cap_{i\in I}\, e_i) \subseteq (\cap_{j\in J}\, f_j)) \qquad (17)$$

$$(\vee_{i\in I}\,(e_i \subseteq f_j)) \;\Rightarrow\; ((\cap_{i\in I}\, e_i) \subseteq f_j) \qquad (18)$$

(12) devient :

$$(\wedge_{i\in I}\,((\cap_{j\in J}\, f_j) \subseteq e_i)) \;\Rightarrow\; (C(\cap_{i\in I}\, e_i) \subseteq C(\cap_{j\in J}\, f_j)) \qquad (19)$$

$$(\vee_{j\in J}\,(f_j \subseteq e_i)) \;\Rightarrow\; ((\cap_{j\in J}\, f_j) \subseteq e_i) \qquad (20)$$

(14) et (16) sont identiques dans leurs significations; idem pour (18) et (20).

Plutôt que d'écrire maintenant les algorithmes, voyons d'abord comment ces principes s'appliquent à nos besoins.

4.2.3. Spécification des fonctions de vérification d'implication

Comme nous l'avons convenu à la section 4.3. de la partie I, les opérateurs **and**, **or**, **not** et **implies** utilisés avec les expressions-NOMADE "masquent" en fait des intersections, réunions, compléments et inclusions au niveau des ensembles désignés par les fragments de ces expressions-NOMADE. Ainsi, les formules (13) à (20) sont immédiatement transposables aux expressions-NOMADE.

Cependant, l'analyse de l'exemple suivant nous amène à introduire une transformation préalable des expressions-NOMADE utilisées :

∀ A, ∀ B, ∀ C : **not** [(A **and** B) **or** C] **implies** [**not** (A **and** B)].

Cette implication est toujours vraie, mais aucune des formules ci-dessus ne pourrait la démontrer, parce que nous sommes en présence de deux expressions-NOMADE de signes contraires ! Bien entendu, il suffirait de ré-écrire le membre de droite sous la forme **not** [(A **and** B)], mais il s'agit d'un exemple trivial ici.

La solution la plus élégante à ce phénomène de déplacement du **not** consiste à mettre préalablement toutes les expressions-NOMADE utilisées sous forme normale disjonctive (disjonction de conjonctions de littéraux).

Cette transformation s'apparente évidemment très fortement à la mise sous forme clausale d'un prédicat, telle qu'elle est décrite par exemple dans [Nilsson 71].

Bien entendu, cette transformation donne lieu à un surcoût en temps de calcul et en place mémoire (s'il y a effectivement des **not** à déplacer vers des noms qualifiés), mais ce double inconvénient est quasi compensé par la rareté de tels **not**.

L'avantage de cette transformation est un énorme gain en puissance de notre technique de vérification d'implication et en élégance de l'algorithme sous-jacent. Une étude approfondie du problème sans passer par une telle transformation a révélé la nécessité de gestion de trois grandes classes de cas particuliers, et s'est soldée par une considérable complexification de certains algorithmes.

Ainsi, une pré-condition des fonctions suivantes sera que toute expression-NOMADE soumise (ou tout fragment d'expression-NOMADE soumis) soit sous forme normale disjonctive.

Partant d'une fonction booléenne vérifiant si une disjonction d'expressions-NOMADE en implique une autre, la structure abstraite des expressions-NOMADE donne lieu aux spécifications suivantes (l'ordre textuel reflétant une hiérarchie "utilise") :

function *LsNeImpliesLsNe* ($LsNe_A$, $LsNe_B$) : **boolean**

paramètres : $LsNe_A ::= \mathbf{or}_{i \in I}\ Ne_i$ (#(I) ≥1)
$LsNe_B ::= \mathbf{or}_{j \in J}\ Ne_j$ (#(J) ≥1)

effet : retourne ($LsNe_A$ **implies** $LsNe_B$), c'est-à-dire, selon (13) :

$$\wedge_{i \in I}\ (\ NeImpliesLsNe\ (\ Ne_i\ ,\ LsNe_B\ )\ )$$

♦

function *NeImpliesLsNe* (Ne_A , $LsNe_B$) : **boolean** {version provisoire}

paramètres : Ne_A ::= expression-NOMADE
$LsNe_B$ ::= $\mathbf{or}_{j \in J}\ Ne_j$ (#(J) ≥1)

effet : retourne (Ne_A **implies** $LsNe_B$), c'est-à-dire, selon (14) :

$$\vee_{j \in J}\ (\ NeImpliesNe\ (\ Ne_A\ ,\ Ne_j\ )\ )$$

♦

function *NeImpliesNe* (Ne_A , Ne_B) : **boolean**

paramètres : $Ne_A ::= \mathbf{or}_{i \in I}\ LsQN_i$ (#(I) ≥1)
$Ne_B ::= \mathbf{or}_{j \in J}\ LsQN_j$ (#(J) ≥1)

effet : retourne (Ne_A **implies** Ne_B), c'est-à-dire, selon (13) :

$$\wedge_{i \in I}\ (\ LsQNImpliesNe\ (\ LsQN_i\ ,\ Ne_B\ )\ )$$

♦

function *LsQNImpliesNe* ($LsQN_A$, Ne_B) : **boolean**

paramètres : $LsQN_A$ ::= liste de noms qualifiés
$Ne_B ::= \mathbf{or}_{j \in J}\ LsQN_j$ (#(J) ≥1)

effet : retourne ($LsQN_A$ **implies** Ne_B), c'est-à-dire, selon (14) :

$$\vee_{j \in J}\ (\ LsQNImpliesLsQN\ (\ LsQN_A\ ,\ LsQN_j\ )\ )$$

♦

function *LsQNImpliesLsQN* ($LsQN_A$, $LsQN_B$) : **boolean**

paramètres : $LsQN_A ::= \mathbf{and}_{i \in I}\ QN_i$ (#(I) ≥1)
$LsQN_B ::= \mathbf{and}_{j \in J}\ QN_j$ (#(J) ≥1)

effet : retourne ($LsQN_A$ **implies** $LsQN_B$), c'est-à-dire, selon (17) :

$$\wedge_{j \in J}\ (\ LsQNImpliesQN\ (\ LsQN_A\ ,\ QN_j\ )\ )$$

♦

function *LsQNImpliesQN* ($LsQN_A$, QN_B) : **boolean**

paramètres : $LsQN_A ::= \mathbf{and}_{i \in I}\ QN_i$ (#(I) ≥1)
QN_B ::= nom qualifié

effet : retourne ($LsQN_A$ **implies** QN_B), c'est-à-dire, selon (18) :

$$\vee_{i \in I}\ (\ QNImpliesQN\ (\ QN_i\ ,\ QN_B\ )\ )$$

♦

Il ne faut pas confondre cette dernière fonction avec la fonction QNImpliesLsQN (qui a été vue en 3.2.2.), qui était en fait le résultat de la fusion en une seule fonction de LsQNImpliesLsQN et LsQNImpliesQN pour le cas particulier I = { 1 }.

function *QNImpliesQN* (QN_A , QN_B) : **boolean**

paramètres : $QN_A ::=$ [**not**] $N_A\ LsQ_A$
$QN_B ::=$ [**not**] $N_B\ LsQ_B$

effet : retourne (QN_A **implies** QN_B), c'est-à-dire :

si QN_A non nié $\wedge$ QN_B non nié

alors NImpliesN (N_A,N_B) $\wedge$ LsQImpliesLsQ (LsQ_A,LsQ_B)

si QN_A nié $\wedge$ QN_B nié

alors NImpliesN (N_B,N_A) $\wedge$ LsQImpliesLsQ (LsQ_B,LsQ_A)

si QN_A non nié $\wedge$ QN_B nié

alors NImpliesNotN (N_A,N_B) $\vee$ LsQImpliesNotLsQ (LsQ_A,LsQ_B)

si QN_A nié $\wedge$ QN_B non nié,

alors NotNImpliesN (N_A,N_B) $\wedge$ NotLsQImpliesLsQ (LsQ_A,LsQ_B)

♦

Cette fonction est une généralisation de la fonction de même nom qui a été spécifiée en 3.2.2. Généralisation, car à présent nous ne connaissons plus d'avance le signe de QN_A, et car QN_A peut maintenant contenir un nom de partition.

La fonction NImpliesN (N_A , N_B) est donc une généralisation de la fonction OEImpliesN (OE_A , N_B) vue en 3.2.2., car il faut maintenant aussi considérer le cas où le premier argument contient une désignation de partition.

Ainsi, le tableau des possibilités de paramétrage (Tableau 3.2.) devient (en continuant le raisonnement esquissé en 3.2.2.) :

N_A \ N_B	OE_B (sans)	OE_B (avec)	$/R_B.A_B/OE_B$ (sans)	$/R_B.A_B/OE_B$ (avec)	$/R_B.A_B/$ (sans)
OE_A (sans méta-caractères)	$OE_A = OE_B$	$OE_A \equiv OE_B$	$OE_A \in /R_B.A_B/$ $\wedge$ $OE_A = OE_B$	$OE_A \in /R_B.A_B/$ $\wedge$ $OE_A \equiv OE_B$	$OE_A \in /R_B.A_B/$
OE_A (avec méta-caractères)	**false**	$OE_A \equiv OE_B$	**false**	**false**	**false**
$/R_A.A_A/OE_A$ (sans)	$OE_A = OE_B$	$OE_A \equiv OE_B$	$OE_A = OE_B$	$OE_A \in /R_B.A_B/$ $\wedge$ $OE_A \equiv OE_B$	$OE_A \in /R_B.A_B/$
$/R_A.A_A/OE_A$ (avec)	**false**	$OE_A \equiv OE_B$	**false**	$(R_A,A_A) = (R_B,A_B)$ $\wedge$ $OE_A \equiv OE_B$	$(R_A,A_A) = (R_B,A_B)$
$/R_A.A_A/$ (sans)	**false**	OE_B = **	**false**	$(R_A,A_A) = (R_B,A_B)$ $\wedge$ OE_B = **	$(R_A,A_A) = (R_B,A_B)$

Tableau 4.1. - Evaluation de N_A **implies** N_B

La contrainte d'absence de méta-caractères dans OE_A a dû être relâchée, mais nous la maintenons pour R_A, A_A, R_B et A_B (car des désignations génériques de partitions nous semblent peu réalistes).

D'où la spécification suivante :

function *NImpliesN* (N_A , N_B) : **boolean**

paramètres : $N_A ::= [\ /R_A.A_A/\ ]\ [\ O_A\ [\ .E_A\ ]\ ]$
$N_B ::= [\ /R_B.A_B/\ ]\ [\ O_B\ [\ .E_B\ ]\ ]$

effet : retourne (N_A **implies** N_B), selon le tableau 4.1.

♦

De même, NImpliesNotN (N_A , N_B) est une généralisation de OEImpliesNotN (N_A , N_B), qui a été vue en 3.2.2. D'où le tableau 4.2. qui est une extension du tableau 3.3. :

N_A \ N_B	OE_B (sans)	OE_B (avec)	$/R_B.A_B/OE_B$ (sans)	$/R_B.A_B/OE_B$ (avec)	$/R_B.A_B/$ (sans)
OE_A (sans méta-caractères)	$OE_A \neq OE_B$	$OE_A \approx OE_B$	$OE_A \notin /R_B.A_B/$ v $OE_A \neq OE_B$	$OE_A \notin /R_B.A_B/$ v $OE_A \approx OE_B$	$OE_A \notin /R_B.A_B/$
OE_A (avec méta-caractères)	$OE_B \approx OE_A$	**false**	$OE_B \approx OE_A$	**false**	**false**
$/R_A.A_A/OE_A$ (sans)	$OE_B \notin /R_A.A_A/$ v $OE_B \neq OE_A$	$OE_A \approx OE_B$	$OE_A \neq OE_B$	$OE_A \notin /R_B.A_B/$ v $OE_A \approx OE_B$	$OE_A \notin /R_B.A_B/$
$/R_A.A_A/OE_A$ (avec)	$OE_B \notin /R_A.A_A/$ v $OE_B \approx OE_A$	**false**	$OE_B \notin /R_A.A_A/$ v $OE_B \approx OE_A$	**false**	**false**
$/R_A.A_A/$ (sans)	$OE_B \notin /R_A.A_A/$	**false**	$OE_B \notin /R_A.A_A/$	**false**	**false**

Tableau 4.2. - Evaluation de N_A **implies not** N_B

On en dérive la spécification suivante :

function *NImpliesNotN* (N_A , N_B) : **boolean**

paramètres : $N_A ::= [\ /R_A.A_A/\ ]\ [\ O_A\ [\ .E_A\ ]\ ]$
$N_B ::= [\ /R_B.A_B/\ ]\ [\ O_B\ [\ .E_B\ ]\ ]$

effet : retourne (N_A **implies not** N_B), selon le tableau 4.2.

♦

Finalement, NotNImpliesN (N_A , N_B) est une nouvelle fonction :

N_A \ N_B	OE_B (sans)	OE_B (avec)	$/R_B.A_B/OE_B$ (sans)	$/R_B.A_B/OE_B$ (avec)	$/R_B.A_B/$ (sans)
OE_A (sans méta-caractères)	**false**	$OE_B = **$	**false**	**false**	**false**
OE_A (avec méta-caractères)	$OE_A = **$	$OE_B = **$	$OE_A = **$	**false**	**false**
$/R_A.A_A/OE_A$ (sans)	**false**	$OE_B = **$	**false**	**false**	**false**
$/R_A.A_A/OE_A$ (avec)	**false**	$OE_B = **$	**false**	**false**	**false**
$/R_A.A_A/$ (sans)	**false**	$OE_B = **$	**false**	**false**	**false**

Tableau 4.3. - Evaluation de **not** N_A **implies** N_B

La spécification est la suivante :

function *NotNImpliesN* (N_A , N_B) : **boolean**

paramètres : $N_A ::= [\ /R_A.A_A/\ ]\ [\ O_A\ [\ .E_A\ ]\ ]$
$N_B ::= [\ /R_B.A_B/\ ]\ [\ O_B\ [\ .E_B\ ]\ ]$

effet : retourne (**not** N_A **implies** N_B), selon le tableau 4.3.

♦

C'est ici que nous allons donc spécifier les fonctions LsQImpliesLsQ et LsQImpliesNotLsQ, qui ont déjà été utilisées (mais non spécifiées) lors de la conception de ChkRight (cf. section 3.2.2.). Ainsi :

function *LsQImpliesLsQ* (LsQ_A , LsQ_B) : **boolean**

paramètres : $LsQ_A ::= \wedge_{i \in I} Q_i$ $(\#(I) \geq 0)$
$LsQ_B ::= \wedge_{j \in J} Q_j$ $(\#(J) \geq 0)$

effet : retourne ($LsQ_A \Rightarrow LsQ_B$), c'est-à-dire, selon (17) :

$$\wedge_{j \in J} (LsQImpliesQ (LsQ_A , Q_j))$$

♦

et :

function *LsQImpliesNotLsQ* (LsQ_A , LsQ_B) : **boolean**

paramètres : $LsQ_A ::= \wedge_{i \in I} Q_i$ $(\#(I) \geq 0)$
$LsQ_B ::= \wedge_{j \in J} Q_j$ $(\#(J) \geq 0)$

effet : retourne ($LsQ_A \Rightarrow \sim LsQ_B$), c'est-à-dire :

$$\vee_{j \in J} (\text{LsQImpliesQ} (LsQ_A , \sim Q_j))$$

♦

(notons que par $\sim Q_j$, nous entendons Q_j après remplacement de son opérateur relationnel par son complémentaire);

Mais la fonction suivante est nouvelle :

function *NotLsQImpliesLsQ* (LsQ_A , LsQ_B) : **boolean**

paramètres : $LsQ_A ::= \wedge_{i \in I} Q_i$ $(\#(I) \geq 0)$

$LsQ_B ::= \wedge_{j \in J} Q_j$ $(\#(J) \geq 0)$

effet : retourne ($\sim LsQ_A \Rightarrow LsQ_B$), c'est-à-dire :

$$\wedge_{j \in J} (\text{NotLsQImpliesQ} (LsQ_A , Q_j))$$

♦

Ensuite on a :

function *LsQImpliesQ* (LsQ_A , Q_B) : **boolean**

paramètres : LsQ_A $::= \wedge_{i \in I} Q_i$ $(\#(I) \geq 0)$

Q_B ::= qualification

effet : retourne ($LsQ_A \Rightarrow Q_B$), c'est-à-dire, selon (18) :

$$\vee_{i \in I} (\text{QImpliesQ} (Q_i , Q_B))$$

♦

ainsi que :

function *NotLsQImpliesQ* (LsQ_A , Q_B) : **boolean**

paramètres : LsQ_A $::= \wedge_{i \in I} Q_i$ $(\#(I) \geq 0)$

Q_B ::= qualification

effet : retourne ($\sim LsQ_A \Rightarrow Q_B$), c'est-à-dire :

$$\wedge_{i \in I} (\text{QImpliesQ} (\sim Q_i , Q_B))$$

♦

Finalement :

function *QImpliesQ* (Q_A , Q_B) : **boolean**

paramètres : $Q_A ::= Attribute_A\ RelOp_A\ LsV_A$
$Q_B ::= Attribute_B\ RelOp_B\ LsV_B$

avec $RelOp_A, RelOp_B \in \{ =, \neq, \geq, >, \leq, < \}$

effet : retourne $(Q_A \Rightarrow Q_B)$.

♦

Vu les deux types de qualifications (définitionnelle - les valeurs sont implicitement séparées par un **or**; factuelle - les valeurs sont implicitement séparées par un **and**), il existe théoriquement quatre variantes de cette fonction. Cependant, seulement deux cas ont un intérêt pratique :

- Q_A factuelle, Q_B définitionnelle (cas de ChkRight);
- Q_A et Q_B définitionnelles (cas de AddRight, DelRight et ChgRight).

Heureusement, le premier cas est un cas particulier du second, car pour démontrer :

$$(A = \wedge_{i \in I} V_i) \Rightarrow (B = \vee_{j \in J} V_j)$$

il **suffit** de prouver que :

$$(A = \vee_{i \in I} V_i) \Rightarrow (B = \vee_{j \in J} V_j).$$

Ainsi, aucune distinction de cas ne devra être faite !

Le détail des comparaisons à effectuer se trouve dans le pseudo-code en annexe E; notons seulement que $Attribute_A$ doit "matcher" $Attribute_B$ pour que l'implication puisse être vraie, et qu'en l'absence de méta-caractères, le tableau QIQ (Tableau 4.4.) ci-dessous permet une prise de décision rapide :

$RelOp_B$ / $RelOp_A$	≠	=	<	>	≤	≥
	= < >	= < >	= < >	= < >	= < >	= < >
≠	V F F	F F F	F F F	F F F	F F F	F F F
=	F V V	V F F	F V F	F F V	V V F	V F V
<	V V F	F F F	V V F	F F F	V V F	F F F
>	V F V	F F F	F F F	V F V	F F F	V F V
≤	F V F	F F F	F V F	F F F	V V F	F F F
≥	F F V	F F F	F F F	F F V	F F F	V F V

Tableau 4.4. - Valeurs de vérité de $Q_A \Rightarrow Q_B$ (valeurs alphanumériques, sans méta-caractères)
(les trois booléens correspondant respectivement à $V_A = V_B$, $V_A < V_B$ et $V_A > V_B$)

Nous pouvons à présent exprimer de manière très précise les conditions (3), (4) et (7) de la section 4.1.2.. Ainsi :

(3) NeImpliesLsNe(Ne,**first**(LoginRights[LsOwnNe,(L ≡ Label)]))
∧ NeImpliesLsNe(Ne,**first**(LoginRights[LsInhNe,(L ≡ Label)]))

(4) ~ NeImpliesLsNe(Ne,TargetMan.LsRight[LsNe,(Label = L)])

(7) LsNeImpliesLsNe(Δ(Ne_1,Ne_2),**first**(LoginRights[LsOwnNe,(L ≡ Label)]))
∧ LsNeImpliesLsNe(Δ(Ne_1,Ne_2),**first**(LoginRights[LsInhNe,(L ≡ Label)]))

Dorénavant, il est clair que si une des commandes de modification échoue (message d'erreur) à cause d'une de ces conditions, alors il ne s'agit pas toujours d'une opération "illégale", mais plutôt (et en toute généralité) d'un manque de puissance de la technique de vérification sous-jacente, qui peut donner **false** alors qu'il aurait fallu retourner **true** (bien entendu, si elle retourne **true**, on est sûr que c'est effectivement la bonne réponse).

La solution présentée jusqu'ici est une fidèle adaptation des principes exposés en 4.2.2. On peut cependant aller encore un peu plus loin, comme le suggère l'exemple suivant :

Exemple :

Ne_A = [L_1 **or** L_2]
$LsNe_B$ = Ne_{B1} **or** Ne_{B2} = [L_1^+ **or** L_3] **or** [L_2^+ **or** L_4]

les L_i étant des conjonctions de noms qualifiés.

La version actuelle de NeImpliesLsNe affirmera que Ne_A **implies** $LsNe_B$ est faux, parce qu'elle comparera Ne_A successivement à Ne_{B1} et à Ne_{B2}. Comme :

Ne_A est équivalent à [L_1] **or** [L_2]

on voit que Ne_A **implies** $LsNe_B$, parce que [L_1] **implies** Ne_{B1} et parce que [L_2] **implies** Ne_{B2}. ♦

Ceci est dû à la grammaire des listes d'expressions-NOMADE, qui est telle que deux disjonctions s'y suivent. Plutôt que d'écrire une routine éclatant chaque expression-NOMADE (non niée) en autant d'expressions-NOMADE qu'il y avait de listes de noms qualifiés, nous allons rompre la symétrie actuelle des fonctions de vérification d'implication en modifiant NeImpliesLsNe pour qu'elle appelle une nouvelle fonction (à savoir LsQNImpliesLsNe) qui elle utilisera directement LsQNImpliesNe. Cependant, il ne faut pas pour autant supprimer NeImpliesNe, car cette fonction constitue toujours un des multiples points d'entrée dans l'algorithme global !

D'où les nouvelles spécifications :

function *NeImpliesLsNe* (Ne_A , $LsNe_B$) : **boolean** {version définitive}

paramètres : Ne_A ::= $\mathbf{or}_{i\in I}$ $LsQN_i$ (#(I) ≥1)
$LsNe_B$ ::= $\mathbf{or}_{j\in J}$ Ne_j (#(J) ≥1)

effet : retourne (Ne_A **implies** $LsNe_B$), c'est-à-dire, selon (13) :

$\wedge_{i\in I}$ (LsQNImpliesLsNe ($LsQN_i$, $LsNe_B$))

♦

function LsQNImpliesLsNe ($LsQN_A$, $LsNe_B$) : **boolean**

paramètres : $LsQN_A$ ::= liste de noms qualifiés
$LsNe_B$ ::= $\mathbf{or}_{j \in J}$ Ne_j (#(J) ≥1)

effet : retourne ($LsQN_A$ **implies** $LsNe_B$), c'est-à-dire, selon (14) :

$$\vee_{j \in J} (\text{LsQNImpliesNe} (LsQN_A , Ne_j))$$

♦

4.3. Variantes de scénario de modification

Dans cette section, nous allons présenter plusieurs variantes de scénario pour les primitives IAddRight, IDelRight et IChgRight. Ces variantes consisteront essentiellement en des modifications des effets explicités pour ces primitives en 4.1.2., voire aussi leur existence ou non (cas de IChgRight).

En ce qui concerne IAddRight, nous avons déjà attiré l'attention sur le fait qu'on peut ou non vérifier si le droit à ajouter est redondant avec les droits existants (compactage de clauses rights).

Nous ne reviendrons pas sur ce cas, car il en existe au moins deux autres qui ouvrent chacun tout un champ de perspectives non considérées jusqu'à présent :

- l'étiquette "*" peut donner lieu à des effets spéciaux lors d'un IAddRight, IDelRight ou IChgRight, car elle a des significations différentes chez le modifieur et chez Target;

- il existe des cas d'apparitions/disparitions d'étiquettes qui sont délicats à gérer.

4.3.1. Effets spéciaux dûs à l'étiquette "*"

4.3.1.1. Hypothèses

Citons d'abord les hypothèses nécessaires à l'apparition de ces effets spéciaux :

- l'étiquette "*" est gérée explicitement (c'est-à-dire qu'elle se trouve dans toute clause rights);

- la primitive IChgRight fait partie des primitives de modification des droits.

La combinaison de ces deux hypothèses implique qu'il peut exister pour l'étiquette "*" des portées autres que [(**)].

4.3.1.2. Effets spéciaux lors d'un AddRight

Illustrons nos propos par un exemple :

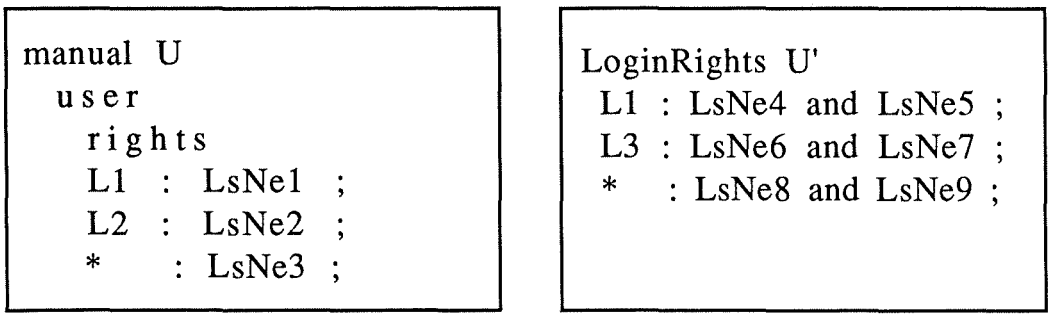


Figure 4.3. - Avant l'AddRight

(supposons que $LsNe_3$ ne contienne pas une expression-NOMADE égale à [(**)], car sinon la suite du raisonnement n'a pas de sens).

Il est primordial de noter que la signification exacte de l'étiquette "*" est impossible à déterminer si on ne regarde qu'une seule clause rights. Ainsi l'ensemble L de toutes les étiquettes connues ne peut être établi que si on consulte toutes les clauses rights.

Supposons connu $L = \{ L_1 , \ldots , L_m \}$ $(m > 2)$ et notons $L_*(U)$ l'ensemble des étiquettes désignées par l'étiquette "*" dans la clause rights (ou les LoginRights, selon le contexte) de U. Ainsi on a:

$L_*(U) = \{ L_3 , L_4 , \ldots , L_m \}$ et
$L_*(U') = \{ L_2 , L_4 , \ldots , L_m \}$.

