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

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Analyse d'une trace Oblog grâce à l'outil ASAX

Limet, Jean-Louis

Award date:
1995

Awarding institution:
Universite de Namur

Link to publication

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

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

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

Download date: 19. Apr. 2024

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue Grandgagnage, 21, 5000 Namur

Analyse d'une trace Oblog grâce à l'outil ASAX

Limet Jean-Louis

Promoteur: *Monsieur Naji Habra*

Mémoire de Licence et Maîtrise en Informatique

Année académique 1994-1995

Je tiens à exprimer ma profonde gratitude à Naji Habra, promoteur de ce mémoire, pour son aide efficace et constante dans l'organisation et la structuration de ce travail, ainsi que pour ses encouragements et ses commentaires constructifs .

Je suis particulièrement reconnaissant à Abdelaziz Mounji pour la disponibilité, la sympathie et la productivité avec lesquelles il fut mon initiateur dans l'environnement ASAX.

Je tiens à souligner le soutien très important, à la fois intellectuel et technique que m'a prodigué Jean Marc Zeippen dans la poursuite des objectifs de ce mémoire et dans ma découverte du monde Oblog.

Je remercie également mes professeurs, Eric Dubois et Baudouin Le Charlier, pour leurs conseils et remarques toujours judicieuses.

Enfin, merci du fond du coeur à mon épouse, Anne-Françoise et à Nicolas, Manu et Simon pour l'affection, la patience et le soutien qu'ils m'ont accordé dans la réalisation de ce travail.

Sommaire

PARTIE 1 : INTRODUCTION ET PLAN DU MEMOIRE 1

1.1 INTRODUCTION 1
1.2 PLAN DU MEMOIRE 3

PARTIE 2 : OBLOG 5

2.1 INTRODUCTION: L'APPROCHE OBLOG 5
2.1.1 Stratégie 5
2.1.2 Concepts primitifs de l'approche 5
2.1.3 Oblog Case Version 1.0 7
2.2 L'OBJET OBLOG: UNE VUE EN TROIS DIMENSIONS 8
2.2.1 La perspective statique 8
2.2.2 La perspective dynamique 9
2.1.3 La perspective comportement 9
2.3 LE MODELE OBLOG ET SON LANGAGE 10
2.3.1 L'objet Oblog 10
2.3.2 Le système d'objets OBLOG 18
2.3.3 Le langage d'expression OBLOG 23
2.3.4 L'interface d'un objet OBLOG 25
2.4 L'OUTIL OBLOG CASE V1.2 27
2.5 OBLOG COMME SUPPORT DES ETAPES DE DEVELOPPEMENT: EVALUATION 27
2.6 CONCEPTION DE L'APPLICATION MONDE BANCAIRE 31
2.7 CONCLUSION 47

PARTIE 3 : ASAX 48

3.1 L'ANALYSE D'UN SYSTEME VIA SA TRACE 48
3.1.1 Contexte et intérêt d'une analyse de trace 48
3.1.2 Analyse d'une trace: problèmes, approches et spécifications d'outil. 48
3.2 OBJECTIFS DU PROJET ASAX 50
3.3 INTRODUCTION AU LANGAGE RUSSEL 51
3.4 DEFINITION LOGIQUE DU LANGAGE RUSSEL 55
3.4.1 Syntaxe abstraite sommaire d'une règle Russel 55
3.4.2 Quelques définitions 56
3.4.3 Sémantique partielle du langage 56
3.5 LE PROTOTYPE ASAX 62
3.5.1 Le format NADF et le Format Adaptator 62
3.5.2 La librairie des procédures préprogrammées 62
3.5.3 Description d'une session ASAX 62
3.6 L'EFFICACITE DE ASAX 63
3.7 LES EXTENSIONS PRECONISEES DE ASAX 63

PARTIE 4 : TRACE OBLOG ET FORMAT ADAPTATOR 64

4.1 LE PROBLEME DE TRADUCTION 64
4.2 LA TRACE OBLOG 65
4.2.1 Contexte et structure globale 65
4.2.2 Commentaires sur la trace Oblog du point de vue conceptuel 69
4.2.3 Syntaxe d'une trace Oblog 71
4.3 DESCRIPTION DETAILLEE DU FORMAT CIBLE NADF 72
4.4 LE FORMAT ADAPTATOR 74
4.4.1 La génération de l'Audit Data Description File de l'application Monde Bancaire 74
4.4.2 La traduction du format natif en format NADF 77
4.4.2.1 Squelette général du corps d'un programme Format Adaptator: 77
4.4.2.2 Choix de granularité de l'opération de traduction 78
4.4.2.3 Construction progressive du programme Format Adaptator 79
4.4.3 Commentaires concernant le Format Adaptator 93

PARTIE 5 : ANALYSE D'UNE TRACE OBLOG VIA ASAX 94

5.1 INTERET D'UNE ANALYSE DE TRACE OBLOG ET PISTES D'ANALYSE 94
5.1.1 La mise au point et le contrôle de la Base de données. 94
5.1.2 Le suivi de l'utilisation réelle de l'application. 94
5.1.3 Le support au développement, aux tests et à la maintenance de l'application. 95
5.1.4 Les mesures quantitatives du comportement d'un système. 95
5.2 ANALYSE D'UNE TRACE OBLOG: PROBLEMATIQUE ET CHOIX D'UN OUTIL 100
5.3 EXEMPLES DE REGLES RUSSEL POUR L'ANALYSE D'UNE TRACE OBLOG 102
5.3.1 Exemple 0 : Produire et compter les événements élémentaires de la trace 102
5.3.2 Exemple 1 : Suivi d'un service macroscopique 104
5.3.3 Exemple 2 : Isoler le comportement des objets d'une classe donnée 108
5.3.4 Exemple 3 : Détecter des tentatives de retrait d'argent suspectes 109
5.3.5 Exemple 4 : Suivi de l'utilisation de la fonction « sélection de clients » 111
5.3.6 Exemple 5 : Utilisation des codes personnels pour les retraits d'argent 113
5.3.7 Exemple 6 : Sélection des événements touchant des objets DBX 115
5.3.8 Exemple 7 : Statistiques sur les interactions directes 116
5.3.9 Exemple 8 : Statistiques sur les événements élémentaires (transitions) 119
5.3.10 Idées d'exemples complémentaires 121
5.4 COMMENTAIRES 122

PARTIE 6 : CONCLUSION ET EXTENSIONS 124

6.1 CONCLUSION 124
6.2 AMELIORATIONS, EXTENSIONS ET NOUVELLES PERSPECTIVES 126

BIBLIOGRAPHIE 130

ANNEXES 132

A) LE LANGAGE REXX 132
B) ELEMENTS DE REXX POUR LA LECTURE DU PROGRAMME FORMAT ADAPTATOR (EXEMPLES) 132
C) QUELQUES ELEMENTS COMPLEMENTAIRES DE LA SPECIFICATION DE MONDE BANCAIRE 140

Partie 1 : INTRODUCTION ET PLAN DU MEMOIRE

1.1 Introduction

Lors de ces dix dernières années sont apparus des modèles abstraits qui décrivent un Système d'Information comme une collection d'objets actifs qui coopérent pour fournir les services attendus du système.
Dans ces modèles, un objet peut être vu localement comme une entité à laquelle sont attachées des propriétés et les règles d'évolution de ces propriétés. Mais il peut également être vu comme un acteur d'une communauté dans laquelle il peut évoluer de manière autonome mais aussi en interaction avec les autres.

Ces modèles sont intéressants pour l'informaticien car ils permettent d'intégrer sur base du concept d'objet actif des spécifications du système généralement éparpillées dans les traditionnels modèles de structuration des données, modèles de la statique des traitements et modèles de la dynamique des traitements.
D'autre part, ces modèles s'avèrent suffisamment polyvalents et expressifs que pour supporter toutes les étapes du cycle de vie d'une application (de l'analyse des besoins à la maintenance).
Ces modèles permettent donc à la fois l'unification des concepts et perspectives de spécification d'un Système d'Information mais aussi leur réutilisation (avec des objectifs éventuellement distincts) dans toute les étapes du cycle de développement.

La vie d'un système opérationnel (prototype ou système terminal) conçu selon un de ces modèles peut être décrite comme une séquence d'événements qui sont des stimuli permettant aux objets du système d'évoluer. Un événement peut concerner un seul objet (événement isolé) ou plusieurs objets de manière simultanée (interaction par partage d'événements). Un événement peut trouver son origine à l'intérieur du système (initiative d'un des objets) ou à l'extérieur du système (utilisateur externe).
Un système opérationnel peut être conçu de manière à conserver une trace de cette séquence d'événements en précisant pour chacun d'entre eux le contexte, les objets impliqués et les conséquences sur leurs états respectifs.

Il apparaît que, quelque soit la qualité de la spécification originale, la vie d'un tel système devient très vite complexe à analyser pour deux raisons essentielles:

- la richesse des scénarii de vie possibles eu égard au comportements indépendants des objets et aux interactions multiples;

- le comportement très peu prévisible des utilisateurs du système.

Cette analyse semble toutefois indispensable pour évaluer le bon fonctionnement du système (c'est à dire un comportement qui assure la poursuite des objectifs du système).

Une idée intéressante serait de mener cette analyse grâce à un outil automatisé qui prendrait tout naturellement comme argument la trace produite par le système étudié.

L'outil ASAX, développé initialement par l'Institut d'Informatique de Namur pour favoriser la sécurité d'un système d'exploitation en supportant l'analyse de sa trace (appelée Audit Trail dans ce contexte), est tout indiqué pour mener cet idée à bien.
En effet, ASAX est capable de traiter en principe tout format de trace moyennant l'écriture d'un programme Format Adaptator qui traduise la trace native dans une trace de format normalisé.
D'autre part, le langage Russel, associé à l'outil, permet de détecter facilement et efficacement des scénarii longs et complexes dans des traces de très grande taille.

Pour matérialiser cette idée, nous nous sommes intéressés plus particulièrement aux traces produites par des systèmes spécifiés grâce au modèle Oblog.
Ce modèle est très représentatif de la classe des modèles d'objets actifs communiquant par événements partagés. Un outil logiciel associé appelé Oblog Case assiste la démarche de spécification et permet de générer automatiquement le système opérationnel à partir des spécifications finales. Il est possible de lancer l'exécution du système en demandant la production d'une trace associée.

L'objectif concret de ce mémoire consiste à étudier l'intérêt et la faisabilité de l'analyse, grâce à l'outil ASAX, de traces produites par un système Oblog.

Nous tâcherons d'atteindre cet objectif en trois étapes:

- construction d'un programme Format Adaptator de manière à traduire toute trace Oblog dans un format normalisé exploitable par ASAX;

- recherche de domaines d'intérêt susceptibles de guider une analyse de trace Oblog;

- construction de règles Russel pour l'analyse de la trace d'une application Oblog particulière afin d'illustrer la faisabilité et les domaines d'intérêt.

Plus généralement, ce travail doit être vu comme une **expérience** destinée à illustrer l'**intérêt** mais aussi les **limites**, les **critères de choix** et les **possibilités d'évolution** d'une **démarche d'analyse**, supportée par l'outil **ASAX**, d'un **système d'objets actifs communiquant par événements partagés**, au travers de sa **trace**.

1.2 Plan du mémoire

Oblog

Dans la deuxième partie du mémoire, nous présenterons l'approche théorique Oblog (OBject LOGic).
Cette approche propose une stratégie de développement d'un Système d'Information basée sur l'adoption du même langage de spécification dans toutes les étapes du cycle de vie de l'application. L'idée est d'augmenter la productivité du processus de développement en supprimant les discontinuités de langage entre les différentes étapes du cycle de vie.
La démarche préconisée consiste à ne travailler qu'avec des spécifications de plus en plus raffinées au fil du processus de développement. La spécification terminale devrait être suffisamment détaillée que pour permettre une génération automatique du code de l'application. Idéalement, chaque spécification intermédiaire devrait être prototypable.

Le développement du langage de l'approche Oblog a démarré en 1986 sous l'impulsion du professeur Amilcar Sernadas de l'INESC (Instituto Nacional de Engenharia de Sistemas e de Computadores, Portugal) initialement comme un langage textuel de modélisation conceptuelle. Un intérêt commercial pour l'approche, apparu en 1989 a contribué à la définition de nouveaux objectifs pour l'approche et pour le langage: (i) définition d'une version diagrammatique du langage; (ii) stratégie d'utilisation du langage dans toutes les étapes du cycle de vie; (iii) génération automatique du code opérationnel à partir des spécifications Oblog.
A partir de 1990, la société ESDI a entamé le développement d'un outil logiciel d'aide au développement pour supporter l'approche et ses nouveaux objectifs. Ce produit porte aujourd'hui le nom de Oblog Case Version 1.2 et est distribué par Oblog Software S. A.
Parallèlement à cette évolution commerciale, l'équipe du professeur A. Sernadas continue à faire progresser la sémantique du langage et l'approche théorique.

Le modèle Oblog du Système d'Information, dont le langage permet d'exprimer les concepts est très représentatif de la classe des modèles d'objets actifs communiquant par événements partagés. Nous décrirons ce modèle avec l'intention d'isoler les concepts et perspectives propres à cette famille de modèles tout en montrant et en critiquant la manière dont ils sont traduits en Oblog.
Nous évaluerons également sommairement l'intérêt du modèle Oblog comme support des différentes étapes du cycle de développement.

Enfin nous présenterons les étapes de conception d'une application particulière (Le Monde Bancaire) grâce à l'outil Oblog Case. En effet, la spécification de cette application et les traces produites par le système opérationnel correspondant nous permettront d'illustrer l'intérêt et la faisabilité d'une analyse de trace Oblog supportée par l'outil ASAX.

ASAX

Dans la troisième partie du mémoire, nous réfléchirons à l'intérêt que peut présenter l'analyse de la trace d'un système informatique quelconque. Nous éclairerons les problèmes principaux et les types d'approches liés à cette forme d'analyse.

L'outil ASAX (**A**dvanced **S**ecurity **A**udit Trail Analysis on uni**X**), développé par l'Institut d'Informatique de Namur et Siemens Nixdorf Software S.A., possède des qualités essentielles pour supporter ce genre d'analyse.
Après avoir éclairé ces qualités, nous présenterons une définition logique de la syntaxe et de la sémantique du langage d'interrogation Russel qui lui est associé.

Finalement, nous décrirons l'environnement d'exécution de l'outil.

La trace Oblog et le Format Adaptator

Toute analyse ASAX s'exerce sur un fichier argument (trace) qui respecte un format normalisé appelé format NADF. Avant même de dégager l'intérêt et la faisabilité d'une analyse de trace Oblog grâce au langage Russel, il importe de s'assurer qu'une trace Oblog peut être traduite dans ce format normalisé.
D'autre part, cette traduction n'a de sens que si la trace présente une qualité d'information suffisante (eu égard aux concepts du modèle) pour envisager une analyse significative.

La quatrième partie de ce mémoire consistera en:
- une présentation détaillée et une évaluation des caractéristiques d'une trace Oblog;
- une description détaillée du format d'un fichier NADF;
- une construction progressive d'un programme Format Adaptator permettant de traduire toute trace Oblog dans un format NADF.

Analyse d'une trace Oblog via ASAX

Dans la cinquième partie de ce mémoire, nous essaierons de dégager l'intérêt que peut présenter l'analyse d'une trace Oblog pour le suivi, le contrôle et l'étude de l'application tracée.

Nous préciserons les problèmes caractéristiques posés par l'analyse d'une trace Oblog et la manière dont ASAX peut contribuer à les résoudre.

Plusieurs exemples de règles Russel viendront illustrer la puissance de Russel comme langage d'analyse d'une trace Oblog et les objectifs variés que peut poursuivre une telle analyse.

Enfin nous formulerons des critiques et des propositions d'améliorations.

Partie 2 : OBLOG

2.1 Introduction: L'approche OBLOG

2.1.1 Stratégie

L'approche Oblog propose une stratégie de développement de systèmes d'information basée sur les progrès récents dans les langages de spécification de haut niveau orientés objets. Cette stratégie est basée sur l'adoption d'un même langage de spécification dans toutes les étapes du cycle de vie de l'application (de l'analyse des besoins à la maintenance du système). Le code final est généré automatiquement à partir des spécifications et n'est jamais manipulé directement. Les spécifieurs (anciennement analystes et programmeurs) développent leur travail grâce à un outil d'aide au développement approprié: Oblog Case.
La démarche idéale consiste à ne travailler qu'avec des spécifications (de plus en plus raffinées au fil du processus de développement). A chaque étape, plus de détails sont incorporés dans la spécification et certains détails préalablement produits sont révisés. Enfin, le système opérationnel est généré automatiquement à partir des spécifications finales. De plus, à chaque instant du développement, il est possible de tester le système spécifié jusque là, de manière à adapter le produit au besoin (prototypage).

2.1.2 Concepts primitifs de l'approche[1]

L'approche Oblog propose une primitive sémantique centrale pour le langage support de cette stratégie: l' «Objet ». L'**Objet** doit être compris comme:

« Quelque chose qui existe (dans le monde réel ou dans le système informatique), qui est créé et éventuellement détruit; lorsqu'il existe, son **état** peut être changé par interaction avec d'autres entités; son état reflète la valeur de ses **attributs**; son état change lorsque des **événements** significatifs apparaissent ».

Pour illustrer cette définition, considérons un *compte bancaire* comme un objet. Un de ses attributs est le *solde*. *Débit* et *Crédit* sont ses événements pertinents.

Donc, spécifier un système correspond à la définition d'un ensemble d'objets. Généralement, il y a beaucoup d'objets semblables, du même type: ils sont groupés en **classes**. En Oblog, la spécification porte toujours sur les classes et non sur les objets qui en constituent les instances. Par exemple, nous spécifions la classe *compte bancaire* considérant qu'elle peut comporter beaucoup d'instances.

Le système spécifié est une communauté d'objets (instances de classes) qui interagissent les uns avec les autres: un *débit* fait par un *client* sur son *compte bancaire* est un événement qui apparaît à la fois pour le *client* (considéré comme un objet) et pour son *compte bancaire*

[1] Ces concepts seront affinés dans les deux chapitres qui suivent

(considéré comme un autre objet). Il est important de souligner que les objets de la communauté évoluent indépendamment les uns des autres, excepté dans les moments d'interaction: le système est perçu comme un ensemble d'objets concourants qui peuvent interagir. La même vue est valide pour le monde réel dans lequel le système est inséré.

Certains objets sont passifs (rien ne leur arrive, à moins que d'autres objets n'interagissent avec eux: c'est le cas du *compte bancaire*), d'autres sont actifs (évoluant de leur propre initiative: c'est le cas des clients de la banque).

Le langage Oblog permet la description des classes d'objets (actifs et passifs) et de leurs interactions.

La notion d'objet intègre les données (attributs) et les processus (événements). Ceci permet de structurer efficacement la spécification et le système décrit. Cette structuration correspond bien à la manière dont l'analyste perçoit la réalité. Ceci est un aspect essentiel des approches orientées objets.

Un autre aspect de ces approches concerne les constructeurs qui favorisent la réutilisation. Le type le plus familier de ces constructeurs est la spécialisation. Pour illustrer, revenons à l'exemple des *comptes bancaires*. Que ce soit dans la phase de spécification, ou même déjà dans la phase de maintenance, il peut s'avérer nécessaire d'introduire des *comptes d'épargne*, qui sont des *comptes bancaires* présentant certaines singularités.

Le langage Oblog possède un mécanisme de spécialisation qui nous permet de spécifier une classe de *comptes d'épargne* en réutilisant tout ce qui est nécessaire de la spécification du *compte bancaire*.

Appartenant à la famille des approches orientées objets (qui peuvent être caractérisées sommairement par une vue intégrée des données et des processus et la présence d'un mécanisme de spécialisation), Oblog présente des aspects originaux.

(i) Tout d'abord, la notion déjà mentionnée d'objets concourants.

(ii) Ensuite la possibilité de raffiner progressivement la spécification. Par exemple, nous pouvons considérer, au début, un *débit* comme une opération atomique (événement).
Plus tard cependant, nous verrons à coup sûr un débit comme une opération avec plusieurs étapes, mais qui doit malgré tout conserver son atomicité: c'est à dire, un *débit* est une **transaction**.
Le langage permet la "**réification**" des spécifications avec des objets de très bas niveau précédemment spécifiés, supportant la primitive transaction au niveau logique: le langage permet de décrire les aspects d'intégration statique et dynamique d'une collection d'objets du niveau de raffinement n+1 leur permettant d' "implémenter" un objet du niveau de raffinement n.

(iii) Enfin, la présence d'un outil CASE, permettant entre autre le support de spécifications diagrammatiques et la génération de code à partir des spécifications.

2.1.3 Oblog Case Version 1.0

Oblog Case est un outil logiciel qui supporte le langage Oblog pour la modélisation (conception) d'un Système d'Information.

Il fournit notamment:
- une interface permettant l'expression des spécifications du SI sous une forme graphique
- une base de spécifications
- des outils de contrôle de la cohérence et de la complétude du système spécifié
- un générateur de code permettant de rendre la spécification finale exécutable.

Cet outil Case a été développé suite à un intérêt commercial pour les travaux du Professeur Amilcar Sernadas (le père du langage Oblog ainsi que du modèle orienté objet y associé) et de son équipe.
La stratégie et les concepts décrits dans les paragraphes qui précèdent constituent respectivement l'ambition et l'état de l'art "académique" de l'approche Oblog [Sernadas1] [Sernadas2].

Toutefois l'outil Oblog Case version 1.0 supporte actuellement un langage moins riche que son équivalent "académique". En voici quelques exemples:

- Le constructeur de spécialisation est absent mais sera présent dans une version ultérieure.
- Plus problématique est l'absence de support pour le mécanisme de réification. En effet, ce dernier est un pré requis à la démarche de conception par raffinements successifs préconisée dans la stratégie associée à l'approche Oblog. Le prototypage d'un système spécifié à un niveau de raffinement intermédiaire n'est par conséquent pas imaginable.
  Seul sera donc prototypable (exécutable en fait via génération de code) et traçable, un système spécifié au niveau ultime de raffinement. Les objets qui apparaissent à ce niveau sont des objets de très bas niveau et l'outil ne permet pas de les relier à des objets de plus haut niveau qu'ils implémenteraient.

Dans ce qui suit, afin de rester concret et cohérent avec l'existant, nous ne ferons plus référence qu'au modèle Oblog « limité » associé à Oblog Case V1.0. En effet les traces Oblog que nous nous proposons d'analyser ont été produites à partir d'applications spécifiées essentiellement avec ce langage moins riche.

2.2 L'Objet OBLOG: une vue en trois dimensions[2]

Etant donné que le concept d'objet est le concept central de l'approche, nous détaillons ici ce que Oblog entend par objet.

Un Objet peut être vu selon 3 axes: son aspect statique, son aspect dynamique et son aspect comportemental. L'aspect statique est relatif à son état, l'aspect dynamique concerne ses changements d'états et son aspect comportemental explique comment l'objet communique avec les autres objets.

2.2.1 La perspective statique

Dans cette perspective, les objets sont des entités identifiables et distinguables, par exemple un *client* de la *banque*, un *compte bancaire* particulier mais aussi par exemple un objet *gestionnaire d'un dialogue d'ajout de client.*

Les objets sont caractérisés par 2 types de propriétés qui constituent leur **état**:

- **des propriétés spatiales** qui donnent une vision instantanée de l'objet à un certain moment. Elles fournissent des informations sur l'état de l'objet. Le *propriétaire*, le *solde*, le *numéro* d'un *compte bancaire* sont des propriétés de ce type. Elles sont appelées **attributs**.
- **une propriété de référence dans le temps** qui fournit une information sur la position de l'objet dans son **cycle de vie** : la **situation**. Par exemple, supposons que dans un dialogue d'ajout de client, un *refus* d'ajout apparaît tout d'abord pour erreur de syntaxe et est suivi d'une *acceptation* de l'ajout corrigé. *Refus* et *acceptation* sont des changements d'état de l'objet *gestionnaire du dialogue* qui se produisent instantanément à des moments particuliers. Ces changements d'état sont appelés **événements**. A chaque instant un objet se trouve dans une **situation** de référence, sorte d' "état remarquable" de son **cycle de vie**, que l'on peut voir comme une abstraction des événements déjà subis par l'objet. Par exemple, à un moment donné, le *gestionnaire de dialogue* se trouve dans la situation *Contrôle*; après le *refus*, il retourne dans la situation *Contrôle*; après l'*acceptation*, il évolue vers la situation *AttenteAjout*. Les occurrences d' **événements** représentent les facteurs déclenchants d'**actions** qui font évoluer un objet d'une situation à une autre. Remarquons qu'un objet Oblog ne mémorise pas son histoire, c'est à dire la suite des événements (et leur contexte) qui l'ont affecté.

Un changement d'état d'objet (événement) aura pour conséquence une modification éventuelle des valeurs de ses attributs et l'occurrence d'une nouvelle situation.

---

[2] Dans ce chapitre et dans le chapitre suivant, notre objectif sera d'éclairer et de définir formellement les concepts et perspectives du modèle Oblog comme modèle très représentatif de la classe des modèles d'objets actifs communiquant par événements partagés. Toutefois, afin de ne pas trop nous écarter de notre objectif concret, plutôt que de présenter un modèle général et de comparer le modèle Oblog à ce modèle général, nous avons choisi de présenter le modèle Oblog [Oblog1] sous la grille d'analyse **implicite** du modèle conceptuel orienté objet O* [Rolland1] en insistant ça et là sur les particularités Oblog.

2.2.2 La perspective dynamique

Selon cette perspective, un objet évolue au cours du temps; ses propriétés changent. Un objet évolue conformément à des règles. Par exemple, un client n'est ajouté que s'il n'existe pas déjà. Deux types de règles décrivent la dynamique d'un objet:

- **les actions** qui modifient les valeurs d'attributs, par exemple diminuer le *solde* d'un *compte bancaire.*
- **les règles d'occurrences des événements** qui déterminent les conditions pour lesquelles les événements peuvent se produire dans le cycle de vie d'un objet, par exemple événement *refus* ne peut se produire que si l'objet *gestionnaire du dialogue* est en situation *Contrôle* et que la syntaxe des caractéristiques du client à ajouter est valide.

2.1.3 La perspective comportement

Dans cette perspective un objet peut être **actif**. Certains événements (appelés événements actifs) peuvent apparaître **spontanément** dans le cycle de vie de l'objet (pour autant que leur condition d'occurrence soit vérifiée). D'autres événements ne peuvent apparaître spontanément (événements passifs), ils n'éclosent qu'en synchronisation avec un ou plusieurs événements du cycle de vie d'autres objets (**interaction** entre objets). L'occurrence d'un événement dans la vie de l'objet **déclenche l'action unique** associée à cet événement et l'objet évolue instantanément vers une nouvelle situation. Dans les cas d'interaction, on peut considérer l'ensemble des événements synchronisés du cycle de vie des différents objets correspondants, comme un "méga-événement" partagé par ces objets (nous parlerons d'**événement partagé** constitué d'un groupe d'événements participants). Les actions déclenchées par les différents événements participants sont également synchronisées et peuvent être vues comme une "méga-action".

Selon les perspectives statique et dynamique, un objet est vu localement (ce qu'il est et comment il évolue), alors que la perspective comportementale permet de considérer l'objet au sein d'une collection d'objets (comment il interagit avec les autres objets).

Remarque importante
Dans le langage Oblog, la notion d'**événement** (qui décrit **pourquoi** un objet évolue) et la notion d'**action** (unique) associée à un événement (qui décrit **comment** un objet évolue) sont amalgamées syntaxiquement sous le terme d'action, bien que sémantiquement différentes.
Cet abus de langage trouve son origine dans le fait que chaque événement élémentaire (isolé ou participant), apparaissant dans le cycle de vie d'un objet, déclenche une et une seule action ayant effet sur cet objet. Cette action est en outre atomique (non décomposable logiquement).
On parlera donc dorénavant de conditions d'occurrence d'une action (événement), d'action (événement) active, d'effet d'une action (action) sur l'objet et d'appel d'action (événement partagé), en laissant le contexte préciser si possible le sens à associer à ces termes.

2.3 Le modèle OBLOG et son langage

2.3.1 L'objet Oblog

Un objet est défini par un triplet de la forme:

$$O = (Id, M, Sch)$$

où Id est l'identifiant de l'objet, M est la mémoire de l'objet et Sch son schéma.

1. Identification d'objet

Chaque objet a sa propre identité qu'il conserve tout au long de sa vie. Il possède un identifiant unique, propre et permanent.

2. Mémoire d'objet

La mémoire d'un objet est définie par:

- les valeurs de ses attributs,
- la situation dans laquelle il se trouve.

$$M = [\{ at_i(v_i) \}_{i = 1..n} ; Sit]$$

où
- $at_i(v_i)$ est la valeur de l'objet pour l'attribut at_i
- Sit est la situation de l'objet.

La séquence des événements qui ont affecté la vie de l'objet (c'est à dire son cycle de vie) ne sont pas mémorisés, seule la situation remarquable Sit résultant de ces événements est retenue.

3. Schéma d'objet

Un schéma d'objet définit un type d'objet; il décrit un moule d'objets de même nature. L'extension d'un schéma est appelée **Classe** et est composée de l'ensemble des objets ayant ce schéma.

Un schéma d'objet comporte 4 éléments:

$$Sch = \langle ATR, ACT, GTE, CALL \rangle$$

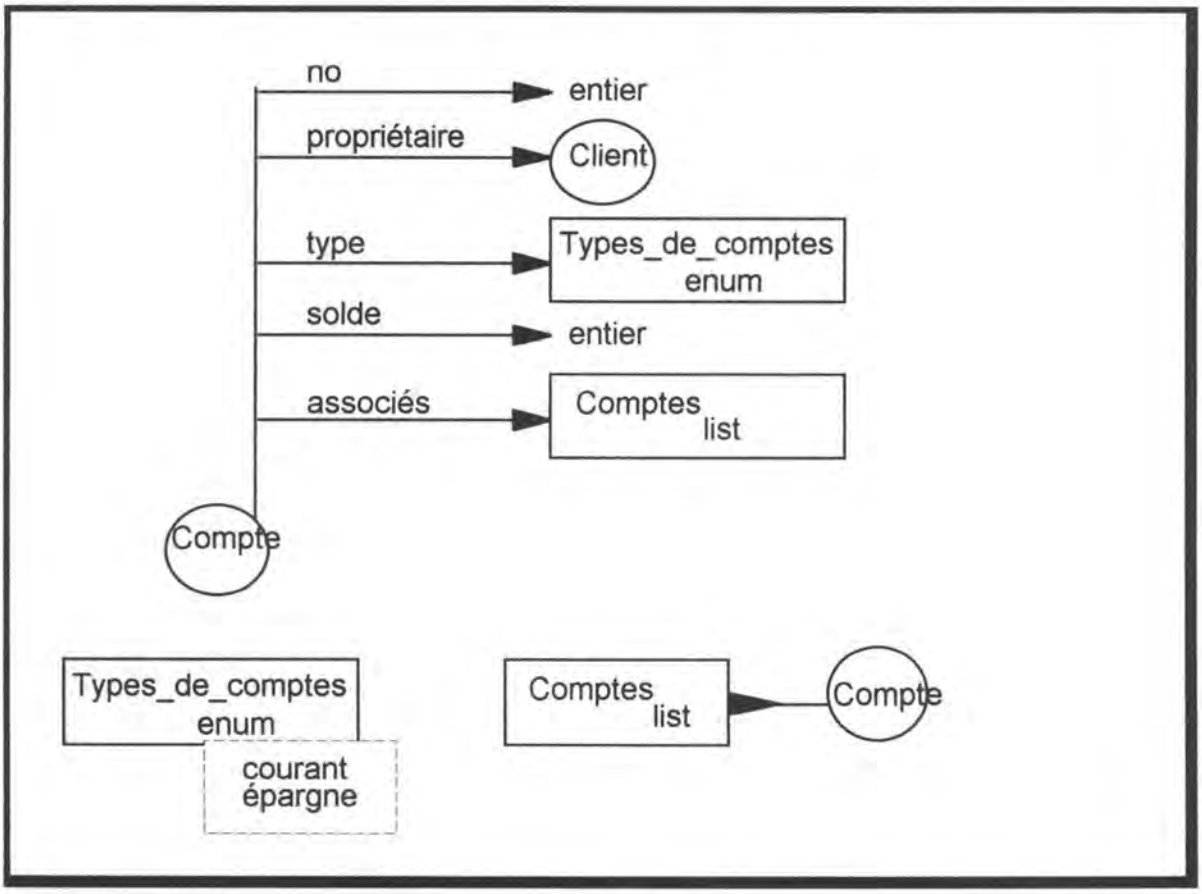
où:
- ATR est la définition des attributs,
- ACT est l'ensemble des actions définies sur les attributs,
- GTE est un graphe de transition d'état,

- CAL décrit les interactions synchrones avec d'autres objets via le mécanisme d'appel d'action.

Ces cinq éléments sont présentés successivement ci-dessous.

<ATR> : ATTRIBUTS ET DOMAINES

La structure d'un objet est définie dans le schéma par un ensemble d'**attributs**.



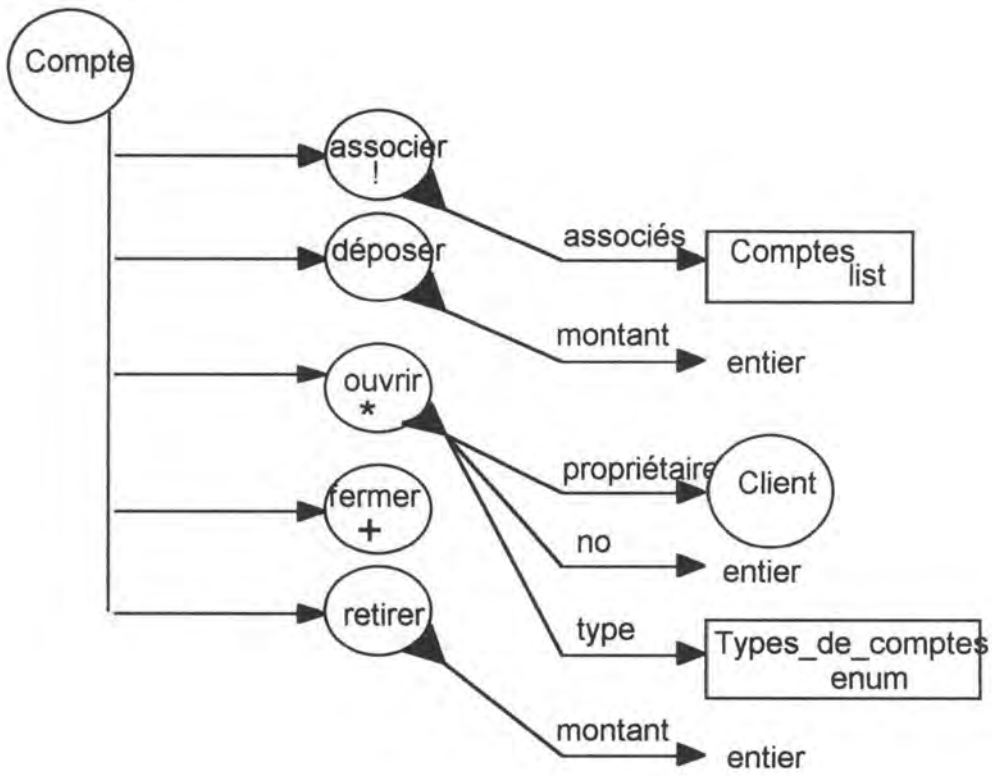
Les attributs peuvent prendre leurs valeurs dans un **domaine.** Les domaines autorisés sont des **types de données** tels que integer, string, natural, real, character ou boolean mais aussi toutes les **classes d'objets** déclarées (attributs "pointeurs"). Des domaines complexes peuvent être définis par l'utilisateur grâce aux constructeurs énumération et liste, par exemple *Comptes* est une liste de *comptes* et *Type_de_comptes* est une énumération des valeurs "courant" et "épargne".

Remarquons que les attributs pointeurs ou liste de pointeurs établissent des liens entre objets dont la sémantique est peu précise. Un tel attribut peut respectivement décrire:

- un lien de composition,
- un lien d'association,
- un lien support d'interaction (via action call ou via instance query, mécanismes que nous décrirons ultérieurement).

<ACT> : ACTIONS

Seules les actions définies dans un schéma d'objet peuvent modifier les valeurs des attributs des objets de ce schéma. Dans un schéma d'objet, une action est définie par un **nom**, des paramètres (**action parameters**) et un effet sur les attributs de l'objet (**attribute update** : liste d'assignations qui définissent les nouvelles valeurs des attributs de l'objet après l'occurrence de l'action).
Par exemple l'action *déposer* de la figure suivante a une signature composée d'un paramètre (le *montant* du dépôt). Son effet sur l'attribut *solde* s'exprime en terme de post-condition.



Attribute Update for action: ouvrir
no : = ouvrir.no
type : = ouvrir.type
propriétaire : = ouvrir.propriétare
solde : = 0

Attribute Update for action: retirer
solde : = solde - retirer.montant

Attribute Update for action: déposer
solde : = solde + déposer.montant

Attribute Update for action: associer
associés : = associer.associés

On peut distinguer des actions **passives** et des actions **actives**. Une action passive ne peut apparaître que suite à un déclenchement externe. Une action active apparaît de sa propre initiative lorsque l'objet se trouve dans une situation permettant cette action.
L'action *associer* est un exemple d'une telle action (marquée du symbole: !)

L'action *ouvrir* est l'action de **naissance** de l'objet (*) et l'action *fermer* est son action de **mort** (+). Les autres actions sont bornées par les deux précédentes et sont qualifiées d'actions de **mise à jour**.

<GTE> : GRAPHE DE TRANSITION D'ETAT

Le GTE permet de spécifier les règles d'évolution d'un objet. Il décrit les cycles de vie possibles de l'objet.

Il comporte:

- des noeuds qui correspondent à des **situations** de l'objet,
- des arcs correspondant à des **transitions**.

Durant sa vie, un objet se trouve toujours dans une situation (état remarquable) bien déterminée.
A partir d'une situation donnée, seules certaines transitions son autorisées.
Une transition est caractérisée par une situation initiale, une situation finale, une éventuelle condition d'occurrence, une action associée unique (et atomique) et une éventuelle instantiation de paramètres de cette action.

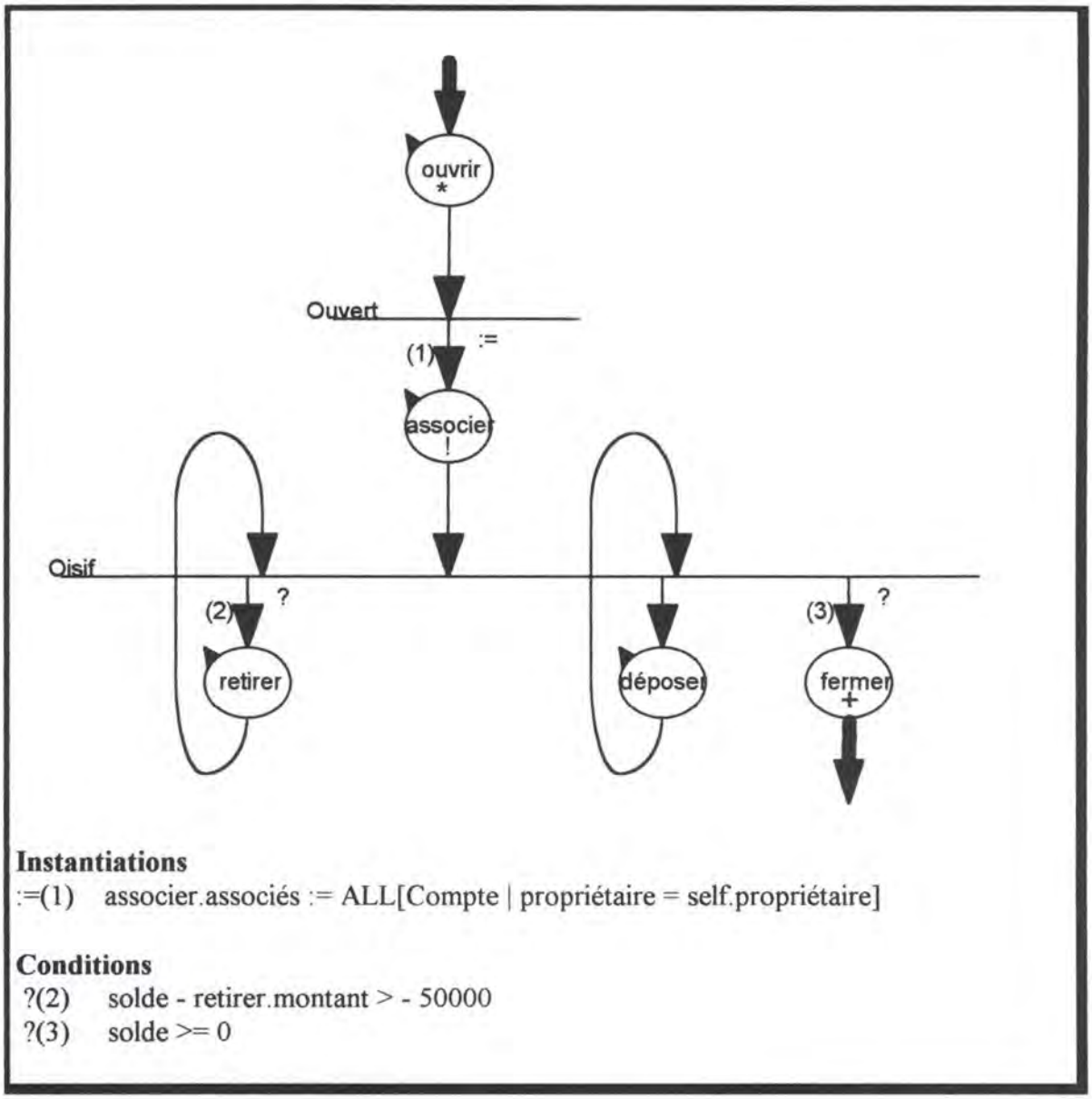
L'occurrence d'une transition induit un changement d'état de l'objet (une transition est en fait un événement):

- modifications éventuelles des attributs de l'objet suite au déclenchement de l'action associée à cette transition,
- évolution de l'objet vers une nouvelle situation de référence.

Une transition est active (respectivement passive) si l'action associée est active (respectivement passive).
Dans une situation donnée, une seule transition peut apparaître dans la vie de l'objet bien que plusieurs puissent être permises. Les transitions passives ont priorité sur les actives. Si plusieurs transitions actives sont permises, une d'entre elles est choisie de manière non déterministe.

Remarquons que la notion de transition (événement) apparaît dans le discours Oblog mais malheureusement pas dans le langage (en particulier une transition Oblog ne porte pas de nom): elle peut toutefois être reconstruite mentalement à partir du GTE.

La figure suivante présente le GTE de l'objet *compte* (encore appelé **Behavior diagram**):



Instantiations
:=(1) associer.associés := ALL[Compte | propriétaire = self.propriétaire]

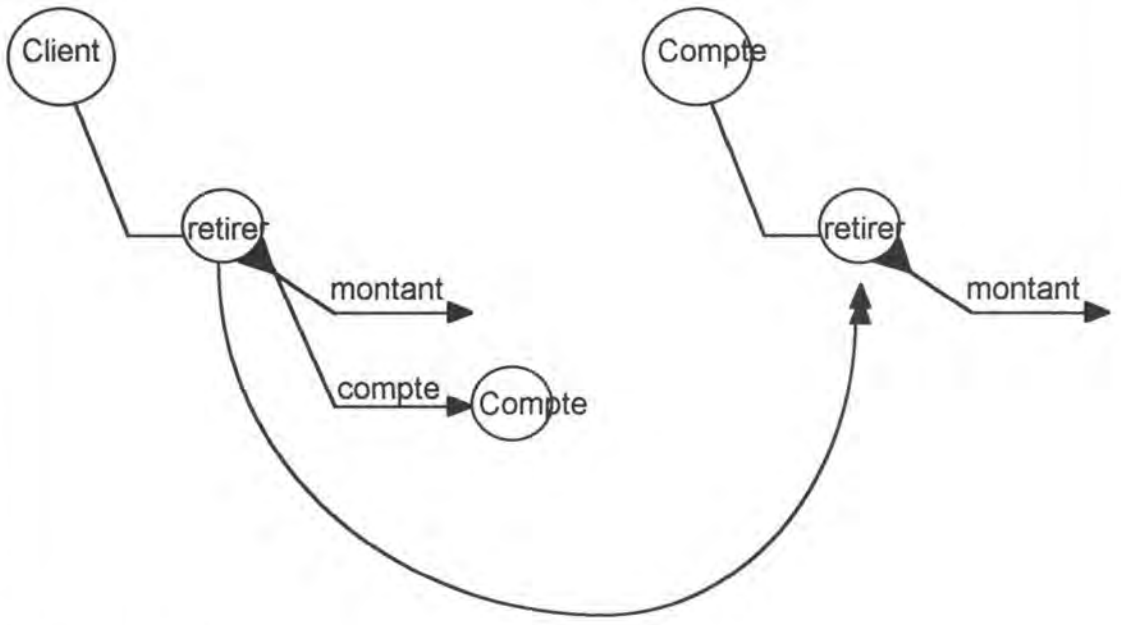
Conditions
?(2) solde - retirer.montant > - 50000
?(3) solde >= 0

Cette figure fait notamment apparaître une transition passive *X* de situation initiale *Oisif*, de situation finale *Oisif*, de précondition ?(2), d'action associée *retirer* et sans instantiation de paramètres de cette action.

<CAL> : APPEL D'ACTION

CAL est l'ensemble des appels d'action (action calls) de l'objet. Un appel d'action est une **méthode d'interaction synchrone** entre objets. Un appel entre 2 actions représente l'apparition **simultanée** des 2 actions dans chacun des objets impliqués.

Un appel d'action (**action call**) est constitué du nom d'une action de l'objet (action appelante), du nom d'une action associée à un autre objet (action appelée), d'une éventuelle condition d'occurrence, d'un facteur d'identification de l'instance d'objet cible (attribut de l'objet appelant ou paramètre de l'action appelante) et d'une clause éventuelle d'instantiation des paramètres de l'action appelée.



ACTION CALL

Client.retirer **calls** Compte.retirer

identified by Client.retirer.compte

Instantiations: Compte.retirer.montant := Client.retirer.montant
Conditions: TRUE

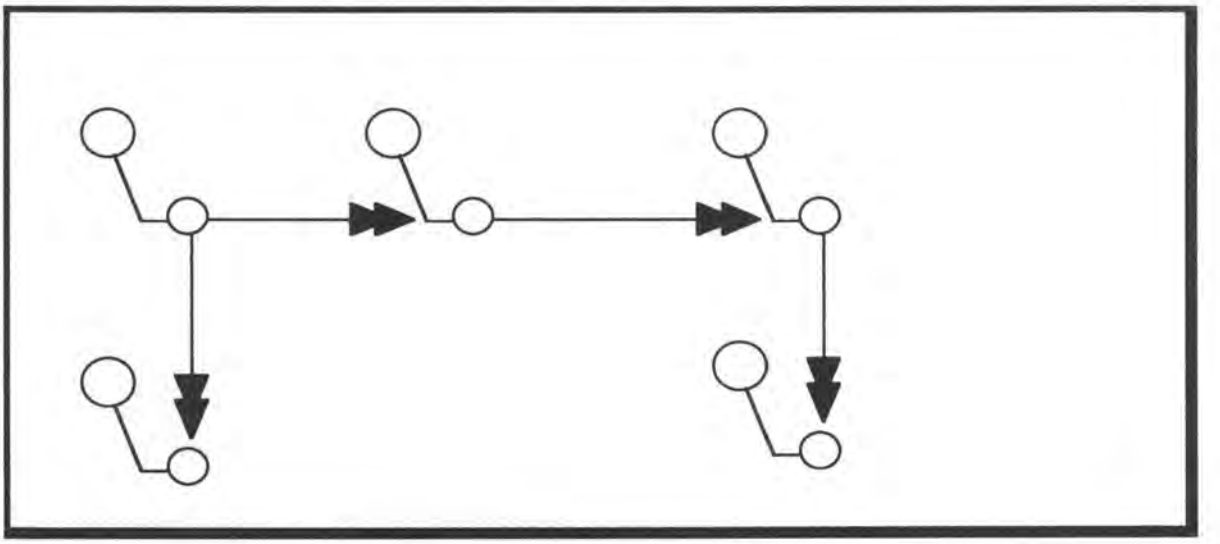
La figure précédente illustre un appel d'action d'un objet de classe *Client*. Lors de l'occurrence de l'action *retirer* dans une instance d'objet de classe *Client*, l'instance *Client.retirer.compte* de la classe *Compte* subit simultanément l'occurrence de l'action *retirer* avec le paramètre *Client.retirer.montant*. Les 2 actions font partie d'une même transaction.

Un call peut seulement apparaître si l'action appelée est autorisée (situation, précondition). Si elle ne l'est pas, aucune des actions n'a lieu et les 2 objets restent dans le même état.

Un call est seulement tenté si la condition de call est satisfaite.

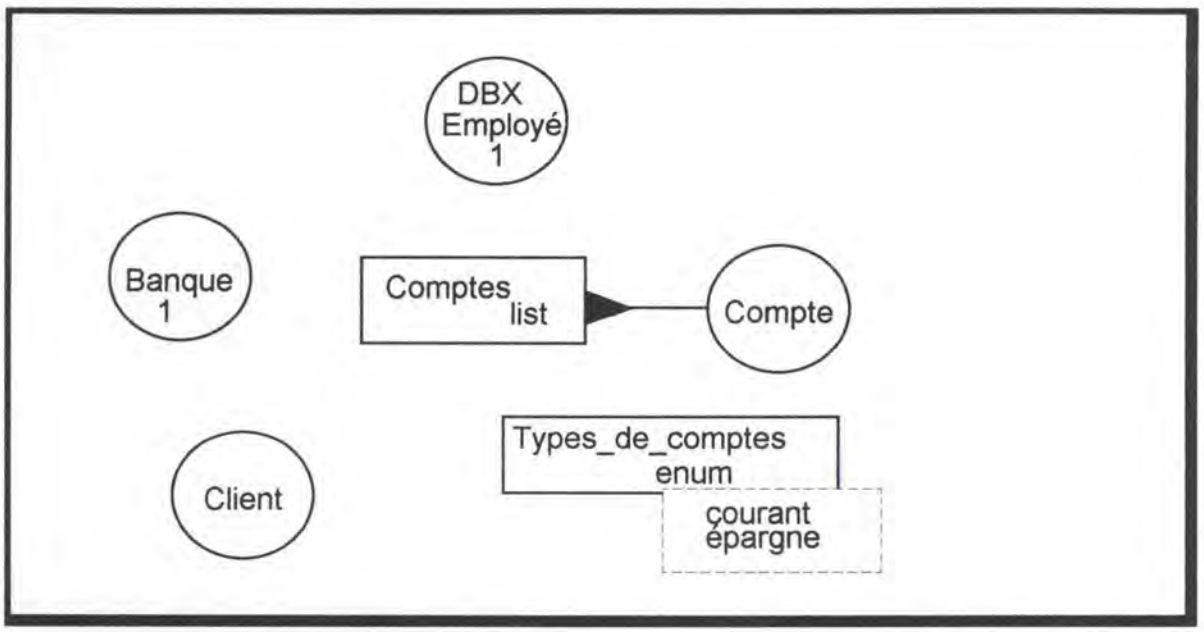
Le facteur d'identification peut également être une liste. Dans ce cas, l'ensemble des instances d'objets de cette liste est appelé (si l'action appelée est autorisée pour chacun d'eux).

Une action active d'un objet peut appeler des actions passives d'autres objets. Une action passive d'un objet peut appeler des actions passives d'autres objets. Les actions actives ne peuvent être appelées. Une action passive ne peut apparaître que via un appel d'une action du système ou suite à un déclenchement externe.
Par conséquent une action active (ou une action passive déclenchée de l'extérieur) peut être à un moment donné la racine d'une arborescence de calls (calls et calls induits dont la condition d'occurrence est vérifiée à cet instant). Si une des actions appelées par un call de l'arborescence n'est pas autorisée à cet instant, toute l'arborescence est inhibée ainsi que l'action racine. Dans le cas contraire, toutes les actions impliquées sont appliquées de manière synchrone sur leurs objets respectifs.
Chaque objet ne peut participer à une telle "méga-action" qu'avec une action à la fois.
Nous pouvons voir le déclenchement d'une telle arborescence de calls comme l'occurrence d'une "méga transition" (événement partagé) composée des transitions élémentaires des objets participants. La figure suivante présente un exemple d'arborescence de calls.



2.3.2 Le système d'objets OBLOG

Un système d'objets Oblog est appelé un **groupe (group).** Ce groupe est un agrégat d'**unités (units)** qui représentent des sous-systèmes d'objets. Les critères de cohésion modulaire sont utilisés pour construire ces unités.
Chaque unité est associée à un Diagramme de Communauté (**Community Diagram**) qui rassemble un ensemble de classes d'objets et de types de données complexes.



Dans la figure ci-avant, les classes *Banque* et *Employé* sont marquées du chiffre 1 de manière à indiquer que ces classes ne comportent pas plus d'une instance.
La classe *Employé* est de type Dialog Box. Les autres classes sont de type OBL.

Les classes d'objets mentionnées dans le Community Diagram peuvent être de 3 types distincts:

- TBL : classes d'objets persistants,
- DBX: classes d'objets boites de dialogue,
- OBL: classes par défaut.