Si maintenant U' avait le droit d'effectuer : AddRight (U , "*" , Ne) et si l'ajout est légal, alors suffit-il d'ajouter Ne à $LsNe_3$ (supposons qu'aucun contrôle de redondance ne soit fait), comme on aurait fait si "*" était une étiquette "normale" ? Non, car en toute rigueur, seul le résultat suivant est correct :

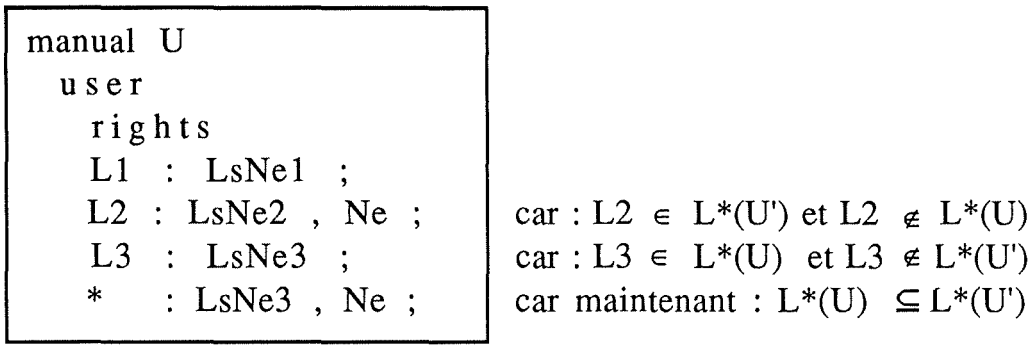


Figure 4.4. - Après l'AddRight (version correcte)

En d'autres termes, toute étiquette appartenant à $L_*(U) \setminus L_*(U')$ est créée chez U avec l'ancien droit de l'étiquette "*"; ceci assure que maintenant $L_*(U) \subseteq L_*(U')$, c'est-à-dire que l'on peut ajouter Ne non seulement au droit de l'étiquette "*" chez U, mais aussi aux droits de toutes les étiquettes appartenant maintenant à $L_*(U') \setminus L_*(U)$. ♦

4.3.1.3. Effets spéciaux lors d'un DelRight

Le même phénomène se présente à l'envers au cours d'un DelRight. En effet, supposons que U' ait le droit d'effectuer (en repartant de la figure 4.4.) : DelRight (U , "*" , Ne) et que cette suppression soit légale.

Comme AddRight et DelRight sont normalement des commandes à effets réciproques, on pourrait s'attendre à retrouver pour U la clause rights de la figure 4.3. Ceci n'est pourtant possible qu'au prix d'un effort de vérification de non-redondance, car autrement le résultat serait le suivant :

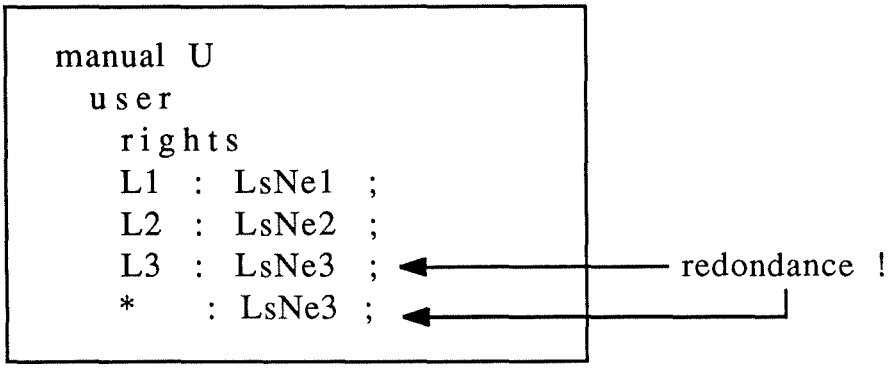


Figure 4.5. - Après DelRight (sans contrôle de redondance)

4.3.1.4. Interprétation de l'étiquette "*"

La question est surtout de savoir si ce sont là les effets souhaité par U'. Tout dépend de l'interprétation donnée à l'étiquette "*" lors d'une modification de droits : soit elle désigne toutes les étiquettes non mentionnées (i.e. $L_*(U)$; et il faut tenir compte de ces effets spéciaux), soit elle est vue comme une étiquette "normale" (et on peut ignorer les effets spéciaux).

4.3.2. Apparitions et disparitions d'étiquettes

Les hypothèses sont les mêmes qu'en 4.3.1.

Nous dirons qu'il y a apparition d'une étiquette E lorsque la commande AddRight (U , E , Ne) fait apparaître "E : Ne" dans le manuel de U.

Nous dirons qu'il y a disparition d'une étiquette E lorsque la commande DelRight (U , E , Ne) fait disparaître "E : Ne" du manuel de U.

4.3.2.1. Disparitions d'étiquette

Illustrons ce phénomène par un exemple :

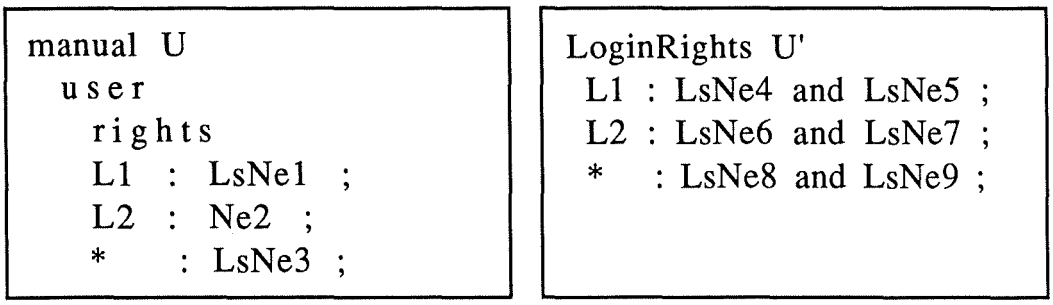


Figure 4.6. - Avant DelRight

(supposons que $LsNe_3$ ne contienne aucune expression-NOMADE égale à [(**)]).

Si à présent, U' avait le droit de lancer : DelRight (U , L_2 , Ne_2), alors il y a deux politiques possibles de gestion de l'étiquette L_2 chez U :

(P1) : **Disparition**, c'est-à-dire dorénavant L_2 se trouve désignée par l'étiquette "*"; 3 variantes de vérification de la légalité de la suppression existent alors :

(P11) : on ne traite que la suppression de Ne_2, c'est-à-dire on ne vérifie que si Ne_2 **implies** $LsNe_6$ **and** $LsNe_7$;

(P12) : (P11) plus traitement de l'éventuel ajout de droit implicite (en effet, L_2 étant à présent couverte par "*", sa portée est maintenant $LsNe_3$ qui peut être plus générale que Ne_2); d'où on vérifie aussi si $LsNe_3$ **implies** $LsNe_6$ **and** $LsNe_7$;

(P13) : si la fonction $\Delta(LsNe_A,LsNe_B)$ est prévue, alors on peut améliorer le processus de vérification de (P12) en contrôlant si $\Delta(Ne_2,LsNe_3)$ **implies** $LsNe_6$ **and** $LsNe_7$;

(P2) : **Maintien et remplacement** de Ne_2 par **not**[(**)], c'est-à-dire qu'on n'a qu'à vérifier si Ne_2 **implies** $LsNe_6$ **and** $LsNe_7$.

En résumé :

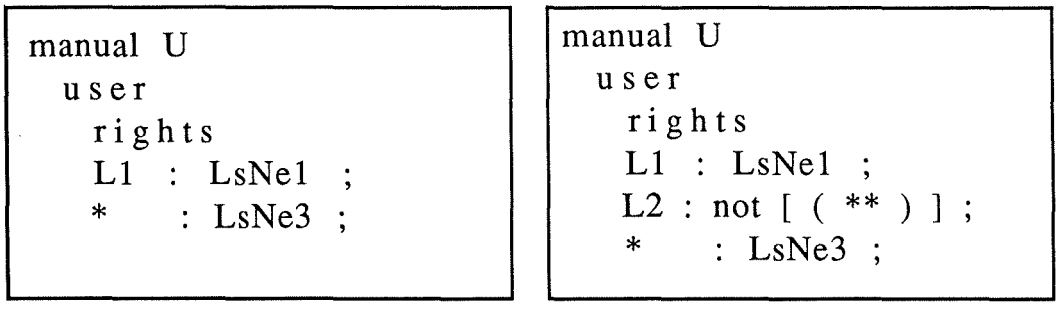


Figure 4.7. - Après DelRight

4.3.2.2. Apparitions d'étiquette

De manière symétrique, si U' avait maintenant le droit de lancer : AddRight (U , L_2 , Ne_2), alors les mêmes deux politiques de gestion de l'étiquette L_2 sont possibles :

$(P1)^{-1}$: **Apparition**, c'est-à-dire dorénavant L_2 n'est plus désignée par l'étiquette "*"; 3 variantes de vérification de la légalité de l'addition existent alors :

- $(P11)^{-1}$: on ne traite que l'addition de Ne_2, c'est-à-dire on ne vérifie que si Ne_2 **implies** $LsNe_6$ **and** $LsNe_7$;
- $(P12)^{-1}$: $(P11)^{-1}$ plus traitement préalable de l'éventuelle suppression de droit implicite (en effet, L_2 n'étant à présent plus couverte par "*", sa portée est maintenant Ne_2 qui peut être plus restrictive que $LsNe_3$); d'où on vérifie aussi si $LsNe_3$ **implies** $LsNe_6$ **and** $LsNe_7$;
- $(P13)^{-1}$: si la fonction $\Delta(LsNe_A,LsNe_B)$ est prévue, alors on pourrait améliorer le processus de vérification de $(P12)^{-1}$ en contrôlant si : $\Delta(Ne_2,LsNe_3)$ **implies** $LsNe_6$ **and** $LsNe_7$;

$(P2)^{-1}$: **Maintien et remplacement** de **not**[(**)] par Ne_2, c'est-à-dire qu'on n'a qu'à vérifier si Ne_2 **implies** $LsNe_6$ **and** $LsNe_7$.

En partant de la figure 4.7., on retrouvera donc la figure 4.6.

Il existe un autre **cas particulier** d'apparition d'étiquette : décrivons-le à l'aide d'un exemple. En partant de la figure 4.3., supposons que U' ait le droit d'effectuer : DelRight (U , L_3 , Ne_3) et qu'il s'agisse d'une suppression légale (i.e. Ne_3 **implies** $LsNe_6$ **and** $LsNe_7$). La spécification actuelle de DelRight signalera l'absence de l'étiquette L_3 dans les droits propres de U, alors qu'on pourrait y ajouter la redondance suivante : "L_3 : $LsNe_3$" sans en changer la sémantique. Dans ce cas, DelRight réussirait à condition que $Ne_3 \in LsNe_3$, c'est-à-dire que le résultat serait le suivant :

```
manual U
  user
    rights
    L1  :  LsNe1  ;
    L2  :  LsNe2  ;
    L3  :  LsNe3 \ Ne3  ;
    *   :  LsNe3  ;
```

Figure 4.8. - Après DelRight

4.3.2.3. Discussion

Les spécifications actuelles de IAddRight et IDelRight assurent la politique (P11) & $(P11)^{-1}$, qui correspond peut-être à l'intuition première, mais n'est pas correcte en toute rigueur. Les politiques (P12) & $(P12)^{-1}$ et (P13) & $(P13)^{-1}$ remédient à cela.

La politique (P2) & $(P2)^{-1}$ peut aussi être vue comme étant correcte, même si elle (tout comme les deux précédentes) ne donne probablement pas le résultat attendu par U'. L'effet de cette politique est qu'une fois créée, une étiquette ne peut jamais disparaître.

Il faut cependant souligner que la commande AddRight (resp. DelRight) peut bien sûr aussi avertir U' si elle doit créer (resp. détruire) une étiquette, afin que U' décide lui-même de la politique, voire d'abandonner la commande.

Finalement, rien n'exclut le cas L_2 = "*", c'est-à-dire qu'il faut combiner la politique choisie avec les considérations de 4.3.1.

5. EVALUATION

Nous avons présenté dans cette deuxième partie le système de sécurité de l'environnement de programmation NOMADE. Etant donné le caractère encore fort expérimental des spécifications de NOMADE et sa non implémentation actuelle, nous ne sommes malheureusement pas en mesure de juger de l'utilisabilité de notre système de sécurité. Voilà pourquoi nous avons, chaque fois que possible, tenté d'explorer un maximum de *voies alternatives*, afin de faciliter les expérimentations une fois que NOMADE sera exécutable.

Il en est ainsi par exemple pour la primitive IChgRight. Nous l'avons incluse tout au long de notre discussion, en laissant ouverte la question de sa nécessité ou non.

D'autre part, vu qu'il s'agit ici d'un noyau d'environnement de programmation, un maximum de responsabilité quant à la mise au point d'un *système de protection "personnalisé"* a été délégué aux utilisateurs eux-mêmes. Cette ouverture de notre système de sécurité a cependant pour le moment l'inconvénient de nous laisser incertains quant à sa réelle applicabilité!

Il faut donc certainement procéder à un réglage de ce système de sécurité en fonction des résultats de la double *expérimentation* tant du côté concepteurs que du côté utilisateurs.

5.1. Problème de la granularité de protection

Malheureusement, les expressions-NOMADE sont telles qu'il n'y a pas moyen de protéger certains attributs seulement d'une clause attributes, ni certaines relations d'une clause relations. En d'autres termes, le grain le plus fin des entités protégeables par notre système de sécurité est la *clause* (attributes, relations, ...) d'un manuel, plutôt qu'une ligne de clause.

Voilà pourquoi nous proposons une extension de la désignation des éléments de type manuel (cf. partie I - section 2.4.) :

- on peut suffixer "manattr" par le nom d'un attribut (contenant éventuellement des méta-caractères); exemple F:I:R.manattr.language

- on peut suffixer "manrel" par le nom d'une relation (contenant éventuellement des méta-caractères); exemple F:I:R.manattr.dep

Cette généralisation permet alors une protection efficace pour certains attributs ou relations.

5.2. Extension possible

Personnellement, nous sommes un peu sceptiques quant à la faculté des usagers d'être au courant des droits effectifs (qui sont donc calculés au moment du "login" à partir des droits propres et des droits hérités) car l'héritage est un mécanisme non trivial. Cette *mé-connaissance éventuelle* risque de gêner la modification des droits (du point de vue "légalité" de la modification) ainsi que de provoquer chez le modifieur un doute au sujet de l'effet réel de son opération (rappelons que IAddRight, IDelRight et IChgRight modifient seulement les droits propres de Target). Il ne faut cependant pas hâtivement conclure que l'héritage de clauses rights devrait être banni du système de sécurité, car c'est justement cet héritage qui semble aussi prometteur d'une bonne factorisation de l'expression des droits usagers.

Nous pensons plutôt à une extension future de notre système de sécurité qui consisterait en une *analyse dynamique préalable des effets d'une modification de droits* , telle que [Trueblood 86] l'a fait pour le modèle de Hartson ([Hartson 84]).

En bref, cette analyse faciliterait les réponses aux deux types de questions suivants (et ce surtout dans un contexte où chaque utilisateur peut appartenir à plusieurs groupes) :

- est-ce que toute décision d'accès (ChkRight) est indépendante d'un nom qualifié N donné ? Si oui, alors N peut être retiré du droit en question. En effet, si on combine les droits de plusieurs groupes, certains prédicats peuvent se compléter ou s'annuler mutuellement;

- quelles sont les conditions sous lesquelles une décision d'accès est dépendante d'un nom qualifié N donné ? En d'autres termes, quelles valeurs de vérité doivent prendre les autres noms qualifiés pour que la décision d'accès dépende de N ?

La mise en oeuvre de cette analyse est basée sur la notion de *différence booléenne* . Soit un droit D s'exprimant en fonction de noms qualifiés :

$D = f (N_1 , \ldots , N_i , \ldots , N_n)$.

Par définition, la **différence booléenne** de D par rapport aux noms qualifiés N_i se calcule comme suit :

$$dD/dN_i = f (N_1 , \ldots , N_{i-1} , \textbf{false} , N_{i+1} , \ldots , N_n) \oplus f (N_1 , \ldots , N_{i-1} , \textbf{true} , N_{i+1} , \ldots , N_n)$$

$$= \begin{cases} \textbf{false}, \text{ si D est toujours indépendant de } N_i; \\ \textbf{true}, \text{ si D est toujours dépendant de } N_i; \\ g (N_1 , \ldots , N_{i-1} , N_{i+1} , \ldots , N_n), \text{ sinon.} \end{cases}$$

Les deux premiers cas correspondent à la première question ci-dessus; le troisième cas correspond à la seconde question, et la réponse s'obtient en évaluant $g (N_1 , \ldots , N_{i-1} , N_{i+1} , \ldots , N_n) = \textbf{true}$.

En guise d'illustration de l'utilité d'une telle extension, donnons deux petits exemples :

Exemple 1 :

Supposons que l'usager U (appartenant seulement à la partition /R.A./) ait comme droit effectif pour l'étiquette L :

$L : LsNe_1$ **and** $LsNe_2$;

(c'est-à-dire "$L : LsNe_1$" est le droit propre de U, et "$L : LsNe_2$" est le droit effectif de /R.A/).

Si à présent, l'usager U' voulait ajouter "$L : Ne_3$"à U (nous ne sommes pas préoccupés ici par les questions de "légalité" et de droit de modification), alors il se peut en toute généralité que :

Ne_3 **and** $LsNe_2$ = **false**

c'est-à-dire que l'ajout de Ne_3 soit sans effets pour U ! Le drame, c'est que U' peut ne pas se rendre compte de l'inutilité de son opération, car il ne sait pas voir les droits effectifs de U. Un outil d'analyse effectuant ce genre de diagnostic serait bien entendu le bienvenu.

Remarquons cependant qu'il existe un autre type d'opérations inutiles : l'ajout d'un droit redondant avec les droits déjà existants; i.e. si Ne_3 **implies** $LsNe_1$, voire si Ne_3 **implies** $LsNe_1$ **and** $LsNe_2$. Ce problème a déjà été discuté en 4.1.2.2. (remarque 1) et s'il est aussi gérable par l'outil proposé, il n'est pas aussi grave que l'exemple traité ici. ♦

Exemple 2 :

Supposons que l'usager U (appartenant seulement à la partition /R.A./) ait comme droit effectif pour l'étiquette L :

L : $LsNe_1$ **and** $LsNe_2$;

tel que $LsNe_1 = Ne_{11}$ **or** Ne_{12} **or** Ne_{13}.

Si à présent, l'usager U' voulait retirer "L : Ne_{12}" à U, alors il serait souhaitable que U' puisse savoir au préalable sous quelles conditions une décision d'accès relative à l'étiquette L était vraiment dépendante de Ne_{12}. Ces conditions s'exprimeront en fonction de Ne_{11}, Ne_{13} et $LsNe_2$, voire de leurs composants. ♦

Ces deux exemples montrent clairement qu'à l'aide d'un pareil outil d'analyse, une personne modifiant les droits de quelqu'un d'autre pourra par la suite rendre ces modifications plus pertinentes (c'est-à-dire que dans l'exemple 1 ci-dessus, U' pourrait étendre $LsNe_2$, et dans l'exemple 2, U' pourrait modifier Ne_{11}, Ne_{13} et/ou $LsNe_2$).

5.3. Le système de sécurité de NOMADE et le modèle HRU

5.3.1. De l'utilité des modèles formels de sécurité

Plusieurs auteurs ([Snyder 81], [Landwehr 81], ...) ont préconisé la nécessité de **modèles formels de sécurité**, et ce afin de pouvoir démontrer qu'un système est effectivement sûr. Ainsi, les **questions** suivantes sont rarement abordées ([Snyder 81]) :

- est-ce que le système de sécurité limite vraiment l'accès à l'information aux utilisateurs désignés par leur propriétaire ?
- est-ce que les règles de délégation de droits permettent vraiment d'établir des relations de partage des données ?
- sous quelles circonstances est-ce que l'information peut être disséminée dans le système ?
- quelle politique de protection est implémentée par le système de sécurité ?

etc.

Ces questions sont très pertinentes, mais il n'est pas facile d'y répondre, car en général, les détails d'implémentation d'un système de sécurité donné masquent le modèle abstrait sous-jacent, dans lequel la formulation précise de ces questions (et donc de leurs réponses; espérons-le !) serait plus facile.

5.3.2. Le modèle HRU

Harrison, Ruzzo et Ullman ([Harrison 76]) ont conçu un modèle formel général (modèle HRU) de l'approche matrice d'autorisation afin de répondre surtout à la troisième question ci-dessus. Après une brève description de ce modèle (en termes de la théorie des graphes, comme l'a fait [Snyder 81]), nous allons l'appliquer au système de sécurité de NOMADE et essayer d'en tirer des conclusions.

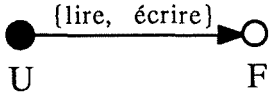
Plutôt que de parler de la matrice d'autorisation proprement dite, considérons-la comme une matrice d'adjacence afin de la transposer en une représentation graphique plus facile à comprendre. Ainsi, sujets et objets deviennent des noeuds et les règles d'autorisation deviennent des arcs étiquetés. Adoptons les conventions suivantes pour les noeuds :

● représente un sujet;

○ représente un objet (au sens strict);

⊗ représente soit un sujet, soit un objet.

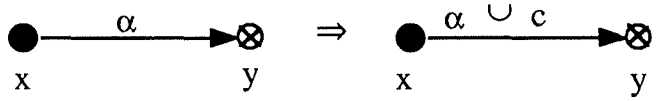
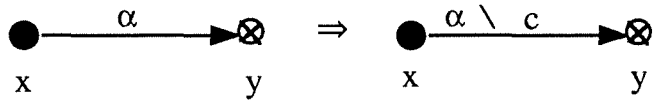
Ainsi :



signifie que l'usager U peut lire/écrire le fichier F.

Appelons graphe de protection la transposition graphique de toute la matrice d'autorisation. Ce graphe peut contenir des boucles, mais les arcs ne peuvent partir que des sujets. Supposons aussi que les ensembles de commandes figurant comme étiquettes sur les arcs soient des sous-ensembles d'un même ensemble fini $C = \{c_1, \ldots, c_m\}$.

Comme nous ne sommes maintenant pas concernés par le problème de renforcement de l'état de protection reflété par le graphe de protection, nous pouvons tout de suite aborder l'aspect transformation de ce graphe (i.e. modification des règles d'autorisation, création/suppression de sujets/objets). A cet effet, le modèle HRU propose les 6 opérations primitives de modification de graphe suivantes :

Opération	Interprétation graphique
insérer c dans (x,y)	
retirer c de (x,y)	
créer_sujet x	
créer_objet x	
détruire_sujet x	
détruire_objet x	

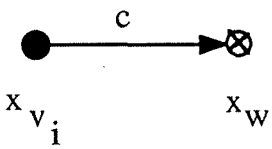
avec c $\in C$ et $\varnothing \subseteq \alpha \subseteq C$.

A partir de ces 6 opérations primitives, on peut créer un nombre arbitraire mais fini de règles de ré-écriture du graphe de protection selon le schéma suivant :

Nom_de_règle $(x_1, x_2, \ldots, x_s)$:

si $c_1 \wedge c_2 \wedge \ldots \wedge c_t$ **alors** a_1; a_2; ...; a_u;

où chaque *condition* c_i prend la forme :



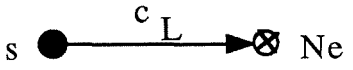
avec $1 \leq v_i, w_i \leq s$

et $c \in C$

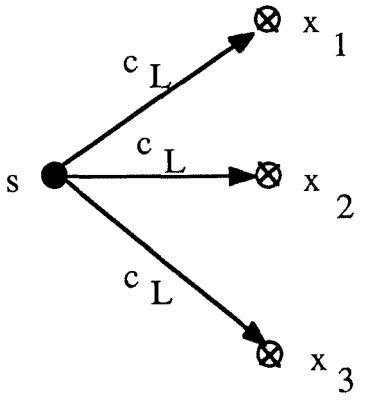
et chaque *action* a_i est une des opérations primitives ci-dessus pour manipuler des *noeuds* parmi $\{x_1, x_2, \ldots, x_s\}$.

5.3.3. Expression du système de sécurité de NOMADE en termes du modèle HRU

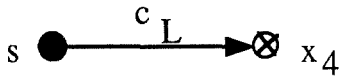
Ecrivons les commandes NOMADE qui modifient les droits usagers en termes du modèle HRU. Malheureusement, cela ne va pas être possible sans adaptation : en effet, dans NOMADE, les objets manipulables sont désignés en intention (expressions-NOMADE) et non pas par énumération. Ainsi, si Ne est une expression-NOMADE désignant (à l'instant t) les objets x_1, x_2 et x_3, et si on trouve "L : Ne" dans la clause rights effective de l'usager s, alors (en notant C_L l'ensemble des commandes NOMADE regroupées par l'étiquette L dans la clause rights effective d'un usager) :



devra être interprété (à l'instant t) comme :



La précision "à l'instant t" est importante, car si en t+Δt est créé l'objet x_4 satisfaisant Ne, alors cette création provoque implicitement l'ajout de l'arc :



De même, la destruction en t+2Δt de l'objet x_4 provoque implicitement la suppression de l'arc correspondant.

Ces considérations rendent extrêmement difficiles (sinon impossibles) l'écriture des commandes NOMADE de création (CrUser pour les usagers; Create pour les autres objets), car elles ne se limitent pas (comme attendu a priori) à des règles sans condition et à une seule action ! Ainsi :

Règle_CrUser (x) :

si vrai alors ⇒ ● x ;

est insuffisant car les droits propres et effectifs de x sont initialisés aux droits effectifs de sa partition P, c'est-à-dire (en considérant P comme un sujet, même s'il ne sera jamais actif) tout arc émanant de P doit être copié vers x !

De même :

Règle_Create (x) :

si vrai alors $\Rightarrow$ ○ x ;

est totalement insuffisant, car il faudrait balayer toutes les clauses rights de tous les usagers et de toutes les partitions d'usagers afin de créer les arcs arrivant en x !