Classe d'objets de type OBL

Les objets d'une classe OBL sont des objets transitoires qui sont utilisés pour spécifier la dynamique du système. Par défaut, une classe est de type OBL.
A la génération de code, ils sont traduits en code C ANSI.

Classe d'objets de type DBX

Une classe DBX ne comporte qu'une instance d'objet au plus. L'objet d'une classe DBX est un objet (transitoire) interface avec l'utilisateur.
Un objet DBX peut être mis en correspondance directe avec une boite de dialogue qui incarne la représentation externe du dialogue. A la génération du code, une boite de dialogue est implémentée comme un dialogue MOTIF sous X-WINDOWS.

La boite de dialogue associée à un objet DBX comporte les éléments suivants:
- des champs de contenu fixe (Un titre par exemple),
- des champs Output-only (OOF),
- des champs Input/Output (IOF),
- des Push buttons (PBT).

Les champs OOF et IOF sont associés à des attributs de domaines simples de l'objet DBX. Un champ OOF affiche toujours la valeur courante de l'attribut associé; si la valeur de l'attribut change suite à l'occurrence d'une action, la valeur affichée est aussitôt mise à jour. Un champ IOF réagit de même mais permet de plus à l'utilisateur d'éditer la valeur affichée. Lorsque la nouvelle valeur est introduite (après pression de la return key), l'attribut associé est immédiatement mis à jour.
Les push buttons (PBT) sont liés à des actions passives de l'objet DBX. Pousser un PBT déclenche l'action passive associée.

Remarquons, d'un point vue méthodologique, qu'un dialogue avec l'utilisateur est rendu possible grâce à 2 objets:

- un objet DBX qui gère les interactions externes au travers d'une fenêtre sur l'écran
- un objet (normal) directement lié au précédent qui contrôle la structure du dialogue (le "script").

Remarquons également que, à côté des actions de naissance et de mort, tout objet DBX doit comporter au moins une action pour ouvrir la fenêtre et éventuellement des actions pour fermer la fenêtre. Une action d'ouverture est typiquement appelée par le script. Une action de fermeture est typiquement déclenchée consécutivement au déclenchement par l'humain d'une action "quit" (PBT).

Classe d'objets de type TBL

Les instances d'une classe d'objets de type TBL sont des objets persistants. L'exécution d'une application Oblog se termine lorsqu'aucune action active n'est autorisée et lorsqu'aucune action passive ne peut être déclenchée de l'extérieur.
La terminaison d'une application détruit tous les objets qu'elle a créés hormis les objets persistants.

Les classes de type TBL sont implémentées comme des tables de bases de données relationnelles. Les instances d'une classe TBL sont implémentées comme des tuples dans une table relationnelle. A la génération de code, un objet TBL est traduit en code C/SQL.

En raison de leur implémentation différente, les classes d'objets de type TBL subissent d'importantes **restrictions de spécifications**:

▪ <ATR> Les domaines d'attributs autorisés sont les types de données simples, les types de données énumérés et les classes d'objets TBL.

▪ <ACT> Les objets TBL ont seulement des actions passives.
Leurs paramètres ont également des domaines restreints.

Les assignations d'une clause d'effet sur les attributs sont restreintes à la forme:
attribut := action.parameter.

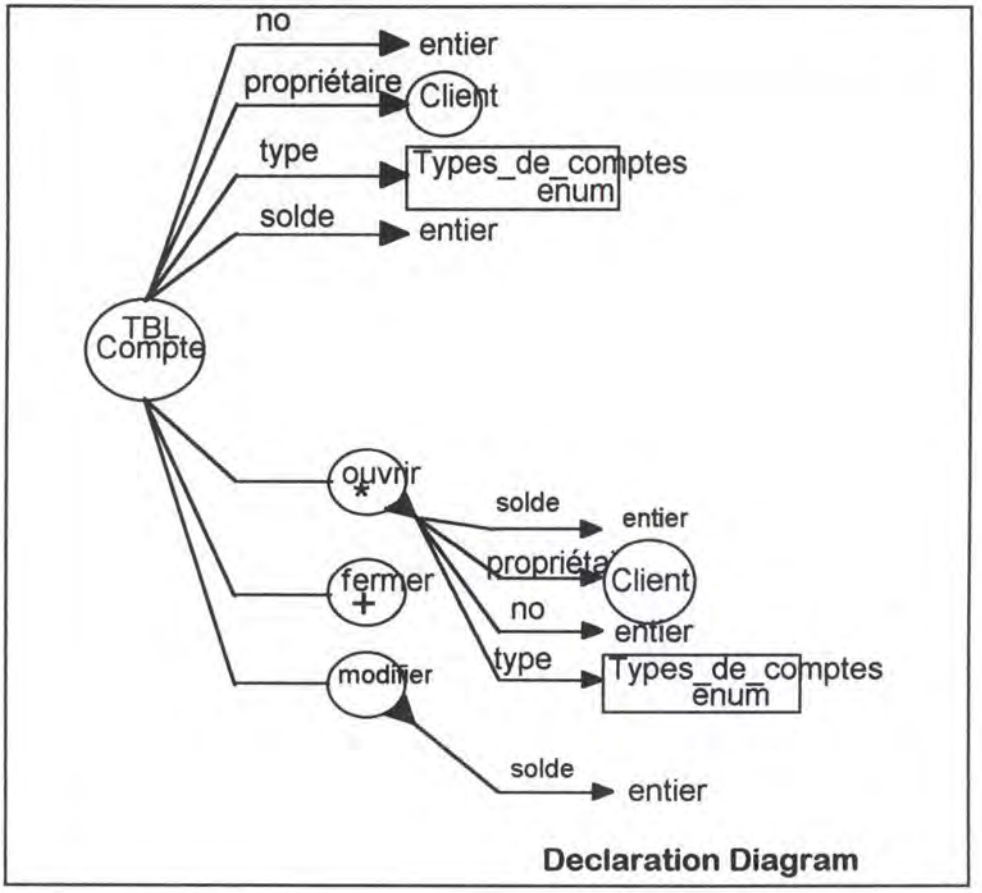
▪ <GTE> Un objet de type TBL a une seule situation.
Il n'y a pas de préconditions de transitions ni d'instantiations de paramètres.

Oblog Case V1.0 définit automatiquement le comportement des objets d'une classe TBL. Par conséquent, il n'y a pas de diagramme de comportement explicite pour de tels objets.

▪ <CAL> Un objet de type TBL ne comporte pas de call.

Un objet de type TBL est donc un objet esclave, au comportement figé, dont les attributs sont persistants.
Un appel d'une action de l'objet correspond à l'insertion (action de naissance), à la mise à jour (action de mise à jour) ou à la suppression (action de mort) d'un tuple, constitués de ses valeurs d'attributs, dans une table relationnelle.

Remarquons par exemple, qu'un objet de classe *Compte* tel que nous l'avons spécifié est sémantiquement trop riche pour rentrer dans le moule d'un objet TBL. En effet, il possède une action active et un attribut liste, les effets sur attributs de ses actions sont trop « complexes », certaines de ses transitions sont instantiées et/ou conditionnées.
Par conséquent, pour devenir persistant, cet objet doit être appauvri, comme l'illustrent les diagrammes qui suivent.

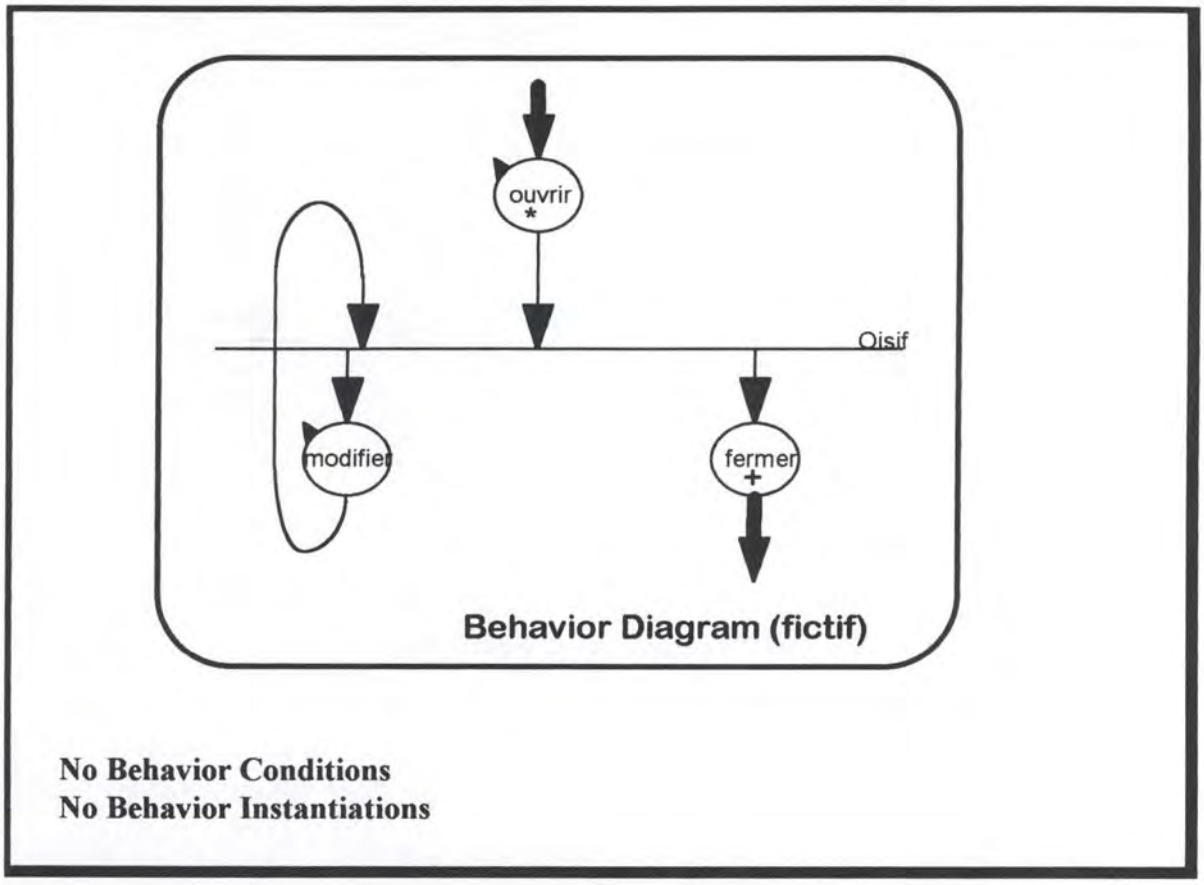


Attribute update for action **ouvrir:**
solde := ouvrir.solde
propriétaire := ouvrir.propriétaire
no := ouvrir.no
type := ouvrir.type

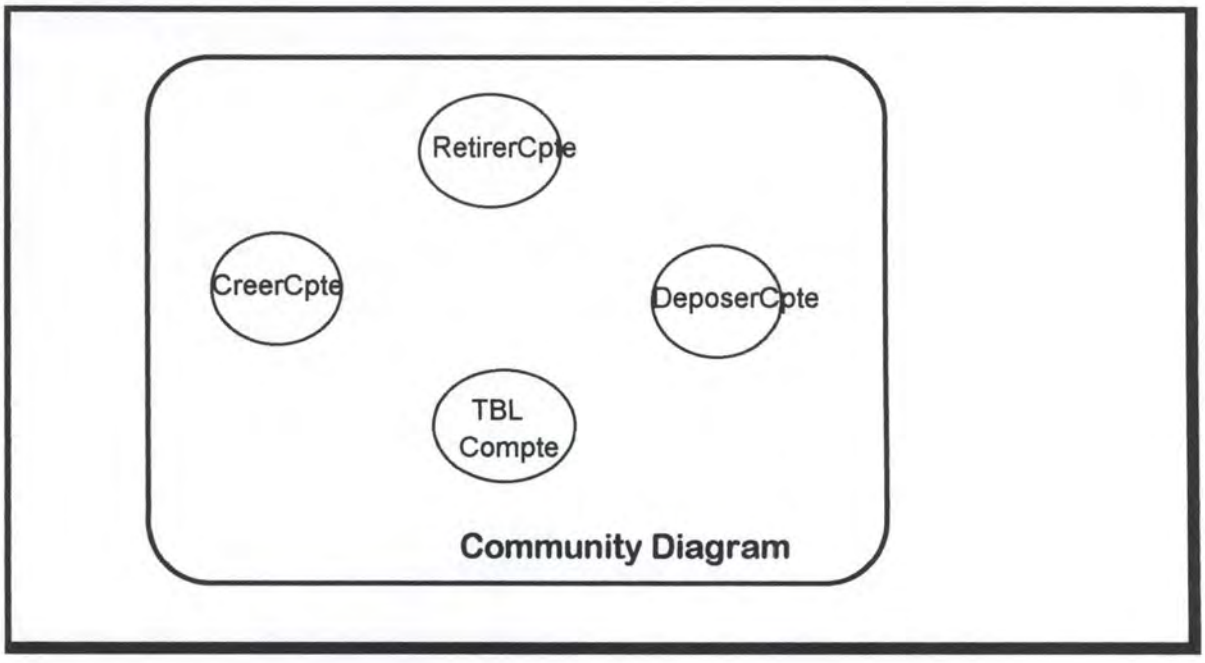
Attribute update for action **modifier:**
solde := modifier.solde

No calls

Les actions *retirer*, *déposer* et *associer* ont disparu. Les « calculs » de mise à jour ne sont plus effectués dans l'objet.



La situation remarquable «*Ouvert* » a disparu. Les transitions ne sont plus ni conditionnées ni instantiées.



Les nouvelles classes d'objets qui apparaissent dans le Community Diagram qui précède comportent des objets transitoires qui réalisent les calculs et/ou les contrôles nécessaires pour mettre à jour des objets comptes. Elles sont appelées classes opérationnelles.
Chaque requête de mise à jour crée une instance de la classe opérationnelle ad hoc. Cette instance trouve une référence de l'objet TBL à mettre à jour, calcule les nouvelles valeurs des attributs à modifier, effectue des contrôles préalables avant d'appeler une action de mise à jour de l'objet TBL.

2.3.3 Le langage d'expression OBLOG

Ce langage permet au spécifieur d'écrire des expressions qui manipulent les valeurs de variables appartenant à des domaines prédéfinis ou construits.

Une expression peut être respectivement:
- une **variable** de domaine x,
- une **constante** de domaine y,
- un **query** de domaine q,
- l'application d'une **opération prédéfinie** de signature z.

Exemples:

variables:

solde
(attribut d'un objet de classe Compte)
entier

ouvrir.no
(paramètre *no* de l'action *ouvrir* d'un objet de classe *Compte*)
entier

constantes:

'Napoléon'
string

[3,5,7]
liste d'entiers

queries:

propriétaire.adresse
(attribut *adresse* du *Client* référencé par l'attribut pointeur *propriétaire* d'un objet *Compte*)
string

ouvrir.propriétaire.nom
(attribut *nom* du *Client* référencé par le paramètre *propriétaire* de l'action *ouvrir* d'un objet *Compte*)
string

ALL[Compte | propriétaire = Self.propriétaire]
(toutes les instances d'objets de la classe *Compte* de même *propriétaire* que l'objet siège du query)
liste d'instances d'objets de classe *Compte*
EXISTS[Compte | solde > 1000000]
booléen

expressions complexes:

solde - retirer.montant > -50000
booléen
solde + déposer.montant
entier
name || ' Bonaparte'
(concaténation de deux strings)
string
fetch(associés,2)
(le deuxième élément de la liste des *comptes associés*)
objet de classe *Compte*
....

Les expressions Oblog sont employées dans différents contextes de spécification:

- **Attribute update clause**

  attrib := expression (**dépourvue de queries** dans ce contexte particulier !)

- **Behaviour condition / Call condition**

  expression booléenne

- **Behavior instantiation**

  act.parameter := expression

- **Call instantiation**

  called_act.param := expression

Chacun de ces contextes est associé à une action unique *act* d'un objet spécifié *Ob*:
attrib est un attribut de *Ob*, *parameter* est un paramètre de *act* et *param* est un paramètre de l'action *called_act* appelée par *act*.
Les variables éventuelles de l'expression sont soit des attributs de *Ob* ou des paramètres de l'action *act*. Ces variables ne peuvent donc être des variables d'un autre objet ni des paramètres d'une autre action de *Ob*.

Par contre, un objet peut « observer » d'autres d'objets via query (uniquement à partir des conditions et instantiations).

Deux types de queries peuvent être distingués:

▪ **query d'instance** : observation par un objet d'un attribut d'un autre objet dont il connaît la référence (via un attribut/paramètre pointeur ou via un autre query d'instance).
Par exemple *pere.pere.age* représente l'âge du grand-père d'un objet de classe *Personne*.

▪ **query de classe** : observation d'une classe d'objets pour:

- en trouver toutes les instances vérifiant une condition
  ALL [X | condition] : liste d'objets de Classe X

- en trouver une instance vérifiant une condition
  ONE [X | condition] : objet de classe X

- déterminer si une instance en vérifie une condition
  EXISTS [X | condition] : booléen

2.3.4 L'interface d'un objet OBLOG

L'interface d'un objet Oblog, c'est-à-dire la partie de l'objet visible de l'extérieur, est constituée:

▪ de la signature de ses actions passives, seules susceptibles d'être appelées par d'autres objets,

▪ de ses attributs, susceptibles d'être observés par d'autres objets (queries)

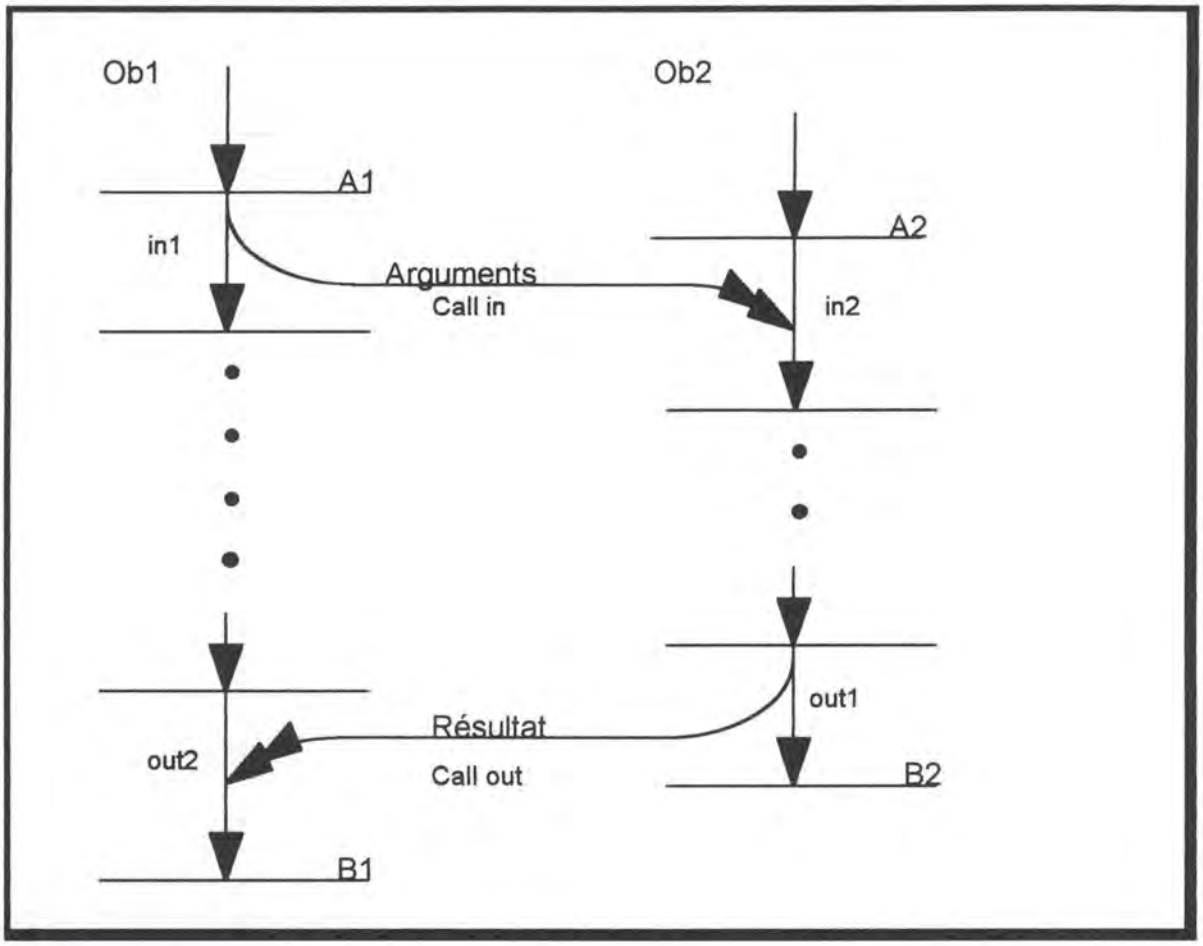
Le diagramme de comportement (GTE), les conditions et instantiations, les calls, les effets sur attributs d'un objet ne sont pas visibles de l'extérieur. Les actions actives ne peuvent être appelées.

Il apparaît donc qu'à côté de l'interaction directe entre objets, mise en place grâce au mécanisme d'action call, existe une forme d'interaction induite mise en place par le mécanisme de query. Cette interaction induite est « cachée » dans certaines expressions des conditions et instantiations diverses d'un objet.

Remarquons également qu'un query peut être considéré comme un appel d'une fonction mathématique de consultation produisant un résultat instantané. Par contre, un appel d'action doit être vu comme un appel d'une fonction mathématique avec arguments et effet immédiat sur l'état de l'objet appelé mais sans résultat. Dans les cas où un résultat est

nécessaire, les objets impliqués doivent retrouver un point de synchronisation ultérieur pour le véhiculer.

Prenons un exemple:



Dans la figure précédente, le premier call (call in) peut être vu comme une fonction mathématique instantanée dont les arguments sont les paramètres des actions in1 et in2, ayant un effet sur un état composé de l'état de Ob1 et de l'état de Ob2 mais sans résultat.
De manière plus macroscopique, la figure peut être vue comme l'illustration d'un service de l'objet Ob2 dont les arguments sont fournis par Call in et dont le résultat est produit par Call out avec effet sur états (états en A1/ A2 -> états en B1/B2). Ce service ne peut être vu comme une fonction mathématique instantanée. Un tel service en deux temps de synchronisation est source de richesse dynamique (parallélisme, distributivité) mais aussi de complexité (influence « technique » sur le schéma comportemental local des objets impliqués) d'autant plus que dans l'intervalle, les objets impliqués jouissent d'une certaine autonomie et interagissent éventuellement avec d'autres objets.

2.4 L'outil OBLOG CASE V1.2

L'outil de développement Oblog Case V1.2 supporte un langage Oblog plus riche que celui que nous venons de décrire.

Le progrès le plus marquant est le support de la spécialisation comme constructeur de réutilisation : le langage permet d'exprimer des relations d'héritage (par surcharge ou par redéfinition) entre classes d'objets.

Le langage enrichi permet également de définir des relations d'association entre classes d'objets (cachées dans les attributs références dans la version 1.0).

Les relations d'interaction (query ou call) peuvent maintenant être formalisées dans le modèle du système d'objets.

Enfin, les actions peuvent être préconditionnées.

Rappelons également que, comme c'est déjà le cas dans la version 1.0, il fournit notamment:

- une interface permettant l'expression des spécifications du SI sous une forme graphique;
- une base de spécifications permettant le maintien des spécifications dans un environnement multi utilisateurs et multi projets;
- des outils de contrôle de la cohérence et de la complétude du système spécifié;
- un générateur de code permettant de rendre la spécification finale exécutable;
- un mécanisme d'interfaçage avec des applications et des librairies externes.

2.5 OBLOG comme support des étapes de développement: évaluation

Oblog est un langage **adapté pour supporter la phase d'analyse conceptuelle**. En effet, il est associé à un modèle de structuration des données de haut niveau (objet, attributs, associations, spécialisation) ainsi qu'à un modèle de la dynamique et du comportement très riche (objet, comportement dynamique local, autonomie et interactions directes et induites). La modélisation Oblog est d'autant plus riche que, contrairement aux approches classiques, les aspects données et traitements sont totalement intégrés. Oblog propose un modèle conceptuel qui rend compte des classes d'entités et des associations d'entités pertinentes pour l'organisation mais aussi des transformations qu'elles peuvent subir dans le temps et des événements qui les provoquent. Cette vision unifiée permet de bâtir un système spécifié plus proche du système organisationnel.

Le modèle Oblog peut très bien être rapproché d'un modèle conceptuel tel que le modèle O* et adopter la démarche de conception associée à ce dernier (modèle de la fontaine) [Rolland1].

Oblog est un langage également **adapté pour couvrir la phase de conception technique**. En effet, une classe d'objets peut être vue comme un module type. Ce module type présente une partie visible (son interface: attributs et signature des actions passives) et une partie cachée (signature des actions actives, effets des actions sur les attributs, comportement local, calls). Les services disponibles dans l'interface d'un tel module sont relatifs à la gestion d'une même structure de données (mémoire de l'objet) et présentent donc une cohésion sur base des données.
Formellement, un service du module classe d'objet Ob peut s'exprimer par une signature de type:

f :	$D_1 \times ... \times D_n$	->	Ob	création (call)
f :	$Ob \times D_1 \times ... \times D_n$	->		destruction (call)
f :	$Ob \times D_1 \times ... \times D_n$	->	D	consultation (query)
f :	$Ob \times D_1 \times ... \times D_n$	->	Ob	mise à jour (call)

où

D_i est le domaine d'un paramètre d'action
D est le domaine d'un attribut de Ob

Les modules/classes peuvent s'insérer dans une architecture logicielle structurée (par exemple une hiérarchie basée sur la relation utilise) et présenter un couplage faible avec leurs partenaires.
L'héritage (dans la version 1.2) permet la réutilisation et la généricité des services.

Remarquons également qu'à ce niveau de la conception technique, le modèle Oblog intègre aussi les aspects dynamiques du système d'objets (autonomie dynamique des objets et interactions entre objets via call ou via query). Les aspects de communication et de synchronisation entre objets peuvent être décrits dans le même modèle après avoir spécifié localement les objets.

Le langage Oblog et le modèle associé sont adaptés au support de la phase d'analyse conceptuelle et de la phase de conception technique pour autant que le spécifieur fasse **l'impasse sur au moins deux restrictions du langage**:

- la faiblesse des clauses d'effets sur attributs des actions Oblog, uniquement constituées d'assignations élémentaires;
- les contraintes intervenant dans la spécification des objets persistants.

Cette impasse faite, il est alors possible de spécifier des objets de haut niveau qui font sens dans les étapes initiales de spécification. Par exemple, l'expression du texte des actions en termes de logique équationnelle dans la phase de conception technique permettrait de

construire de tels objets. Ces derniers objets pourraient même théoriquement constituer un système exécutable sur un prototype lent.