Cependant (soient DestrUser et Destroy les commandes de suppression d'un usager et objet respectivement) :

Règle_DestrUser (x) :

si vrai alors x ● $\Rightarrow$;

et :

Règle_Destroy (x) :

si vrai alors x ○ $\Rightarrow$;

ne posent pas de problèmes, car on suppose que les deux opérations primitives utilisées détruisent aussi tous les arcs partant de x et arrivant en x.

En ce qui concerne les commandes de mise à jour proprement dite du graphe de protection, on a:

Règle_ChgRight (U' , U , Ne_1 , Ne_2) :

si U' ●$\xrightarrow{\{ChgRight\}}$⊗ U.manrights **et** U' ●$\xrightarrow{c_L}$⊗ $\Delta (Ne_1 , Ne_2)$

alors U ●$\xrightarrow{\alpha}$⊗ $Ne_1 \setminus Ne_2$ $\Rightarrow$ U ●$\xrightarrow{\alpha \setminus c_L}$⊗ $Ne_1 \setminus Ne_2$

U ●$\xrightarrow{\beta}$⊗ $Ne_2 \setminus Ne_1$ $\Rightarrow$ U ●$\xrightarrow{\beta \cup c_L}$⊗ $Ne_2 \setminus Ne_1$

5.3.4. Théorèmes de sécurité

Revenons au modèle HRU :

Intuitivement, on dit qu'un état de protection est sûr (**safe**) pour un droit donné si ce droit n'y figure pas et s'il ne peut y apparaître.

De manière plus formelle, soient un graphe de protection initial G_i, un ensemble fini de règles de ré-écriture R, une commande c ∈ C, un sujet s et un objet/sujet o. On définit le prédicat **safe (s , o , c , G_i , R)** comme prenant la valeur vrai si, partant de G_i (tel que c n'y figure pas dans l'ensemble des commandes attachées à l'arc éventuel de s vers o) il n'y a pas moyen d'obtenir G_f en utilisant des règles de R, tel que c figure dans l'ensemble des commandes attachées à l'arc de s vers o, et faux sinon.

Théorème 1 :

safe (s , o , c , G_i , R) est un prédicat indécidable. ♦

Il en est de même pour les prédicats **can_share (s , o , c , G_i , R)** et **can_steal (s , o , c , G_i , R)** définis par [Snyder 81], le premier supposant une coopération totale de la part des sujets pouvant déjà manipuler o avec c, le second excluant cette coopération (ainsi on ne peut voler quelque chose qu'on a déjà, ni quelque chose qui a été volontairement donné, même indirectement).

Ces résultats théoriques sont assez déconcertants, car nous ne pouvons pas démontrer (par le biais d'un algorithme) qu'un état de protection donné est effectivement à 100 % sûr. Cependant, le cadre formel du modèle HRU étant très général, il se peut que pour un système de sécurité particulier, les prédicats mentionnés ci-dessus soient quand même décidables. Ainsi, [Snyder 81] a montré que le modèle "Take/Grant" était tel que ces prédicats sont décidables !

Conscients de la trop grande généralité de leur modèle, Harrison, Ruzzo et Ullman ont trouvé des théorèmes un peu plus positifs :

Théorème 2 :

safe (s , o , c , G_i , R) est un prédicat décidable, si pour toute règle de R, il y a une et une seule action (i.e. u = 1). ♦

Théorème 3 :

safe (s , o , c , G_i , R) est un prédicat décidable, si pour toute règle de R, il y a une et une seule condition (i.e. t = 1) et si les trois primitives : retirer, détruire_sujet et détruire_objet ne figurent dans aucune action. ♦

5.3.5. Application des théorèmes de sécurité à NOMADE

Notre système de sécurité ne peut satisfaire au théorème 2 qu'à condition de :

- supprimer ChgRight;
- considérer comme opérations primitives les créations /destructions d'arcs vers des expressions-NOMADE;
- considérer comme opérations primitives les créations d'objets/sujets, telles que décrites ci-dessus.

D'autre part, le théorème 3 est aussi trop restrictif, car il exige non seulement la suppression de ChgRight (ce qui en somme n'est pas très grave), mais aussi celle de Destroy, DestrUser et DelRight !

Le modèle HRU ne permet dès lors malheureusement pas de répondre à des questions concernant la sécurité effective offerte par le système de protection de NOMADE.

PARTIE III

EXTENSION DU META-MODELE DE L'ATELIER LOGICIEL ALMA AUX ASPECTS DYNAMIQUES

(Atelier Logiciel sur Machine Abstraite)

1. INTRODUCTION

1.1. Etude bibliographique

Classiquement, les modèles conceptuels de représentation des données ne tiennent compte que des **aspects statiques** du réel perçu. On se borne à décrire des types d'entité, relations et attributs, c'est-à-dire uniquement les liens structurels entre les objets. Les inconvénients d'une telle modélisation ont été reconnus dès le milieu des années 70, parce que, par exemple, les vérifications des contraintes d'intégrité doivent être programmées à la main, car elles sont inconnues aux SGD.

Entretemps, de nombreuses recherches ont été menées pour incorporer des **aspects dynamiques**, comme le suggère l'**étude bibliographique** suivante :

Dans le contexte des BD relationnelles, [Smith 77] introduit le concept de *type générique* pour définir les mécanismes d'agrégation et de généralisation. Ce type générique est formalisé par des *invariants relationnels* , qui sont des propriétés des relations qui doivent rester invariantes. Pour chaque opération de mise à jour, une table indique les correspondances entre invariant relationnel, *condition de violation* et *action de correction* . Lors d'une opération de mise à jour, un mécanisme automatique détecte les invariants relationnels qui ne sont plus respectés et déclenche les actions de correction associées pour rétablir les invariants.

La méthodologie **ACM/PCM** ([Brodie 81, 82]) préconise la prise en compte explicite des propriétés dynamiques (de comportement) afin de simplifier la conception d'une BD. Par propriété dynamique d'un objet, on entend une *action* effectuant une seule mise à jour dans la BD, des pré- et post-conditions garantissant le respect des contraintes d'intégrité. Ces actions servent alors de base à l'écriture de *transactions* , qui deviennent ainsi des unités d'intégrité sémantique (plutôt que physique uniquement). La notion d'*événement* , bien que présente, ne joue pas un rôle "déclencheur" d'actions, car la vérification dynamique des contraintes d'intégrité est implémentée dans les pré- et post-conditions, plutôt qu'en tant qu'effet de bord d'une mise à jour.

Le SGBDR **RUBIS** ([Lingat 86]) étend le modèle relationnel avec les notions d'*événements* et *opérations* pour mieux représenter la sémantique des données, maintenir leur intégrité et assurer une gestion sûre et efficace des applications.

Du côté des ateliers logiciels, le projet **PMDB** ([Penedo 85, 87]) suggère la nécessité des *composants actifs* qui "surveillent" les données et déclenchent des actions si une contrainte ou règle est violée. On y craint cependant que de tels mécanismes dégradent les performances du SGD auquel ils sont ajoutés, et l'espoir semble venir de l'intelligence artificielle.

Le projet **SKB** ([Meyer 85a]) propose une expression mathématique de contraintes que doit vérifier la BD logicielle pour être cohérente. A toute *contrainte* est associé un *démon* dont la mission est de détecter des violations de cette contrainte et de déclencher une *action* associée à cette contrainte.

Les environnements **ADELE** et **NOMADE** ([Belkhatir 87]) introduisent les concepts d'*événement* et *action* pour contrôler essentiellement les effets de bord d'une mise à jour. Il y est reconnu l'importance primordiale de distinguer le moment de génération d'un événement et son moment de prise en charge.

La notion d'*assistant "intelligent"* ([Feiler 87]) pilotant les utilisateurs d'un atelier logiciel (selon leur niveau d'expertise) à travers les différentes phases du cycle de vie d'un logiciel est récemment venue enrichir le problème. En effet, un tel assistant (dirigé par une *base de connaissances* des processus de développement de logiciels) se grefferait au-dessus des outils actifs travaillant sur la BD projet et utiliserait alors des mécanismes d'événements et d'actions.

D'autres disciplines s'intéressent aussi à la modélisation d'aspects dynamiques, ne citons que l'attachement procédural dans les "frames" (*intelligence artificielle*), les "action routines" en *édition syntaxique* (GANDALF), etc.

1.2. Fil conducteur

Cette troisième partie sera consacrée à l'élaboration d'une proposition d'extension du méta-modèle de l'atelier logiciel **ALMA** ([van Lamsweerde 86, 88b]), afin d'y incorporer les aspects dynamiques du cycle de vie d'un logiciel. Le lecteur familier avec ALMA pourra donc omettre la section 2 qui décrit brièvement les concepts fondamentaux d'ALMA qui seront indispensables à la compréhension de la suite. Dans la section 3, nous ferons quelques réflexions préliminaires pour bien cerner l'étendue du problème et la philosophie à respecter par la solution développée. La section 4 proposera une taxonomie détaillée des événements retenus, ainsi que l'extension correspondante du méta-modèle. La section 5 consistera en une proposition d'architecture du moteur événements/actions qui doit être greffé sur l'atelier actuel. En conclusion (section 6), nous indiquerons comment poursuivre ce travail, car il ne s'agit ici que de directives de recherche.

Remarquons tout de suite que celle-ci est une sorte de compromis et synthèse des approches de RUBIS et NOMADE, sans en être pourtant l'adaptation exacte sous ALMA, car nous avons tenté de les généraliser partout où cela était possible.

2. PRESENTATION SUCCINCTE DE L'ATELIER LOGICIEL ALMA

Remarque :

La section 2.1. ci-dessous est essentiellement une traduction du début de [van Lamsweerde 85].

2.1. Présentation

Les ateliers logiciels visent à fournir un ensemble intégré et cohérent d'outils pour automatiser les nombreux aspects non créatifs des différentes tâches effectuées pendant le cycle de vie d'un projet logiciel développé en de multiples versions par de multiples personnes.

Le besoin d'articuler des outils autour d'une *base centrale d'informations* relatives au projet est maintenant largement reconnue. L'idée consiste à modéliser tous les types d'information pertinents pour un modèle de cycle de vie (MCV) particulier en termes d'objets et de relations entre ces objets, et à maintenir cette information d'une manière cohérente et "traçable" dans une base de données d'atelier logiciel.

D'autre part, il a aussi été reconnu que certains produits logiciels spécifiques sont des objets *structurés* écrits dans des formalismes variés. Les outils qui manipulent de tels objets sont dirigés par la syntaxe et articulés autour de leur représentation interne unique. Le plus souvent, cette représentation prend la forme d'un *arbre syntaxique abstrait* .

Ces deux types de tendances ont jusqu'à présent évolué sans beaucoup d'interpénétration. Le noyau d'atelier logiciel qui a été conçu dans le contexte du projet ALMA les intègre parce qu'il fournit essentiellement les deux fonctionnalités suivantes :

- support intégré à la gestion d'une documentation de projet cohérente, complète, "traçable" et enregistrée dans une base de données de projet (PDB - Project Database);

- support dirigé par la syntaxe à la manipulation de textes trouvés dans cette documentation et écrits dans des formalismes variés.

Les principales caractéristiques de ce noyau sont les suivantes :

2.1.1. Stratégie d'intégration

Une base de données de projet (*Project Database* - PDB) est l'unique moyen pour intégrer des outils autonomes qui autrement ne se connaissent pas. Elle a une structure entité/relation avec les composantes suivantes :

(i) *Objets logiciels* de types divers à prédéfinir. Par exemple, les objets FONCTION ou DONNEES pourraient être trouvés dans une phase d'analyse fonctionnelle; une architecture logicielle pourrait être constituée d'objets MODULE. Un modèle de cycle de vie plus complet devrait aussi comprendre des types d'objet comme PROGRAMME SOURCE, FICHIER, JEU DE TEST, PROGRAMMEUR, RESSOURCE, etc.

(ii) *Relations logicielles n-aires* définies sur des types d'objet. Par exemple, une relation binaire DERIVE DE pourrait être définie sur les types d'objet FONCTION et MODULE. Des MCV plus riches devraient comprendre d'autres relations comme par exemple une relation IMPLEMENTE PAR définie dur les types d'objet MODULE et PROGRAMME SOURCE, une relation réflexive VERSION DE définie sur des types d'objet tels que FONCTION, MODULE ou PROGRAMME SOURCE, une relation ternaire ASSIGNE A définie sur les types d'objet MODULE, PROGRAMMEUR et CONTRAT, une relation TESTE définie sur les types d'objet PROGRAMME SOURCE et JEU DE TEST, etc.

(iii) *Propriétés logicielles* caractérisant les objets logiciels *et/ou* les relations logicielles. Par exemple, les types d'objet MODULE ou PROGRAMME SOURCE devraient avoir des propriétés telles que leur NOM, DATE de création/mise à jour, STATUT, ou DECISIONS DE CONCEPTION prises; une propriété ECHEANCE pourrait être attachée à la relation ternaire ASSIGNE A mentionnée ci-dessus, etc.

Une caractéristique importante qui rend ALMA unique vis-à-vis d'autres systèmes de support au cycle de vie (comme par exemple ISDOS, SODOS ou PMDB) est que *des propriétés peuvent avoir des textes formels comme valeurs* . Ceci permet d'attacher des propriétés du genre SPECIFICATION FORMELLE, PSEUDOCODE, ou EXPLICATION FORMELLE au type d'objet MODULE. De même, une propriété CODE DE PROTOTYPE pourrait être attachée au type d'objet FONCTION, la propriété CODE SOURCE devrait apparaître dans la liste des propriétés caractérisant des objets PROGRAMME SOURCE, etc. N'importe quel texte formel peut ainsi être attaché à une instance d'objet logiciel ou relation et sera *manipulé moyennant une connaissance de sa syntaxe* .

2.1.2. Couverture du cycle de vie complet

Comme les exemples ci-dessus devraient le suggérer, les divers objets logiciels, relations et textes formels associés qui sont manipulés dans le noyau peuvent se référer à toutes les étapes du cycle de vie d'un logiciel.

2.1.3. Paramétrisation sur les modèles de cycle de vie des logiciels et les langages

Le génie logiciel n'a pas encore atteint l'état d'une discipline scientifique et mûre dans laquelle un ensemble standard de concepts, modèles, langages, théories et méthodes ont émergé. Par exemple, les adeptes des techniques de spécification par flux de données (du style SADT ou Structured Analysis) ne forceront pas les adeptes de techniques plus formelles de spécification de besoins à changer leurs habitudes, et vice-versa. D'autre part, une compagnie peut souhaiter avoir les informations de gestion de projet structurées dans la PDB selon ses propres standards spécifiques, standards qui se révèlent être très différents de ceux d'autres compagnies. Pour certains domaines d'application, ils pourraient ne pas être concernés par des informations sur les jeux de test, etc. Ainsi, un noyau d'atelier logiciel doit s'adapter aux modèles de cycle de vie des logiciels, méthodes et langages propres aux utilisateurs, plutôt que de leur imposer encore une autre méthodologie. Le noyau ALMA est *automatiquement adaptable aux types d'objet logiciels, relations, propriétés et formalismes qui sont spécifiques à un MCV particulier* . Ce but est atteint par une architecture en deux couches :

Au niveau ***instancié*** , un atelier logiciel est associé avec un MCV spécifique qui doit être méta-défini au niveau instanciant. Un atelier instancié se base sur un schéma spécifique de base de données projet correspondant à ce MCV particulier; il est constitué d'outils génériques tels que, par exemple, un outil de mise à jour de la PDB, un système de requête de la PDB, des éditeurs structurels et des analyseurs. Ces outils génériques sont automatiquement instanciés au MCV approprié parce qu'ils sont dirigés par une base de données du modèle (*Model Database* - MDB) qui contient la méta-description des divers types d'objet, relations, propriétés et formalismes trouvés dans ce modèle.

Au niveau ***instanciant*** , le méta-système génère toutes les informations nécessaires à la production automatique d'ateliers instanciés. Il accepte deux types de méta-définitions : une méta-définition, en termes d'un méta-modèle entité/relation étendu, des types d'objet logiciels, relations et propriétés trouvés dans un MCV particulier, et des méta-définitions des divers formalismes trouvés dans ce modèle. Le méta-système génère le contenu de la MDB qui dirige les outils génériques du niveau instancié. D'autres méta-informations sont aussi générées, comme par exemple des méta-définitions des langages utilisés pour mettre à jour ou interroger la PDB; celles-ci sont nécessaires pour générer à leur tour des éditeurs structurels instanciés à ces langages.

2.1.4. Utilisation systématique d'interfaces abstraites

Nous avons introduit une série d'interfaces abstraites dans notre architecture afin que les outils soient raisonnablement indépendants de leur environnement. De plus, ces interfaces abstraites permettent aux outils d'être implémentés en termes de leurs propres abstractions. Une *interface entité/relation* fournit un ensemble de primitives pour l'accès uniforme à la PDB et à la MDB en termes de concepts entité/relation seulement. Le SGBD relationnel utilisé est ainsi caché et l'implémentation de tous les outils manipulant la PDB en est rendue indépendante. Une interface appelée *META interface* fournit un ensemble de primitives à utiliser dans les ateliers instanciés par les outils génériques pour l'accès uniforme aux tables qui les dirigent. L'organisation complexe des méta-informations est ainsi cachée. La *"Abstract Workstation Interface"* fournit un ensemble de primitives pour l'interaction uniforme avec les usagers. Les particularités du terminal, des mécanismes d'affichage/acquisition, et les protocoles de dialogue choisis sont ainsi cachés et l'implémentation de tous les outils interactifs en est rendue indépendante.

ALMA était un projet de recherche universitaire et industriel sous la tutelle du Ministère Belge des Affaires Scientifiques. Il représentait un effort de trente années-homme. Le noyau est écrit en C et tourne sur un VAX 750 sous UNIX avec RTI INGRES comme SGBDR.

2.2. Absence d'aspects dynamiques

La principale faiblesse d'ALMA réside dans le fait que seuls les composants *statiques* d'un modèle de cycle de vie peuvent être représentés. Il est ainsi impossible de modéliser les aspects *dynamiques* , c'est-à-dire des actions qui doivent être déclenchées à la survenance d'événements. La mission de ces actions serait par exemple d'analyser, évaluer, contrôler, voire générer des objets logiciels et les relations y associées.

3. REFLEXIONS PRELIMINAIRES

La description précédente de l'atelier ALMA montre clairement la philosophie à respecter : la solution proposée doit avoir un très haut degré de paramétrabilité, afin de permettre aux utilisateurs de définir leurs propres stratégies. Ainsi, il faut qu'idéalement toute composante actuelle d'ALMA qui soit supprimée ou modifiée puisse être instanciée sous la nouvelle version d'ALMA.

Conformément à l'état de l'art actuel dans le domaine, nous allons séparer les notions d'*événement* et d'*action* , qui seront les concepts clés dans le développement ci-après. Les définitions des différents événements et actions font partie de la définition du modèle de cycle de vie dans lequel ces événements et actions prennent place; ces définitions doivent dès lors figurer dans la MDB. Il en découle la nécessité d'une extension du **méta-modèle ALMA**. Nous ajouterons à sa partie statique (sémantique et syntaxe) une **partie dynamique** qui sera donc, en première approximation, la suivante :

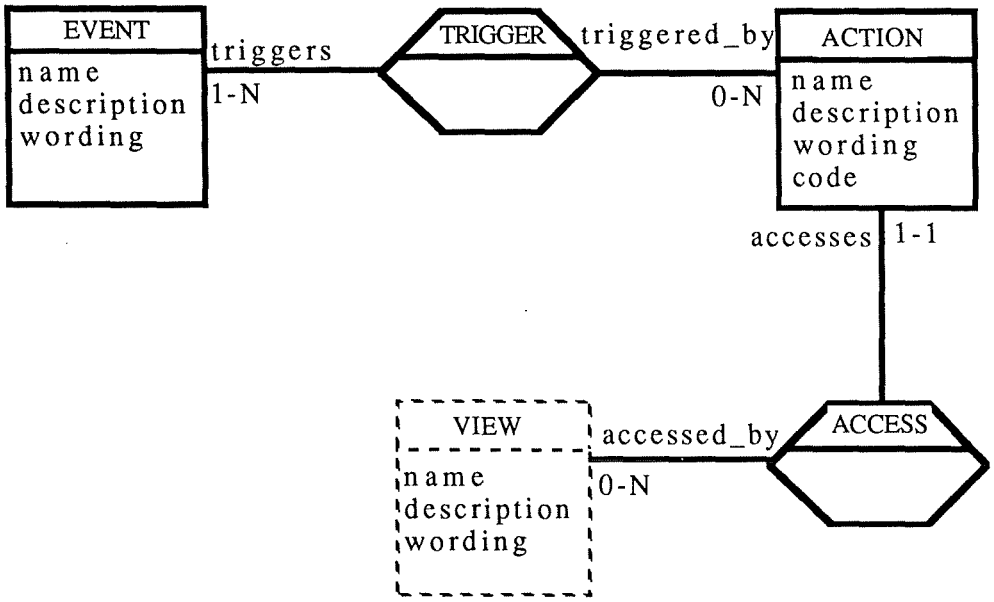


Figure 3.1. - Première approximation de la partie dynamique du méta-modèle ALMA (le méta-type VIEW appartient déjà au méta-modèle existant)

Remarquons que les méta-attributs *name* , *description* et *wording* apparaissent dans la plupart des méta-types. *Name* sera un nom abrégé pour l'usage courant, alors que *wording* prendra pour valeur un nom plus parlant qui sera utilisé dans des rapports par exemple. *Description* contiendra un texte libre définissant le concept en question pour des besoins de documentation du modèle de cycle de vie.

On peut aussi envisager un *partitionnement de la MDB* en deux partitions, une pour la partie statique, une pour la partie dynamique du méta-modèle. Ce partitionnement ne peut l'être au sens mathématique, car les deux partitions se recouvrent partiellement. L'avantage de cette séparation est que le méta-système peut être subdivisé lui aussi en deux composants. Ceci empêche notamment une reconstruction de la PDB au cas où, par exemple, seulement le code d'une action a été changé.

Bien entendu, il faut aussi ajouter au noyau d'ALMA des mécanismes prenant en charge des occurrences de EVENT pour déclencher des occurrences de ACTION. Cette nouvelle composante sera appelée **moteur événements/actions**; son architecture et flux de données seront grosso-modo les suivants :

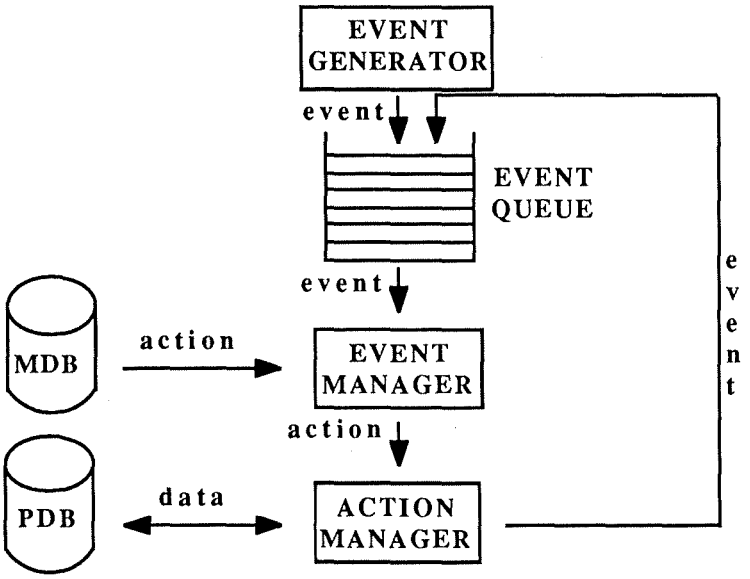


Figure 3.2. - Première approximation du moteur événements/actions

D'autre part, un enrichissement en primitives offertes par ALMA sera nécessaire pour permettre la programmation des stratégies relatives à la dynamique. Il est clair aussi que tout doit s'insérer dans un milieu multi-utilisateurs.

4. TAXONOMIE DES EVENEMENTS

Dans ce chapitre, nous allons décrire les quatre *types* d'événement que nous avons retenus, à savoir :

- les événements relatifs à l'exécution de commandes (**CommandEvent : CE**);
- les événements temporels (**TemporalEvent : TE**);
- les événements relatifs aux effets de bord d'une commande (**SideEffect : SE**);
- les événements relatifs aux violations de contraintes suite à une mise à jour de la PDB (**ConstraintViolation : CV**).

Pour chaque type, nous donnerons l'extension correspondante du méta-modèle (les méta-types d'entité en pointillé existant déjà dans la partie statique) et nous indiquerons les *moments de prise en charge* possibles.

4.1. Evénements relatifs à l'exécution de commandes

Par souci d'homogénéité, nous considérons comme un événement l'émission d'une commande C portant sur un objet O, et ce à la date D par l'agent A (la notion d'agent sera explicitée ci-dessous). Nous proposerons à la section 4.3 une notation particulière pour de tels événements, notation qui sera en parfaite harmonie avec les trois autres types d'événement.

L'objet O peut en toute généralité être une occurrence d'un type d'entité, une valeur d'attribut ou une occurrence de relation. Une même commande peut porter sur des objets de types différents, mais déclencher des actions différentes. D'où la méta-relation ternaire TRIGGER. Notons qu'on aurait aussi pu en faire une méta-relation binaire, et ce par le biais d'un méta-attribut *object* dans COMMAND-EVENT. Ceci nécessiterait évidemment une contrainte d'intégrité pour vérifier s'il existe dans le modèle un concept ayant pour nom la valeur d'*object*.

Le moment de prise en charge d'un tel événement sera toujours (et donc implicitement) **Immediate**, c'est-à-dire qu'il sera pris en charge aussitôt que généré.

Quelques précisions quant au méta-attribut *code* de ACTION s'imposent : de type FormalText, il prend sa valeur (appelée *programme d'action*) dans un langage de programmation dont les caractéristiques restent à définir (composition séquentielle, alternative, itérative, ... ; primitives; etc). Les *primitives* sont câblées et en nombre fixé à la livraison. Certaines seront introduites dans les sections suivantes : celles qui sont relatives aux notions d'événements et actions.

Dans la suite, nous distinguerons les primitives et les *commandes*. Ces dernières sont les seules disponibles à l'usager et correspondent à l'invocation d'une action (que nous considérons synonyme avec outil). Elles sont aussi les seules à donner lieu à des événements de type CE. Les programmes d'action d'une action sont donc implémentés en termes des primitives et d'invocations d'autres actions.

Nous devrons donc distinguer les commandes émises par les usagers (UCE : User-issued CommandEvent) et celles émises lors de l'exécution d'un programme d'action (ACE : Action-issued CommandEvent). Ainsi, la notion d'agent regroupe celles d'usager et d'action.

Dans un programme d'action, il y a un **BeginTransaction** implicite avant le premier accès à la PDB. Implicitement, la dernière instruction d'un programme d'action est **EndOfTransaction**, dont l'interprétation (**CommitTransaction** ou **AbortTransaction**) sera élucidée à la section 4.3.2.2.

Voici donc l'extension correspondante du méta-modèle :

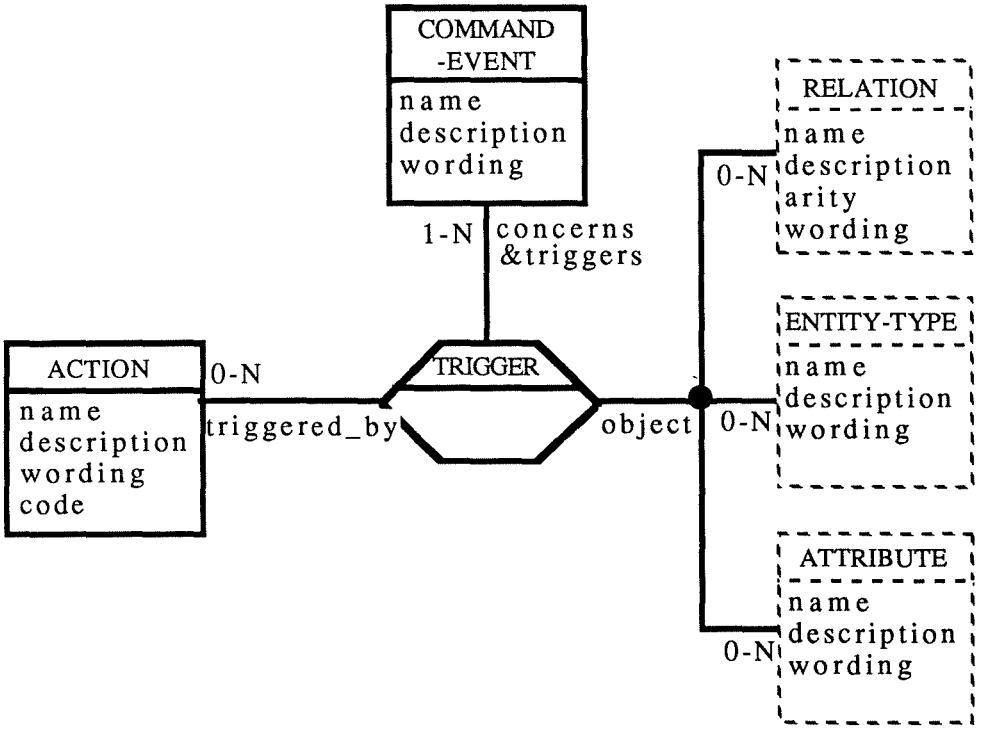


Figure 4.1. - Evénements relatifs à l'exécution de commandes

D'autre part, la philosophie ALMA veut que tout outil (donc toute action) accédant à la PDB soit attaché à au moins une **vue**. D'où :

Figure 4.2. - Une action accède à une vue

Finalement, nous laissons ouverte la question du mode d'exécution d'un programme d'action : soit il sera interprété par le moteur en tant que code source, soit le méta-système le compile une fois pour toutes.

4.2. Evénements temporels

Un événement temporel est caractérisé par son moment de survenance D (une notation pour ce type d'événement sera proposée à la section 4.3.). Il appartient au moteur de générer un tel événement chaque fois que D est atteint. Un formalisme précis de définition de D reste à définir (point temporel, intervalle de temps, durée, périodicité, ...) pour typer le méta-attribut *time* . Voilà donc l'extension proposée du méta-modèle :

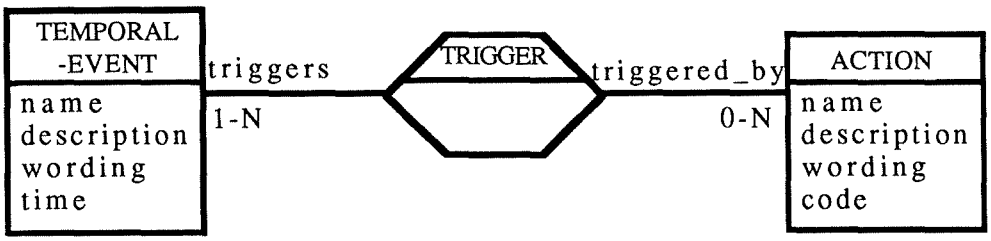
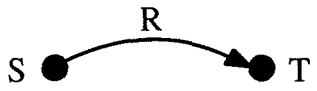


Figure 4.3. - Evénements temporels

Ici encore, le moment de prise en charge **Immediate** est implicite.

4.3. Evénements relatifs aux effets de bord d'une commande (SE)

Soit :



Si T est modifié suite à la commande C émise par l'agent A à la date D, alors S est *susceptible* de devoir subir aussi une modification. On appelle cela un effet de bord sur S et l'événement en question se note SE(S,R,T,C,A,D).

Afin d'assurer un maximum d'homogénéité dans la notation des événements, l'événement associé à l'émission par l'agent A de la commande C sur l'objet O sera noté : CE(O,C,O,**Command**,A,D,PositionalParameters) (c'est-à-dire qu'on considère qu'il a exécuté la commande **Command** sur une relation réflexive C portant sur O). De même, l'événement temporel de date D sera noté TE(**Clock**,**Time**,**Clock**,**Command**,**System**,D) (c'est-à-dire qu'on considère que le système a exécuté la commande **Command** sur une relation réflexive fictive **Time** portant sur l'objet fictif **Clock**).

Tous ces paramètres réels sont passés au programme d'action associé, dans lequel on peut utiliser les paramètres formels SOURCE, RELATION, TARGET, COMMAND, AGENT et DATE.

Mais la génération d'événements de type SE n'est pas câblée dans la commande C, car beaucoup d'événements inutiles risqueraient alors d'être produits. Voilà pourquoi la génération des événements pertinents est à charge du programme d'action P attaché à la commande C.

Avant de montrer comment ce but est atteint, il se pose aussi la question du *moment de prise en charge* de ces événements. Afin de laisser un maximum de choix au concepteur de P, nous offrons quatre moments de prise en charge (méta-attribut *moment*) :

- **BeforeCommit** : juste avant l'**EndOfTransaction** implicite de P;
- **AfterCommit** : juste après l'**EndOfTransaction** implicite de P, et à condition que celui-ci soit interprété (cf. section 4.3.2.2.) comme un **CommitTransaction**;
- **Access** : au prochain accès à S par un agent quelconque;
- **Demand** : à la demande d'un agent quelconque.

4.3.1. Effets de bord de moments Access et Demand

Ces moments de prise en charge nécessitent l'enregistrement dans S de l'événement. Voilà pourquoi l'attribut *LsRecEvents* doit être ajouté systématiquement par le méta-système à chaque occurrence de ENTITY-TYPE lors du remplissage de la MDB.

La primitive **RecordEvent** (S,R,T,C,A,D) placée dans P permet d'enregistrer l'événement SE(S,R,T,C,A,D) dans (S.LsRecEvents). Le "réveil" de cet événement est maintenant à charge des agents : ce sera soit une demande explicite par la primitive **Make** (S), soit au prochain accès à S par une primitive autre que **Make**. Le programmeur de P ignore donc le moment d'activation de l'événement enregistré. De plus, le programme d'action déclenché par le réveil d'un événement enregistré peut être différent selon qu'il l'est à la demande ou sur accès.

4.3.2. Effets de bord de moments BeforeCommit et AfterCommit

Ces moments peuvent être explicitement programmés dans P par la primitive : **PropagateEvent**(S,R,T,C,A,D,**BeforeCommit** | **AfterCommit**). Il s'en suit une mise en file d'attente de l'événement propagé. La question est de savoir s'il faut ou non imposer un certain ordre dans cette file d'attente pour tous les effets de bord propagés par le même programme d'action ?

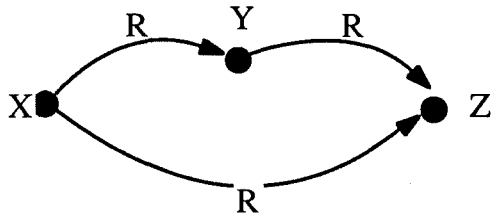
4.3.2.1. Ordonnancement des effets de bord

Si un tel ordre doit être respecté, alors il doit évidemment être assuré lors de l'insertion d'un événement dans la file d'attente, plutôt que d'imposer au programme d'action d'effectuer les **PropagateEvent** dans cet ordre. Une analyse plus fine du problème a révélé qu'il se subdivise en deux sous-questions :

- comment ordonnancer deux effets de bord dont les relations sont différentes ? ;
- comment ordonnancer deux effets de bord dont les relations sont identiques, mais les sources différentes ?

Quant à la *première question* , [Belkhatir 87] a donné un exemple où il faut d'abord gérer les effets de bord relatifs à une certaine relation, avant de passer à ceux d'une autre relation. Il s'agit en l'occurrence d'une politique de recompilation de configurations. Face à ce besoin réel, une solution naturelle consiste à affecter des **priorités** (nouveau méta-attribut *precedence* pour RELATION) à toutes les relations susceptibles d'être porteuses d'effets de bord. L'ordonnancement des effets de bord propagés par la même action doit donc se faire par priorités décroissantes.

La réponse à la *seconde question* est malheureusement plus laborieuse. Un petit exemple est nécessaire à son explication. Ainsi, supposons :

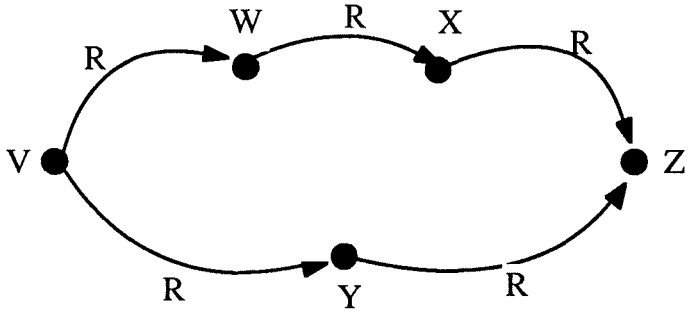


et qu'une manipulation de Z propage les effets de bord de moment **BeforeCommit** :
SE_1(Y,R,Z,...) associé au programme d'action P,
et SE_2(X,R,Z,...) associé au programme d'action P.

Si SE_2 est pris en charge avant SE_1 et si P (en traitant SE_1) propage l'effet de bord de moment **BeforeCommit** : SE_3(X,R,Y,...) associé au programme d'action P, alors X sera manipulé deux fois par P !

Par contre, si SE_1 est pris en charge avant SE_2, et si P (en traitant SE_1) propage l'effet de bord SE_3, alors (en raison du fonctionnement en profondeur d'abord du mécanisme de traitement des effets de bord de moment **BeforeCommit**) SE_2 se trouve en file d'attente au moment où P traite SE_3. Il suffit donc de prévoir une primitive **ResetEvents** (S,R,T,C,A,D,EventType) et que P émette : **ResetEvents** (SOURCE,RELATION,*,*,*,*,**BeforeCommit**) pour supprimer SE_2 lors du traitement de SE_3 et empêcher ainsi la double invocation de P sur X.

Comment maintenant imposer la prise en charge de SE_1 avant SE_2 ? Il faudrait effectuer une **décomposition en niveaux** (tri topologique) des noeuds de chaque graphe, ce qui nécessite une extension obligatoire du modèle de cycle de vie afin de mémoriser les niveaux affectés à chaque noeud d'un graphe. Cependant, une telle détection d'actions doubles semble extrêmement coûteuse (gestion permanente des niveaux) et elle risque de n'être utile que dans certains cas particuliers, d'autant plus que le lecteur constatera aisément à l'aide de l'exemple suivant que la solution n'est pas sans défaillance :



Bien entendu, si les actions sont de longue durée, une telle technique d'évitement est envisageable.

4.3.2.2. Transactions emboîtées

Il faut un mot d'explication pour les programmes d'action déclenchés par un effet de bord de moment **BeforeCommit**. Etant donné qu'ils sont exécutés juste avant l'**EndOfTransaction** implicite de P et que leur mission est d'évaluer si oui ou non les mises à jour effectuées par P peuvent être validées, ils doivent retourner une valeur booléenne. Si toutes les actions déclenchées par des effets de bord de moment **BeforeCommit** générés par P réussissent (retournent **true**), alors l'**EndOfTransaction** de P est interprété comme un **CommitTransaction** et les effets de bord de moment **AfterCommit** sont réveillés. Si par contre, au moins une action déclenchée par des effets de bord de moment **BeforeCommit** générés par P échoue (retourne **false**), alors P échoue et toute modification de la PDB doit être défaite; l'**EndOfTransaction** est interprété comme un **AbortTransaction**.

Pour contourner les difficultés inhérentes à la notion de transactions emboîtées ([Date 82]), il faut que seule la transaction la plus "externe" ait un **CommitTransaction** implicite. Une transaction déclenchée par un effet de bord de moment **BeforeCommit** peut donc seulement propager des effets de bord de moment **BeforeCommit**.

En synthèse, l'extension du méta-modèle sera la suivante, afin de garantir que la MDB contienne toutes les informations nécessaires au moteur événements/actions :

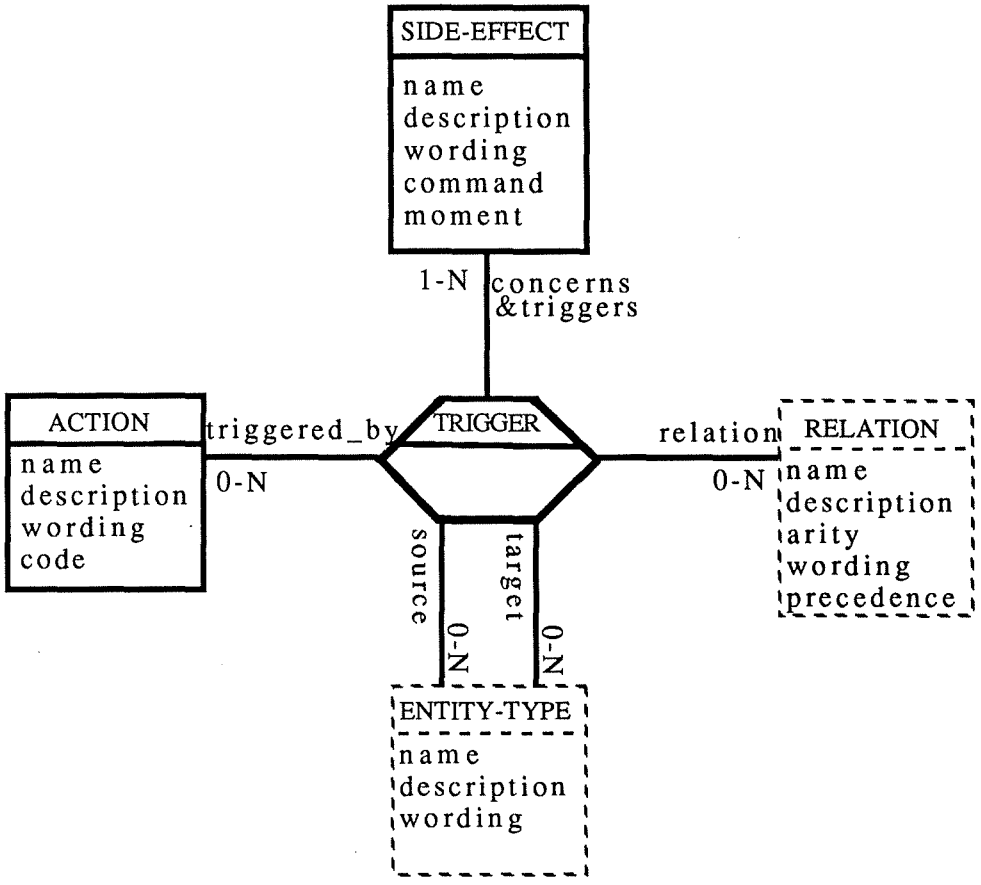


Figure 4.4. - Evénements relatifs aux effets de bord d'une commande

La méta-relation quinternaire TRIGGER modélise le lien entre les paramètres (S,R,T,C) d'un effet de bord et l'action correspondante. Au prix d'une contrainte d'intégrité, on aurait pu en faire une méta-relation binaire.

4.4. Evénements relatifs aux violations de contraintes

Ce type d'événement est une espèce de généralisation de la notion de violation de contrainte d'intégrité.

Faisons d'abord un petit rappel sur les **contraintes d'intégrité**. Comme leur nom l'indique, il s'agit de contraintes (de prédicats pour être précis) que la BD doit satisfaire à tout moment (c'est-à-dire maintenir à **true**) pour être dans un état d'intégrité. Parmi les contraintes d'intégrité, on distingue les contraintes d'intégrité classiques (qui ont reçu un nom bien spécifique) et les contraintes spécifiques à un système d'information donné. Parmi les contraintes d'intégrité classiques, on distingue encore celles qui ont trouvé une représentation explicite au niveau d'un modèle (schéma) de BD et qui sont donc automatiquement vérifiées par le SGBD, et celles qui n'ont pas de représentation (graphique) explicite et dont la vérification est à charge des programmes d'application.

Nous proposons donc de généraliser ce concept de contrainte d'intégrité pour introduire celui de contrainte tout court. En effet, suite à une violation de contrainte d'intégrité, il faut normalement **entreprendre quelque chose** pour ramener la BD dans un état intègre : pourquoi donc ne pas aussi faire quelque chose chaque fois qu'une contrainte (au sens général) est violée ? Le but d'une telle action peut alors être quelconque.

En ce qui concerne le cas particulier des contraintes d'intégrité, l'atelier ALMA permet de modéliser les contraintes relatives aux notions d'identifiant d'un type d'entité, de connectivité du rôle joué par un type d'entité dans une relation, de domaine des attributs et de répétitivité des attributs.

Par contre, des contraintes relatives à des attributs/rôles exclusifs, attributs calculés, dépendances fonctionnelles/multivaluées/de jointure, ... et des contraintes référentielles (ou contraintes d'inclusion) ne sont actuellement pas modélisables par le méta-modèle, sans parler des contraintes non classiques.

Quant à la gestion des contraintes d'intégrité, la solution actuelle consiste à :

(1) vérifier (suite à toute mise à jour de la PDB) les contraintes explicitées dans le modèle. En cas de violation d'une telle contrainte d'intégrité, il y a soit une erreur fatale (faute de domaine), soit un avertissement (pour les autres types de contraintes d'intégrité modélisées);

(2) programmer les autres contraintes d'intégrité sous forme de requêtes, qui sont alors soumises régulièrement au système de requêtes (éventuellement en mode "batch") afin de détecter les incohérences actuelles de la PDB.

L'inconvénient de la stratégie (1) réside dans son *câblage* : elle fait partie intégrante du code de l'outil de mise à jour de la PDB et n'est pas modifiable par un administrateur ALMA. D'autre part, (2) les contraintes d'intégrité non représentables dans le modèle ne peuvent être vérifiées qu'*à la demande* . Finalement, les contraintes d'intégrité ne sont pas gérées de manière homogène.

L'amélioration proposée consiste donc à stocker de manière déclarative des contraintes dans la MDB et à leur associer des actions qui seront déclenchées suite à un nouveau type d'événement : les **violations de contraintes**.

Un événement de ce type sera noté : CV (C,**Violation**,C,**Command**,**System**,D), c'est-à dire qu'on considère que le système a exécuté la commande **Command** sur une relation réflexive fictive **Violation** portant sur la contrainte C.

La mécanique sous-jacente doit être telle que la solution actuellement adoptée sous ALMA et relative au cas particulier des contraintes d'intégrité en soit une instanciation possible. Par souci de standardisation, les contraintes d'intégrité représentables au niveau d'un modèle doivent également être déclarées de cette nouvelle manière.

Précisons aussi la différence avec les effets de bord : la génération d'un effet de bord traduit une modification *potentielle* de sa source S, alors qu'une violation de contrainte représente un *fait*.

Bien entendu, nous ne pouvons pas imposer le *moment de détection* d'une violation de contrainte. Voilà pourquoi chaque contrainte est caractérisée par un moment de détection (méta-attribut *detection*) parmi:

Update : juste après une mise à jour par la primitive **Update**;

Demand : à la demande explicite d'un agent.

Cette dernière alternative nécessite alors l'introduction d'une nouvelle primitive, à savoir : **DetectViolations** (O), dont la mission est donc de générer tous les événements de type CV correspondant à une violation de contrainte portant sur O.

Cependant, si la détection d'une violation de contrainte nécessite la confrontation des états avant et après mise à jour, seul le moment **Update** est permis, car on ne peut conserver ces deux états jusqu'au prochain **DetectViolations**.

Une deuxième caractéristique d'une contrainte est le *moment souhaité de prise en charge* de l'événement généré si la contrainte est effectivement violée (méta-attribut *event-moment*). Les alternatives sont :

BeforeCommit (uniquement si detection = **Update**) : juste avant l'**EndofTransaction** implicite;

Immediate : immédiatement;

Deferred : au prochain **Make** (O) ou au prochain accès à l'objet O.

La dernière valeur nécessite l'enregistrement automatique de l'événement : l'attribut obligatoire de tout type d'entité (à savoir *LsRecEvents*) accueille aussi les événements de ce type.

La principale difficulté réside bien sûr dans l'expression obligatoirement formelle de la contrainte (méta-attribut *predicate*). Un tel formalisme en termes du modèle ERA d'ALMA reste à définir.

Nous suggérons l'extension suivante du méta-modèle :

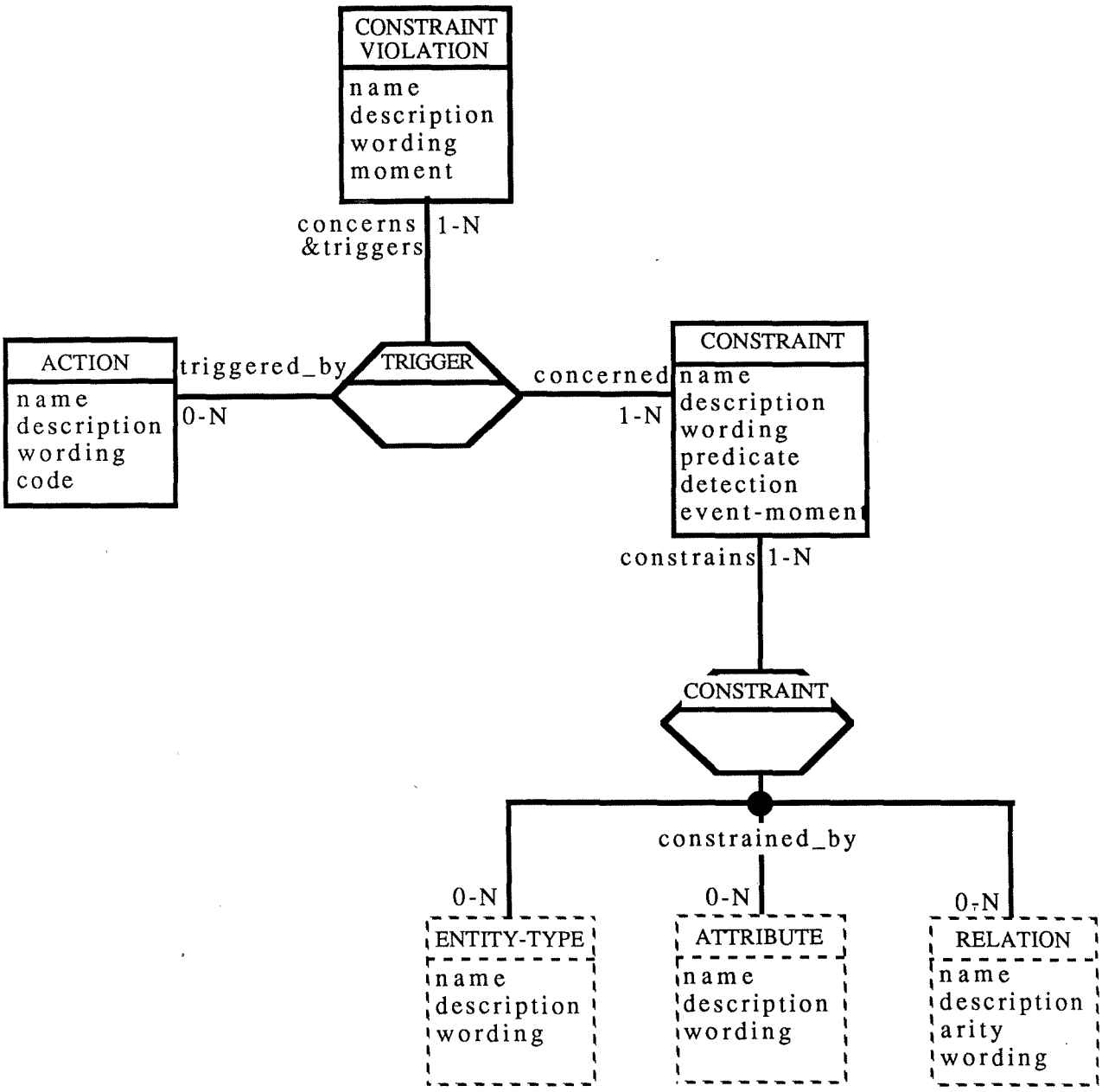


Figure 4.5. - Evénements relatifs aux violations de contraintes

4.5. Synthèse : extension du méta-modèle ALMA

Résumons en une seule figure l'extension proposée du méta-modèle d'ALMA :