Cependant, pour être **exécutable** via l'outil de génération de code, une spécification Oblog doit tenir compte de ces restrictions. Une telle spécification (appelée terminale) comporte des objets de bas niveau. Elle relève des **étapes ultimes de conception technique** et sous certains aspects, de problèmes d'implémentation.
Dans une spécification terminale, un module type peut voir sa spécification diluée dans les spécifications d'une collection d'objets de bas niveau qui coopérent pour assurer les services du module abstrait.
Une trace ne peut être produite que par l'exécution du code généré à partir d'une spécification terminale à défaut de possibilités de prototypage.

De manière à assurer la **transition entre les niveaux de raffinements successifs** du processus de spécification, le langage Oblog « académique », développé par le professeur Sernadas, propose un mécanisme de réification [Oblog1]. Ce mécanisme repose sur des primitives d'intégration structurelle et comportementale d'objets de niveau n+1 de manière à « implémenter » un objet de niveau n. Par exemple, un objet Pile peut être vu comme une intégration structurelle et comportementale des objets Pointeur et Vecteur. Toutefois le mécanisme de réification n'est pas à ce jour supporté par l'outil Case V1.2.

Dans la suite de l'exposé, nous nous référerons dorénavant à une spécification terminale. Rappelons quelques conséquences des restrictions qui lui sont attachées:

(i) Objet persistant: les services d'un objet persistant sont éclatés dans des objets Oblog de classes opérationnelles étant donné la pauvreté de spécification imposée pour un objet TBL.

(ii) Effet sur attributs d'une action: réduit à de simples assignations; tout service de haut niveau comportant par exemple des aspects répétitifs ou conditionnels élémentaires donne lieu dans la spécification terminale à des éclatements d'objets ou à l'apparition de complexités du niveau implémentation dans la description du comportement local et externe

(iii) Interface: les consultations dont le résultat n'est pas uniquement la valeur d'un attribut ou les mises à jour dont le résultat n'est pas limité à l'état de l'objet, c'est à dire des fonctions d'un des deux types suivants:

$f : Ob \times D_1 \times ... \times D_n \rightarrow E_1 \times ... \times E_m$ consultation
$f : Ob \times D_1 \times ... \times D_n \rightarrow Ob \times E_1 \times ... \times E_m$ mise à jour

où

D_i est le domaine d'un paramètre d'action
E_j est un domaine quelconque

ne peuvent être réalisées que via un service en deux temps de synchronisation (Call in , Call out) (voir l'exemple du paragraphe 2.3.4).

Face à cet éparpillement des objets et des services au niveau d'une spécification terminale, le seul mécanisme fédérateur est le concept sommaire d'unité (unit) que l'on peut à son tour voir, du point de vue conceptuel, comme un module, d'interface relativement floue cette fois.

2.6 Conception de l'application Monde Bancaire

Dans ce chapitre, nous décrirons les étapes de conception d'un système d'information qui supporte l'activité d'une banque fictive [Oblog1]. L'application Monde Bancaire nous servira ultérieurement de référence pour l'analyse d'une trace Oblog et pour l'illustration de règles d'analyse ASAX d'une trace Oblog.

Univers du discours: le Monde Bancaire

La banque est une institution qui fournit divers services financiers à ses clients via des comptes. Un client peut avoir un ou plusieurs comptes mais un compte est toujours associé à un seul client.

Un client est caractérisé par:
- un nom
- un titre (facultatif)
- un numéro d'identification unique
- une adresse (rue, code zonal,ville)
- un numéro de téléphone (facultatif)

Un compte est caractérisé par:
- un propriétaire Client
- un numéro d'identification unique
- un solde
- une limite de crédit
- une carte bancaire (facultative)

La banque offre des produits et services:
- comptes (avec ou sans carte bancaire)
- dépôt d'argent
- retrait d'argent

Un client peut ouvrir autant de comptes qu'il le souhaite. Pour chaque compte, le client peut demander l'usage d'une carte bancaire. A l'ouverture d'un compte, le client dépose une somme d'argent qui va devenir le solde initial du compte et il négocie la limite de crédit pour ce compte. Les comptes peuvent être fermés si leur solde est positif. Un retrait peut être réalisé si le montant retiré n'excède pas le solde augmenté de la limite de crédit.

Les clients peuvent aisément gérer leurs comptes grâce à un automate (ATM). Pour cela, une carte bancaire avec un code personnel est émise et associée à chaque compte concerné.
L'automate dispose d'un écran, d'un lecteur de carte, de touches numériques et de touches d'introduction spéciales ainsi que d'un émetteur de billets.
Quand la machine est au repos, un message d'accueil est affiché. Les touches restent passives en attente de l'insertion d'une carte.
Lorsqu'une carte est insérée, le lecteur de carte commence son contrôle et sa validation. Si la validation échoue, le client est prévenu à l'écran et la carte est éjectée sans autres

commentaires. Dans le cas contraire, le client est invité à entrer son code personnel via les touches numériques.
Si ce code est correct, un message demande au client de choisir l'opération souhaitée.
Si le code est incorrect, la carte est éjectée. Si un code incorrect est introduit à trois reprises, la machine conserve la carte. La politique de la banque est d'ensuite détruire ce type de cartes.
La seule opération offerte au client par l'automate est le retrait de fonds d'un compte.
Cette opération sera réalisée en pressant une des touches spéciales qui spécifie le montant exact à retirer. Le montant requis sera délivré par l'émetteur de billets, si le compte concerné est suffisamment garni.
Une fois l'opération terminée, le système retourne dans la situation « attente carte ».
A chaque moment, durant l'acceptation de la carte, ou pendant l'opération de retrait, le client peut arrêter l'action en pressant la touche d'annulation. Dans ce cas, la carte est éjectée et la machine retourne en état d'attente. Durant l'introduction du code personnel, le client peut presser la touche de correction de manière à réintroduire un code correct.

Tous les autres services de la banque sont réalisés via le personnel de la banque:

- enregistrement de nouveaux clients
- suppression de clients ne possédant pas de comptes ouverts
- création de comptes pour un client
- suppression de comptes non découverts
- sélection et consultation d'un client sur base de son numéro, de son nom ou de son adresse
- consultation d'un compte sur base de son numéro ou consultation de tous les comptes d'un client sur base du numéro client

Le système d'information a pour objectif de modéliser le dialogue du client avec l'automate, de simuler la machine automate et l'application « front-office » destinée au personnel de la banque.

Découpe du système en sous-systèmes

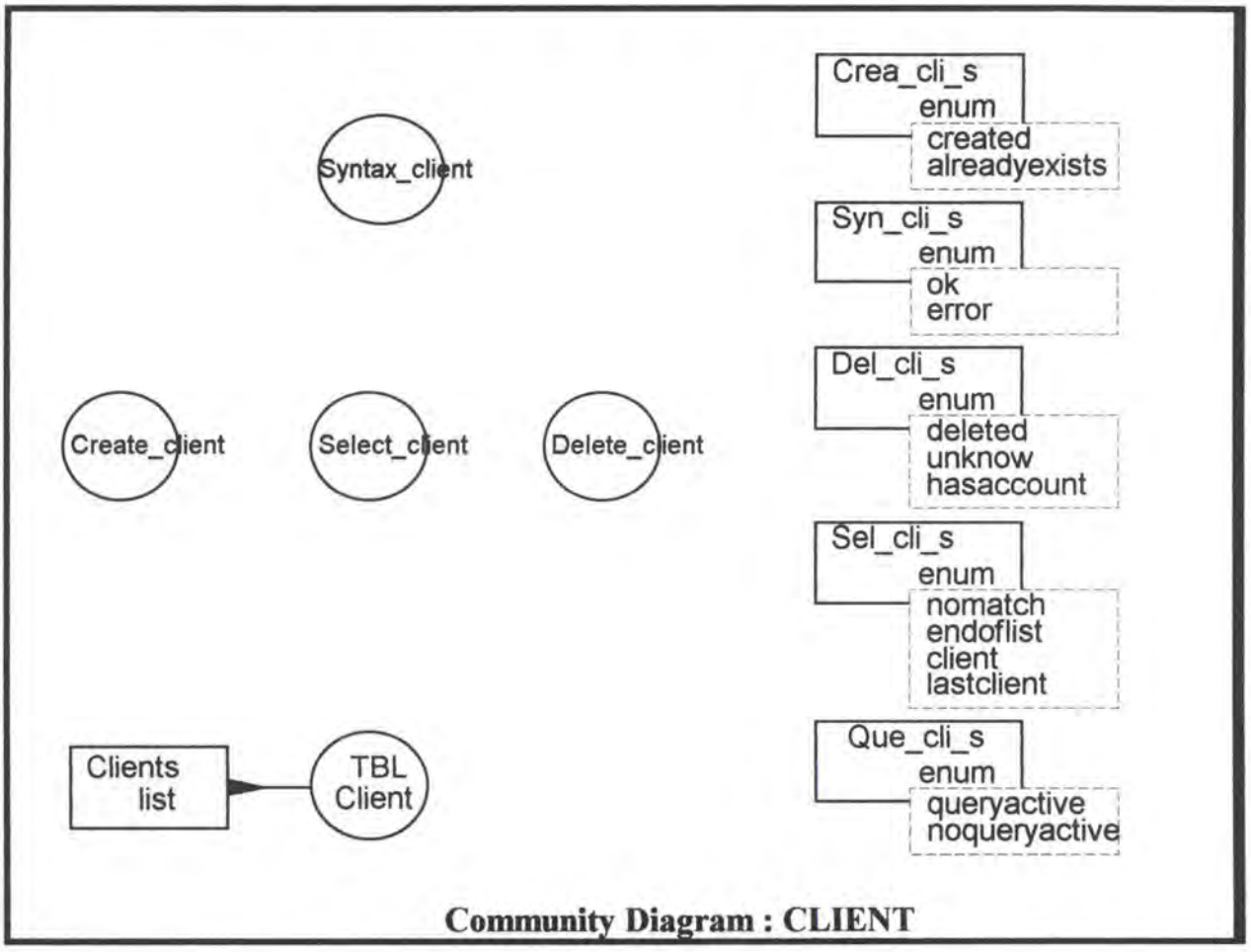
Le groupe (système) *Bank* est divisé en six unités (sous-systèmes):

- ACCOUNT
- ACCOUNT_X
- CLIENT
- CLIENT_X
- ATM_X
- NAVIGATOR_X

En effet, le Monde Bancaire peut être décomposé en deux composants fonctionnels bien distincts: l'automate et le guichet de la banque. Ces deux composants permettent la manipulation de deux concepts de base, ACCOUNT et CLIENT (concepts units).

Les fonctions de base du composant guichet sont ici rassemblées en deux groupes suivant les concepts respectifs qu'elles utilisent. ACCOUNT_X et CLIENT_X représentent les interfaces (interface units) donnant respectivement accès à ces deux groupes de fonctions. ATM_X est l'unité gérant l'interface avec l'automate.
NAVIGATOR_X permet la navigation de l'utilisateur entre les services de l'automate et les services du guichet (eux-mêmes raffinés en services client et services compte).

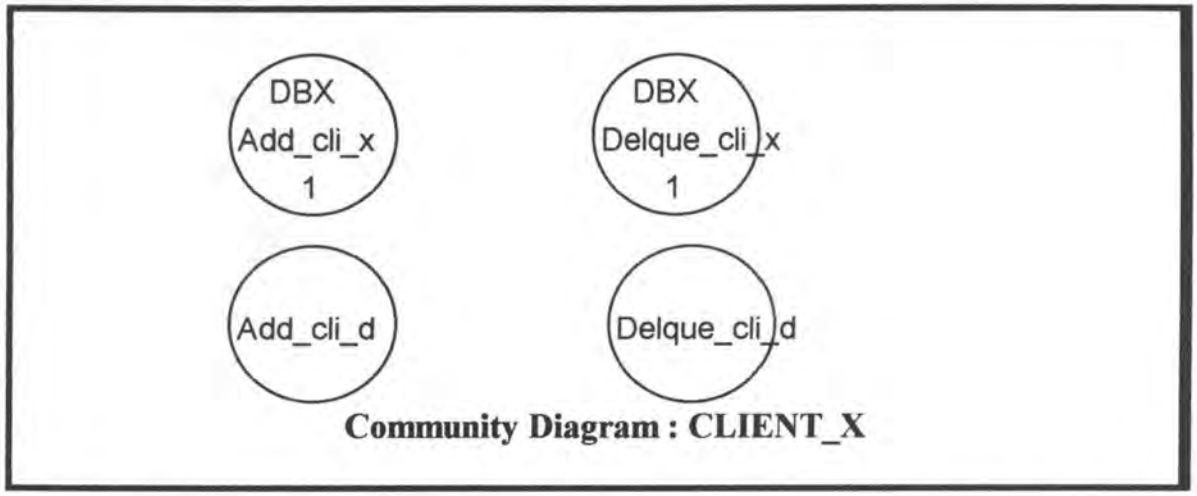
Définition des classes d'objets d'une unité concept: unité CLIENT



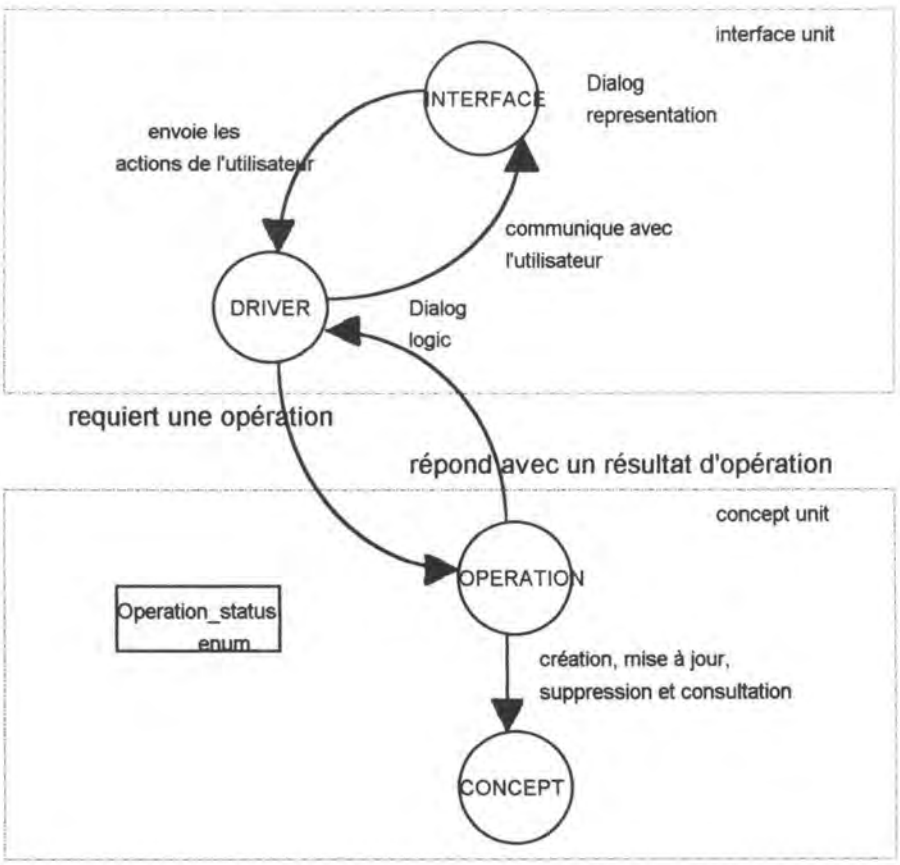
Community Diagram : CLIENT

Client est un objet persistant de type TBL. Les autres classes sont des classes opérationnelles qui assurent en fait les services du concept CLIENT (création, mise-à-jour, suppression, sélection et contrôle syntaxique). Ces services, indépendamment de leur effet sur le concept, fournissent éventuellement un résultat externe à l'objet demandeur. Par exemple, le type *Crea_cli_s* définit les résultats externes possibles de l'opération/objet *Create_client* (service en deux temps).

Définition des classes d'objets d'une unité interface: unité CLIENT_X

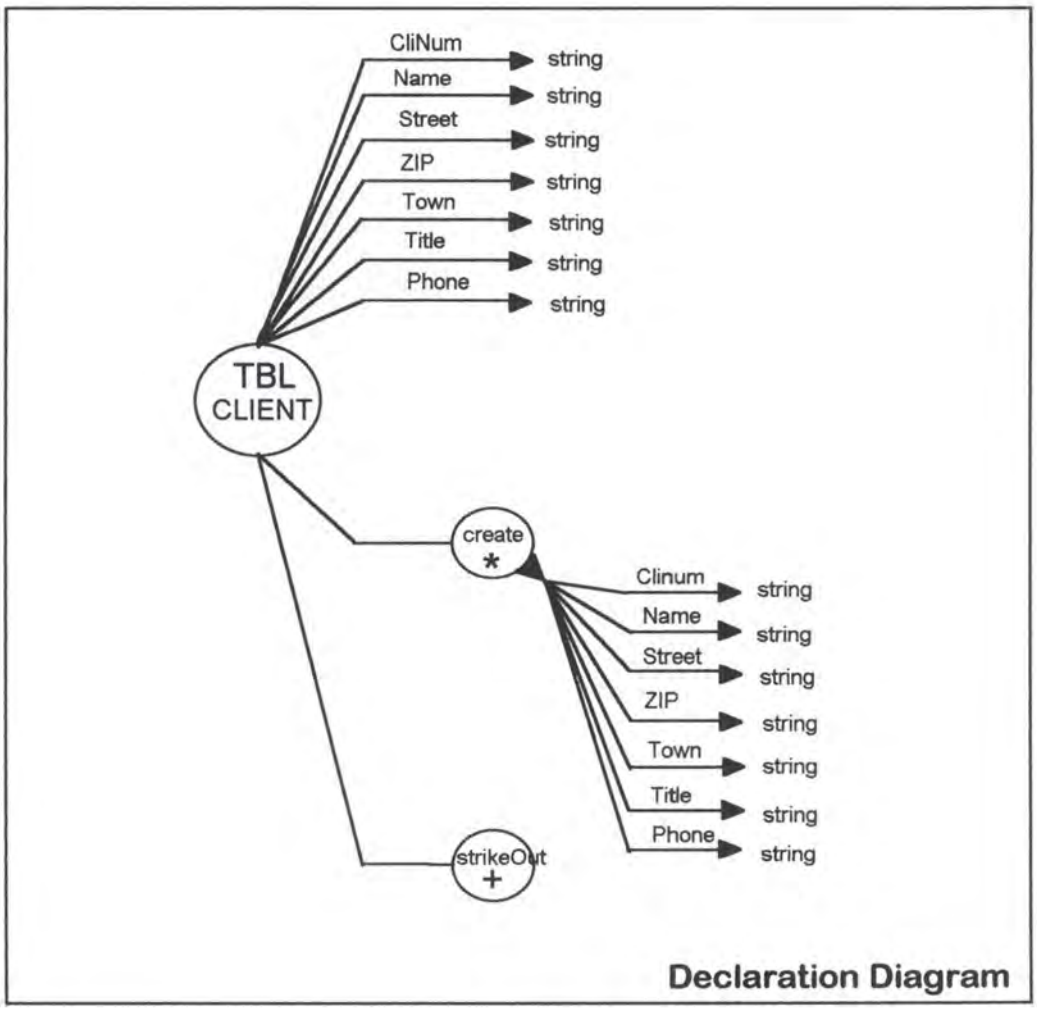


Pour manipuler les objets *Clients*, deux dialogues utilisateur sont définis: l'un permet d'enregistrer de nouveaux clients et l'autre de sélectionner et supprimer des clients.
A chaque dialogue est associé un couple de classes d'objets : un objet de la première classe est le conducteur du dialogue, dont il implémente la logique (dialog logic); un objet de la deuxième classe est de type DBX, il est la représentation concrète de la boite de dialogue utilisateur (dialog representation).



Schèma global

Spécification détaillée d'une classe concept: classe CLIENT

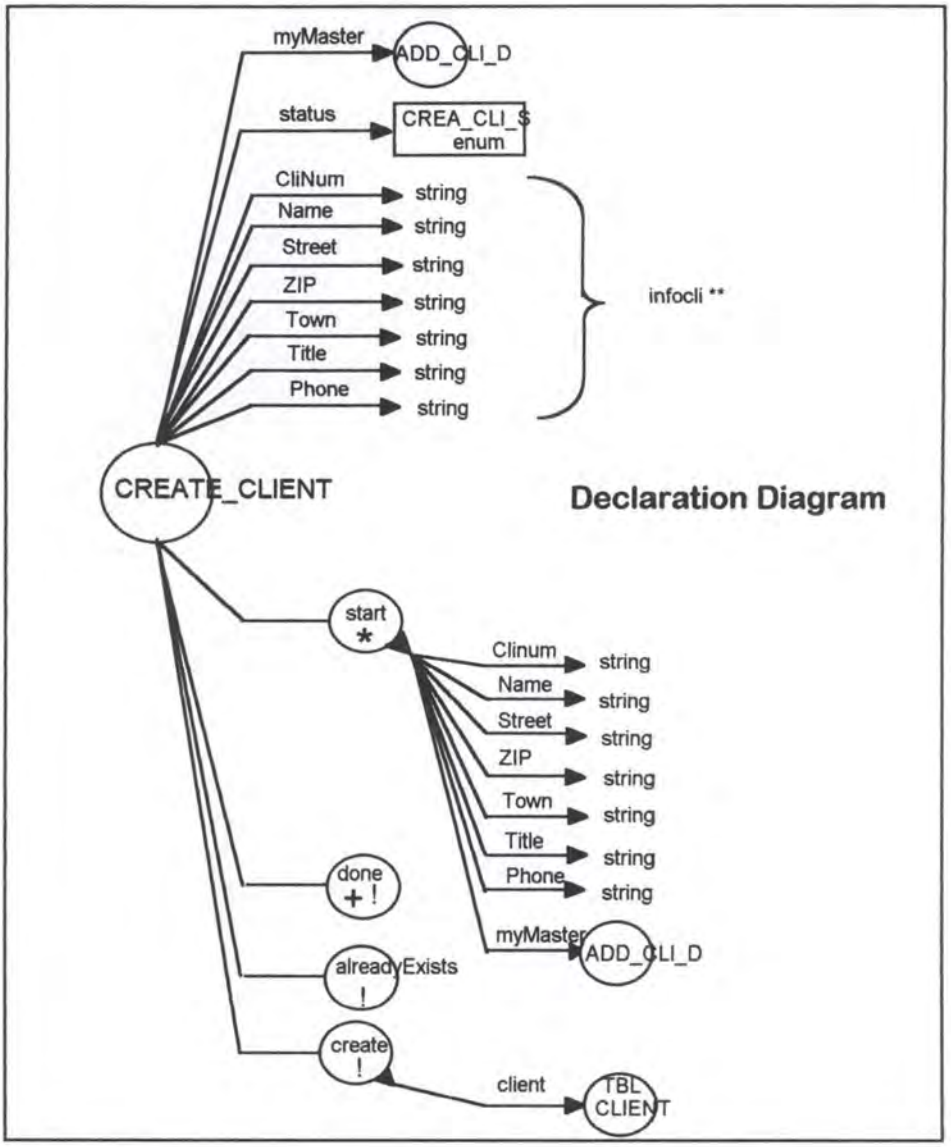


Attribute Updates of object class **CLIENT** are:

For Action **create**

Name := create.Name
Street := create.Street
Town := create.Town
ZIP := create.ZIP
CliNum := create.CliNum
Title := create.Title
Phone := create.Phone

Spécification détaillée d'une classe opérationnelle: classe CREATE_CLIENT



Attribute Updates of object class **CREATE_CLIENT** are:

For Action **start**

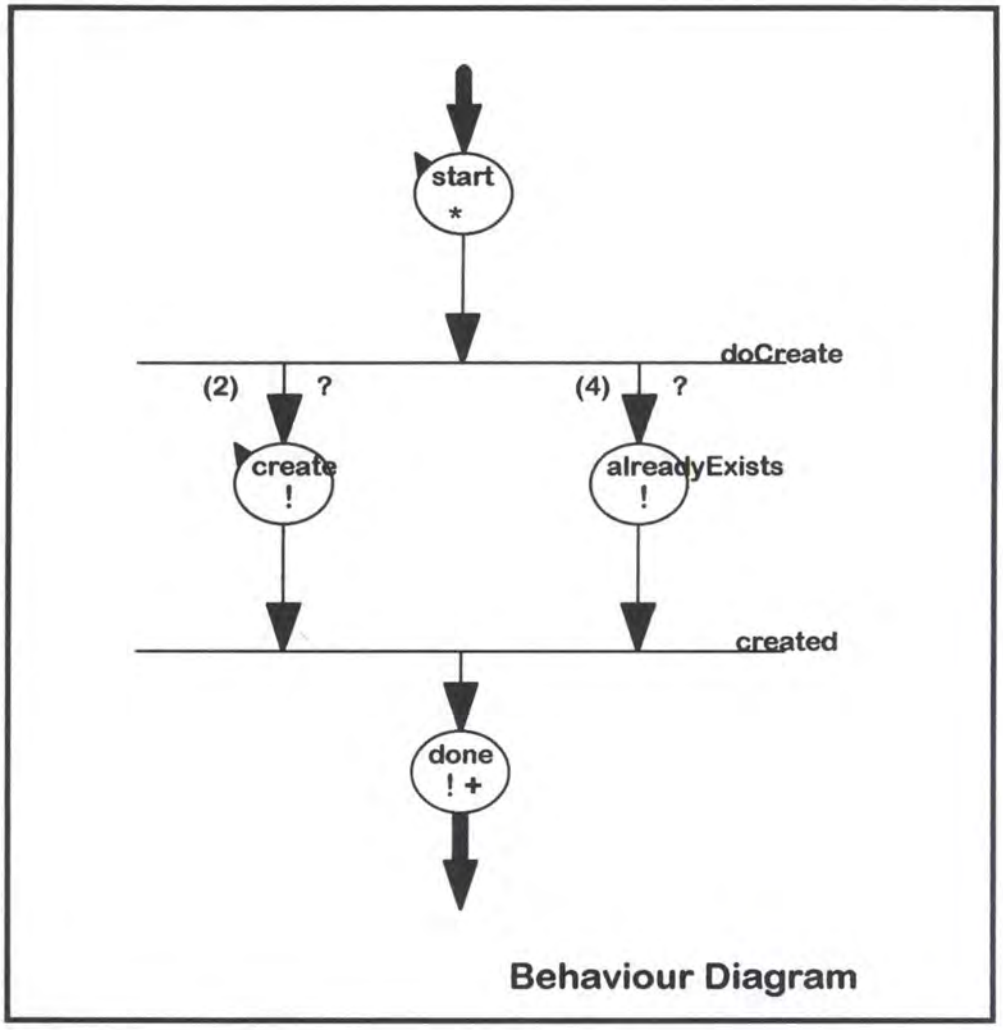
Name := start.Name
Street := start.Street
Town := start.Town
ZIP := start.ZIP
CliNum := start.CliNum
Title := start.Title
Phone := start.Phone
status := CREA_CLI_SSCREATED
myMaster := start.myMaster