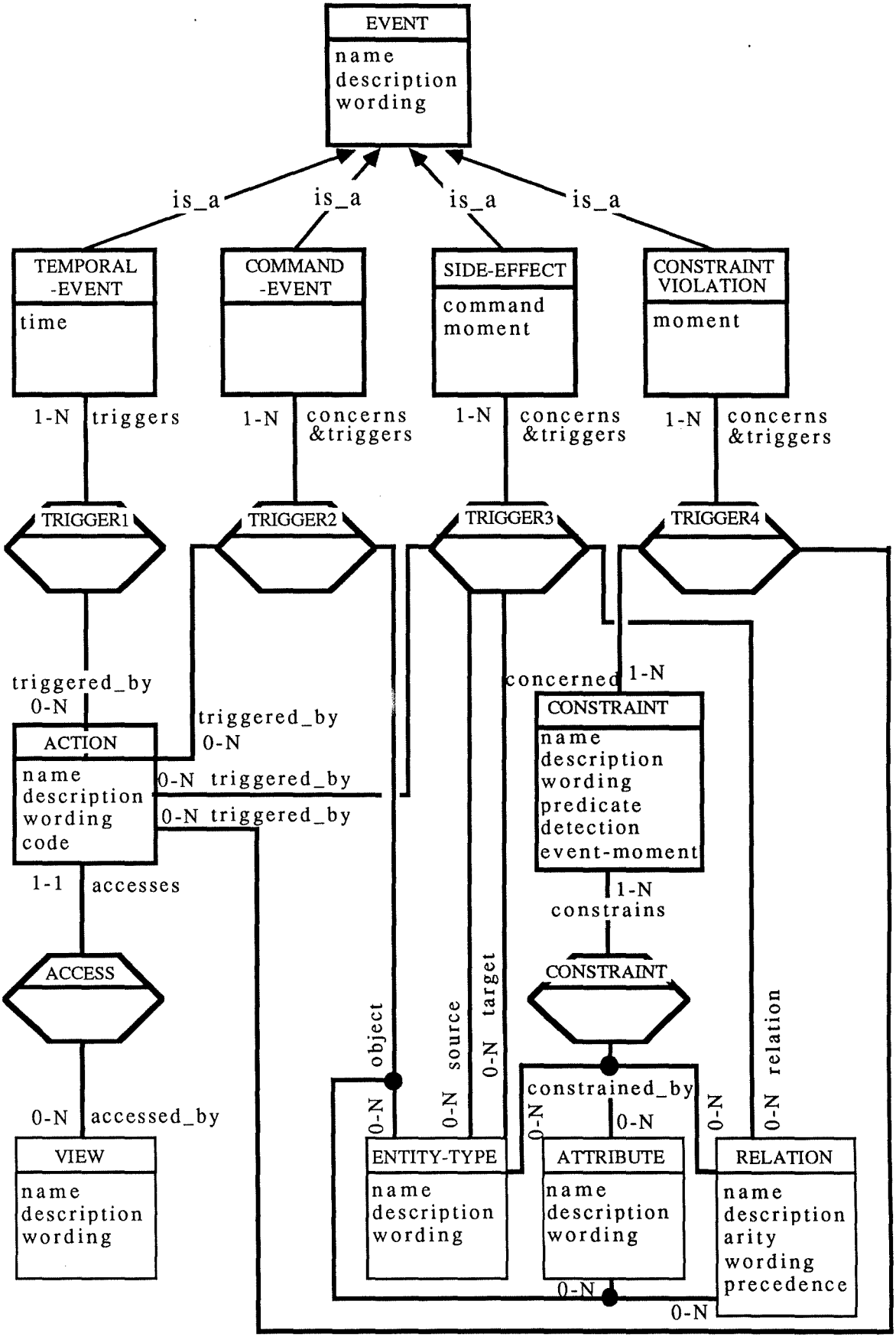


Figure 4.6. - Méta-modèle d'ALMA : partie dynamique

Le reste de cette section contient une série d'exemples représentatifs des nouvelles sections d'entité du langage de mise à jour de la MDB.

```
TEMPORAL-EVENT backup-time;
   TIME = "every friday at 18:00" ;
   TRIGGERS (action) make-backup;

COMMAND-EVENT compile;
   CONCERNS (entity type | relation | attribute) source-text
      AND TRIGGERS (action) compile-source-text;
   CONCERNS (entity type | relation | attribute) configuration
      AND TRIGGERS (action) compile-configuration;

SIDE-EFFECT inconsistent-source;
   COMMAND = update;
   MOMENT = AfterCommit;
   CONCERNS (entity type) realization AS SOURCE
      RELATED TO (entity type) interface AS TARGET
      VIA (relation) depends_on
      AND TRIGGERS (action) recompile;

CONSTRAINT-VIOLATION funct-dep-1-violation;
   MOMENT = CommitTime;
   CONCERNS (constraint) funct-dep-1
      AND TRIGGERS (action) ask-user;

CONSTRAINT funct-dep-1;
   DETECTION = update;
   EVENT-MOMENT = CommitTime;
   PREDICATE = "∀ m ∈ module : m.language → m.compiler";
   CONSTRAINS (entity type | relation | attribute) module;

ACTION recompile;
   CODE = compile SOURCE.sourcetext;
   ACCESSES (view) compiler-view;
```

5. ARCHITECTURE ET FLUX DES DONNEES DU MOTEUR EVENEMENTS/ACTIONS

Nous allons à présent donner une description détaillée du moteur événements/actions. La section 4 suggère l'architecture et le flux des données suivants (qui sont donc plus complets et réalistes que ce qui a été proposé dans la section 3) :

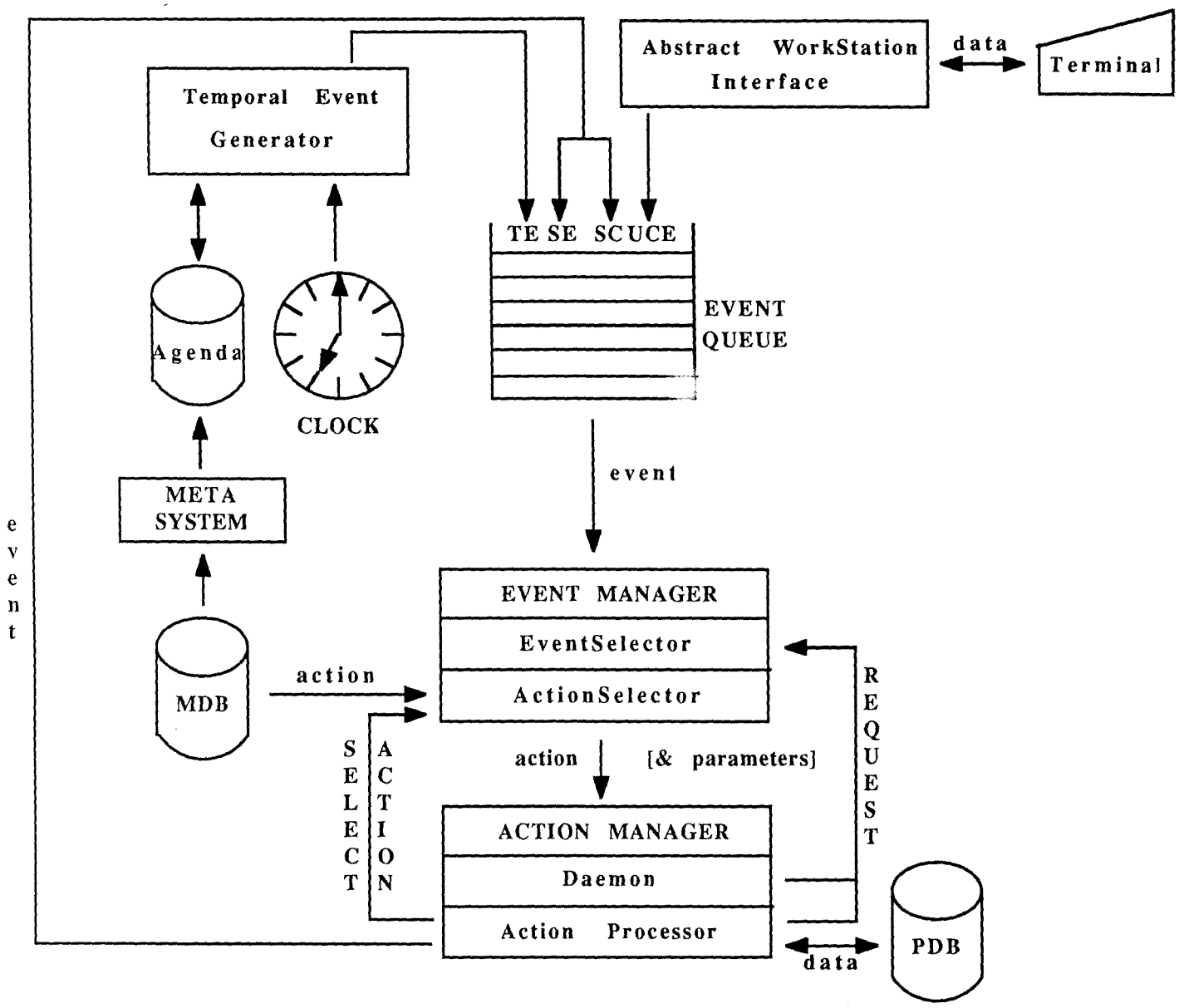


Figure 5.1. - Architecture et flux des données du moteur événements/actions

Chacune des quatre sous-sections suivantes sera consacrée à l'étude d'une de composantes actives de ce moteur, les composantes passives (structures de données) étar décrites aux endroits appropriés.

5.1. Abstract Workstation Interface (AWI)

La mission de l'AWI a été légèrement modifiée par rapport à son fonctionnement initial : il s'agit maintenant de transformer les commandes émises par les usagers en événements de type UCE et de les insérer dans la file d'attente des événements (EVENT QUEUE - EQ), dont la structure est la suivante :

```
Event ( Source,
        Relation,
        Target,
        Command,
        Agent,
        Date
        [ , PositionalParameters ] )
EventMoment
ProcessIdentifier
```

Seuls des événements de type ACE ne peuvent pas se trouver dans cette file d'attente, nous verrons pourquoi en 5.4.2.2. Nous avons aussi vu dans la section 4 que l'emploi de certaines constantes (par exemple : **Clock**, **Time**, **Command**, **System**, ...) permet de distinguer des événements de types différents.

5.2. Temporal Event Generator (TEG)

Comme son nom l'indique, le TEG a pour but de mettre dans la file d'attente des événements de type TE, et ce aux moments de leurs survenances. Pour réaliser cette tâche, il lui faut non seulement l'accès à l'horloge système (CLOCK), mais aussi un fichier (dénommé AGENDA ci-après) contenant dans l'ordre chronologique les dates de survenance des événements temporels à venir. La génération de l'AGENDA est une nouvelle mission du META-SYSTEM, qui consulte les sections d'entité relatives au type d'entité TEMPORAL-EVENT suite à chaque mise à jour de la partie dynamique de la MDB.

Dans le cas particulier des événements temporels de périodicité P, une survenance à la date D d'un tel événement doit être suivie d'une insertion dans l'AGENDA d'un nouvel événement temporel de date D+P. Cette mise à jour ne peut être assurée par le META-SYSTEM, car celui-ci n'est invoqué qu'après des modifications de la MDB. C'est donc le TEG qui s'occupe de telles mises à jour.

5.3. Event Manager (EM)

L'Event Manager se subdivise en deux composantes :

- l'*EventSelector* , qui extrait de la file d'attente un événement correspondant à un profil d'événement explicité dans une demande d'extraction;
- l'*Action Selector* , qui extrait de la MDB l'action correspondant à un événement donné.

5.3.1. Event Selector (ES)

Toute extraction d'événement de la file d'attente se fait via une requête spéciale fournie par l'ES :

SELECT-EVENT(EventType , EventMoment , [ProcessIdentifier])→ Event

Le dernier paramètre (ProcessIdentifier) désigne un identificateur de processus qui n'est nécessaire que si EventType = **SideEffect** ou si EventType = **ConstraintViolation** et EventMoment = **CommitTime**. En effet, étant donné que nous sommes dans un environnement multi-utilisateurs, chaque action doit pouvoir se distinguer des autres pour retrouver ses événements. Notons qu'une autre solution aurait consisté en la gestion d'une file d'attente par action.

5.3.2. Action Selector (AS)

Une autre requête (fournie par l'AS) sert à sélectionner dans la partie dynamique de la MDB l'action à déclencher en réponse à un événement :

SELECT-ACTION (Event , EventMoment) → Action

Cette requête sera utilisée par l'Action Manager (AM; cf. section 5.4.) au cas où une mise en file d'attente d'un événement n'aurait pas de sens, car suivie immédiatement de sa prise en charge.

Cependant, étant donné que seulement l'extraction d'un événement n'est pas utile au AM, les deux requêtes ci-dessus ont été combinées pour lui en une seule :

REQUEST (EventType , EventMoment , [ProcessIdentifier]) → (Action , Parameters)

dont l'implémentation est la suivante :

```
Event ← SELECT-EVENT (EventType,EventMoment,[ProcessIdentifier])
IF Event = NIL
   THEN RETURN ( NIL , NIL )
   ELSE RETURN ( SELECT-ACTION ( Event , EventMoment ) , Event.Parameters )
FI
```

5.4. Action Manager (AM)

L'Action Manager se subdivise aussi en deux composantes, dont la première utilise la deuxième :

- le *Daemon* , qui est un processus permanent créé par ALMA et qui prend en charge les événements de types UCE et TE stockés dans la file d'attente;

- l'*Action Processor* , qui exécute des programmes d'action.

5.4.1. Daemon

Voici donc en gros la procédure sous-jacente au Daemon :

```
WHILE TRUE DO
   ( A , P ) ← REQUEST (UserCommandEvent | TemporalEvent , / , /)
   I ← ALLOC-PID ()
   FORK ( A , P , I )
OD
```

Deux nouvelles opérations fournies par l'Action Processor (AP; cf. section 5.4.2.) sont nécessaires :

ALLOC-PID () : retourne un nouvel identificateur de processus;

FORK (Action , Parameters , ProcessIdentifier) : exécute Action en tant que processus *concurrent* identifié par ProcessIdentifier et lui transmet Parameters.

Cette technique permet au démon de tourner en permanence, car il n'a pas à se préoccuper de la bonne fin des processus créés.

Notons que nous réservons le terme "opération" aux activité des composantes du moteur : les opérations ne sont pas à confondre avec les primitives du langage de programmation des actions et les commandes.

5.4.2. Action Processor (AP)

L'Action Processor est donc utilisé par le démon, mais nous verrons qu'il s'utilise lui-même dans le cas des actions emboîtées. Il se subdivise en deux parties :

- *exécution d'action* ;
- *fin d'action* : choix de l'interprétation du **EndOfTransaction** implicite.

Notons d'abord que les paramètres formels TYPE (désignant le type de l'événement ayant déclenché l'action en cours) et PROCESS-IDENTIFIER (désignant l'identificateur du processus de l'action en cours) sont utilisées par le moteur. Leur utilité deviendra évidente à la lecture des pseudo-codes ci-dessous.

Soient aussi les nouvelles opérations fournies par l'AP :

RUN (Action , Parameters , ProcessIdentifier) : exécute Action en tant que processus identifié par ProcessIdentifier, lui transmet Parameters et passe *après* cette exécution à l'opération suivante;

PUT-QUEUE (Event , EventMoment , ProcessIdentifier) : met les trois paramètres dans l'EVENT-QUEUE;

DELETE-QUEUE (Event , EventMoment , ProcessIdentifier) : enlève toutes les entrées de l'EVENT-QUEUE qui correspondent aux trois paramètres.

Analysons à présent les deux parties du AP :

5.4.2.1. Exécution d'action

Ayant laissé ouvertes les questions relatives au langage de programmation utilisé et au mode d'exécution (compilation ou interprétation), il ne nous reste à détailler que ce qui se cache derrière les commandes et les primitives offertes par ALMA.

Quant aux commandes, elles sont prioritaires sur les primitives en cas de conflit de nom. Leur traduction en événements de type ACE s'ensuit de la prise en charge immédiate de cet événement.

Les primitives **Make**, **RecordEvent**, **PropagateEvent**, **ResetEvents Update** et **DetectViolations** ont été suffisamment décrites jusqu'à présent pour permettre une compréhension immédiate de leurs portions de pseudo-code ci-dessous.

La primitive **Fail** provoque l'échec de l'action courante et le retrait de l'EQ de tous les événements qu'elle y a déchargés.

Les autres primitives sont celles qu'il y avait déjà sous ALMA et ne seront donc pas décrites ici.

D'où le fragment de pseudo-code (en supposant qu'une primitive peut y figurer plusieurs fois) :

```
CASE ... OF
  "command" :
        e ← ACE(Source,Command,Source,Command,System,DATE(),
                PositionalParameters)
        RUN ( SELECT-ACTION ( e , / ) , e.Parameters , ALLOC-PID () )
  (* primitives *)
  "all primitives accessing x" :
      (* handle recorded events on access *)
      FOR EACH e ∈ (x.LsRecEvents) DO
        remove e from (x.LsRecEvents)
        RUN ( SELECT-ACTION ( e , Access ) , e.Parameters , ALLOC-PID () )
      OD
  DetectViolations ( x ) :
        FOR EACH c ∈ set of constraints attached to the type of X DO
          IF (c.detection = Demand) AND (NOT CHECK(c.predicate))
            THEN  e ← CV(c,Violation,c,Command,System,DATE())
                  IF c.event-moment = Deferred
                    THEN store e into (x.LsRecEvents)
                    ELSE (* Immediate *)
                         RUN ( SELECT-ACTION ( e , Immediate ) ,
                               / , ALLOC-PID () )
                  FI
            ELSE
          FI
        OD
  Fail :  DELETE-QUEUE ( * , * , PROCESS-IDENTIFIER )
        END-EXEC ( FALSE )
  Make ( x ) :
        FOR EACH e ∈ (x.LsRecEvents) DO
          remove e from (x.LsRecEvents)
          RUN ( SELECT-ACTION ( e , Demand ) , e.Parameters , ALLOC-PID () )
        OD
```

```
   PropagateEvent ( Event , EventMoment ) :
         PUT-QUEUE ( Event , EventMoment , PROCESS-IDENTIFIER )
   RecordEvent ( Event ) :
         store Event into ((Event.Source).LsRecEvents)
   ResetEvents ( Event , EventMoment ) :
         DELETE-QUEUE ( Event , EventMoment , PROCESS-IDENTIFIER )

   Update ( x ) :
         UPDATE ( x )
         FOR EACH c ∈ set of constraints attached to the type of x DO
            IF (c.detection = Update) AND (NOT CHECK(c.predicate))
               THEN e ← CV(c,Violation,c,Command,System,DATE())
                    CASE c.event-moment OF
                       Deferred : store e into (x.LsRecEvents)
                       BeforeCommit : PUT-QUEUE ( e , BeforeCommit ,
                                                  PROCESS-IDENTIFIER )
                       Immediate : RUN(SELECT-ACTION( e , Immediate ) ,
                                       e.Parameters , ALLOC-PID () )
                    ESAC
               ELSE
            FI
         OD
   ... : ...
ESAC
```

5.4.2.2. Fin d'action

Implicitement, la dernière chose à faire dans une action est un **EndOfTransaction**. Cela consiste à exécuter la séquence des actions correspondant aux événements (de type SE ou CV et de moment **BeforeCommit**) que cette action aura mis en file d'attente.

Si une seule de ces actions échoue (retourne **false**), alors le **EndOfTransaction** est interprété comme un **AbortTransaction** : toutes les mises à jour dans la PDB faites par cette action (et ses actions emboîtées) sont défaites et l'action échoue elle-même.

Si toutes ces actions réussissent (retournent **true**), alors le **EndOfTransaction** est interprété comme un **CommitTransaction** s'il s'agit d'une action non emboîtée : toutes les mises à jour de la PDB faites par cette action (et ses actions emboîtées) sont validées et l'action réussit elle-même. Finalement, il faut exécuter la séquence des actions correspondant aux événements (de type SE et de moment **AfterCommit**) que cette action aura mis en file d'attente D'où le pseudo-code :

```
(* handle events with moment = BeforeCommit *)
(A,P) ← REQUEST ( SideEffect | ConstraintViolation ,
                  BeforeCommit , PROCESS-IDENTIFIER )
WHILE A ≠ NIL DO
   IF RUN ( A , P , ALLOC-PID () )
      THEN (A,P) ← REQUEST ( SideEffect | ConstraintViolation,
                             BeforeCommit , PROCESS-IDENTIFIER )
      ELSE ABORT-TRANSACTION ( PROCESS-IDENTIFIER )
           DELETE-QUEUE ( * , * , PROCESS-IDENTIFIER )
           END-EXEC ( FALSE )
   FI
OD
```

```
IF not nested action
   THEN  COMMIT-TRANSACTION ( PROCESS-IDENTIFIER )
         (* handle side-effects with moment = AfterCommit *)
         (A,P) ← REQUEST ( SideEffect , AfterCommit , PROCESS-IDENTIFIER)
         WHILE A ≠ NIL DO
            RUN ( A , P , ALLOC-PID () )
            (A,P) ← REQUEST ( SideEffect , AfterCommit ,
                                          PROCESS-IDENTIFIER )
         OD
         END-EXEC ( TRUE )
   ELSE
FI
```

6. CONCLUSION

Nous avons élaboré tout au long de cette troisième partie une proposition d'extension du méta-modèle d'ALMA afin d'y incorporer les aspects dynamiques d'un modèle de cycle de vie d'un logiciel. Une architecture de moteur implémentant les mécanismes sous-jacents aux notions d'événement et d'action a été développée.

Voyons à présent les **extensions possibles** de ce travail, puisque certaines questions ont été laissées ouvertes.

Il reste notamment à définir de manière précise un *langage de programmation des actions*, à savoir ses mécanismes de composition algorithmique (séquentielle, alternative, itérative, etc.) ainsi que ses primitives. Certaines de ces primitives ont été mises en évidence lors de cette partie, d'autres existent déjà sous ALMA (accès à la PDB, interface homme/machine, etc.). Il faudra aussi choisir si le méta-système compile les codes source écrits dans ce langage, ou si le moteur doit les interpréter à chaque déclenchement des actions correspondantes.

Ensuite, un *formalisme de définition des moments de survenance des événements temporels* devra être développé. Nous pensons par exemple à des points temporels, intervalles temporels, durées, périodicités, etc. Un nouveau composant devra ête ajouté au méta-système afin de mettre à jour l'AGENDA suite à des extensions de la partie dynamique de la MDB. La complexité de ce composant sera proportionnelle à celle des types de moment exprimables.

Finalement, il faudra identifier un *langage formel (déclaratif ?) d'expression de contraintes* afin que celles-ci soient automatiquement vérifiables. Etant donné la grande variété de contraintes imaginables, cette tâche sera loin d'être facile. D'autre part, il n'est pas impossible que ce langage soit tel que les effets de bord apparaissent comme un cas particulier des violations de contrainte.

Bien entendu, toute ré-utilisation de composants déjà existants dans ALMA doit être favorisée lors de ces recherches. De même, il n'est pas exclus que celles-ci aboutissent à des modifications des solutions proposées ci-dessus.

Ultimement, on peut envisager d'implémenter un *assistant "intelligent"* en termes de ces mécanismes d'événements/actions. Cet assistant sera dirigé par une base de connaissances sur le processus de développement de logiciels et guidera les utilisateurs d'ALMA (selon leur niveau d'expertise) à travers les différentes phases du cycle de vie d'un logiciel.

CONCLUSION

CONCLUSION

En conclusion de ce mémoire, reprenons les trois parties une à une afin de préciser nos apports personnels, d'énumérer nos principaux résultats, de cerner les limites des solutions adoptées et d'énoncer des perspectives de recherche ultérieure.

C.1. Contribution personnelle et principaux résultats

L'apport personnel de ce mémoire s'échelonne à plusieurs niveaux.

Ainsi, dans la **partie I**, l'*étude détaillée* de l'environnement de programmation NOMADE constitue un effort original d'articulation en un tout homogène des divers documents produits lors de la conception de NOMADE (année 1987). L'effet de bord de cette analyse se traduisait pendant notre stage par un "feedback" continu sur les spécifications de NOMADE. Cette analyse a aussi servi de base à l'*étude critique* de NOMADE (sections 11 à 15) qui clôture cette première partie.

En ce qui concerne la **partie II**, le système de sécurité proposé (ainsi que les variantes examinées) sont le fruit de nos propres réflexions, excepté l'aspect "expression des droits usagers" qui était déjà présent sous une forme primitive à notre début de stage. Les principaux résultats de ces recherches sont le développement d'une technique de vérification des droits usagers, ainsi que d'une technique de vérification d'inclusion ensembliste servant à la vérification de "légalité" d'une modification des droits usagers.

Finalement, la **partie III** présente les résultats de nos investigations personnelles visant à incorporer les aspects dynamiques d'un modèle de cycle de vie d'un logiciel dans le méta-modèle de l'atelier logiciel ALMA.

C.2. Limites

Les limites de l'environnement de programmation NOMADE ont été énumérées dans une étude critique à la fin de la **partie I**.

En ce qui concerne notre système de sécurité exposé dans la **partie II**, il est difficile d'en évaluer à présent l'*applicabilté* vu sa non implémentation actuelle. Toutefois, nous avons déjà proposé une solution au problème de la *granularité* de protection. D'autre part, il est certain que les *performances d'accès* seront dégradées par l'ajout d'un système de protection; il reste donc à en évaluer le coût supplémentaire. Finalement, il est impératif de trouver une solution au probème de la méconnaissance potentielle des droits effectifs

La **partie III** donnant surtout des orientations de recherche, il est évident que seule une étude approfondie permettra d'en cerner les limites.

C.3. Perspectives de recherche

Le présent travail ne constitue évidemment pas un traitement complet des aspects abordés.

L'étude critique de NOMADE qui clôture la **partie** Ⅰ ne doit pas être vue comme une mise en question de l'utilité de cet environnement de programmation, mais plutôt comme une suggestion de modification du modèle sous-jacent. Il serait donc certainement intéressant d'étudier des *modèles alternatifs* .

Le système de sécurité proposé dans la **partie** Ⅱ, ainsi que les variantes examinées, ouvrent tout un champ d'expérimentation, tant du côté concepteurs de NOMADE, que du côté de ses utilisateurs, afin d'en évaluer l'applicabilité; un prototypage serait donc utile en vue de juger de son caractère réalisable. La conception de l'*outil d'analyse dynamique des effets d'une modification des droits* proposé dans la section d'évaluation du système de protection devrait aboutir à une meilleure maîtrise du flux des autorisations et à une meilleure connaissance des droits effectifs.

La **partie** Ⅲ joue aussi un rôle fortement *exploratoire* . De nombreuses questions ont été laissées ouvertes lors de l'élaboration de la partie dynamique du méta-modèle d'ALMA, ainsi que du mécanisme sous-jacent (moteur événements/actions). Ainsi, il reste à *définir* le langage de programmation des actions, le langage de définition des moments de survenance des événements temporels et le langage d'expression formelle des contraintes. Une rétroaction de ces recherches sur le résultat actuel n'est évidemment pas exclue. Ultimement, un *assistant "intelligent* " pourra être greffé sur les mécanismes d'activité d'ALMA.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[Belkhatir 86a]
Noureddine BELKHATIR and Jacky ESTUBLIER; Protection and Cooperation in a Software Engineering Environment; IFIP WG2.4 - International Workshop on Advanced Programming Environments; Trondheim (Norway), June 1986.

[Belkhatir 86b]
Noureddine BELKHATIR and Jacky ESTUBLIER; Experience with a Data Base of Programs; Proceedings of the 2nd ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments; Palo Alto (California), December 9 -11, 1986; ACM Sigplan Notices, Volume 22, No. 1, pp. 84 - 91, January 1987.

[Belkhatir 87]
Noureddine BELKHATIR and Jacky ESTUBLIER; Software Management Constraints and Action Triggering in the ADELE Program Database; Proceedings of the 1st European Software Engineering Conference; Strasbourg (France), September 9-11, 1987, pp. 47-57.

[Bernard 87]
Y. BERNARD, M. LACROIX, P. LAVENCY and M. VANHOEDENAGHE; Configuration Management in an Open Environment; Proceedings of the 1st European Software Engineering Conference; Strasbourg (France), September 9-11, 1987, pp. 37-46.

[Bernstein 87]
Philip A. BERNSTEIN; Database System Support for Software Engineering - An Extended Abstract; Proceedings of the 9th International Conference on Software Engineering; Monterey, California, 1987, pp. 166 - 178.

[Bodart 83]
François BODART et Yves PIGNEUR; Conception assistée des applications informatiques - Volume 1 : Etude d'opportunité et analyse conceptuelle; Masson, Paris 1983.

[Boehm 81]
Barry W. BOEHM; Software Engineering Economics; Prentice Hall,Inc., Englewood Cliffs, New Jersey,1981.

[Brodie 81]
Michael L. BRODIE; On Modelling Behavioural Semantics of Databases; Proceedings of the 1981 International Conference on Very Large Databases; Cannes (France), September 1981.

[Brodie 82]
Michael L. BRODIE and Erico SILVA; Active and Passive Component Modelling; in : T. W. OLLE, H. G. SOL and A. A. VERRIJN-STUART (editors); Information Systems Design Methodologies : A Comparative Review; North-Holland Publishing Company, IFIP,1982.

[Cox 86]
Brad J. COX; Object Oriented Programming : An Evolutionary Approach; Addison-Wesley, Reading (Massachusetts), 1986.

[Date 82]
C. J. DATE; An Introduction to Database Systems; Addison-Wesley, Reading (Massachusetts), 1982.

[Estublier 84]
Jacky ESTUBLIER, Said GHOUL and Sacha KRAKOWIAK; Preliminary Experience with a Configuration Control System for Modular Programs; Proceedings of the 1st ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments; April 1984.

[Estublier 86]
Jacky ESTUBLIER; ADELE - A Database of Programs : Presentation Manual; June 1986 (revised September 1987).

[Estublier 88a]
Jacky Estublier; Configuration Management - The Notion and the Tools; Proceedings of a Workshop on Version Control and Configuration Management; Grassau (Germany), January 1988.

[Estublier 88b]
Jacky Estublier; Spécifications de NOMADE; IMAG - LGI : Document interne; Janvier 1988.

[Feiler 87]
Peter E. FEILER and Gail E. KAISER; An Architecture for Intelligent Assistance in Software Development; Proceedings of the 9th International Conference on Software Engineering; Monterey, California, 1987, pp. 180 - 188.

[Feldman 79]
S. I. FELDMAN; Make - A Program for Maintaining Computer Programs; Software Practice and Experience; Volume 9, pp. 255 - 265; April 1979.

[Gallo 86]
F. GALLO, R. MINOT and I. THOMAS; The Object Management System of PCTE as a Software Engineering Database Management System; Proceedings of the 2nd ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments; Palo Alto (California), December 9 - 11, 1986; ACM Sigplan Notices, Volume 22, No. 1, pp. 12 - 15, January 1987.

[Goldberg 83]
Adele GOLDBERG and David ROBSON; SMALLTALK-80 : The Language and its Implementation; Addison-Wesley, Reading (Massachusetts), 1983.

[Griffiths 76]
Patricia P. GRIFFITHS and Bradford W. WADE; An Authorization Mechanism for a Relational Database System; ACM Transactions on Database Systems, Volume 1, No.3, pp. 242 - 255; September 1976.

[Hainaut 86]
Jean-Luc HAINAUT; Conception assistée des applications informatiques - Volume 2 : Conception de la base de données; Masson, Paris 1986.

[Harrison 76]
M. A. HARRISON, W. L. RUZZO and J. D. ULLMAN; Protection in Operating Systems; CACM, Volume 19, 1976.

[Hartson 84]
H. Rex HARTSON; Implementation of Predicate-Based Protection in MULTISAFE; Software - Practice and Experience, Volume 14, No. 3, pp.207 - 214; March 1984.

[Horowitz 86]
Ellis HOROWITZ & Ronald C. WILLIAMSON; SODOS : A Software Documentation Support Environment - Its Definition; IEEE Transactions on Software Engineering; Volume SE-12, No. 8, pp. 849 - 859, August 1986.

[Ingalls 81]
Daniel H.H. Ingalls; Design Principles behind SMALLTALK; BYTE August 1981, pp. 286-298.

[Jones 76]
A. K. JONES, R. J. LIPTON and L. SNYDER; A Linear Time Algorithm for Deciding Security; Proceedings of the 17th Symposium on Foundations of Computer Science, 1976.

[Landwehr 81]
Carl E. LANDWEHR; Formal Models for Computer Security; Computing Surveys, Volume 13, No. 3, pp. 247 - 278, September 1981.

[Leblang 84]
David B. LEBLANG and Robert P. CHASE, Jr.; Computer-Aided Software Engineering in a Distributed Workstation Environment; Sigplan/Sigsoft Symposium on Practical Software Development Environments; ACM; April 1984.

[Linden 76]
Theodore A. LINDEN; Operating System Structures to Support Security and Reliable Software; Computing Surveys, Volume 8; No. 4, pp. 409 - 445, December 1976.

[Lingat 86]
Jean-Yves LINGAT, Philippe NOBECOURT and Colette ROLLAND; Managing the Behavior of Database Applications; 1986.

[LRG 81]
The Xerox Learning Research Group; The SMALLTALK-80 System; BYTE August 1981, pp. 36-47.

[Meyer 85a]
Bertrand MEYER; The Software Knowledge Base; Proceedings of the 8th International Conference on Software Engineering; London, 1985, pp. 158 - 165.

[Meyer 85b]
Bertrand MEYER; Eiffel : A Language for Software Engineering; Technical Report TRCS 85-19; University of California, Santa Barbara, Computer Science Department, November 1985 (Revised August 1986).

[Meyer 87a]
Bertrand MEYER; Eiffel : Programming for Reusability and Extendibility; ACM Sigplan Notices, Volume 22, No. 2, pp. 85-94, February 1987.

[Meyer 87b]
Bertrand MEYER; Reusability : The Case for Object-Oriented Design; IEEE Software, pp. 50-64, March 1987.

[Meyer 87c]
Bertrand MEYER, Jean-Marc NERSON and Masanobu MATSUO; Eiffel : Object-Oriented Design for Software Engineering; Proceedings of the 1st European Software Engineering Conference; Strasbourg (France), September 9 -11, 1987, pp. 237-245.

[Minsky 81]
Naftaly MINSKY; Synergistic Authorization in Database Systems; Proceedings of the 1981 International Conference on Very Large Databases; Cannes (France), September 1981.

[Nilsson 71]
Nils NILSSON; Problem-Solving Methods in Artificial Intelligence; McGraw - Hill Book Company; 1971.

[Osterweil 81]
Leon OSTERWEIL; Software Environment Research : Directions for the Next Five Years; IEEE Computer; Volume 14, Number 4, pp. 35 - 43; April 1981.

[Penedo 85]
Maria H. PENEDO and E. Don STUCKLE; PMDB - A Project Master Data Base for Software Engineering Environments; Proceedings of the 8th International Conference on Software Engineering; London, 1985, pp. 150 - 157.

[Penedo 87]
Maria H. PENEDO; Prototyping a Project Master Data Base for Software Engineering Environments; Proceedings of the 2nd ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments; Palo Alto (California), December 9 -11, 1986; ACM Sigplan Notices, Volume 22, No. 1, pp. 1 - 11, January 1987.

[Robson 81]
David ROBSON; Object-Oriented Software Systems; BYTE August 1981, pp. 74-86.

[Rochkind 75]
Marc J. ROCHKIND; The Source Code Control System; IEEE Transactions on Software Engineering; Volume SE-1, No. 4, pp. 364 - 370; December 1975.

[Smith 77]
John Miles SMITH and Diane C. P. SMITH; Database Abstractions : Aggregation and Generalization; ACM Transactions on Database Systems; Volume 2, Number 2, pp. 105-133; June 1977.

[Snyder 81]
Lawrence SNYDER; Formal Methods of Capability-Based Protection Systems; IEEE Transactions on Computers, Volume C-30, No. 3, pp.172 - 181, March 1981.

[Taylor 87]
Richard N. TAYLOR et al.; Next Generation Software Environments : Principles, Problems and Research Directions; University of California at Irvine, Department of Information and Computer Science, Technical Report Number 87-16; July 15, 1987.

[Tichy 82]
Walter TICHY; Design, Implementation and Evaluation of a Revision Control System; Proceedings of the 6th International Conference on Software Engineering; Tokyo, 1982; pp. 58 - 67.

[Trueblood 86]
Robert P. TRUEBLOOD and A. SENGUPTA; Dynamic Analysis of the Effects Access Rule Modifications Have Upon Security; IEEE Transactions on Software Engineering, Volume SE-12, No. 8, pp. 866 - 870, August 1986.

[van Lamsweerde 82]
Axel van LAMSWEERDE; Les outils d'aide au développement de logiciels : un aperçu des tendances actuelles; Proceedings JIIA, Paris, June 1982.

[van Lamsweerde 85]
Axel van LAMSWEERDE; Cadre général pour un modèle de cycle de vie d'un projet informatique; FNDP - Namur (Belgium); Révision août 1985.

[van Lamsweerde 86]
Axel van LAMSWEERDE et al.; The Kernel of a Generic Software Development Environment; Proceedings of the 2nd ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments; Palo Alto (California), December 9 -11, 1986; ACM Sigplan Notices, Volume 22, No. 1, pp. 208 - 217, January 1987.

[van Lamsweerde 87]
Axel van LAMSWEERDE; A Generic Knowledge-Based Assistant; Summary of a Research Project; FNDP - Namur (Belgium); December 1987.

[van Lamsweerde 88a]
Axel van LAMSWEERDE; Situation de l'atelier ALMA par rapport à l'état de l'art actuel; January 1988.

[van Lamsweerde 88b]
Axel van LAMSWEERDE et al.; Generic Lifecycle Support in the ALMA Environment; IEEE Transactions on Software Engineering; Special Issue on Software Engineering Environment Architectures; June 1988.

ANNEXES

Annexe A

Grammaire de NOMADE

A.1. Conventions

Les symboles terminaux sont imprimés en **gras**.

Les symboles non terminaux apparaissent entre "<" et ">".

"::=" sépare les parties gauche et droite d'une définition.

" | " sépare deux choix possibles.

Tout ce qui apparaît entre "[" et "]" est optionnel (0-1).

Tout ce qui apparaît entre "{" et "}" est répétitif (0-N).

A.2. Méta-caractères utilisés

Toutes les informations fournies par l'utilisateur, c'est-à-dire ne faisant pas partie de la syntaxe concrète (comme par exemple les noms d'objet, d'élément, d'attribut, de relation, etc.), correspondent dans la grammaire de NOMADE au symbole non terminal <Identifier>. La philosophie de NOMADE veut que, chaque fois que possible et pertinent, un tel identifiant puisse désigner plusieurs valeurs possibles. Ce but est atteint via le mécanisme des **expressions régulières**, qui utilisent les **méta-caractères** suivants (fort proches des conventions du shell Unix).

Soient :

Alpha = {a, b, ..., z, A, B, ..., Z}
Digit = {0, 1, ..., 9}
Spec = {-, _}
Sep = {:, ., /} (symboles terminaux séparateurs)

on a:

- le méta-caractère "?" désigne n'importe quel caractère c, tel que c ∈ Alpha ∪ Digit ∪ Spec;
- le méta-caractère "*" désigne une chaîne (éventuellement vide) de caractères c, telle que c ∈ Alpha ∪ Digit ∪ Spec;
- les méta-caractères "**" désignent une chaîne (éventuellement vide) de caractères c, telle que c ∈ Alpha ∪ Digit ∪ Spec ∪ Sep;

- l'expression "[α-β]" désigne n'importe quel caractère c tel que ASCII(α) ≤ ASCII(c) ≤ ASCII(β);

- l'expression "[αβ...γ]" désigne n'importe quel caractère c tel que c ∈ S = {α, β, ..., γ}; si on veut que le tiret "-" fasse partie de S, il faudra le placer soit en première, soit en dernière position de S, afin d'éviter toute ambiguïté avec les expressions du type précédent.

Exemples : (notons "≡" l'opérateur de "matching" entre une chaîne de caractères et une expression régulière)

Nomade ≡ [A-Z]*
Adele.V5 ≡ **5

A.3. Grammaire

```
<Manual>          ::= manual <Object>
                        <ObjType>
                            [ <AttrClause> | <FamAttrClause> ]
                            [ <RelClause> ]
                            [ <ActClause> ]
                            [ <RightsClause> ]
                        { <PartitionBlock> }
                        [ <SelImpClause> ]
                        [ <SelCondClause> ]
                        [ <SelDefClause> ]
                        [ <Special> ]
                      end
<ObjType>             ::= Family | Interface | Realization | User
<Special>             ::= Library | WithoutBody

<SelImpClause>        ::= selimp <NOMADEexpr>
<SelCondClause>       ::= selcond <NOMADEexpr>
<SelDefClause>        ::= seldef <NOMADEexpr>


<PartitionBlock>      ::= partition <Relation>
                            [ <FamAttrClause> ]
                            [ <RelClause> ]
                            [ <ActClause> ]
                            [ <RightsClause> ]

<FamAttrClause>       ::= attributes { def <Domain> { <AttrDef> ; } }
<Domain>              ::= <NOMADEexpr>
<AttrClause>          ::= attributes { <AttrDef> ; }
<AttrDef>             ::= <Attribute> = <Value> { , <Value> } | stopinh
<Attribute>           ::= <Identifier>
<Value>               ::= <Identifier> | stopinh

<RelClause>       ::= relations { <RelDef> ; }
```

```
<RelDef>        ::= <Source> <Relation> <Target> | stopinh
<Source>        ::= <NOMADEexpr> | reflex
<Relation>      ::= <Identifier>
<Target>        ::= <NOMADEexpr> | stopinh

<ActClause>     ::= actions
                      [ command { <ForStat> } ]
                      [ post-action { <ForStat> } ]
                      [ obligations { <ForStat> } ]
                      [ demand { <ForStat> } ]
                      [ access { <ForStat> } ]
<ForStat>       ::= for <Relation> ( <Command> )
                      do
                          { <Statement> ; }
                      done;
<Relation>      ::= <Identifier>
<Command>       ::= <Identifier>
<Statement>     ::= <CondStat> | <Action>
<CondStat>      ::= if <Condition>
                      then { <Action> ; }
                      [ else { <Action> ; } ]
                    fi ;
<Condition>     ::= <NOMADEexpr> | <BooleanExpr> | not ( <BooleanExpr> )
<BooleanExpr>   ::= <NOMADEfct> | <OScommand> | <Variable> |
                      <Expression> <RelOp> <Expression>
<Action>        ::= <NOMADEfct> | <OScommand> |
                      <Variable> := <Expression>
<Expression>    ::= <Variable> | <StringExpr> | <NOMADEfct>
<StringExpr>    ::= <AttrExpr> <CharString> { + <AttrExpr> <CharString } 
<AttrExpr>      ::= [ <Variable> ] \ <Attribute>
<Variable>      ::= $S |$R |$T |$C |$U |$D |
                      $0 |$1 |$2 |$3 |$4 |$5 |$6 |$7 |$8 |$9
<Attribute>     ::= <Identifier>
<NOMADEfct>     ::= <Function> [ ( <Parameter> { , <Parameter> } ) ]
<Function>      ::= <Identifier>
<Parameter>     ::= <Variable> | <CharString>
<OScommand>     ::= <Identifier>

<RightsClause>  ::= rights { <RightDef> }
<RightDef>      ::= <Label> : <Scope> { , <Scope> } | stopinh
<Label>         ::= <Identifier>
<Scope>         ::= <NOMADEexpr> | stopinh

<NOMADEexpr> ::= <LsOrLsAndQN>  | not ( <LsOrLsAndQN> )
<LsOrLsAndQN>::= <LsAndQN> | ( <LsAndQN> ) { or ( <LsAndQN> ) }
<LsAndQN>       ::= <QualName> { and <QualName> }  |
                      not ( <QualName> { and <QualName> } )

<QualName>      ::= [ not ] <Name> [ ( <LsAndQual> ) ]
<LsAndQual>     ::= <Qualification> { and <Qualification> }
```

<Qualification> ::= <Attribute> <RelOp> <Value> { , <Value> }
<RelOp> ::= = | <> | < | <= | > | >=

<Name> ::= [<Partition>] [<Object> [. <Element>]]

<Partition> ::= / <PartRel> . <PartAnc> /
<PartRel> ::= <Relation>
<PartAnc> ::= <Family> | <User>

<Object> ::= <Family> [: <Interface> [: <Realization>]] | <User>
<Family> ::= <Identifier>
<User> ::= <Identifier>
<Interface> ::= <Identifier>
<Realization> ::= <Identifier>

<Element> ::= <Revision> | <Document> | <UserDoc> [. <Revision>]
<Revision> ::= [<Digit> [<Digit> [<Digit>]]]
<Document> ::= **hist** | **man** | **manattr** | **manrel** | **manact** | **manright** | **mansel** | **manother** | **manpart**
<UserDoc> ::= <Identifier>

<Identifier> ::= [<FirstChar> { <NonFirstChar> }]
<CharString> ::= " { <NonFirstChar> } "
<FirstChar> ::= <AlphaChar> | <SpecialChar> | <MetaChar>
<NonFirstChar> ::= <FirstChar> | <Digit>
<AlphaChar> ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
<SpecialChar> ::= - | _
<MetaChar> ::= ? | * | [|]
<Digit> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

A.4. Sémantique statique

D'un point de vue **sémantique statique**, résumons en quels endroits dans les manuels peuvent (O) ou non (N) apparaître des méta-caractères :

clause	attributes		relations			actions		rights	
contexte	A	V	S	R	T	R	C	L	S
partition de familles	O	O	O	O	O	O	O	/	/
partition d'usagers	O	O	O	O	O	O	O	N	O
famille	O	O	O	O	O	O	O	/	/
interface	N	N	N	N	N	O	O	/	/
réalisation	N	N	N	N	N	O	O	/	/
usager	N	N	N	N	N	O	O	N	O

Tableau A.1. - Endroits possibles des méta-caractères

Annexe B

Règles par défaut dans la désignation des objets/éléments

Si dans une désignation d'objet/élément (i.e. nom simple), au moins un des identifiants de famille, interface ou réalisation est la chaîne vide, alors les règles par défaut (RD) suivantes sont appliquées successivement :

(**RDFam**) l'identifiant de famille est remplacé par la famille **courante**, c'est-à-dire la famille la plus récemment utilisée.

(**RDInt**) l'identifiant d'interface est remplacé par l'interface **par défaut** de la famille correspondante; si une telle interface par défaut n'a pas été désignée explicitement par la clause **seldef** de sa famille, alors l'interface la plus récente (la dernière introduite dans le système) de cette famille sera prise en compte;

(**RDReal**) l'identifiant de réalisation est remplacé par la réalisation **par défaut** de son interface; si une telle réalisation par défaut n'a pas été désignée explicitement (par la clause **seldef** de son interface, cf. section 10), alors la réalisation la plus récente de cette interface sera prise en compte. Les corps sont prioritaires sur les configurations dans la recherche de la réalisation la plus récente.

Si, dans une désignation d'élément d'une réalisation, on ne met que le point séparateur (exemples : $F:I_2:R_{21}.$ ou $F:I_2:R_{21}$.spécif.), alors on applique :

(**RDRev**) on complète le nom simple par le numéro de la révision la plus récente de la réalisation en question.

Remarque : pour l'exemple suivant, le lecteur se référera à l'annexe A pour la signification des méta-caractères.

Exemple : (cf. partie I - section 2.1.3. - figure 2.2.)

Supposons que :

- F soit la famille courante;
- l'ordre chronologique de création des interfaces et des réalisations soit l'ordre de gauche à droite sur la figure;
- que R_{32} soit la réalisation par défaut de I_3 et que ce soit la seule désignation par défaut explicite dans F.

Alors :

$:I_2$	désigne	$F:I_2$	(RDFam)
:	désigne	$F:I_3$	(RDFam, puis RDInt)
::	désigne	$F:I_3:R_{32}$	(RDFam, puis RDInt, puis RDReal)

F::. désigne $F:I_3:R_{32}.22$ (si on suppose que cette réalisation totalise 22 révisions); (RDInt, puis RDReal, puis RDRev);

F:*:R*1 désigne toutes les réalisations dont le nom commence par "R" et se termine par "1" de toutes les interfaces de la famille F, c'est-à-dire : $\{F:I_1:R_{11}, F:I_2:R_{21}, F:I_3:R_{31}\}$;

F::.[0-9]* désigne toutes les révisions de $F:I_3:R_{32}$;

F:*.** désigne tous les éléments de $F:I_1$, $F:I_2$ et $F:I_3$;

$F:I_2$:*.* désigne toutes les révisions et tous les documents (mais pas les documents de révision) de $F_1:I_2:R_{21}$;

* désigne toutes les familles et tous les usagers du projet;

** désigne tous les objets du projet et tous leurs éléments. ♦

Annexe C

Fonctionnement des opérateurs d'héritage

Soient A_i des noms d'attribut, V_i des valeurs d'attribut, S_i, T_i et Ne_i des expressions-NOMADE, R_i des relations (binaires ou réflexives), C_i des commandes (relations réflexives), P_i des programmes d'actions, L_i des étiquettes, M_i des moments parmi { **command**, **postaction**, **obligation**, **access**, **demand**}.

Dans les figures suivantes, la clause de gauche est la clause héritante, la clause du milieu est la clause effective héritée et la clause de droite est la clause effective.

C.1. Opérateur de réunion

C.1.1. Réunion de deux clauses attributes

La réunion de deux clauses attributes fonctionne de la manière suivante :

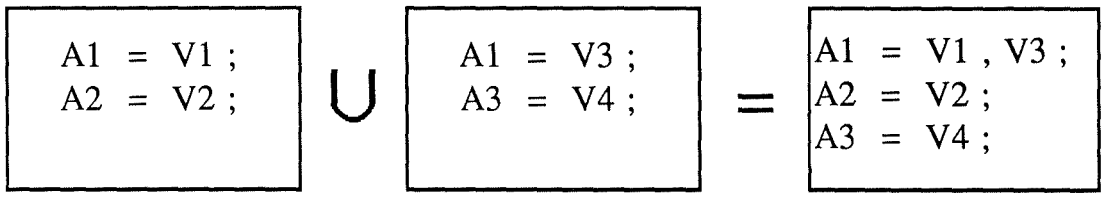


Figure C.1. - Réunion de deux clauses attributes

Les valeurs d'attributs possibles étant données par énumération, il y a élargissement de la liste des valeurs possibles (rôle définitionnel).

C.1.2. Réunion de deux clauses relations

La réunion de deux clauses relations fonctionne de la manière suivante:

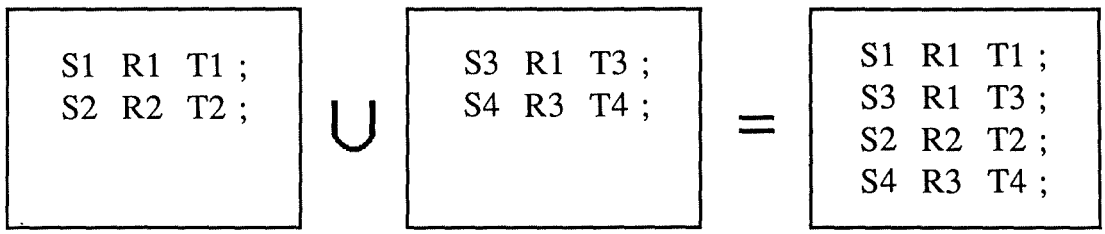


Figure C.2. - Réunion de deux clauses relations (supposons p(R2) > p(R3))

Le problème de la *priorité* des relations dans la clause effective est résolu du fait qu'elles sont déclarées une fois pour toutes au niveau de la famille projet (cf. partie I - section 8.5). Sinon, sachant que $p(R_1) > p(R_2)$ (clause héritante) et que $p(R_1) > p(R_3)$ (clause effective héritée), on n'aurait pas pu dégager que $p(R_2) > p(R_3)$.