For Action **create**

status := CREA_CLI_SSCREATED

For Action **alreadyExists**

status := CREA_CLI_SSALREADYEXISTS



Behaviour Diagram

Behavior Conditions and Instantiations of
object class **CREATE_CLIENT** are:

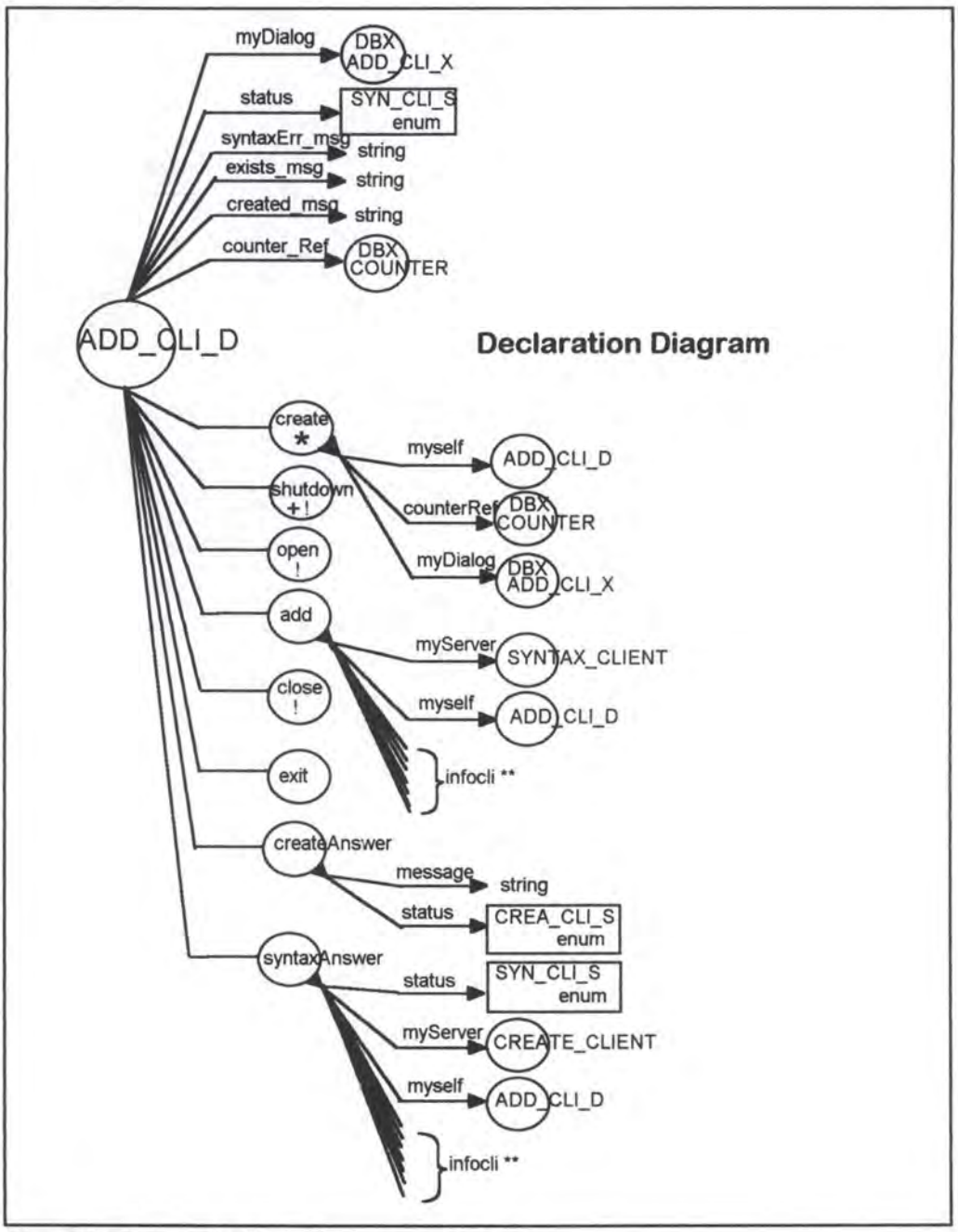
(2) **create**

? **NOT(EXISTS[CLIENT | (CliNum = SELF.CliNum)])**

(4) **alreadyExists**

? **EXISTS[CLIENT | (CliNum = SELF.CliNum)]**

Spécification détaillée d'une classe dialog logic: classe ADD_CLI_D



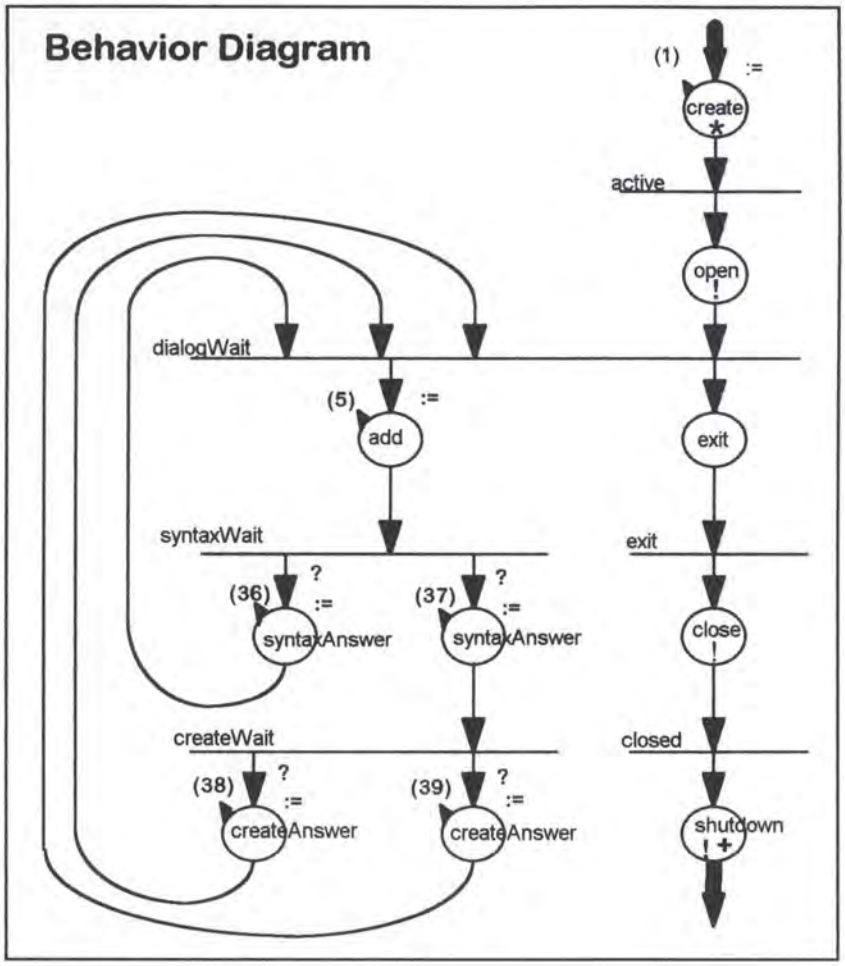
Attribute Updates of object class **ADD_CLI_D** are:

For Action **create**

myDialog := create.myDialog
status := SYN_CLI_S$ERROR
syntaxErr_msg := « Syntax Error ! »
created_msg := « Client was created! »
exists_msg := « Client already exists! »
counterRef := create.counterRef

For Action **syntaxAnswer**

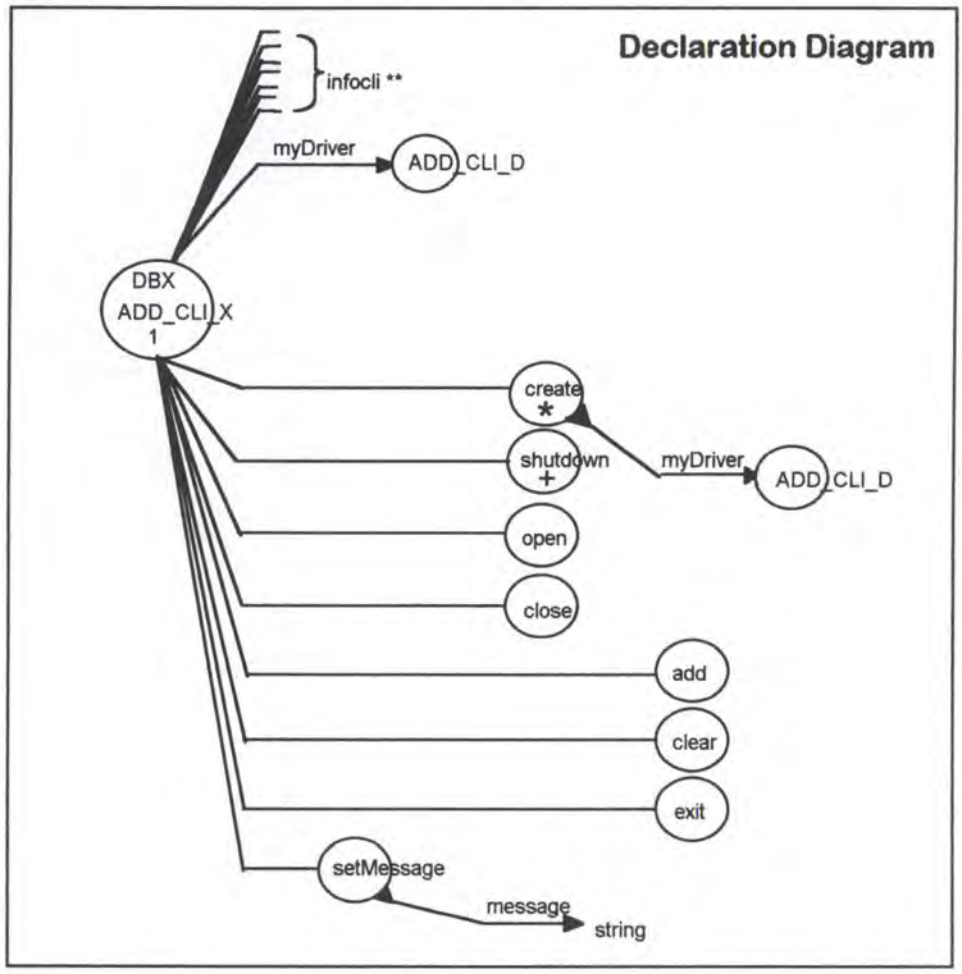
status := syntaxAnswer.status



Behavior Conditions and Instantiations of object class **ADD_CLI_D** are:

(1) **create**
:= **create.myself := SELF**
(38) **createAnswer**
? **createAnswer.status = CREA_CLI$ALREADYEXISTS**
(5) **add**
:= **add.myself := SELF**
(39) **createAnswer**
? **create Answer.status = CREA_CLI_S$CREATED**
:= **createAnswer.message := created_msg**
(36) **syntaxAnswer**
? **syntaxAnswer.status = SYN_CLI_S$ERROR**
:= **syntaxAnswer.myself := SELF**
(37) **syntaxAnswer**
? **syntaxAnswer.status = SYN_CLI_S$OK**
:= **syntaxAnswer.myself := SELF**

Spécification détaillée d'une classe dialog representation: classe ADD_CLI_X



Attribute Updates of object class **ADD_CLI_X** are:

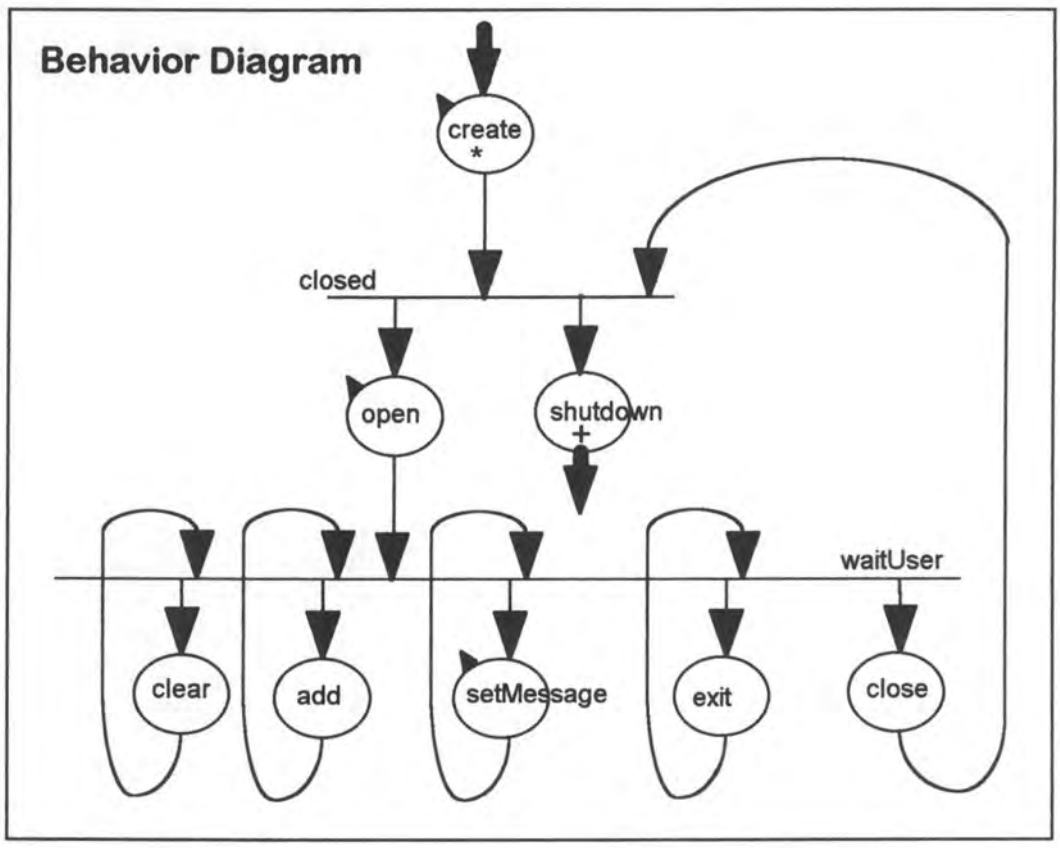
For Action **create**
- **Name := « »**
- **Street := « »**
- **Town := « »**
- **ZIP := « »**
- **CliNum := « »**
- **Title := « »**
- **Phone := « »**
- **message := « »**
- **myDriver := create.myDriver**

For Action **exit**
- **message := « »**

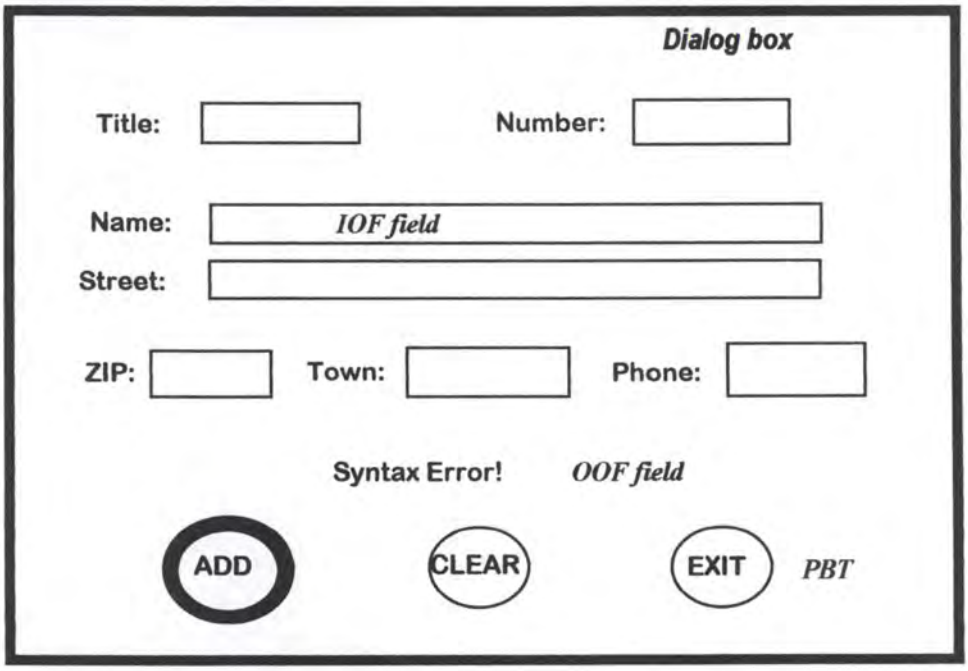
For Action **clear**
- **Name := « »**
- **Street := « »**
- **Town := « »**
- **ZIP := « »**
- **CliNum := « »**
- **Title := « »**
- **Phone := « »**
- **message := « »**

For Action **add**
- **message := « »**

For Action **setMessage**
- **message := setMessage.message**



No Instantiations
No Conditions



Interactions directes entre classes spécifiées localement (partim)

Calls of class **ADD_CLI_X**:

Caller action: **add**
Called action: **ADD_CLI_D.add**
Identification: **myDriver**
Instantiations:

add.Title := Title
add.Phone := Phone
add.Clinum := Clinum
add.Town := Town
add.ZIP := ZIP
add.Street := Street
add.Name := Name

...

Calls of class **ADD_CLI_D**:

Caller action: **syntaxAnswer**
Condition : **(syntaxAnswer.status = SYN_CLI_S$OK)**
Called action: **CREATE_CLIENT.start**
Identification: **syntaxAnswer.myServer**
Instantiations:

start.Title := syntaxAnswer.Title
start.Phone := syntaxAnswer.Phone
start.Clinum := syntaxAnswer.Clinum
start.Town := syntaxAnswer.Town
start.ZIP := syntaxAnswer.ZIP
start.Street := syntaxAnswer.Street
start.Name := syntaxAnswer.Name
start.myMaster := syntaxAnswer.mySelf

Caller action: **syntaxAnswer**
Condition : **(syntaxAnswer.status = SYN_CLI_S$ERROR)**
Called action: **ADD_CLI_X.setMessage**
Identification: **myDialog**
Instantiations:

setMessage.message := syntaxErr_msg

Caller action: **add**
Called action: **SYNTAX_CLIENT.start**
Identification: **add.myServer**
Instantiations:

start.Title := add.Title
start.Phone := add.Phone
start.Clinum := add.Clinum
start.Town := add.Town

start.ZIP := add.ZIP
start.Street := add.Street
start.Name := add.Name
start.myMaster := add.myself

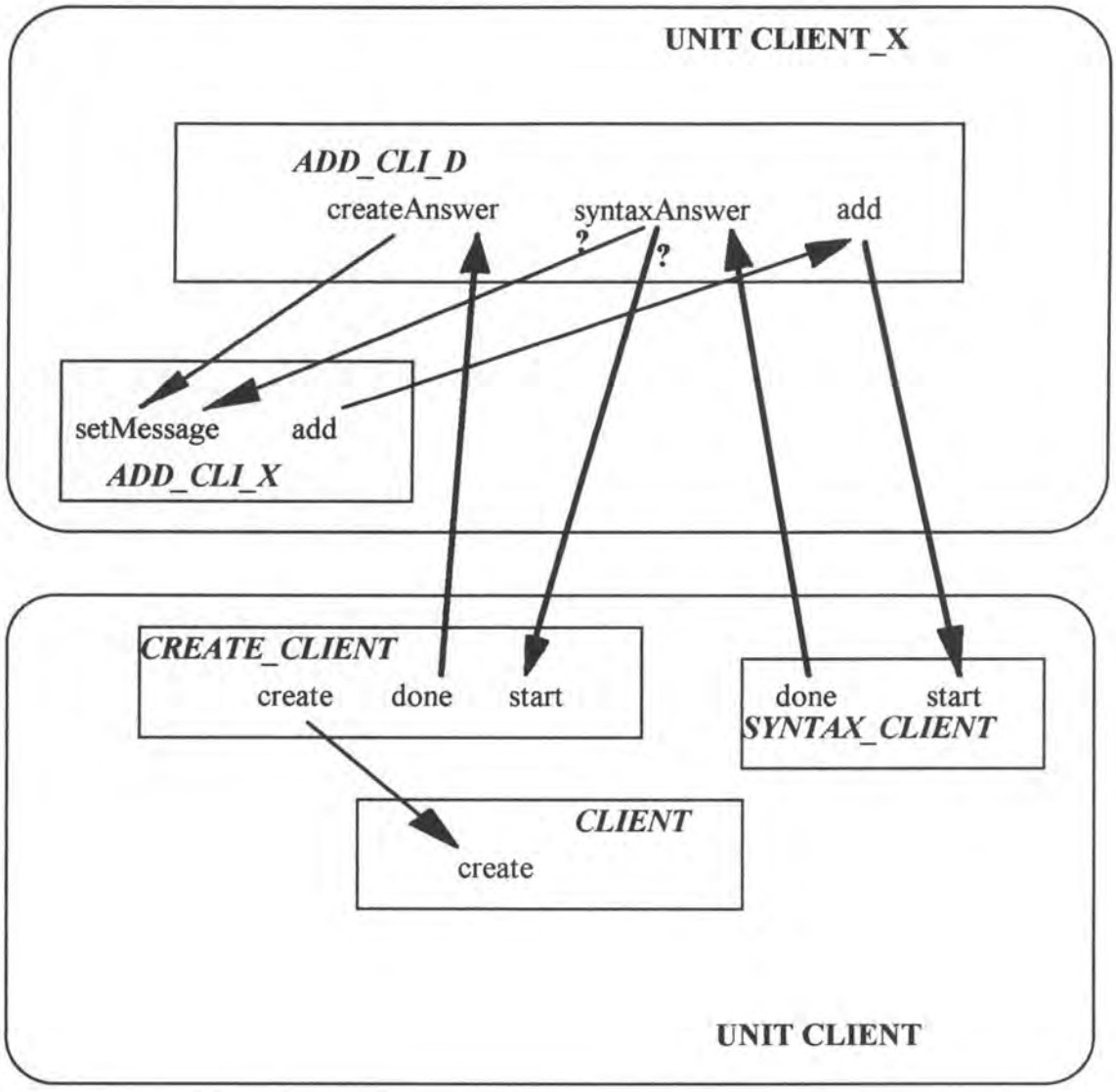
Caller action: **createAnswer**
Called action: **ADD_CLI_X.setMessage**
Identification: **myDialog**
Instantiations:
setMessage.message := createAnswer.message
...

Calls of class **CREATE_CLIENT**:
Caller action: **create**
Called action: **CLIENT.create**
Identification: **create.client**
Instantiations:
create.Title := Title
create.Phone := Phone
create.Clinum := Clinum
create.Town := Town
create.ZIP := ZIP
create.Street := Street
create.Name := Name

Caller action: **done**
Called action: **ADD_CLI_D.createAnswer**
Identification: **myMaster**
Instantiations:
createAnswer.status := status

Calls of class **SYNTAX_CLIENT**:
Caller action: **done**
Called action: **ADD_CLI_D.syntaxAnswer**
Identification: **myMaster**
Instantiations:
syntaxAnswer.Title := Title
syntaxAnswer create.Phone := Phone
syntaxAnswer create.Clinum := Clinum
syntaxAnswer.Town := Town
syntaxAnswer.ZIP := ZIP
syntaxAnswer.Street := Street
syntaxAnswer.Name := Name
syntaxAnswer.status := status

SCHEMA DES INTERACTIONS DIRECTES (partim)



La figure précédente illustre uniquement les interactions directes permettant d'implémenter le service guichet « Ajout d'un client »

Spécification terminale complète dans le manuel Oblog User Guide [Oblog1]