C.1.3. Réunion de deux clauses actions

La réunion de deux clauses actions fonctionne de la manière suivante :

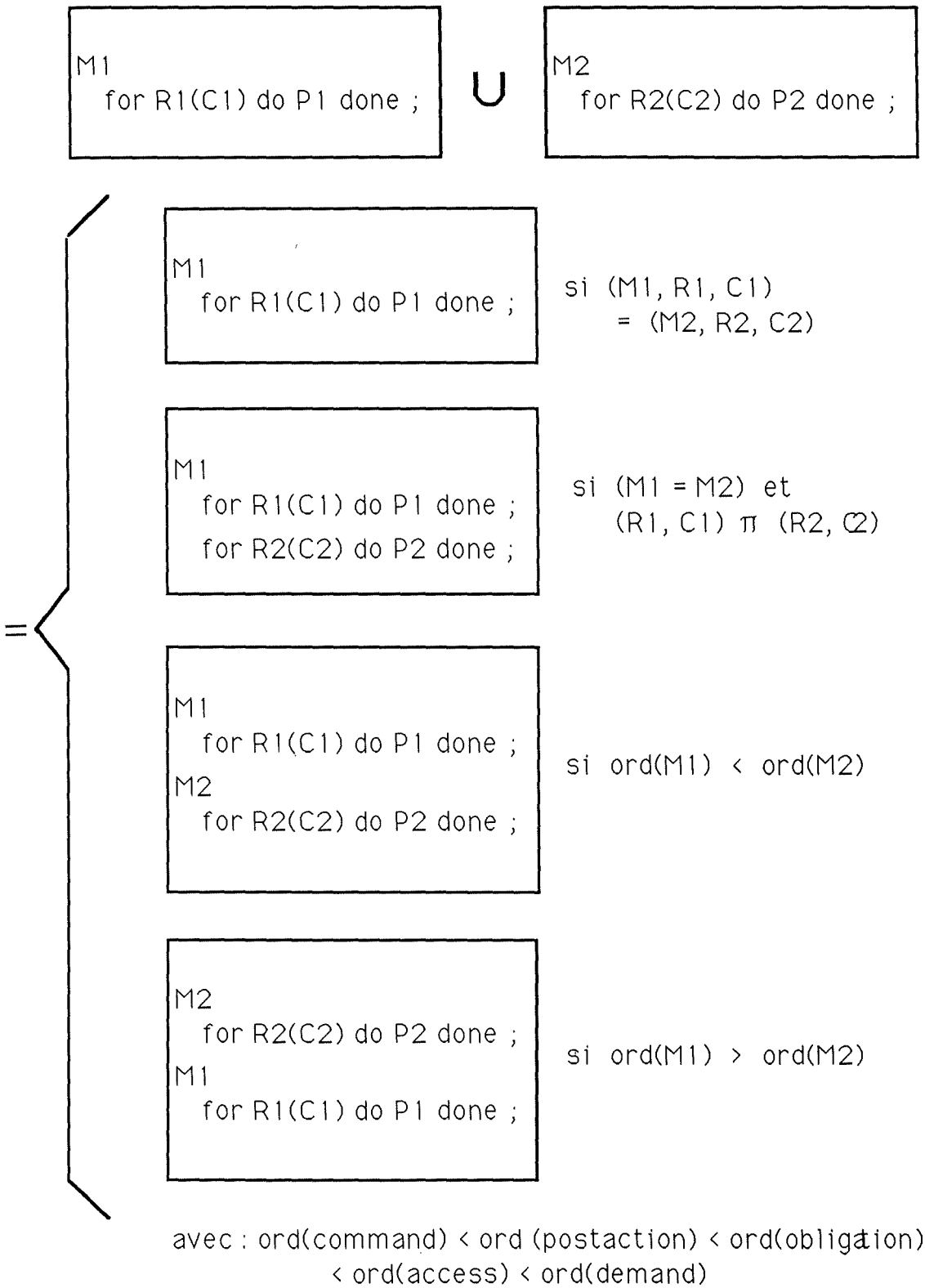


Figure C.3. - Réunion de deux clauses actions

Nous constatons que la réunion de deux clauses actions n'est pas une opération symétrique : dans le cas $(M_1, R_1, C_1) = (M_2, R_2, C_2)$, nous sommes en présence d'un phénomène de **redéfinition** : la clause héritante a redéfini le programme d'actions P_2 défini par la clause effective héritée en le substituant par P_1.

C.1.4. Réunion de deux clauses rights

La réunion de deux clauses rights fonctionne de la manière suivante:

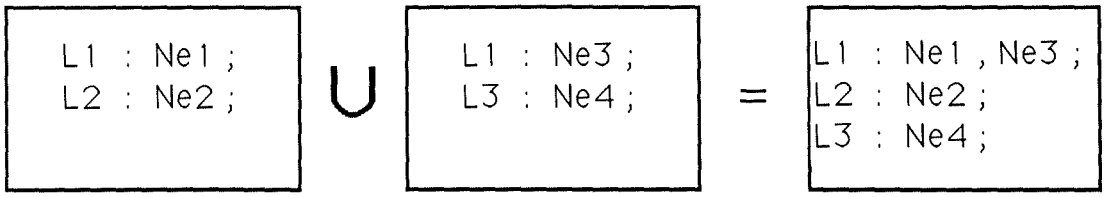


Figure C.4. - Réunion de deux clauses rights

Ici, de nouveau, il y a extension des possibilités (cas de l'étiquette L_1).

C.2. Mécanisme d'inhibition de l'héritage linéaire

Nous avons vu (section C.1.3.) que seule la clause actions dispose d'un mécanisme de redéfinition : un tel mécanisme pourrait s'avérer utile pour les trois autres clauses, qui autrement seraient "obligées" d'hériter. Le mot-clé **stopinh** ("stop inheritance"), placé au bon endroit, permet d'obtenir cet effet. Il en résulte que la réunion de deux clauses attributes, relations ou rights n'est plus une opération symétrique.

C.2.1. Inhibition dans une clause attributes d'une partition

Au niveau de la clause attributes d'une partition, le mot-clé **stopinh** peut se trouver :

- en tant que valeur d'attribut : aucune valeur d'attribut n'est héritée pour cet attribut;

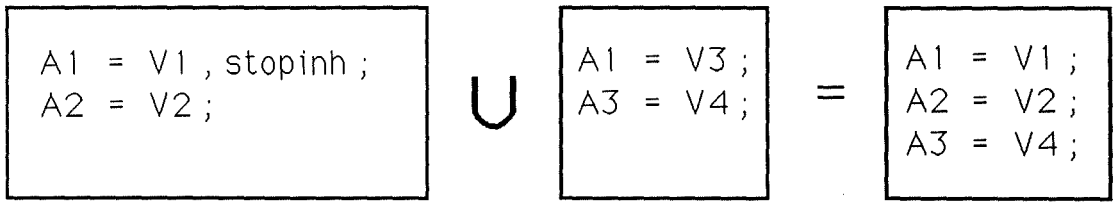


Figure C.5. - Inhibition pour A1

- seul sur la ligne : aucune valeur d'attribut n'est héritée pour tous les attributs (et non seulement pour ceux de la clause héritante);

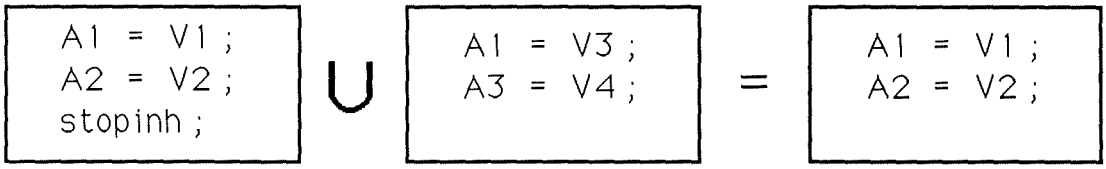


Figure C.6. - Inhibition pour tous les attributs

Il faut remarquer que la ligne où se trouve le mot-clé **stopinh** est sans importance; d'autre part, cette possibilité n'est pas équivalente à placer des **stopinh** en tant que valeur d'attribut derrière tous les attributs de la clause héritante, car :

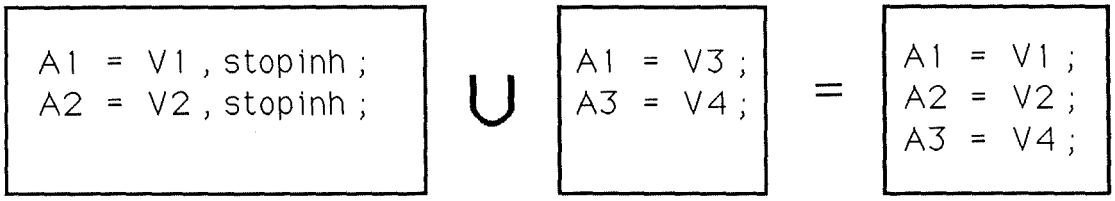


Figure C.7. - Inhibition pour A1 et A2

Il faudrait donc en plus ajouter "* = **stopinh** ;" à la clause héritante pour obtenir l'effet de la figure C.6.

C.2.2. Inhibition dans une clause relations d'une partition

Au niveau de la clause relations d'une partition, le mot-clé **stopinh** peut se trouver :

- en tant que co-domaine (avec domaine = **) : il n'y a pas d'héritage pour cette relation;

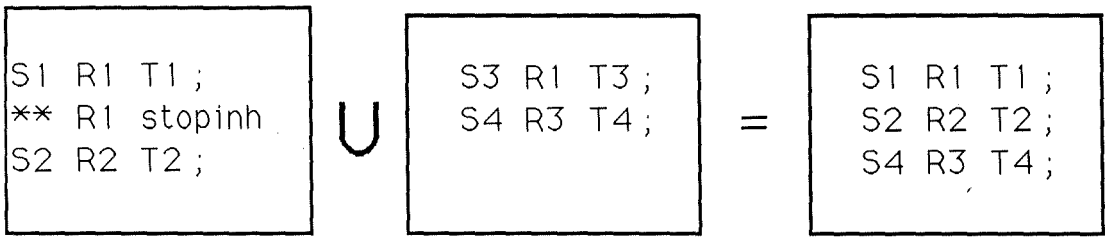


Figure C.8. - Inhibition pour R1 (supposons p(R2) > p(R3))

- seul sur la ligne : il n'y a pas d'héritage pour toutes les relations (et non seulement pour celles de la clause héritante);

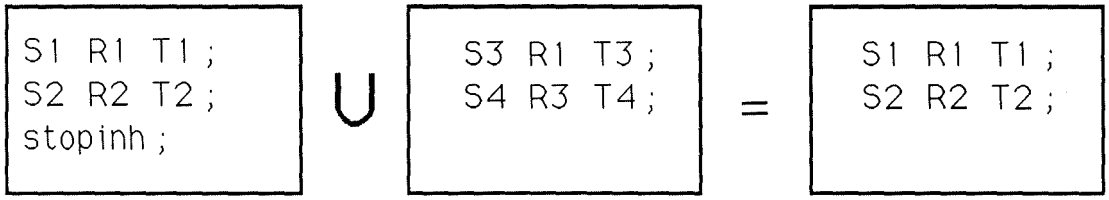


Figure C.9. - Inhibition pour toutes les relations

Il faut remarquer que la ligne où se trouve le mot-clé **stopinh** est sans importance; d'autre part, cette possibilité n'est pas équivalente à placer des **stopinh** en tant que co-domaine pour toutes les relations reprises dans la clause héritante, car :

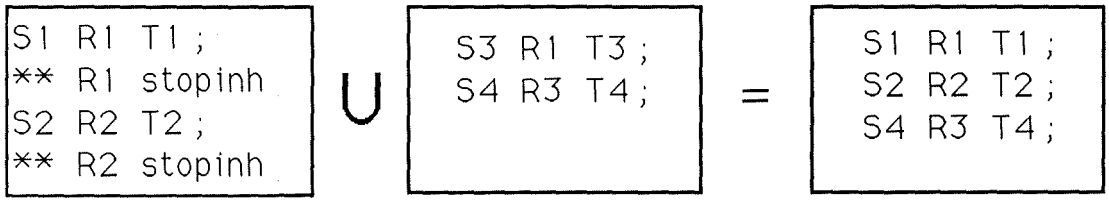


Figure C.10. - Inhibition pour R1 et R2 (supposons p(R2) > p(R3))

Il faudrait donc en plus ajouter " ** * **stopinh** ; " à la clause héritante pour obtenir l'effet de la figure C.9.

C.2.3. Inhibition dans une clause actions d'une partition

Au niveau de la clause actions d'une partition, le mot-clé **stopinh** est interdit, car l'inhibition d'héritage de clauses actions n'a pas de sens.

C.2.4. Inhibition dans une clause rights d'une partition

Au niveau de la clause rights d'une partition, le mot-clé **stopinh** peut se trouver :

- en tant que portée : il n'y a pas d'héritage pour cette étiquette;

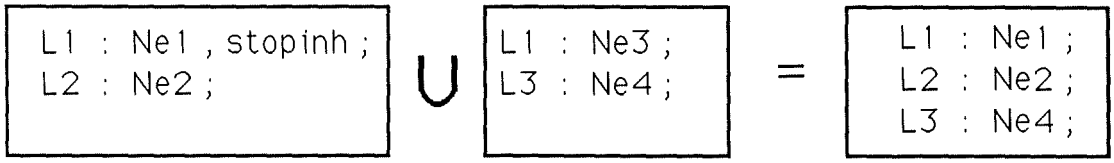


Figure C.11. - Inhibition pour L1

- seul sur la ligne : il n'y a pas d'héritage pour toutes les étiquettes (et non seulement pour celles de la clause héritante);

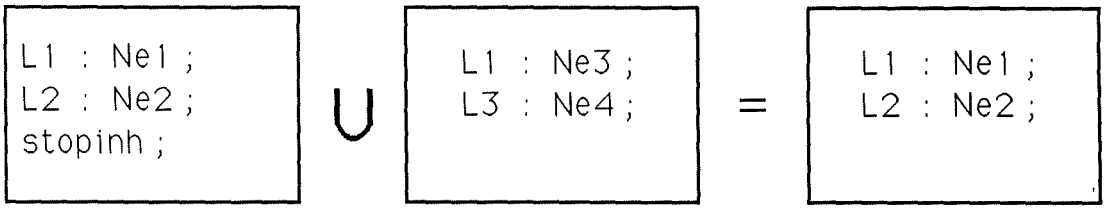


Figure C.12. - Inhibition pour toutes les étiquettes

Il faut remarquer que la ligne où se trouve le mot-clé **stopinh** est sans importance; d'autre part, cette possibilité n'est pas équivalente à placer des **stopinh** en tant que portée derrière toutes les étiquettes de la clause héritante, car :

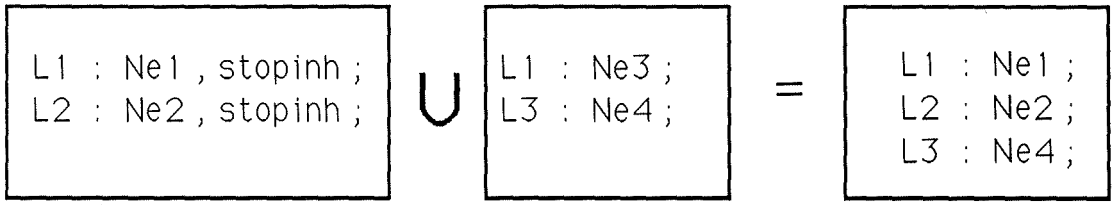


Figure C.13. - Inhibition pour L1 et L2

Il faudrait donc en plus ajouter " * : **stopinh** ; " à la clause héritante pour obtenir l'effet de la figure C.12.

C.3. Opérateur d'intersection

C.3.1. Intersection de deux clauses attributes

L'intersection de deux clauses attributes fonctionne de la manière suivante (la définition "* = *" étant implicite) :

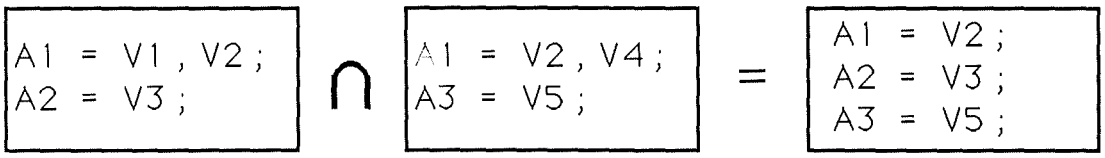


Figure C.14. - Intersection de deux clauses attributes

C.3.2. Intersection de deux clauses relations

L'intersection de deux clauses relations fonctionne de la manière suivante (la définition "** * **" étant implicite) :

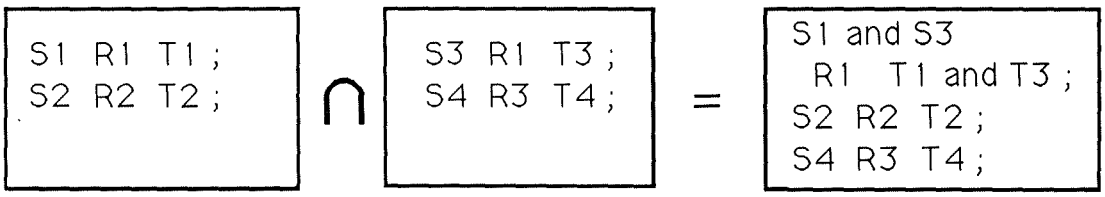


Figure C.15. - Intersection de deux clauses relations

Notons qu'ici, pour la première fois, nous devons sortir de la grammaire des expressions-NOMADE (cf. Annexe A) pour exprimer le résultat de l'application d'un opérateur à deux clauses. En effet, le connecteur **and** ne s'y trouve pas au niveau le plus externe (rappelons que les expressions-NOMADE sont des disjonctions de conjonctions de noms qualifiés). Mais il faut aussi remarquer que les opérateurs de réunion et d'intersection n'existent pas nécessairement en tant que primitives internes du système NOMADE, ce sont plutôt des moyens d'explication des mécanismes d'héritage.

C.3.3. Intersection de deux clauses actions

L'intersection de deux clauses actions fonctionne de la même manière que la réunion (!); ceci est dû au fait qu'une clause actions n'énumère pas des possibilités, mais donne des implémentations. Ainsi, les termes "intersection" et "réunion" sont en fait inadaptés dans ce contexte.

C.3.4. Intersection de deux clauses rights

L'intersection de deux clauses rights fonctionne de la manière suivante (la définition de "* : **" étant implicite :

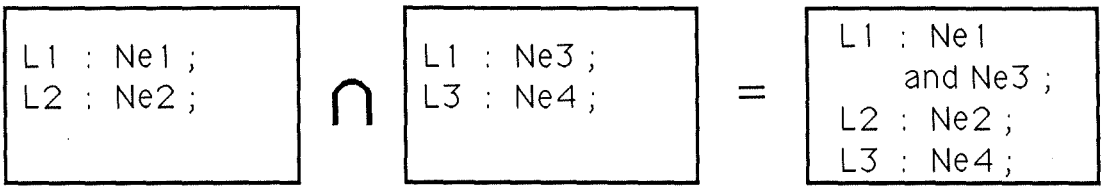


Figure C.16. - Intersection de deux clauses rights

Ici encore, le **and** de la clause effective constituerait une violation de la grammaire des expressions-NOMADE.

Annexe D

Technique de vérification d'inclusion
Démonstrations et illustrations

Rappelons d'abord quelques propriétés fondamentales de la relation d'inclusion ($\subseteq$), qui nous seront utiles :

[AD] Augmentabilité à droite

$$\forall A, B : A \subseteq B \Rightarrow \forall C : A \subseteq (B \cup C) \quad \blacklozenge$$

[AG] Augmentabilité à gauche

$$\forall A, B : A \subseteq B \Rightarrow \forall C : (A \cap C) \subseteq B \quad \blacklozenge$$

[A$\cup$] Additivité (réunion)

$$\forall A, B, C, D : A \subseteq B \wedge C \subseteq D \Rightarrow (A \cup C) \subseteq (B \cup D) \quad \blacklozenge$$

[A$\cap$] Additivité (intersection)

$$\forall A, B, C, D : A \subseteq B \wedge C \subseteq D \Rightarrow (A \cap C) \subseteq (B \cap D) \quad \blacklozenge$$

[TP] Transposition

$$\forall A, B : A \subseteq B \Leftrightarrow C(B) \subseteq C(A) \quad \blacklozenge$$

Théorème : soit $E = \bigcup_{i \in I} e_i$, on a $E^- = \bigcup_{i \in I^-} e_i^-$ ♦

Démonstration :

par définition :

$$\forall\, i \in I^- : e_i^- \subseteq e_i$$

par $(\#(I^-) - 1)$ applications successives de [A$\cup$] :

$$\bigcup_{i \in I^-} e_i^- \subseteq \bigcup_{i \in I^-} e_i$$

par $(\#(I \setminus I^-))$ applications successives de [AD] :

$$E^- = \bigcup_{i \in I^-} e_i^- \subseteq \bigcup_{i \in I} e_i = E$$

cqfd. ♦

Démonstration de $E^+ = \bigcup_{i \in I^+} e_i^+$: analogue.

Interprétation graphique du procédé (supposons m = 2)

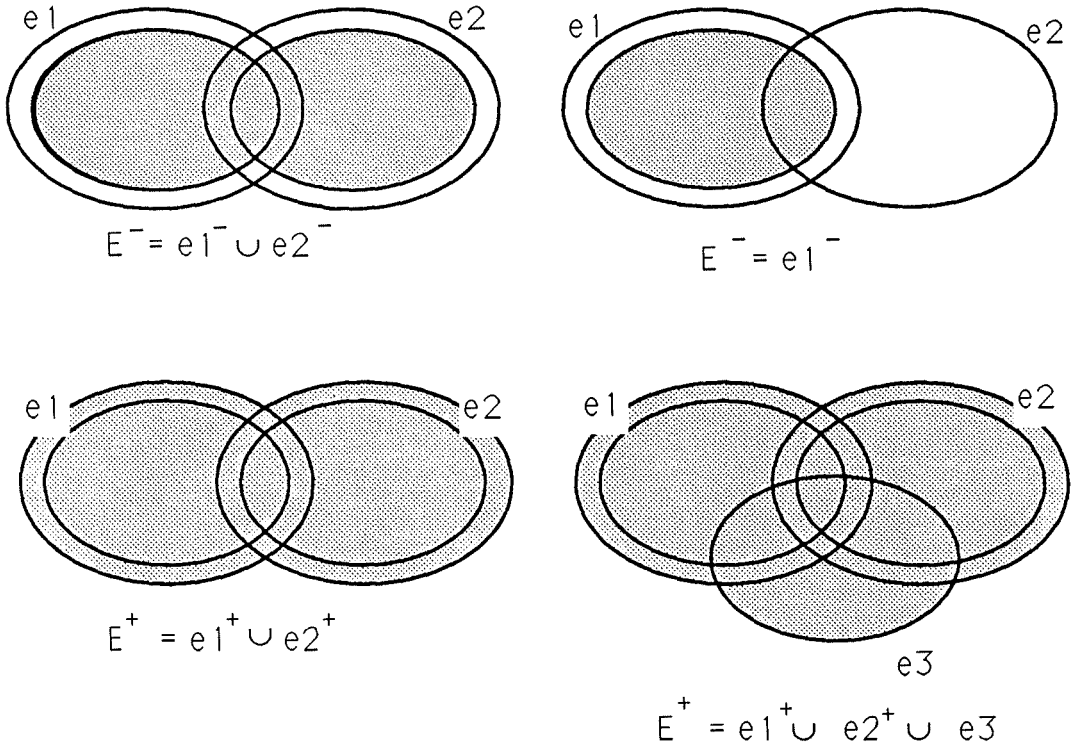


Figure D.1. - Interprétation graphique

Théorème : soit $E = C(\cup_{i\in I} e_i)$, on a $E^- = C(\cup_{i\in I^+} e_i^+)$ ♦

Démonstration :

par définition :

$$\forall\, i \in I : e_i \subseteq e_i^+$$

par (m - 1) applications successives de [A$\cup$] :

$$\cup_{i\in I} e_i \subseteq \cup_{i\in I} e_i^+$$

par ($\#(I^+ \setminus I)$) applications successives de [AD] :

$$\cup_{i\in I} e_i \subseteq \cup_{i\in I^+} e_i^+$$

[TP] :

$$E^- = C(\cup_{i\in I^+} e_i^+) \subseteq C(\cup_{i\in I} e_i) = E$$

cqfd. ♦

Démonstration de $E^+ = C(\cup_{i\in I^-} e_i^-)$: analogue.

Interprétation graphique du procédé (supposons m = 2)

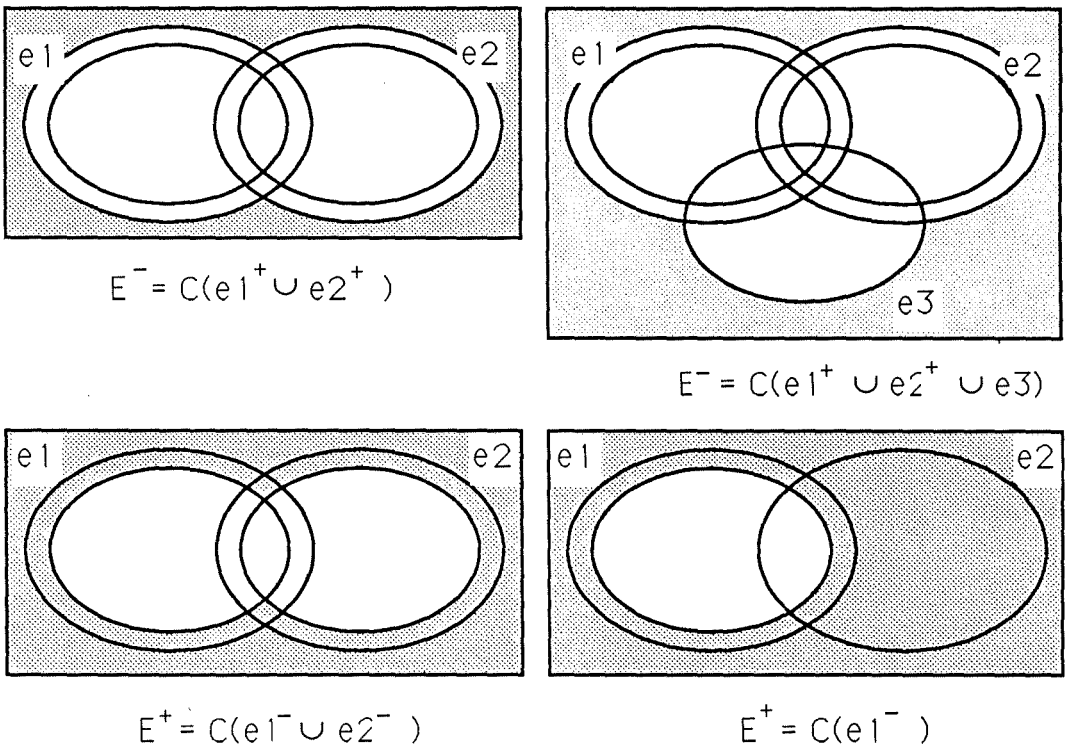


Figure D.2. - Interprétation graphique

Théorème : soit $E = \bigcap_{i \in I} e_i$, on a $E^- = \bigcap_{i \in I^+} e_i^-$ ♦

Démonstration :

par définition :

$$\forall\, i \in I : e_i^- \subseteq e_i$$

par (m - 1) applications successives de [A$\bigcap$] :

$$\bigcap_{i \in I} e_i^- \subseteq \bigcap_{i \in I} e_i$$

par ($\#(I^+ \setminus I)$) applications successives de [AG] :

$$E^- = \bigcap_{i \in I^+} e_i^- \subseteq \bigcap_{i \in I} e_i = E$$

cqfd. ♦

Démonstration de $E^+ = \bigcap_{i \in I^-} e_i^+$: analogue.

Interprétation graphique du procédé (supposons m = 2)

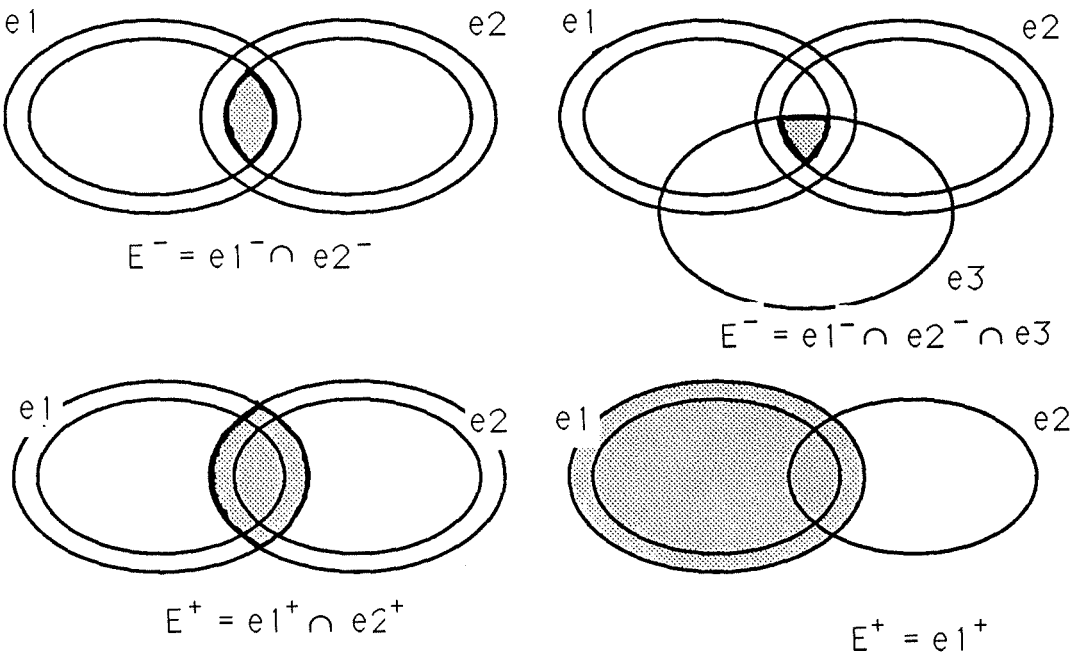


Figure D.3. - Interprétation graphique

Théorème : soit $E = C(\bigcap_{i\in I} e_i)$, on a $E^- = C(\bigcap_{i\in I^-} e_i^+)$

Démonstration :

par définition :

$$\forall\, i \in I^- : e_i \subseteq e_i^+$$

par $(\#(I^-) - 1)$ applications successives de $[A\bigcap]$:

$$\bigcap_{i\in I^-} e_i \subseteq \bigcap_{i\in I^-} e_i^+$$

par $(\#(N^-))$ applications successives de [AG] :

$$\bigcap_{i\in I} e_i \subseteq \bigcap_{i\in I^-} e_i^+$$

[TP] :

$$E^- = C(\bigcap_{i\in I^-} e_i^+) \subseteq C(\bigcap_{i\in I} e_i) = E$$

cqfd. ♦

Démonstration de $E^+ = C(\bigcap_{i\in I^+} e_i^-)$: analogue.

Interprétation graphique du procédé (supposons m = 2)

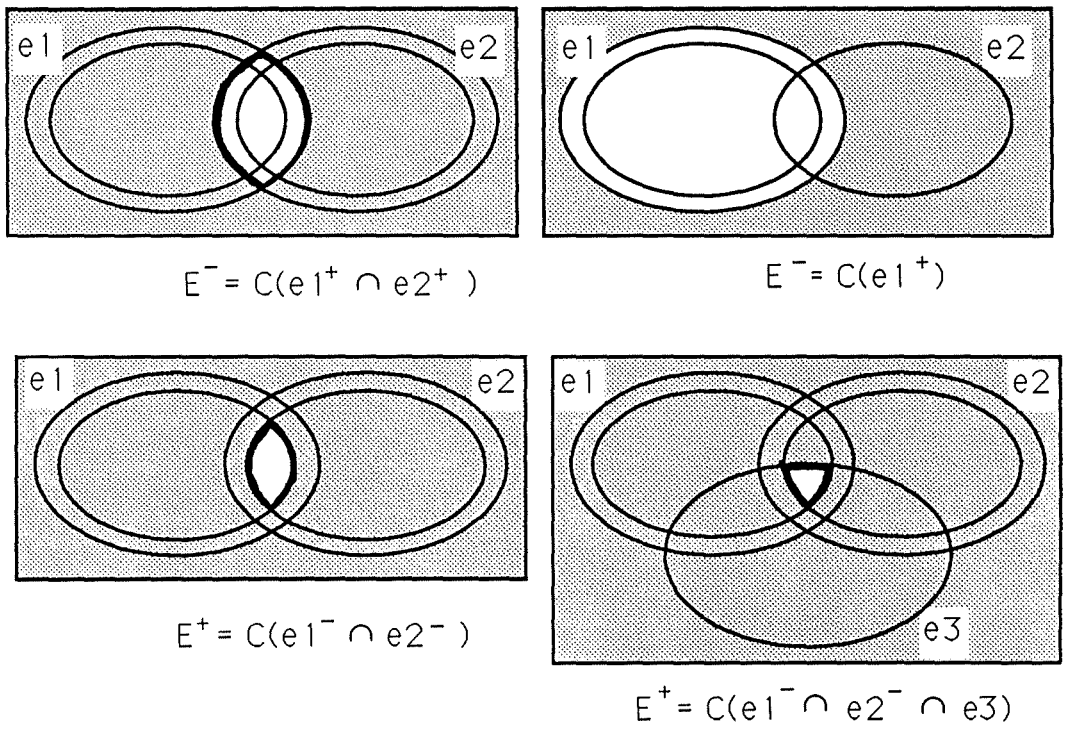


Figure D.4. - Interprétation graphique

Annexe E

Pseudocodes des algorithmes du système de protection de NOMADE

```
procedure IAddRight ( Target , L , Ne )                    { version of section 4.1.2.2. }
if Target ∈ BD [ ObjName ]                                                  { condition (1) }
   then if ValidLabel ( L ) ∧ ValidNe ( Ne )                                { condition (2) }
           then if NeImpliesLsNe(DisjNormForm(Ne),
                        first(LoginRights[LsOwnNe,(L≡Label)]))
                   ∧ NeImpliesLsNe(DisjNormForm(Ne),
                        first(LoginRights[LsInhNe,(L≡Label)]))              { condition (3) }
                   then TargetMan ← BD [ ( ObjName = Target ) , Man ]
                        if L ∈ TargetMan.LsRight [ Label ]
                           then TargetMan.LsRight[LsNe,(Label=L)] ∪= Ne
                           else TargetMan.LsRight ∪= (L,(Ne))
                        fi
                   else exit ( "Unable to prove that this right is a subset of your effective
                                rights" )
                fi
           else exit ( "Invalid parameter(s)" )
        fi
   else exit ( "Non existent target" )
fi

procedure IDelRight ( Target , L , Ne )                    { version of section 4.1.2.3. }
if Target ∈ BD [ ObjName ]                                                  {condition (1) }
   then TargetMan ← BD [ ( ObjName = Target ) , Man ]
        if L ∈ TargetMan.LsRight [ Label ]                                  { condition (5) }
           then if Ne ∈ TargetMan.LsRight[LsNe,(Label=L)]                   { condition (6) }
                   then if NeImpliesLsNe(DisjNormForm(Ne),
                                first(LoginRights[LsOwnNe,(L≡Label)]))
                           ∧ NeImpliesLsNe(DisjNormForm(Ne),
                                first(LoginRights[LsInhNe,(L≡Label)]))
                                                                            { condition (3) }
                           then if size(TargetMan.LsRight[LsNe,(Label=L)])=1
                                   then TargetMan.LsRight \= (L,(Ne))
                                   else TargetMan.LsRight[LsNe,(Label=L)] \= Ne
                                fi
                           else exit ( "Unable to prove that this right is a
                                        subset of your effective rights" )
                        fi
                   else exit ( "Non existent NOMADE-expression for this label" )
                fi
           else exit ( "Non existent label" )
        fi
   else exit ( "Non existent target" )
fi
```

```
procedure IChgRight ( Target , L , Ne1 , Ne2 )                { version of section 4.1.2.4. }
if Target ∈ BD [ ObjName ]                                                    { condition (1) }
   then if ValidNe ( Ne2 )                                                    { condition (2) }
           then TargetMan ← BD [ ( ObjName = Target ) , Man ]
                if L ∈ TargetMan.LsRight [ Label ]                           { condition (5) }
                   then if Ne1 ∈ TargetMan.LsRight[LsNe,(Label=L)]           { condition (6) }
                              then if LsNeImpliesLsNe(DisjNormFormL(Δ(Ne1,Ne2)),
                                                first(LoginRights[LsOwnNe,(L≡Label)]))
                                        ∧ LsNeImpliesLsNe(DisjNormFormL(Δ(Ne1,Ne2)),
                                                first(LoginRights[LsInhNe,(L≡Label)]))
                                                                             { condition (7) }
                                     then TargetMan.LsRight[LsNe,(Label=L)] \= Ne1
                                          TargetMan.LsRight[LsNe,(Label=L)] U= Ne2
                                     else exit ( "Unable to prove that this right is
                                                        a subset of your effective rights" )
                                   fi
                              else exit ( "Non existent NOMADE-expression for this label" )
                           fi
                   else exit ( "Non existent label" )
                fi
           else exit ( "Invalid NOMADE-expression" )
        fi
   else exit ( "Non existent target" )
fi

function IChkRight ( Lbl , QN ) : boolean
if size ( LoginRights [ Label ≡ Lbl ] ) > 0
   then return(QNImpliesLsNe(QN,first(LoginRights[(Label ≡ Lbl),LsOwnNe])
             ∧ QNImpliesLsNe(QN,first(LoginRights[(Label ≡ Lbl),LsInhNe]))
   else return ( false )
fi
```

```
function InitRights (User) : LoginRight
LoginRights ← ()
{ load User.manrights }
for each OR ∈ BD[(ObjName = User), Man].LsRight do
   LoginRights ∪= (OR.Label, DisjNormFormL (OR.LsNe), (), true, false)
od
{ process inherited rights }
LsPart ← GetLsEmbPart (User)
for each (R, U) ∈ LsPart do                    { multiple inheritance : User ∈ /R.U/ }
   StopInh ← false
   for each LR ∈ LoginRights do
      LR.InhPoss ← true                        { always true : stopinh not allowed }
   od
   repeat                                                    { linear inheritance }
      for each LR ∈ LoginRights do
         LR.Already ← false
      od
      MergeRights (LoginRights, BD[(ObjName = /R.U/), Man].LsRight, StopInh)
      (R, U) ← GetEmbPart (R, U)
   until (R, U) = ("", "") v StopInh
od
return (LoginRights)

procedure MergeRights (LoginRights, PartRights, StopInh)
{ merge PartRights into LoginRights and change StopInh if needed }
StarRight ← ∅
for each PR ∈ PartRights do
   case PR.Label of
      "*" : StarRight ← PR                     { will be processed separately }
      "stopinh" : StopInh ← true
      otherwise :
         if PR.Label ∈ LoginRights [Label]
            then    LR ← LoginRights (Label = PR.Label)
                    LR.Already ← true
                    if LR.InhPoss
                       then Process (PR.LsNe, LR)
                       else
                    fi
            else SR ← LoginRights (Label = "*")
                 if SR.InhPoss
                    then LR ← (PR.Label, DisjNormFormL (SR.LsOwnNe),
                                 DisjNormFormL (SR.LsInhNe), true, true)
                         Process (PR.LsNe, LR)
                         LoginRights ∪= LR
                 fi
         fi
   esac
od
```

```
if StarRight ≠ ∅
   then for each LR ∈ LoginRights do
            if (~ LR.Already) ∧ (LR.InhPoss)
               then Process (StarRight.LsNe, LR)
               else
            fi
         od
   else
fi

procedure Process (PartLsNe, LR)
{ LR.LsInhNe U= PartLsNe [ \ "stopinh" ]     and switch LR.InhPoss to false if needed }
if "stopinh" ∈ PartLsNe
   then    LR.InhPoss ← false                                  { handle multiple "stopinh"s }
           Without ← PartLsNe \ "stopinh"
           while "stopinh" ∈ Without do
              Without \= "stopinh"
           od
           LR.LsInhNe U= DisjNormFormL (Without)
   else LR.LsInhNe U= DisjNormFormL (PartLsNe)
fi

function LsNeImpliesLsNe ( LsNe_A , LsNe_B ) : boolean
for each Ne_i ∈ LsNe_A do
   if NeImpliesLsNe ( Ne_i , LsNe_B )
      then
      else return ( false )
   fi
od
return ( true )

function NeImpliesLsNe ( Ne_A , LsNe_B ) : boolean
for each LsQN_i ∈ Ne_A.LsLsQN do
   if LsQNImpliesLsNe ( LsQN_i , LsNe_B )
      then
      else return ( false )
   fi
od
return ( true )

function NeImpliesNe ( Ne_A , Ne_B ) : boolean
{ pre-condition : disjunctive normal form }
for each LsQN_i ∈ Ne_A.LsLsQN do
   if LsQNImpliesNe ( LsQN_i , LsNe_B )
      then
      else return ( false )
   fi
od
return ( true )
```

```
function LsQNImpliesLsNe ( LsQN_A , LsNe_B ) : boolean
for each Ne_j ∈ LsNe_B do
   if LsQNImpliesNe ( LsQN_A , Ne_J )
      then return ( true )
      else
   fi
od
return ( false )

function LsQNImpliesNe ( LsQN_A , Ne_B ) : boolean
for each LsQN_j ∈ Ne_B.LsLsQN do
   if LsQNImpliesLsQN ( LsQN_A , LsQN_J )
      then return ( true )
      else
   fi
od
return ( false )

function LsQNImpliesLsQN ( LsQN_A , LsQN_B ) : boolean
{ pre-condition : disjunctive normal form }
for each QN_j ∈ LsQN_B.LsQN do
   if LsQNImpliesQN ( LsQN_A , QN_J )
      then
      else return ( false )
   fi
od
return ( true )

function LsQNImpliesQN ( LsQN_A , QN_B ) : boolean
for each QN_i ∈ LsQN_A.LsQN do
   if QNImpliesQN ( QN_i , QN_B )
      then return ( true )
      else
   fi
od
return ( false )

function QNImpliesQN ( QN_A , QN_B ) : boolean
case (QN_A.Sign , QN_B.Sign ) of
   (yes , yes) : return ( NImpliesN ( QN_A.N , QN_B.N )
                        ∧ LsQImpliesLsQ (QN_A.LsQ , QN_B.LsQ) )
   (no , no)   : return ( NImpliesN ( QN_B.N , QN_A.N )
                        ∧ LsQImpliesLsQ (QN_B.LsQ,QN_A.LsQ) )
   (yes , no)  : return ( NImpliesNotN ( QN_A.N , QN_B.N )
                        v LsQImpliesNotLsQ ( QN_A.LsQ , QN_B.LsQ ) )
   (no , yes)  : return ( NotNImpliesN ( QN_A.N , QN_B.N )
                        ∧ NotLsQImpliesLsQ ( QN_A.LsQ , QN_B.LsQ ) )
esac
```

```
function NImpliesN ( N_A , N_B ) :  boolean
case (N_A , N_B) of
   (OE_A , OE_B)                  :  return ( OE_A ≡ OE_B )
   (OE_A , /R_B.A_B/OE_B)         :  if ContMetaChar ( OE_A )
                                        then return ( false )
                                        else return ( (OE_A ∈ /R_B.A_B/) ∧ (OE_A ≡ OE_B) )
                                     fi
   (OE_A , /R_B.A_B/)             :  if ContMetaChar ( OE_A )
                                        then return ( false )
                                        else return ( OE_A ∈ /R_B.A_B/ )
                                     fi
   (/R_A.A_A/OE_A , OE_B)         :  return ( OE_A ≡ OE_B )
   (/R_A.A_A/OE_A , /R_B.A_B/OE_B) :  if ContMetaChar ( OE_A )
                                        then return (((R_A,A_A)=(R_B,A_B)) ∧ (OE_A≡OE_B))
                                        else return ((OE_A ∈ /R_B.A_B/) ∧ (OE_A≡OE_B))
                                     fi
   (/R_A.A_A/OE_A , /R_B.A_B/)    :  if ContMetaChar ( OE_A )
                                        then return ( (R_A,A_A) = (R_B,A_B) )
                                        else return ( OE_A ∈ /R_B.A_B/ )
                                     fi
   (/R_A.A_A/ , OE_B)             :  return ( OE_B = ** )
   (/R_A.A_A/ , /R_B.A_B/OE_B)    :  return ( ( (R_A,A_A) = (R_B,A_B) ) ∧ (OE_B = **) )
   (/R_A.A_A/ , /R_B.A_B/)        :  return ( (R_A,A_A) = (R_B,A_B) )
esac

function NImpliesNotN ( N_A , N_B ) : boolean
case (N_A , N_B) of
   (OE_A , OE_B)                  :  if ContMetaChar ( OE_A )
                                        then if ContMetaChar ( OE_B )
                                                then return ( false )
                                                else return ( OE_B ≈ OE_A )
                                             fi
                                        else return ( OE_A ≈ OE_B )
                                     fi
   (OE_A , /R_B.A_B/OE_B)         :  if ContMetaChar ( OE_A )
                                        then if ContMetaChar ( OE_B )
                                                then return ( false )
                                                else return ( OE_B ≈ OE_A )
                                             fi
                                        else return ((OE_A ∉ /R_B.A_B/) v (OE_A ≈ OE_B))
                                     fi
   (OE_A , /R_B.A_B/)             :  if ContMetaChar ( OE_A )
                                        then return ( false )
                                        else return (OE_A ∉ /R_B.A_B/)
                                     fi
```

```
   (/R_A.A_A/OE_A , OE_B)          : if ContMetaChar ( OE_B )
                                       then if ContMetaChar ( OE_A )
                                                then return ( false )
                                                else return ( OE_A ≈ OE_B )
                                            fi
                                       else return ((OE_B ∉ /R_A.A_A/) v (OE_B ≈ OE_A ))
                                     fi
   (/R_A.A_A/OE_A , /R_B.A_B/OE_B) : if ContMetaChar ( OE_A )
                                       then if ContMetaChar ( OE_B )
                                                then return ( false )
                                                else return ((OE_B ∉ /R_A.A_A/)
                                                             v (OE_B ≈ OE_A ))
                                            fi
                                       else return ((OE_A ∉ /R_B.A_B/) v (OE_A ≈ OE_B ))
                                     fi
   (/R_A.A_A/OE_A , /R_B.A_B/)     : if ContMetaChar ( OE_A )
                                       then return ( false )
                                       else return (OE_A ∉ /R_B.A_B/)
                                     fi
   (/R_A.A_A/ , OE_B)              : if ContMetaChar ( OE_B )
                                       then return ( false )
                                       else return (OE_B ∉ /R_A.A_A/)
                                     fi
   (/R_A.A_A/ , /R_B.A_B/OE_B)     : if ContMetaChar ( OE_B )
                                       then return ( false )
                                       else return (OE_B ∉ /R_A.A_A/)
                                     fi
   (/R_A.A_A/ , /R_B.A_B/)         : return ( false )
esac

function NotNImpliesN ( N_A , N_B ) : boolean
case (N_A , N_B) of
   (OE_A , OE_B)                   : if ContMetaChar ( OE_B )
                                       then return ( OE_B = ** )
                                       else return ( OE_A = ** )
                                     fi
   (OE_A , /R_B.A_B/OE_B)          : if ContMetaChar ( OE_B )
                                       then return ( false )
                                       else return ( OE_A = ** )
                                     fi
   (OE_A , /R_B.A_B/)              : return ( false )
   (/R_A.A_A/OE_A , OE_B)          : return ( OE_B = ** )
   (/R_A.A_A/OE_A , /R_B.A_B/OE_B) : return ( false )
   (/R_A.A_A/OE_A , /R_B.A_B/)     : return ( false )
   (/R_A.A_A/ , OE_B)              : return ( OE_B = ** )
   (/R_A.A_A/ , /R_B.A_B/OE_B)     : return ( false )
   (/R_A.A_A/ , /R_B.A_B/)         : return ( false )
esac
```

```
function LsQImpliesLsQ ( LsQA , LsQB ) : boolean
for each Qj ∈ LsQB do
   if LsQImpliesQ ( LsQA , QJ )
      then
      else return ( false )
   fi
od
return ( true )

function LsQImpliesNotLsQ ( LsQA , LsQB ) : boolean
for each Qj ∈ LsQB do
   if LsQImpliesQ ( LsQA , ~QJ )
      then return ( true )
      else
   fi
od
return ( false )

function NotLsQImpliesLsQ ( LsQA , LsQB ) : boolean
for each Qj ∈ LsQB do
   if NotLsQImpliesQ ( LsQA , QJ )
      then
      else return ( false )
   fi
od
return ( true )

function LsQImpliesQ ( LsQA , QB ) : boolean
for each Qi ∈ LsQA do
   if QImpliesQ ( Qi , QB )
      then return ( true )
      else
   fi
od
return ( false )

function NotLsQImpliesQ ( LsQA , QB ) : boolean
for each Qi ∈ LsQA do
   if QImpliesQ ( ~Qi , QB )
      then
      else return ( false )
   fi
od
return ( true )
```

```
function QImpliesQ ( QA , QB ) : boolean
if QA.Attribute ≡ QB.Attribute
   then for each Vi ∈ LsVA do
           if VInLsV ( RelOpA , Vi , RelOpB , LsVB )
              then
              else return ( false )
           fi
        od
        return ( true )
   else return ( false )
fi

function VInLsV ( RelOpA , VA , RelOpB , LsVB ) : boolean
for each VJ ∈ LsVB do
   if ContMetaChar ( VA ) v ContMetaChar ( VJ)
      then case ( RelOpA , RelOpB ) of
                (= , =)   : return (VA ≡ Vj)
                (≠ , ≠)   : return (Vj ≡ VA)
                (= , ≠)   : return (VA ≈ Vj)
                (≠ , =)   : return (VA ≈ Vj)
                otherwise : return (false)
           esac
      else if VA < VJ
              then return ( QIQ [ RelOpA , RelOpB , < ] )
              else if VA > VJ
                      then return  ( QIQ [ RelOpA , RelOpB , > ] )
                      else return  ( QIQ [ RelOpA , RelOpB , = ] )
                   fi
           fi
od
return (false)

function QNImpliesLsNe ( QN , LsNe ) : boolean
for each Ne ∈ LsNe do
   if QNImpliesNe ( QN , Ne )
      then return ( true )
      else
   fi
od
return ( false )

function QNImpliesNe ( QN , Ne ) : boolean
for each LsQN ∈ Ne.LsLsQN do
   if QNImpliesLsQN ( QN , LsQN )
      then return ( true )
      else
   fi
od
return ( false )
```

```
function QNImpliesLsQN ( QN_A , LsQN_B ) : boolean
for each QN_j ∈ LsQN_B.LsQN do
   if QNImpliesQN ( QN_A , QN_J )
      then
      else return ( false )
   fi
od
return ( true )

function DisjNormFormL ( LsNe ) : list of NOMADEexpression
ResultLsNe ← ""
for each Ne ∈ LsNe do
   ResultLsNe or= DisjNormForm ( Ne )
od
return ( ResultLsNe )

function DisjNormForm ( Ne ) : NOMADEexpression
ResultNe ← ""
if Ne.Sign = no
   then { steps A and B simultaneously }
        Conj ← ""
        for each LsQN_1 ∈ Ne.LsLsQN do
           if LsQN_1.Sign = no
              then Conj and= LsQN_1.LsQN
              else if size ( LsQN_1.LsQN ) = 1
                      then Conj and= ChgSign ( first ( LsQN_1.LsQN ) )
                      else for I ← 1 to size ( LsQN_1.LsQN ) - 1 do
                              ResultNe or= ResultNe
                           od
                           for each QN_1 ∈ LsQN_1.LsQN do
                              for each LsQN_2 ∈ ResultNe.LsLsQN do
                                 LsQN_2.LsQN and= ChgSign ( QN_1 )
                              od
                           od
                   fi
           fi
        od
        for each LsQN_2 ∈ ResultNe.LsLsQN do
           LsQN_2.LsQN and= Conj
        od
   else { step A only }
        for each LsQN ∈ Ne.LsLsQN do
           if LsQN.Sign = no
              then for each QN ∈ LsQN.LsQN do
                      ResultNe or= QNToLsQN ( ChgSign ( QN ) )
                   od
              else ResultNe or= LsQN
           fi
        od
fi
return ( ResultNe )
```

function *ValidLabel* (L) : **boolean**
return ((L = "* ") v (~ ContMetaChar (L)))

function Δ ($LsNe_A$, $LsNe_B$) : **list of** NOMADEexpression
{ simplified version }
return ($LsNe_A$ **or** $LsNe_B$)

REPARTITION DU TRAVAIL DE CONCEPTION ET DE REDACTION

REPARTITION DU TRAVAIL DE CONCEPTION ET DE REDACTION

Tout le travail de **conception** de ce mémoire a été effectué conjointement par les **deux** auteurs. Le travail de **rédaction** se répartit comme suit ("C" signifie "rédaction commune"; "PF" signifie "rédigé par Pierre FLENER"; "BJ" signifie "rédigé par Bernard JABAS") :