Schéma agrégé des interactions directes (calls) : Figure 1 ci-après

Schéma agrégé des interactions induites (queries) : Figure 2 ci-après

FIGURE 1

Interactions directes

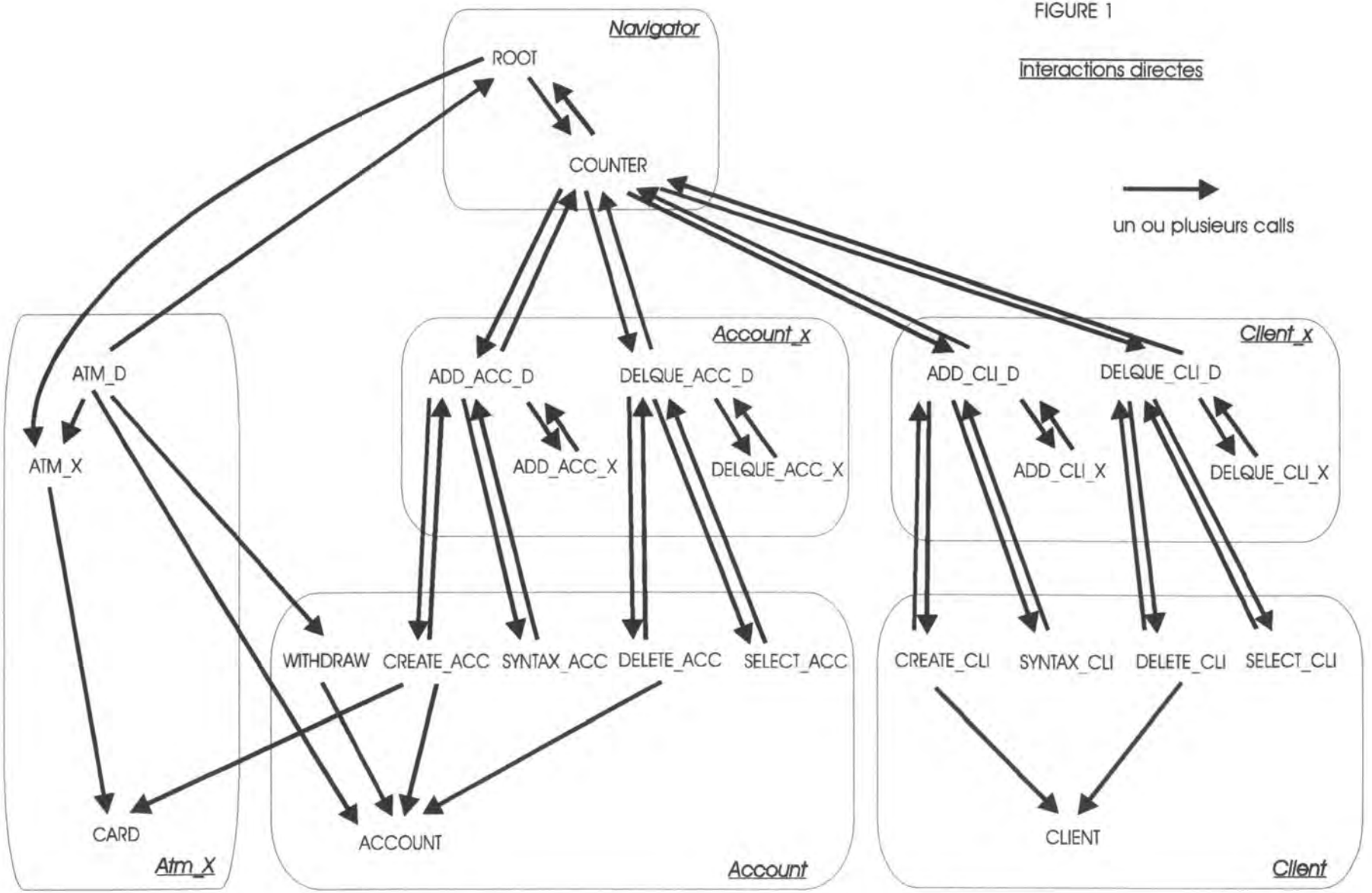
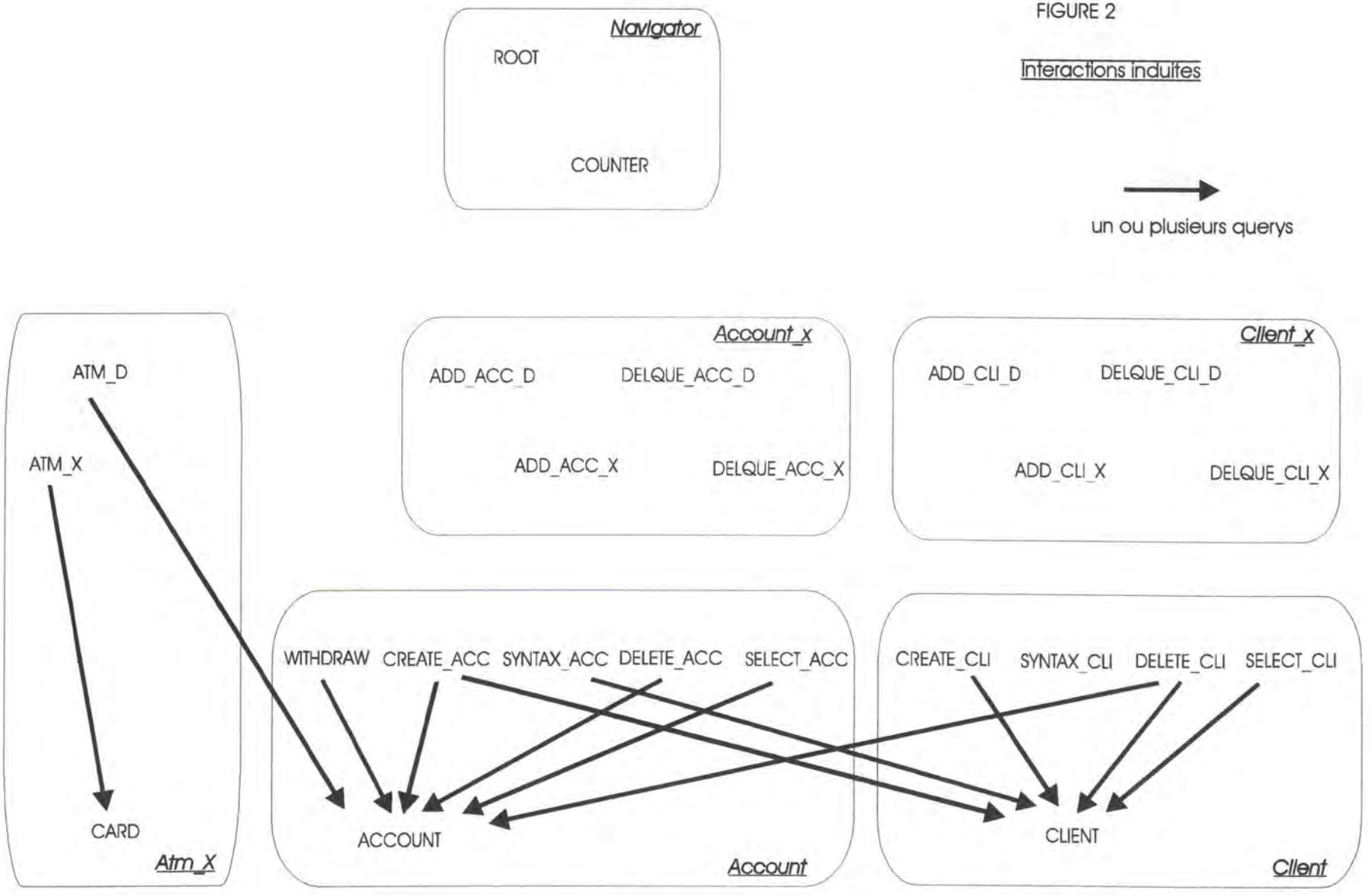


FIGURE 2

Interactions induites



2.7 Conclusion

Le modèle Oblog permet de supporter la phase **d'analyse conceptuelle**. Il **intègre** à la fois un modèle des données puissant (attribut, association, héritage) et un modèle des traitements riche (GTE: comportement local; CALL, queries: comportement externe). Le mécanisme de réification permet la spécification en raffinements successifs.

Oblog est également apte à supporter la phase de **conception technique** (design). L'objet Oblog peut être vu comme un module autonome, uniquement connu de l'extérieur via son interface et dont les services sont organisés de manière cohérente autour de la gestion de son état. Un tel objet peut s'insérer dans une architecture logicielle structurée: hiérarchie basée sur la relation use [Parnas1] .
Les aspects comportementaux sont **intégrés** à cette vue statique des traitements: la relation use est implémentée via interaction directe (partage d'événement; synchronisation) ou induite (consultation externe; autonomie).
Héritage et réification permettent généricité et réutilisation.

Cependant, ce support reste « théorique ». En effet, les restrictions de Oblog Case ne lui permettent de supporter que l' étape de spécification terminale de la phase de design.
La pauvreté du code des traitements (effets sur état) et les limitations des objets persistants conduisent à des éclatements d'objets et à la complexification des comportements locaux et externes de ces objets.
D'autre part, le mécanisme de réification, base de la démarche de spécifications en raffinements successifs est absent dans l'outil. Ce mécanisme aurait permis notamment d'intégrer structurellement (composition; attributs dérivés) et comportementalement (transaction logique) des objets éclatés implémentant un objet de plus haut niveau.

Un mécanisme limité (que l'on peut plutôt voir comme documentaire) permet toutefois de rassembler des objets suivant des critères de cohésion modulaire : l'unité (unit).
Toutefois son interface est peu précise, elle délimite mais n'intègre pas les objets composants et elle ne permet qu'un niveau de regroupement. Ce concept reste donc à être précisé et développé.

Concernant notre objectif d'étude de l'intérêt et de la faisabilité de l'analyse, via ASAX, d'une trace Oblog, les illustrations de notre démarche concernerons essentiellement une trace produite par un système opérationnel dérivé d'une spécification terminale (en l'occurrence Monde Bancaire).
En effet, dans l'état actuel des choses, une trace ne peut être produite et donc analysée que par une application générée à partir d'une spécification terminale (absence de possibilité de prototypage).

Partie 3 : ASAX

3.1 L'analyse d'un système via sa trace

3.1.1 Contexte et intérêt d'une analyse de trace

Le comportement d'un système informatique en activité peut être décrit comme une séquence d'événements/résultats, d'origine interne ou externe, dont ce système peut conserver la trace: audit trail d'un système d'exploitation, trace d'une application, ...
Ce comportement est régi par des règles destinées à assurer le bon fonctionnement du système, c'est à dire un fonctionnement adapté aux respects des objectifs.

Toutefois, bon nombre de scénarii (séquences particulières d'événements/résultats), bien que respectant ces règles, sont susceptibles de nuire au bon fonctionnement du système et/ou d'indiquer la nécessité de son adaptation.

Certains scénarii peuvent être connus mais se révéler trop complexes à maîtriser par le système. D'autres peuvent ne pas avoir été imaginés ou apparaître en raison de lacunes ou d'incohérence des règles du système lui même.

L'analyse rétrospective des événements décrits dans la trace du système s'avère donc particulièrement intéressante pour détecter des comportements nuisant ou susceptibles de nuire à la poursuite des objectifs du système.
Par exemple, une analyse des audit trails produits par un système d'exploitation peut être menée pour détecter des atteintes à la sécurité mais aussi pour suivre l'évolution de la charge du système. Les mêmes raisons peuvent justifier une analyse de la trace d'une application.

3.1.2 Analyse d'une trace: problèmes, approches et spécifications d'outil.

Les traces produites par les systèmes informatiques présentent généralement des caractéristiques susceptibles de poser des problèmes d'analyse.

- Un premier problème réside dans la disparité et l'éventuelle complexité des scénarii susceptibles de nuire au bon fonctionnement du système. Il suffit pour se convaincre de ce fait de tenter de produire la liste des scénarii connus d'atteintes à l'objectif de sécurité d'un système d'exploitation.

Face à cette diversité et à cette complexité, deux approches d'analyse se dégagent. Une première approche consiste à construire un modèle statistique du comportement normal d'un système et de ses acteurs. Une détection et un suivi des scénarii suspects eu égard à ce modèle permettra un approfondissement d'analyse.

Une deuxième approche consiste à détecter des scénarii connus d'entorses au bon fonctionnement du système.
Les deux approches peuvent bien sûr s'enrichir mutuellement.

▪ Un deuxième problème naît du volume souvent énorme de données contenues dans la trace d'un système. L'analyse d'une telle trace s'avère en pratique impossible sans techniques à la fois efficaces et appropriées de filtrage et/ou de sélection.

Un outil automatisé capable de supporter l'analyse d'une trace doit s'avérer suffisamment puissant et efficace pour résoudre ces problèmes.
Cet outil devrait également pouvoir supporter l'analyse de traces de tout format produites par des systèmes divers (réutilisabilité physique).

Idéalement, l'outil devrait permettre à l'analyste d'exprimer ses requêtes au travers d'un modèle logique (modèle logique de sécurité [Denning1], modèle d'objets actifs [Rolland1]) propre à son centre d'intérêt. L'outil effectuerait la traduction de questions de haut niveau, exprimées en termes de composants abstraits propres au modèle, en questions de bas niveau appropriées au traitement de la trace d'un système particulier. L'analyse serait dans ce cas portable d'un système à l'autre (réutilisabilité logique) et favoriserait les comparaisons entre systèmes.

Dans le domaine de la sécurité, par exemple le modèle IDES [Denning1] de détection d'intrusion propose de voir l'activité d'un système comme une suite événements (Sujet, Objet, résultat, référence temporelle). L'outil associé permet de construire un modèle statistique des comportements des sujets et de leur associer des profils standards. L'analyste définit des règles abstraites, indépendantes du système natif, chargées de détecter des scénarii logiques prédéfinis et des écarts aux profils standards.

Remarquons enfin que, contrairement à des outils existants qui n'apportent des réponses qu'à des requêtes prédéfinies, l'outil devrait permettre à l'utilisateur de définir de manière souple de nouvelles requêtes.

3.2 Objectifs du projet ASAX

Le projet ASAX (**A**dvanced **S**ecurity **A**udit Trail Analysis on uni**X**), initié par l'Institut d'Informatique de Namur et Siemens Nixdorf Software S.A., a pour but de définir et d'implémenter un outil de support à l'analyse d'audit trails dans le domaine de la sécurité [ASAX1] [ASAX2] .
Les qualités majeures attendues de l'outil, à savoir l'universalité, l'efficacité et la puissance, ont guidé la spécification, la conception et l'implémentation d'une version prototype.
Ces qualités ont été éprouvées à travers la mise en oeuvre de systèmes d'analyse d'audit trails correspondant au niveau de sécurité B3, selon la classification du Département de la Défense américain [DOD1], sur différents systèmes d'exploitation.

L'**Universalité** est visée en proposant un format normalisé de fichier d'audit (NADF) particulièrement simple et souple. Ce format est universel en ce sens que tout audit trail peut théoriquement être traduit en fichier NADF. L'analyse sera uniquement réalisée sur des audit trails normalisés. Des programmes appelés Format Adaptators seront fournis pour traduire les audit trails natifs en format normalisé.
La faisabilité de cette approche a été démontrée par la mise au point de Format Adaptators pour les audit trails de nombreux systèmes d'exploitation.
L'universalité assure la réutilisabilité au niveau physique.

La **Puissance** est fournie par une sorte de langage à base de règles (**RU**le-ba**S**ed Sequence **E**valuation **L**anguage) permettant d'exprimer des critères de sélection complexes concernant des séquences d'enregistrements arbitrairement longues mais aussi permettant de traiter séquentiellement le fichier d'audit en une seule passe. Ce dernier point est obligatoire pour assurer l'efficacité dés que la masse de données à traiter est très importante.
Le principe de base est que l'information sur le passé est stockée dans un ensemble de règles d'évaluation actives qui seront utilisées pour analyser l'enregistrement suivant. Ces règles pourront également déclencher de nouvelles règles pour l'analyse du reste.

L'**Efficacité** sera fournie d'une part par le principe même de RUSSEL permettant de traiter chaque enregistrement une seule fois et, d'autre part, par des techniques d'implémentation efficaces.

Ces trois qualités fondamentales font de ASAX un outil adéquat et désormais éprouvé pour résoudre les problèmes précédemment évoqués de l'analyse des audits trails dans le domaine de la sécurité. Ces problèmes apparaissent toutefois plus généralement dans l'analyse des traces de systèmes informatiques quelconques. D'autre part, le domaine de la sécurité n'est qu'un cas particulier des domaines d'intérêt que peut recouvrir ce type d'analyse.
Dans la suite du mémoire, nous nous efforcerons d'illustrer l'adéquation de l'outil ASAX pour l'analyse d'une trace produite par un système d'objets actifs communiquant par événements partagés, spécifié grâce à l'outil Oblog.
Un Format Adaptator de trace Oblog sera fourni et des exemples de règles RUSSEL illustrerons la puissance du langage dans des domaines d'intérêt divers.

3.3 Introduction au langage Russel

Avant de présenter la description logique du langage Russel, il nous semble adéquat de l'illustrer par un exemple introductif et de dégager les points clés du langage.

L'exemple suivant présente une règle Russel dont l'objectif est de détecter, sur base d'un audit trail de format NADF, des tentatives répétées et infructueuses de 'login' sur un système, d'un même utilisateur, dans un intervalle de temps donné.
Précisons que les enregistrements d'audit décrivant les événements de login ne sont pas nécessairement consécutifs étant donné que bien d'autres événements systèmes peuvent être audités entre deux tentatives du même utilisateur.

Deux règles sont nécessaires pour détecter une séquence de login's infructueux du même utilisateur.

La première (*failed_login*) analyse l'enregistrement courant.
Si cet enregistrement décrit une tentative de login refusée, *failed_login* déclenche (trigger off) la règle *count_rule* en lui confiant l'identité de l'utilisateur, un compteur de login's infructueux et un délai d'expiration.
Elle se perpétue ensuite pour l'analyse de l'enregistrement suivant.

La deuxième règle (*count_rule*), d'argument *userid*, *countdown* et *expiration*, se perpétue (en se redéclenchant explicitement) tant qu'elle ne constate pas l'épuisement de *countdown* ou le dépassement de *expiration.*

Si elle détecte un login infructueux pour l'utilisateur userid sans dépassement de *expiration*, elle se redéclenche elle même en décrémentant l'argument *countdown*. Si ce dernier atteint 0, la règle envoie un message d'alarme et s'arrête (non redéclenchement).
Si un dépassement de *expiration* est constaté, la règle s'arrête.
Dans les autres cas, elle se perpétue avec les mêmes arguments.

De manière à initialiser le processus d'analyse, la règle spéciale d'initialisation *init_action* rend la règle *failed_login* active pour le premier enregistrement. Elle requiert également l'activation d'une troisième règle *print_results* à la fin du processus d'analyse.
Cette dernière règle est utilisée pour imprimer le nombre total de séquences de login's infructueux repérées durant toute l'analyse.
Elle utilise pour ce faire une variable globale *v*, permanente à la session d'analyse, qui est incrémentée après l'envoi de chaque message d'alarme.

Ces règles s'expriment en Russel de la manière suivante:

```
global v: integer;

rule failed_login(maxtimes, duration: integer);
begin
 if    EVENT = 'login' and RESULT = 'failure'
       --> trigger off for_next
                count_rule(USER, maxtimes - 1, TIMESTP + duration)
 fi;
 trigger off for_next failed_login(maxtimes, duration: integer)
end;

rule count_rule(userid: string; countdown, expiration: integer);
if     EVENT = 'login' and RESULT = 'failure'
       and USER = userid and TIMESTP < expiration
       --> if countdown > 1
            -->    trigger off for_next
                        count_rule(userid, countdown - 1, expiration);
            countdown = 1
            -->    begin
                    SendMessage( 'Too much failed login's for ', userid);
                    v := v + 1
                   end
          fi;
       TIMESTP > expiration
       --> skip;
       true
       --> trigger off for_next count_rule(userid, countdown, expiration)
fi;

rule print_results;
Println( v , ' sequences of failed login''s found');

init_action;
begin
 v := 0;
 trigger off for_next failed_login(3, 120);
 trigger off at_completion print_results
end.
```

Remarquons que dans la figure précédente, les noms des champs d'audit sont représentés en majuscules, les noms de procédures externes en italique et les mots clés du langage en gras.

Cet exemple illustre les aspects novateurs du langage Russel qui a été spécialement conçu pour résoudre le problème de la recherche de 'patrons' arbitraires d'enregistrements dans des fichiers séquentiels (où l'ordre des enregistrements joue un rôle essentiel).

L'aspect le plus caractéristique du langage est le mécanisme 'built-in' de déclenchement de règle qui permet l'analyse en une seule passe du fichier séquentiel, de 'gauche à droite'.
Russel introduit un nouveau type de programmation, que l'on pourrait qualifier de 'programmation chronologique'.
Le langage, bien qu'essentiellement procédural, et tirant parti de la puissance du procédural, permet cependant d'écrire des règles dans une 'intention' déclarative du type
($Condition_1$ --> $Action_1$; ... ; $Condition_n$ --> $Action_n$)

Le langage Russel fournit des structures de contrôle traditionnelles telles que la condition, la répétition, et la composition des actions. Les actions primitives incluent l'assignation, l'appel de routines externes et le déclenchement.
Un programme Russel consiste simplement en un ensemble de déclarations de règles qui sont composées d'un nom de règle, d'une liste de paramètres formels et de variables locales ainsi que d'une partie action.
Russel supporte aussi le partage de variables globales entre modules et les déclarations de règles exportées. Le lecteur intéressé par ces deux dernières possibilités peut consulter la référence [ASAX3] pour plus de détails.

La sémantique opérationnelle de Russel peut être résumée comme suit:

- les enregistrements sont analysés séquentiellement. L'analyse de l'enregistrement courant consiste à exécuter toutes les règles actives. L'exécution d'une règle active peut déclencher de nouvelles règles, émettre des alarmes, écrire des messages de compte rendu ou modifier des variables globales, etc;

- le déclenchement de règle est un mécanisme spécial grâce auquel une règle est rendue active pour l'enregistrement courant ou pour le suivant. En général, une règle est active pour l'enregistrement courant parce qu'un préfixe d'une séquence particulière d'enregistrements d'audit a été détecté (Le reste de cette séquence doit encore éventuellement être trouvée dans le reste du fichier). Les paramètres réels dans l'ensemble des règles actives représentent la connaissance au sujet de la sous séquence déjà trouvée. Cette connaissance est utile pour sélectionner les enregistrements ultérieurs éventuels de la séquence;

- lorsque toutes les règles actives pour l'enregistrement courant ont été exécutées, l'enregistrement suivant est lu et les règles déclenchées pour l'analyser, dans l'étape précédente sont exécutées à leur tour;

- pour initialiser le processus, un ensemble de règles, appelées règles d'initialisation, sont déclenchées pour le premier enregistrement.

Des routines C, prédéfinies ou construites par l'utilisateur, peuvent être appelées à partir du corps d'une règle. Une interface simple et clairement spécifiée avec le langage C permet de compléter le langage Russel notamment pour simuler des structures de données complexes, pour envoyer des messages d'alarme, pour verrouiller un compte bancaire en cas de violation de la sécurité, ...

3.4 Définition logique du langage RUSSEL

3.4.1 Syntaxe abstraite sommaire d'une règle Russel

Domaines syntaxiques

A	actions	O	opérateurs arithmétiques
B	opérateurs logiques	P	paramètres formels
C	conditions	Q	noms de règles
E	expressions	R	règles
F	noms de champs	S	opérateurs relationnels
G	déclarations de paramètres	T	types
H	déclarations de variables	V	variables locales
L	littéraux	X	noms de fonctions externes
M	modes de déclenchement	Y	noms de procédures externes

R ::= **rule** Q (... ; G ; ...) ; ... ; H ; ... ; A

G ::= P : T

H ::= V : T

A ::= V := E | Y (... , E , ...)
| **trigger off** M Q (... , E , ...) | **begin** ... ; A ; ... **end**
| **do** ... ; C --> A ; ... **od** | **if** ... ; C--> A ; ... **fi**

C ::= **true** | **present** F | **not** C | C B C | E S E

E ::= L | V | F | P | -E | E O E | X (... , E , ...)

B ::= **and** | **or**

O ::= **+** | **-** | ***** | **div** | **mod**

S ::= **>** | **<** | **=** | **!=** | ≥ | ≤

M ::= **for_current** | **for_ next** | **at_completion**

T ::= **integer** | **string**

où ... ; W ; ... séquence éventuellement vide de W séparés par des ;

3.4.2 Quelques définitions

Les définitions qui suivent permettront de faciliter la description de la sémantique du langage.

▪ Enregistrement courant (CR)
L'exécution d'un programme ASAX est associée à l'analyse d'un audit trail entier. A un moment donné de cette exécution, un certain ensemble de règles actives traitent un simple enregistrement. Lorsque toutes ces règles ont été exécutées eu égard à cet enregistrement, l'enregistrement d'audit suivant est traité à son tour. Ce traitement est répété jusqu'à la fin du fichier d'audit. L'enregistrement traité à un moment donné est appelé l'enregistrement courant. L'enregistrement courant est constitué d'un ensemble de champs. Un champ contient logiquement deux composants : un nom de champ et une valeur de champ (au niveau physique, nous verrons plus tard qu'un champ sera en pratique implémenté à l'aide de trois composants: un identifiant, une longueur et une valeur).

▪ Environnement courant (CE)
L'environnement courant est à un moment donné constitué des éléments suivants:

. l'enregistrement courant;
. l'environnement local: un ensemble de variables;
. Dstrig: l'ensemble des règles actives (ou déclenchées);
. Dsnext: l'ensemble des règles à déclencher pour l'enregistrement suivant;
. Dscompl: l'ensemble des règles de terminaison, c'est à dire l'ensemble des règles à déclencher quand le traitement du fichier complet est achevé.

▪ Schéma de règle
Un schéma de règle est une déclaration de règle qui respecte la syntaxe décrite dans le paragraphe précédant.

▪ Règle
Une règle consiste en un schéma de règle et une liste de valeurs à substituer à chaque nom de paramètre.

3.4.3 Sémantique partielle du langage

▪ <u>Evaluation des expressions et des conditions</u>

Cette évaluation est réalisée de manière classique et nous n'en donnerons pas le détail ici.
Les éléments de base pour cette évaluation sont les champs d'audit (de type string), les variables et paramètres de règles (de type integer ou string), les constantes (de type integer ou string ou booléennes) et les fonctions externes (de type integer ou string).
Les opérateurs traditionnels (arithmétiques, relationnels et de comparaisons) associés à ces types, interviennent dans l'évaluation des expressions et conditions complexes.

Précisons toutefois deux points importants:

. la condition

present nom_de_champ

est évaluée à **true** si CR contient un nom de champ de nom *nom_de_champ*; elle est évaluée à **false** sinon;

. l'expression

nom_de_champ

est évaluée à *s* si *s* est la valeur du champ *nom_de_champ* dans CR. Elle est évaluée à une valeur arbitraire (sans détection d'erreur) si CR ne contient pas de champ de ce nom.

▪ Exécution d'une action

A) **instruction skip**

skip est l'instruction vide.

B) **assignation**

Soit rexpr une expression droite ;
var un nom de variable;

l'exécution de l'action:

var := rexpr

eu égard à CE est évaluée comme suit:

1. rexpr est évaluée eu égard à CE. Soit v sa valeur;
2. var reçoit la valeur v.

C) **actions conditionnelles**

Soit $cond_1$, ... , $cond_n$ des expressions booléennes
$action_1$, ..., $action_n$ une séquence d'actions (n >= 1)

l'exécution de l'instruction:

if
$cond_1$ --> $action_1$;
.
.
.
$cond_n$ --> $action_n$
fi

est effectuée comme suit:

1. $cond_1$ est exécutée eu égard à CE. Soit v sa valeur.
2. si v = **true** l'action $action_1$ est exécutée eu égard à CE et l'exécution est terminée; sinon

(a) si n > 1 l'instruction:

if
$cond_2$ --> $action_1$;
.
.
.
$cond_n$ --> $action_n$
fi

est exécutée eu égard à CE.
(b) sinon (v = **false** et n = 1) l'exécution est terminée.

D) **actions répétitives**

Sous les hypothèses du point précédent, l'exécution de l'instruction:

do
$cond_1$ --> $action_1$;
.
.
.
$cond_n$ --> $action_n$
od

est effectuée comme suit eu égard à CE:

1. les conditions $cond_1$, ... , $cond_n$ sont successivement évaluées eu égard à CE jusqu'à ce que l'une d'entre elles (s'il en existe) retourne une valeur **true**. Soit $cond_i$ ($1 \leq i \leq n$) une telle condition. Dans ce cas, l'action $action_i$ est exécutée et ensuite toute l'instruction

do

$cond_1$ --> $action_1$;
.
.
.
$cond_n$ --> $action_n$

od

est exécutée une nouvelle fois.

2. si aucune des conditions $cond_1$, ... , $cond_n$ n'est évaluée à **true**, l'exécution est terminée.

E) **actions composées**

Soit $action_1$, ..., $action_n$ une séquence d'actions ($n > 0$)

L'exécution de l'instruction:

begin $action_1$; ... ; $action_n$ **end**

eu égard à CE consiste à exécuter successivement chacune des actions $action_1$, ..., $action_n$ eu égard à CE.

F) **déclenchement de règles**

Supposons que

rule rule_name (parameter_list);
local_varible_list;
rule_body

est un schéma de règle R.

L'exécution de l'action:

Trigger off triggering_mode rule_name ($expr_1$, ..., $expr_n$) , (n>=0)

est la suivante:

1. les expressions $expr_1$, ..., $expr_n$ sont évaluées eu égard à CE.
   Soit L = { v1,..., vn } la liste de leurs valeurs respectives;

2. (a) si triggering_mode = **for_current**
alors l'effet de l'exécution est d'ajouter une nouvelle règle à l'ensemble des règles actives:

$$Dstrig = Dstrig \cup \{ (R,L) \}$$

où (R,L) est la règle déclenchée identifiée par son nom et sa liste de paramètres effectifs L.

(b) si triggering_mode = **for_next**
alors l'effet de l'exécution est de définir un nouvel environnement courant NCE identique à CE excepté pour Dsnext qui devient:

$$Dsnext = Dsnext \cup \{ (R,L) \}$$

où (R,L) est la règle déclenchée identifiée par son nom et sa liste de paramètres effectifs L.

(b) si triggering_mode = **at_completion**
alors l'effet de l'exécution est de définir un nouvel environnement courant NCE identique à CE excepté pour Dscompl qui devient:

$$Dscompl = Dscompl \cup \{ (R,L) \}$$

où (R,L) est la règle déclenchée identifiée par son nom et sa liste de paramètres effectifs L.

G) **appel de procédure prédéfinie**

L'exécution de l'appel de procédure prédéfinie:

$$procedure_name\ (expr_1, ..., expr_n),\ (n>=0)$$

se déroule comme suit:

1. les expressions $expr_1, ..., expr_n$ sont évaluées eu égard à CE. Soit v1,..., vn la liste de leurs valeurs respectives;
2. l'appel de procédure prédéfinie:

$$procedure_name\ (expr_1, ..., expr_n),\ (n>=0)$$

est exécuté eu égard à CE.

- Vue globale du traitement complet du fichier d'audit

inputs:

. l'audit trail en format NADF
. un ensemble de schémas de règles dont une règle d'initialisation init_action décrite comme suit:

init_action;
begin
trigger off for_next RI_1;
:
trigger off for_next RI_n;
trigger off at_completion RC_1;
:
trigger off at_completion RC_p
end

outputs: tout espèce d'output généré durant le traitement.

Algorithme:

Dstrig := { init_action };
Dscompl := { };
Dsnext := { };
Ouvrir l'audit trail;

Tant que pas fin de audit trail faire
begin
lire l'enregistrement d'audit suivant;
Traiter Dstrig qui produira Dsnext et Nscompl;
Dstrig := Dsnext;
Dsnext := { };
Dscompl := Dscompl ∪ Nscompl;
end;

Fermer l'audit trail;
Traiter Dscompl.

Remarque: l'audit trail contient un premier enregistrement de contrôle bidon.

3.5 Le prototype ASAX

Le chapitre précédent a décrit la spécification de l'évaluateur de règles, indépendamment du système d'exploitation sous-jacent et du format d'audit trail.
Le prototype s'exécute sur un système Unix et traite des audit trails en format NADF.
Ce chapitre décrit l'environnement d'exécution du prototype : arguments et formats des arguments fournis par l'utilisateur; description d'une session ASAX.

3.5.1 Le format NADF et le Format Adaptator

ASAX traite des enregistrements d'audit dans un format normalisé (NADF) souple et efficace dont nous présenterons le détail ultérieurement (section 4.3).
Un enregistrement de données d'audit de format NADF consiste en une liste de données d'audit dont chacune est représentée par son identifiant entier, sa longueur réelle et sa valeur. La liste est triée dans l'ordre ascendant des identifiants pour accroître la performance du traitement des données d'audit.
La souplesse du format NADF permet, en principe, de traduire tout audit trail natif en un audit trail de format normalisé. Le programme qui assure cette traduction est appelé un Format Adaptator. Outre cette traduction, ce programme doit fournir une table d'association entre les noms externes des données d'audit et leurs identifiants NADF.

3.5.2 La librairie des procédures préprogrammées

ASAX ne permet pas de programmer tous les types de procédures. Les seuls types de données qu'il autorise à ce jour sont les strings et les entiers pour les variables et les paramètres et les strings pour les audit data.
Toutefois, les services d'une librairie de procédures prédéfinies ou préprogrammées peuvent être invoqués à partir des règles RUSSEL: envoi de messages d'alerte, écriture d'enregistrements sélectionnés dans un fichier temporaire, impression d'une table statistique, ...
Cette librairie peut être enrichie de manière très souple par l'utilisateur.

3.5.3 Description d'une session ASAX

Durant une session ASAX, l'utilisateur doit fournir:

- le fichier d'audit en format NADF;
- un éventuel nom de fichier output;
- un fichier texte associant noms externes et identifiants des données d'audit;
- un fichier texte contenant les schémas de règles et la règle d'initialisation;
- la librairie des procédures prédéfinies susceptibles d'être invoquées dans les règles.

Les règles sont d'abord analysées et contrôlées pour les erreurs syntaxiques et sémantiques. En l'absence de détection de telles erreurs, le système sauve ces règles dans une

représentation interne. Si ces règles référencent certaines procédures externes, ces dernières sont linkées avec l'évaluateur de règles. Le résultat est un programme prêt pour exécution. La règle initiale est alors exécutée et le fichier d'audit ouvert. Le traitement d'analyse peut alors commencer et le premier enregistrement est analysé par les règles actives courantes déclenchées par la règle initiale. Le fichier output est garni avec les messages et sélections émises par les règles.

3.6 L'efficacité de ASAX

L'efficacité de ASAX est assurée grâce à la structure du langage Russel, qui permet de réaliser l'analyse d'un audit trail en une seule passe, et en optimisant les étapes répétitives par des techniques d'implémentation efficaces dont le détail sort du cadre de ce travail. Décrivons toutefois deux techniques simples mais efficaces concourant à cette optimisation:

- lors de l'évaluation de ($expr_1$ **or** $expr_2$ ... **or** $expr_n$), les expressions composantes sont évaluées dans une séquence qui peut ne pas correspondre à leur ordre d'apparition dans le texte de l'expression. Dès que, dans cette séquence, une expression composante est évaluée à **true**, l'expression composée est évaluée à **true** et les expressions composantes restantes ne sont pas évaluées.

- avant d'appliquer les règles actives, l'enregistrement courant est lu en mémoire et est ensuite parcouru pour garnir une table d'indirection constituée d'identifiants et de pointeurs vers les données d'audit associées à ces identifiants. En fait un squelette de cette table est alloué une fois pour toutes en mémoire au début du processus. Les identifants possibles y sont rangés en ordre ascendant de sorte que la lecture séquentielle des audit data d'un enregistrement (eux aussi rangés en ordre ascendant sur les identifiants) peut être menée en parallèle avec la garniture de la table.
Toutes les références ultérieures à une donnée d'audit effectuées par les règles actives s'effectueront de manière optimisée via cette table d'indirection.

3.7 Les extensions préconisées de ASAX

Les auteurs du projet ont prévu diverses possibilités d'extension de ASAX:

- le support d'un modèle logique de sécurité (réutilisabilité logique);
- le développement d'une interface conviviale pour les non experts;
- le développement d'un générateur de Format Adaptator sur base d'une description déclarative d'un audit trail natif:
- la gestion de structures de données complexes dans le langage de règles.

Nous proposerons d'autres possibilités d'extensions à la fin de ce mémoire.

Partie 4 : Trace Oblog et Format Adaptator

4.1 Le problème de traduction

Toute analyse ASAX s'exerce sur un fichier argument qui respecte un format normalisé appelé NADF. Il importe donc de s'assurer qu'une trace Oblog peut être traduite sans trop de difficultés dans ce format normalisé, avant même de réfléchir à l'intérêt et à la faisabilité d'une analyse de cette trace grâce au langage d'interrogation Russel.
Une analyse ASAX réclame également comme deuxième argument un fichier auxiliaire décrivant la correspondance entre les noms externes (utilisés dans les requêtes Russel) des champs de données d'un enregistrement NADF et leurs identifiants numériques internes. La manière de construire ce fichier intermédiaire doit également être étudiée.

Avant même de se préoccuper de la construction de ces fichiers arguments, une première appréciation de la trace Oblog s'avère toutefois indispensable, du point de vue conceptuel: la vie d'un système Oblog peut-elle être raisonnablement reconstituée, en termes des concepts du modèle, à partir des informations présentes dans la trace?

4.2 La trace OBLOG

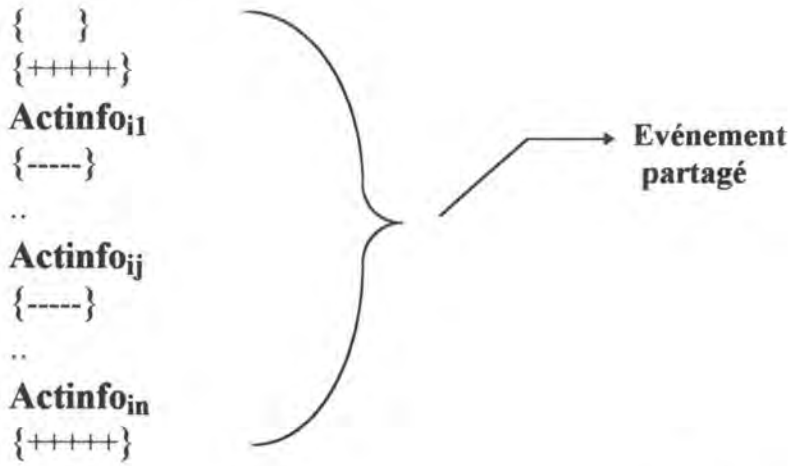
4.2.1 Contexte et structure globale:

Une fois réalisée la spécification terminale d'un système d'objets, l'outil Oblog Case permet de générer le code source de l'application. La compilation de ce code produit une version exécutable de l'application.
Il est possible de lancer une exécution de l'application en demandant la production d'une trace associée. A l'apparition de tout événement dans le cycle de vie d'un objet (événement isolé) ou de plusieurs objets (événement partagé), des informations (Evtinfo) concernant cet événement et ses effets sont ajoutées chronologiquement dans la trace. Rappelons l'assimilation qui est faite en Oblog entre un événement élémentaire (événement isolé ou événement participant) et l'action qu'il déclenche dans un certain contexte.

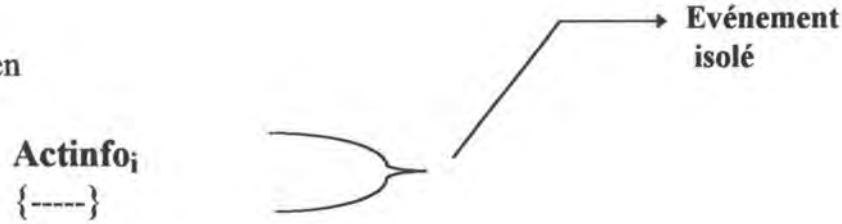
La trace prend la forme suivante:

Evtinfo$_1$
..
Evtinfo$_i$
..
Evtinfo$_n$

où **Evtinfo$_i$** est de la forme:



ou bien



Exemple d'une trace partielle d'exécution de Monde Bancaire:

```
Object ROOT      (3319488) Action boot      Situation booted
Attribute ATMApp = NULL
Attribute bankApp = NULL
-------------------------------------------------------------------------------
Object ROOT      (3319488) Action open      Situation waitUser
Attribute ATMApp = NULL
Attribute bankApp = NULL
-------------------------------------------------------------------------------

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Object COUNTER   (3325104) Action create    Situation started   Caller ROOT     (3319488)
Event Attribute myMaster = OBJECT NAME: ROOT            OBJECT ADDRESS:3319488
Attribute myMaster = OBJECT NAME: ROOT            OBJECT ADDRESS:3319488
-------------------------------------------------------------------------------
Object ROOT      (3319488) Action optionBank Situation userChose
Event Attribute selfRef = OBJECT NAME: ROOT             OBJECT ADDRESS:3319488
Attribute ATMApp = NULL
Attribute bankApp = OBJECT NAME: COUNTER          OBJECT ADDRESS:3325104
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Object COUNTER   (3325104) Action open      Situation waitUser
Attribute myMaster = OBJECT NAME: ROOT            OBJECT ADDRESS:3319488
-------------------------------------------------------------------------------
Object ROOT      (3319488) Action close     Situation tempClosed
Attribute ATMApp = NULL
Attribute bankApp = OBJECT NAME: COUNTER          OBJECT ADDRESS:3325104
-------------------------------------------------------------------------------

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Object ADD_CLI_X (3327088) Action create    Situation closed    Caller ADD_CLI_D (3325904)
Event Attribute myDriver = OBJECT NAME: ADD_CLI_D       OBJECT ADDRESS:3325904
Attribute Name = STRING: Empty String.
Attribute Street = STRING: Empty String.
Attribute message = STRING: Empty String.
Attribute ZIP = STRING: Empty String.
Attribute Town = STRING: Empty String.
Attribute CliNum = STRING: Empty String.
Attribute Phone = STRING: Empty String.
Attribute Title = STRING: Empty String.
Attribute myDriver = OBJECT NAME: ADD_CLI_D       OBJECT ADDRESS:3325904
-------------------------------------------------------------------------------
Object ADD_CLI_D  (3325904) Action create    Situation active    Caller COUNTER  (3325104)
Event Attribute myDialog = OBJECT NAME: ADD_CLI_X       OBJECT ADDRESS:3327088
Event Attribute myself = OBJECT NAME: ADD_CLI_D       OBJECT ADDRESS:3325904
Event Attribute counterRef = OBJECT NAME: COUNTER       OBJECT ADDRESS:3325104
Attribute created_msg = STRING: 'Client was created !' (len=20)
Attribute exists_msg = STRING: 'Client already exists!' (len=22)
Attribute syntaxErr_msg = STRING: 'Syntax Error !' (len=14)
Attribute status = SYN_CLI_S: ERROR
Attribute counterRef = OBJECT NAME: COUNTER       OBJECT ADDRESS:3325104
Attribute myDialog = OBJECT NAME: ADD_CLI_X       OBJECT ADDRESS:3327088
-------------------------------------------------------------------------------
Object COUNTER   (3325104) Action insClient  Situation userChose
Event Attribute selfRef = OBJECT NAME: COUNTER          OBJECT ADDRESS:3325104
Event Attribute addCliDRef = OBJECT NAME: ADD_CLI_D     OBJECT ADDRESS:3325904
Attribute myMaster = OBJECT NAME: ROOT            OBJECT ADDRESS:3319488
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

La partie de trace précédente contient six événements dont quatre isolés (actions) et deux partagés (méga-actions). Elle comporte neuf événements élémentaires (actions).

A chaque événement élémentaire (événement isolé ou événement participant) sont associées les informations suivantes concernant l'objet concerné:

Ligne d'en-tête:

c	Nom de la classe de l'objet
i	Valeur de l'identifiant unique de l'objet
a	Nom de l'action déclenchée par cet événement
s	Situation finale de l'objet après cette action
cc	Nom de la classe de l'objet appelant (**Si** l'action est appelée)
ci	Valeur de l'identifiant de l'objet appelant (**Si** l'action est appelée)

Object c (i) **Action** a **Situation** s [**Caller** cc (ci)]

Pour chaque paramètre éventuel de l'action:

Nom (pn) et valeur (pv) du paramètre

Event Attribute pn = pv

Pour chaque attribut éventuel de l'objet concerné:

Nom (an) et valeur (av) de l'attribut après exécution de l'action:

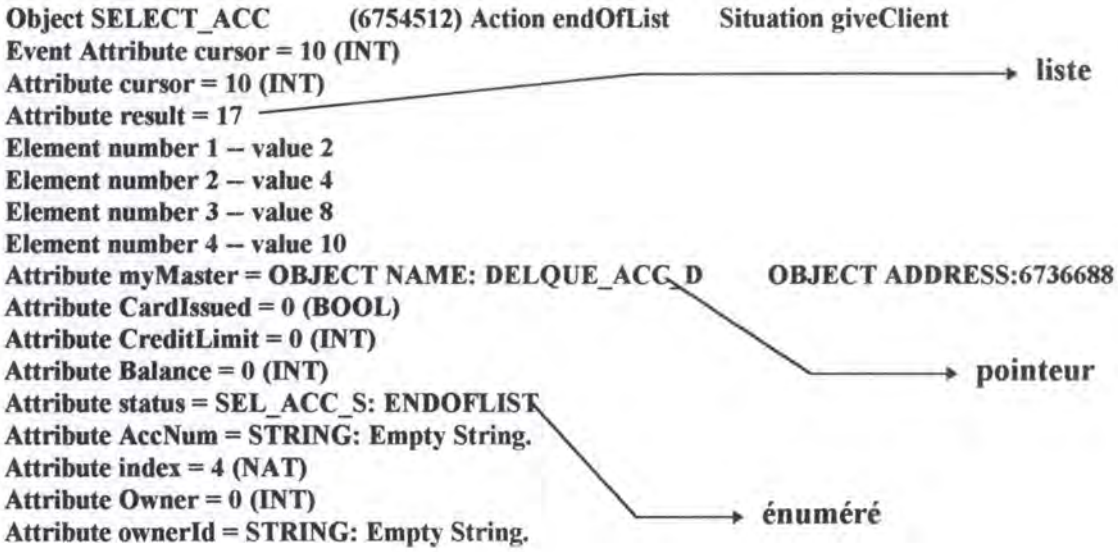
Attribute pn = pv

Remarques:

▪ Si *a* est une action de mort de l'objet, la situation *s* est conventionnellement posé à '???'. Dans ce cas les paramètres d'action et les attributs de l'objet n'apparaissent pas dans la trace.

▪ Si *a* n'est pas une action de mort et si l'objet est de type TBL, *s* est conventionnellement posé à '...'.

▪ Les éléments qui composent la valeur d'un attribut de type liste peuvent apparaître sur plusieurs lignes comme le montre l'exemple suivant.



▪ Les événements participants à un événement partagé sont simultanés. Ils apparaissent toutefois dans la trace en fonction de leur position dans l'arborescence de calls associée à cet événement partagé. Cette arborescence est parcourue de gauche à droite et de bas en haut (une action appelée apparaît dans la trace avant l'action appelante bien que simultanée). L'événement participant (action) associé à la racine de l'arborescence apparaît donc le dernier du groupe des participants.

▪ La trace produite par une application Oblog ne produit pas en standard les champs *i*, *cc* et *ci* (identifiant d'objet , classe et identifiant de l'éventuel appeleur).
A notre demande, un test satisfaisant d'adaptation du générateur de trace a été réalisé par Oblog Software de manière à produire des traces enrichies de ces champs. Une trace significative de l'application Monde Bancaire a été produite suivant ce modèle de trace enrichi et servira de base pour les exemples d'analyse d'une trace Oblog grâce à l'outil ASAX.
En effet, ces champs sont essentiels à l'analyse. Le champ *i* est nécessaire : supposons, par exemple que trois objets instances d'une même classe X évoluent de manière concourante; sans ce champ, il s'avère impossible d'associer tout événement de classe X à l'une des trois instances; par conséquent un suivi de la dynamique d'une instance est impossible. Un raisonnement de même type montre la nécessité des champs *cc* et *ci* pour le suivi du comportement (interactions directes) des instances.
Ces champs permettent de détecter « qui est qui » et « qui parle avec qui ».

Le modèle de trace enrichie présenté ci-avant constitue une spécification minimale de trace Oblog permettant une analyse significative via l'outil ASAX.

4.2.2 Commentaires sur la trace Oblog du point de vue conceptuel

Oblog propose un modèle du SI qui fait partie de la classe plus importante des modèles de système d'objets actifs communiquant par partage d'événement. Le modèle de trace Oblog est, à quelques détails près, suffisamment riche et souple pour décrire l'exécution d'une application spécifiée dans le cadre d'un modèle quelconque de cette classe.

Logiquement, une trace Oblog peut être vue comme une séquence chronologique d'enregistrements décrivant l'activité du système. Un enregistrement décrit:

- soit un événement isolé qui dans un certain contexte déclenche une action sur une instance d'objet (instance de classe *c* et d'identifiant *i*) avec un certain résultat sur sa mémoire (attributs, situation)

- soit un événement partagé (composé d'événements participants dans leurs contextes respectifs) qui déclenche une action dans chacun des objets concernés avec pour chacun des objets un certain résultat sur sa mémoire.

Evt --> Action --> (Class,Id) --> (Attributes,Situation)

$$\mathbf{Evt} \rightarrow \left\{ \begin{array}{l} .. \\ \mathbf{Evt_j} \rightarrow \mathbf{Action_j} \rightarrow \mathbf{(Class_j, Id_j)} \rightarrow \mathbf{(Attributes_j, Situation_j)} \\ .. \end{array} \right.$$

Toute requête d'analyse qui s'inscrit dans la logique du modèle de trace pourra donc être réutilisable pour l'étude de toute trace de système d'objets actifs communiquant par partage d'événements.

Remarquons toutefois une faiblesse importante dans le contenu informatif de la trace Oblog qui sera de nature à réduire le champ d'investigation des requêtes d'analyse:
le contexte des événements n'apparaît que partiellement dans la trace au travers des valeurs des paramètres d'action; le texte des conditions d'occurrence des événements et l'évaluation des éléments qui le composent sont absents de la trace. Il en est de même pour les clauses d'instantiation d'action.

Pour illustrer, prenons deux exemples.

(i) Supposons que nous désirions analyser les interactions induites entre les objets d'un système Oblog. Ces interactions induites sont caractérisées par des queries d'instance ou de classe qui ne peuvent s'exprimer que dans les conditions et instantiations. La trace ne nous permettra donc pas de déceler ces interactions induites.

(ii) Supposons que nous souhaitions analyser pourquoi un événement spécifié ne se produit jamais à partir d'une situation donnée.

Deux cas de figure sont possibles:

- toutes les évaluations de sa condition d'occurrence ont retourné une valeur fausse et au moins une évaluation a eu lieu;
- la condition d'occurrence n'a jamais été évaluée bien que l'objet ait atteint la situation donnée.

Dans le premier cas, nous pourrions être particulièrement intéressé par l'évaluation des différents composants de la condition afin, par exemple pour découvrir un composant systématiquement responsable de l'avortement.

Dans le deuxième cas, un détail des conditions des autres événements effectivement déclenchés à partir de la situation donnée, pourrait s'avérer très instructif.

Toutefois, dans l'état actuel, la trace ne contient pas ces éléments intéressants.
Remarquons que ce deuxième exemple suggère non seulement d'insérer dans la trace le contexte (conditions et instantiations) des événements effectivement déclenchés mais aussi l'évaluation du contexte des événements inhibés.

4.2.3 Syntaxe d'une trace Oblog

Une trace Oblog est implémentée comme une séquence de lignes d'un fichier texte ascii. Elle est organisée suivant la syntaxe suivante (définition BNF) :

<trace> ::= <eof> |
<événement> <trace>
<événement> ::= <action> <{---}> |
<{ }> <{+++}> <action> <{---}> <séquence_actions>
<séquence_actions> ::= <action> <{+++}> |
<action> <{---}> <séquence_actions>
<action> ::= <entête> |
<entête> <état>
<entête> ::= <objinfo> <eol> |
<objinfo> <bl> <callinfo> <eol>
<objinfo> ::= 'Object' <bl> <c> <bl> '(' <i> ')' <bl>
'Action' <bl> <a> <bl> 'Situation' <s>
<callinfo> ::= 'Caller' <bl> <cc> <bl> '(' <ci> ')'
<état> ::= <parameters> <attributes>
<parameters> ::= < > |
<parameter> <eol> <parameters>
<attributes> ::= < > |
<attribute> <eol> <attributes>
<parameter> ::= 'Event Attribute ' <pn> ' = ' <pv>
<attribute> ::= 'Attribute ' <an> ' = ' <av>
<pv> ::= <val>
<av> ::= <val>

où
- **c, ci, a, s, cc, ci, pn, an** sont de type STRING;
- **eof** est un indicateur de fin de fichier;
- **eol** est une suite éventuellement vide de caractères blancs suivie d'un indicateur de fin de ligne;
- **bl** est une suite non vide de caractères blancs;
- **{xxx}** est une séquence non vide de caractères x suivie de **eol**;
- **val** est complexe: nous n'en donnons pas le détail (il ne contient pas de **eol** sauf s'il est valeur de liste; dans ce dernier cas, les éléments listés apparaissent répétitivement à la ligne comme **Element Number n -- Value x** sur une seule ligne);

4.3 Description détaillée du format cible NADF [ASAX4]

ASAX est un outil universel pour l'analyse de fichiers séquentiels. Pour assurer cette universalité, une traduction du fichier natif est assurée vers un format de fichier universel appelé Normalized Audit Data Format. Cette traduction est réalisée grâce à un programme appelé Format Adaptator. Le format des enregistrements d'un fichier NADF est suffisamment flexible que pour rendre théoriquement possible la traduction de tout fichier séquentiel. Le prix à payer pour cette « universalité » consiste toujours en la rédaction d'un Format Adaptator.

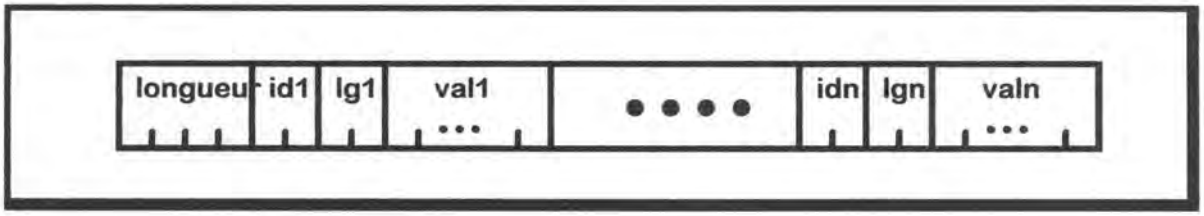
Un fichier NADF est un fichier séquentiel d'enregistrements en format NADF.
Un enregistrement NADF consiste en:

- un entier sur quatre bytes représentant la longueur en bytes de l'enregistrement NADF (y compris le champ longueur);

- un certain nombre de champs contigus de données d'audit. Chacun de ces champs contient les trois éléments contigus qui suivent:
  identifiant: un entier sur deux bytes qui identifie la donnée d'audit. Cet élément doit être aligné sur des frontières de deux bytes;
  longueur: un entier sur deux bytes qui indique la longueur de la valeur de la donnée d'audit;
  valeur: la valeur proprement dite de la donnée d'audit.

D'autre part, les identifants de données d'audit apparaissant dans un enregistrement NADF doivent être **triés en ordre ascendant strict** pour des raisons d'efficacité.

La figure qui suit illustre la forme générale d'un enregistrement NADF



Pour respecter la contrainte d'alignement des identifiants de donnée d'audit, la valeur de la donnée d'audit doit éventuellement être suffixée par un byte de remplissage (un espace ❐) pour atteindre un nombre pair de bytes. Ce byte n'est pas pris en compte dans la longueur de la valeur.
D'autre part, pour des raisons de facilité de calcul, la longueur d'un enregistrement NADF doit être alignée sur une frontière de quatre bytes. Par conséquent, les enregistrements sont éventuellement complétés par des caractères de remplissage (caractères **\0**) de manière à atteindre une longueur multiple de quatre. De nouveau, la longueur du record n'inclut pas ces caractères de remplissage.

Enfin tout fichier NADF commence par un enregistrement de contrôle caractéristique: un entier sur quatre bytes contenant la valeur 15 suivi de la chaîne '__NADF__1|\0' sur 11 bytes et d'un caractère de remplissage \0.

Remarques:

- Les identifiants de données d'audit intervenant dans un enregistrement NADF doivent être **distincts**.

Dans une requête RUSSEL, une donnée d'audit de l'enregistrement NADF courant est référencée par un **nom externe** propre au domaine d'application. Il est donc nécessaire d'associer **un nom externe distinct** à chaque champ de données apparaissant dans l'enregistrement correspondant du fichier séquentiel natif.

- Préalablement à la construction d'un Format Adaptator, un ensemble des identifiants de données d'audit et un ensemble correspondant des noms externes du domaine d'application sont construits en tenant compte des contraintes ci-avant. Un fichier texte auxiliaire appelé 'Audit Data Description File' (ADDF) est construit. Il décrit une bijection entre l'ensemble des identifants de données d'audit et l'ensemble des noms externes.

Au sens large, on entend par Format Adaptator à la fois la génération de l'ADDF et la traduction du format natif en NADF.

4.4 Le Format Adaptator

4.4.1 La génération de l'Audit Data Description File de l'application Monde Bancaire

L'ensemble des noms externes de Monde Bancaire est construit en collectant tous les noms d'attributs et de paramètres des objets de l'application (exprimés en majuscules). De manière à distinguer, au sein d'un événement/action élémentaire, un paramètre de l'action, d'un attribut (éventuellement de même nom) de l'objet impacté, un suffixe caractéristique est ajouté a tous les noms de paramètres. Par exemple, le paramètre *solde* de l'action *créer* de l'objet *Compte* et l'attribut *solde* du même objet deviennent respectivement les noms externes SOLDE__E et SOLDE.

De plus, nous ajoutons à l'ensemble des noms externes ainsi constitué, les sept noms externes suivant:

ACTTYPE
ACTCLAS
ACTADDR
ACTACTI
ACTSITU
ACTCCLA
ACTCADD

Le nom ACTTYPE permettra de référencer dans un enregistrement NADF le type de l'événement élémentaire correspondant dans la trace Oblog. Il pourra prendre les valeurs suivantes en fonction du contexte de détection de l'événement élémentaire dans la trace Oblog:

'Isolated'	Evénement isolé
'First'	Evénement participant rencontré le premier dans un événement partagé (c'est à dire dernière action appelée de cet événement partagé)
'Last'	Evénement participant rencontré le dernier dans un événement partagé (c'est à dire action racine de l'arborescence de calls associée)
'Next'	Evénement participant intermédiaire

Les six autres noms externes correspondent respectivement au champs *c,i,a,s,cc,ci* dont nous avons donné le sens dans la définition des informations de la trace Oblog décrivant un événement élémentaire.

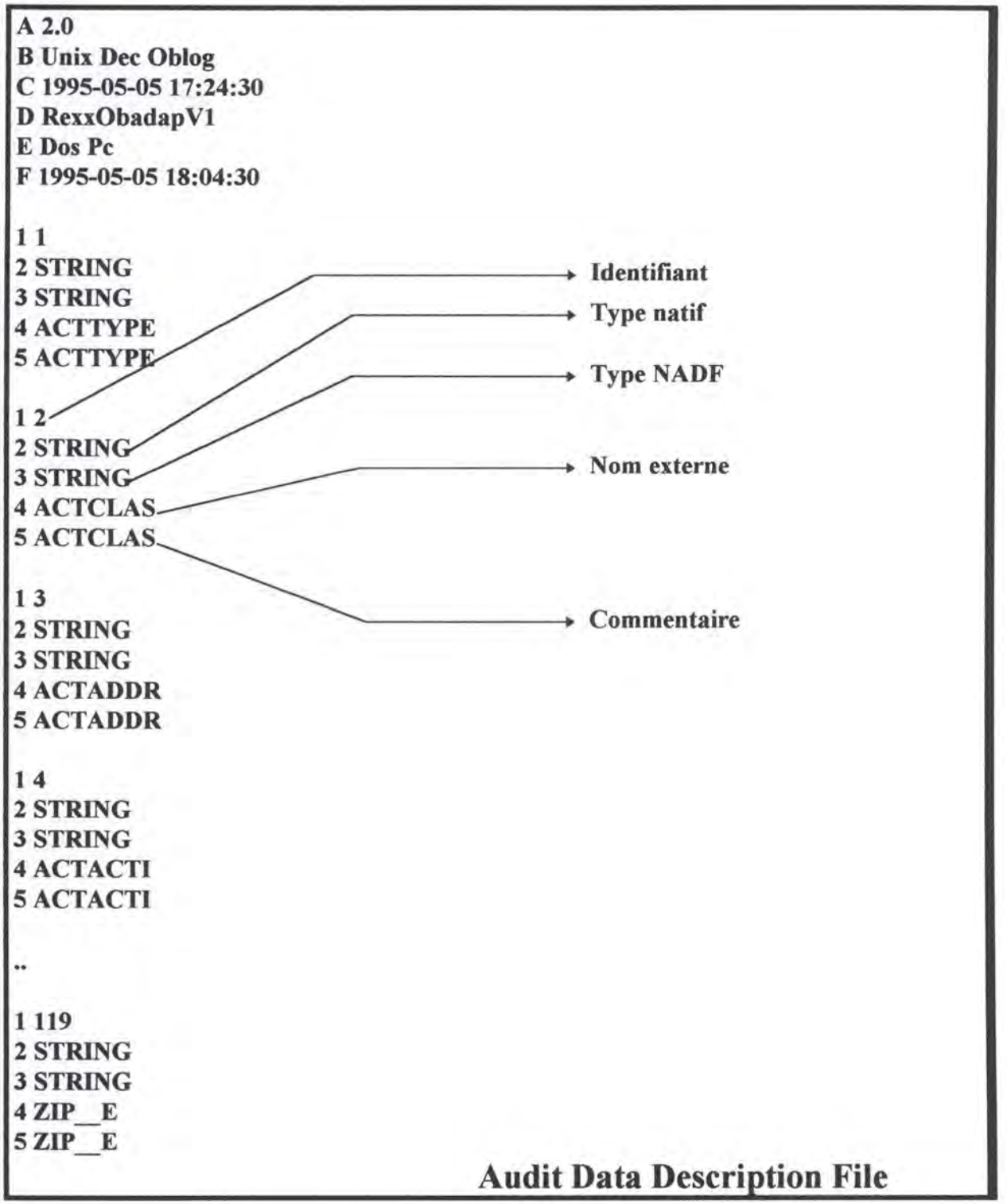
L'ensemble des noms externes ainsi constitué est sauvé dans un fichier texte dont le contenu partiel apparaît à la figure suivante:

ACTTYPE
ACTCLAS
ACTADDR
ACTACTI
ACTSITU
ACTCCLA
ACTCADD

ACCNUM
ACCNUM__E
ACCOUNT__E
ACC_NO__E
ADDACCDREF__E
ADDCLIDREF__E
AMOUNT
AMOUNT__E
ATMAPP
BALANCE
BALANCE__E
BANKAPP
BOT
CARD__E
CARDISSUED
CARDISSUED__E
CARDNUM
CARDNUM__E
CARDPINCODE
CARDPINCODE__E
CEN
CLIENT__E
CLIENT_MSG
CLINUM
CLINUM__E
COUNT
COUNTERREF
COUNTERREF__E
CREATED_MSG
...
...
TOWN
TOWN__E
UNKNOWN_MSG
ZIP
ZIP__E

Fichier des noms externes

Le fichier Audit Data Description File est ensuite construit en associant à chaque nom externe un identifiant de donnée d'audit qui correspond à la position relative du nom externe en question dans le fichier des noms externes. Un aperçu du contenu du fichier Audit Data Description File de Monde Bancaire apparaît dans la figure ci-après.



La figure illustre le fait que l'identifiant 2 est associé au nom externe ACTCLAS.
Les champs 2, 3 et 5 sont actuellement purement documentaires.

4.4.2 La traduction du format natif en format NADF

Dans ce paragraphe, nous effectuerons une construction progressive du programme de traduction guidée par la structuration des données d'une trace Oblog, en précisant les choix opérés.

4.4.2.1 Squelette général du corps d'un programme Format Adaptator:

Version 0

```
Begin
    ..
    ..
    While not end of native file do
        begin
            read native file record;
            convert native record to NADF format in output buffer;
            write output buffer to NADF file;
        end;
    ..
    ..
End.
```

4.4.2.2 Choix de granularité de l'opération de traduction

Nous choisissons de traduire chaque action élémentaire (isolée ou participante) rencontrée dans la trace Oblog par un enregistrement NADF. Un enregistrement de fichier natif sera assimilé dans ce cadre à une séquence de lignes caractérisant une action/événement élémentaire.

Par exemple, l'action **isolée**:

Object CREATE_ACC (6741904) Action createACC Situation accCreated
Event Attribute Owner = 6 (INT)
Event Attribute account = 8 (INT)
Attribute ownerId = STRING: '1000' (len=4)
Attribute myMaster = OBJECT NAME: ADD_ACC_D OBJECT ADDRESS:6735968
Attribute status = CREA_ACC_S: CREATED
Attribute CardNum = STRING: '20' (len=2)
Attribute AccNum = STRING: '20' (len=2)
Attribute Balance = 0 (INT)
Attribute CreditLimit = 100000 (INT)
Attribute noOfAttempts = 0 (NAT)
Attribute cardPinCode = STRING: '1020' (len=4)
Attribute CardIssued = 1 (BOOL)

sera traduite en l'enregistrement NADF:

[212](1,1)I□(2,10)CREATE_ACC(3,7)6741904□(4,9)createACC□(5,10)accCreated(6,0)(7,0)(10,2)20(12,1)8□(19,1)0□(24,1)1□(26,2)20(28,4)1020(39,6)100000(76,56)OBJECT NAME: ADD_ACC_D OBJECT ADDRESS:6735968(85,1)0□(88,1)6□(89,4)1000(105,7)CREATED□

[longueur totale]
(identifiant,longueur donnée d'audit)

Cette traduction sera effectuée par une procédure **StoreAction(type)** du Format Adaptator. Cette procédure reçoit en argument le **type** d'action élémentaire (**I**solated, **F**irst,**N**ext,**L**ast) et travaille sur une strucure de données globale **Q** contenant les lignes de l'action à traduire.
Conventionnellement, si **act** est une action de type **t**, on appellera **Nadf(act,t)** l'enregistrement NADF construit par **StoreAction** à partir de **act**.

Remarques:

- le premier Audit Data de l'enregistrement NADF ci-avant ((1,1)I□) indique que l'action est isolée (ACTTYPE = 'I')

- le détail de la procédure StoreAction sera uniquement présenté dans la version finale du programme de traduction. Son fonctionnement sera ensuite illustré de manière intuitive.

4.4.2.3 Construction progressive du programme Format Adaptator

Ce programme **P** a une structure basée sur l'organisation syntaxique de la trace Oblog. Il a pour démarche d'isoler l'une après l'autre les actions élémentaires de la trace Oblog stockée dans un fichier **T** en déterminant contextuellement leur **type**.

Chaque fois qu'une action a pu être isolée, les lignes de cette action sont rangées en séquence dans une structure de données globale de type File [Meyer1] que nous appelerons la queue **Q**.

Une fois l'action stockée dans la queue **Q** et son **type** déterminé, une procédure **StoreAction** effectue le parsing des lignes de l'action pour retrouver les couples (nom externe, valeur associée). Elle construit ensuite les triplets correspondants (identifiant,longueur valeur associée, valeur associée), les trient sur base de l'identifiant. Elle produit enfin le record NADF correspondant à l'action (longueur totale, $triplet_1$, ..., $triplet_n$) et ajoute ce dernier en appendice d'un fichier NADF **N**.

Structures de données globales du programme P

T un fichier trace Oblog ouvert

> Remarquons qu'il est possible de **numéroter** les n actions élémentaires de cette trace en fonction de leur ordre d'apparition dans T. Il est également possible de déterminer le **type** de chacune des actions de la trace en prenant pour référence l'organisation syntaxique de trace Oblog.
>
> On définit $\mathbf{Map(T) = ((a_i, Type(a_i)))_{i = 1..n}}$
>
> où a_i est la ième action élémentaire de T
> n entier >=0 est le nombre d'actions élémentaires de T

line buffer de lecture des lignes de T

Q une file

N un fichier NADF ouvert

Fonctions/instructions utilitaires

Lines (fonction) [output = function_name(input)]

Syntaxe: n = **Lines**(F) n : integer ; F fichier texte de pointeur *F

Spécification:

Pré: F = F0
*F = *F0 pointe vers une ligne de F (string suivi de **eol**) ou vers **eof**
n indéterminé

Post: F = F0
*F = *F0
n=0 si *F0 pointe **eof** sinon n=1

Linein (fonction)

Syntaxe: line = **Linein**(F) line : string ; F fichier texte

Spécification:

Pré: **Lines**(F) > 0
F = F0
*F = *F0 pointe vers une ligne x de F
line indéterminé

Post: F = F0
*F pointe vers la ligne de F suivant x ou vers **eof** si x est la dernière ligne de F
line contient la valeur de la ligne x

Queue (instruction)

Syntaxe: **Queue** line line : type ; Q $File_{type}$ (implicite)

Pré: Q = Q0
line = lo

Post: Q = Enfile(Q0,lo) [Meyer1]
line = lo

Corps du programme P (Version 1)

P:
...
Traite_trace;
...
exit

Traite_trace:
    if Lines(T) > 0 then do
        Traite_événement;
        Traite_trace
    enddo;
    return

Traite_événement:
    line = Linein(T);
    if line = {□□□} then do
        line = Linein(T);
        line = Linein(T);
        Traite_action('First');
        Traite_séquence_actions
    enddo
    else Traite_Action('Isolated');
    return

Traite_séquence_actions:
    line = Linein(T);
    Traite_action('Next');
    if line not = {+++} then **Traite_séquence_actions**;
    return

Traite_action(type):
    GetAction;
    if type = 'Next' then if line = {+++} then type = 'Last;
    StoreAction(type);
    return

GetAction:
    queue line;
    line = Linein();
    if line not in { {+++} , {---} } then **GetAction**;
    return

StoreAction(type): pas de détail ! (voir programme définitif)

Corps du programme P: version itérative (Version 2)

P:
...
Traite_trace;
...
exit

Traite_trace:
```
    do while Lines(T) > 0
        Traite_événement
      enddo;
    return
```

Traite_événement:
```
    line = Linein(T);
    if line = {□□□} then do
        line = Linein(T);
        line = Linein(T);
        Traite_action('First');
        Traite_séquence_actions
      enddo
    else Traite_Action('Isolated');
    return
```

Traite_séquence_actions:
```
    do until line = {+++}
        line = Linein(T);
        Traite_action('Next')
      enddo;
    return
```

Traite_action(type):
```
    GetAction;
    if type = 'Next' then if line = {+++} then type = 'Last;
    StoreAction(type);
    return
```

GetAction:
```
    do until line in { {+++} , {---} }
        queue line;
        line = Linein()
      enddo;
    return
```
StoreAction(type): pas de détail ! (voir programme définitif)

Spécifications

Dans les spécifications qui suivent, T reste toujours inchangé si ce n'est son pointeur associé *T

- **Traite_trace**

  Pré: *T pointe sur la première ligne de T (éventuellement EOF(T))
  line est indéterminée
  Q est vide
  *N pointe sur EOF(N)
  N est vide

  Post: *T pointe sur EOF(T)
  line contient la valeur de la ligne précédant *T si elle existe
  Q est vide
  *N pointe sur EOF(N)
  $N = ((Nadf(Map(T)_i))_{i=1..n}$

- **Traite_événement**

  Pré: *T pointe sur la première ligne d'un événement E de T
  line contient la valeur de la ligne précédant *T si elle existe
  Q est vide
  *N pointe sur EOF(N)
  N = N0

  Post: *T pointe sur la ligne suivant la dernière ligne de E (éventuellement EOF(T))
  line contient la valeur de la ligne précédant *T
  Q est vide
  *N pointe sur EOF(N)
  $N = Append(N0 , ((Nadf(Map(T)_i))_{i=p..q})$
  où p..q sont les numéros des actions de E

- **Traite_séquence_actions**

  Pré: *T pointe sur la première ligne d'une action a_k d'un événement E de T dont elle n'est pas la première action
  line contient la valeur de la ligne précédant *T
  Q est vide
  *N pointe sur EOF(N)
  N = N0

Post: *T pointe sur la ligne suivant la dernière ligne de E (éventuellement EOF(T))
line contient la valeur de la ligne précédant *T
Q est vide
*N pointe sur EOF(N)
N = Append(N0 , ((Nadf(Map(T)$_i$)) $_{i=k..q}$)
où q est le numéro de la dernière action de E

- **Traite_action**

Pré: *T pointe sur la ligne suivant la première ligne d'une action a_k de T
line contient la valeur de la ligne précédant *T (première ligne de a_k)
Q est vide
*N pointe sur EOF(N)
N = N0

Post: *T pointe sur la deuxième ligne suivant la dernière ligne de a_k
(éventuellement EOF(T))
line contient la valeur de la ligne précédant *T
Q est vide
*N pointe sur EOF(N)
N = Append(N0 , ((Nadf(Map(T)$_i$)) $_{i=k}$)

- **GetAction**

Pré: *T pointe sur la ligne suivant la première ligne d'une action a_k de T
line contient la valeur de la ligne précédant *T (première ligne de a_k)
Q est vide

Post: *T pointe sur la deuxième ligne suivant la dernière ligne de a_k
(éventuellement EOF(T))
line contient la valeur de la ligne précédant *T
Q contient les lignes de a_k dans leur ordre d'apparition dans T

- **StoreAction(type)** type = 'I' ou 'F' ou 'N' ou 'L'

Pré: Q = Q0 non vide type = t0
*N pointe sur EOF(N)
N = N0

Post: Q est vide
*N pointe sur EOF(N)
N = Append(N0 , ((Nadf(Q0,t0)))

Programme Format Adaptator définitif (Version 3)

```
Init:
X = 'nomsexte.txt'
T = 'obltrace.txt'
N = 'oblnadf.sun'
rc = Stream(N,'Command','OPEN CREATE')
nadf_first_record = D2C(15,4) || '__NADF__1|' || D2C(0,2)
rc = Stream(N,'Command','WRITE', nadf_first_record )
ext_number = 0
do while Lines(X)
   ext_name = Strip(Linein(X))
   ext_number = ext_number + 1
   if ext_name ^= '' then identifier.ext_name = ext_number
end

Traite_trace:
do while Lines(T)
    line = Linein(T)
    if line = ' ' then
      do
        line = Linein(T)
        line = Linein(T)
        call Traite_action 'F'
        do until Left(line,5) = '+++++'
            line = Linein(T)
            call Traite_action 'N'
        end
      end
    else call Traite_action 'I'
end

Clot:
x = Stream(N,'Command','CLOSE')
exit

Traite_action: procedure expose T N line identifier.
parse arg type
call GetAction
if type = 'N' then if Left(line,5)='+++++' then type = 'L'
call StoreAction type
return

GetAction: procedure expose T line
do until Left(line,5) = '-----' | Left(line,5) = '+++++'
   queue line
   line = Linein(T)
end
return
```

```
StoreAction: procedure expose N identifier.
parse arg type                                          /* Phase 1 */
parse pull . c '(' i ')' . a . s . cc '(' ci ')'
m.1 = 'ACTTYPE' type
m.2 = 'ACTCLAS' Strip(c)
m.3 = 'ACTADDR' i
m.4 = 'ACTACTI' a
m.5 = 'ACTSITU' s
m.6 = 'ACTCCLA' Strip(cc)
m.7 = 'ACTCADD' ci
j = 7
do queued()
   parse pull mot1 mot2 mot3 mot4 reste
   select
     when mot1 = 'Attribute' then do
          j=j+1
          m.j= Translate(mot2) Clean(mot4 reste)
        end
     when mot1 = 'Event' & mot2 = 'Attribute' then do
          j=j+1
          m.j= Translate(mot3) || '__E' Clean(reste)
        end
     when mot1 = 'Empty' | mot1 = 'Element' then do
          m.j= m.j mot1 mot2 mot3 mot4 reste
        end
     otherwise
   end
end

liste_identifiants = ''                                 /* Phase 2 */
longueur_totale = 4
do k=1 to j
   nom_externe = Word(m.k,1)
   identifiant = identifier.nom_externe
   liste_identifiants = liste_identifiants identifiant
   valeur = Subword(m.k,2)
   longueur = Length(valeur)
   long2 = longueur + longueur // 2
   longueur_totale = longueur_totale + 2 + 2 + long2
   audit_data.identifiant =  D2C(identifiant,2) || D2C(longueur,2) || Left(valeur,long2)
end
liste_identifiants = Subword(liste_identifiants,1,7) Sort(Subword(liste_identifiants,8))
nadf_record=''
do k=1 to j
   identifiant = Word(liste_identifiants,k)
   nadf_record = nadf_record || audit_data.identifiant
end
long_totale4 = longueur_totale + longueur_totale // 4
nadf_record = D2C(longueur_totale,4) || nadf_record
if long_totale4 > longueur_totale then nadf_record = nadf_record || D2C(0,2)
x = Stream(N,'Command','WRITE',nadf_record)
return
```

```
Clean: procedure
parse arg v1 v2
select
  when v1 = 'STRING:' then do
     val = Strip(Delword(v2,Words(v2),1))
     if val = 'Empty' then val = """"
     val = Strip(val,,"""")
    end
  when Right(v1,1) = ':' then val = v2
  when v2='(NAT)' | v2='(INT)' | v2='(BOOL)' | v2='(CHAR)' then val = v1
  otherwise val = v1 v2
end
return val

Sort: procedure
parse arg list
slist = ''
n = Words(list)
do i = 1 to n
   max = Word(list,1)
   imax = 1
   do j = 1 to Words(list)
     if max < Word(list,j) then do
        max = Word(list,j)
        imax = j
       end
   end
   slist = max slist
   list =Delword(list,imax,1)
end
return slist
```

Commentaires concernant le Programme Format Adaptator définitif

(i) Le programme Format Adaptator définitif a été écrit dans le langage REXX. Ce langage est particulièrement puissant notamment pour le traitement des chaînes de caractères dont est essentiellement constituée la trace Oblog. Concis, expressif et souple, il permet une lecture aisée, débarrassée de l'encombrement technique des traditionnels langages de troisième génération. Enfin, il est implémenté sur de très nombreuses plates-formes et est en voie de normalisation ANSI.

Le lecteur non familier de ce langage trouvera en annexe une présentation très intuitive du langage destinée à faciliter la lecture des détails du programme.
Une présentation plus formelle et plus détaillée du langage peut être trouvée dans [Cowlishaw1] et [IBM1].

(ii) **Initialisation** (label Init:)

Le fichier NADF N est créé et ouvert en écriture. Le premier enregistrement de contrôle est écrit dans N.
Le fichier texte X des noms externes de l'application et le fichier texte T contenant la trace Oblog seront ouverts implicitement.

Un « tableau » de variables composées, de préfixe **identifier.** , est construit en lisant le fichier des noms externes. Il associe à chaque nom externe lu sur une ligne de X, un identifiant numérique correspondant à la position relative de la ligne lue dans X.
Par exemple, si la septième ligne de X contient le nom externe ACTCADD, l'assignation **identifier.ACTADD = 7** sera effectuée. Ce tableau sera utilisé par la procédure StoreAction.

(iii) **Corps du programme** (label Traite_trace:)

Le corps du programme est issu de la consolidation au sein du programme principal du code des procédures **Traite_trace**, **Traite_événement** et **Traite_séquence_actions** de la version deux du programme.

(iv) **Clôture** (label Clot:)

Le fichier N est fermé explicitement. La sortie (exit) du programme ferme également implicitement X et T.

(v) **Procédures Traite_Action et GetAction**

Ces procédures ne diffèrent pas de celles de la version 2.

(vi) **Fonction Sort**

Cette fonction reçoit en argument un string s dont les mots sont des nombres et retourne un string composé des mots de s rangés en ordre ascendant

Ex: Sort('87 10 12') = '10 12 87'

(vii) **Fonction Clean**

Cette fonction reçoit en argument un string et retourne un string issu d'un « nettoyage » plus ou moins arbitraire de ce string que nous allons illustrer sur 2 exemples. Elle est employée pour « nettoyer » le suffixe, situé derrière le caractère '=', de chaque ligne, de type 'Attribute ...' ou 'Event Attribute ...' dans la trace.

Prenons 2 exemples d'actions de la trace:

Exemple 1

Object SELECT_ACC (6754512) Action endOfList Situation giveClient
Event Attribute cursor = ***10*** *(INT)*
Attribute cursor = ***10*** *(INT)*
Attribute result = ***17***
Element number 1 -- value 2
Element number 2 -- value 4
Attribute myMaster = ***OBJECT NAME: DELQUE_ACC_D*** ***OBJECT ADDRESS:6736688***
Attribute CardIssued = ***0*** *(BOOL)*
Attribute CreditLimit = ***0*** *(INT)*
Attribute Balance = ***0*** *(INT)*
Attribute status = *SEL_ACC_S:* ***ENDOFLIST***
Attribute AccNum = *STRING: Empty String.*
Attribute index = ***4*** *(NAT)*
Attribute Owner = ***0*** *(INT)*
Attribute ownerId = *STRING: Empty String.*

Exemple 2

Object CREATE_ACC (6741904) Action createACC Situation accCreated
Event Attribute Owner = ***6*** *(INT)*
Event Attribute account = ***8*** *(INT)*
Attribute ownerId = *STRING:* ***'1000'*** *(len=4)*
Attribute myMaster = ***OBJECT NAME: ADD_ACC_D*** ***OBJECT ADDRESS:6735968***
Attribute status = *CREA_ACC_S:* ***CREATED***
Attribute CardNum = *STRING:* ***'20'*** *(len=2)*
Attribute AccNum = *STRING:* ***'20'*** *(len=2)*
Attribute Balance = ***0*** *(INT)*
Attribute CreditLimit = ***100000*** *(INT)*
Attribute noOfAttempts = ***0*** *(NAT)*
Attribute cardPinCode = *STRING:* ***'1020'*** *(len=4)*
Attribute CardIssued = ***1*** *(BOOL)*

Dans ces deux exemples, les arguments successifs de *Clean* sont représentés en italique et les résultats en gras (string vide si absence de gras). Ce « nettoyage », bien que discutable, nous permettra d'écrire des règles Russel légèrement plus lisibles car seules seront sauvées les valeurs nettoyées dans le fichier NADF. D'autre part, ce nettoyage implique, dans certains cas, une perte d'information sur le type de la donnée traduite. Lorsque le produit ASAX permettra d'exploiter les types des données natives via le fichier ADDF, ce problème sera toutefois résolu.

(viii) **Procédure StoreAction**

Cette procédure a pour argument le type de l'action courante et pour argument implicite les lignes de cette action rangées dans la File système Q.

Par exemple, type = 'I' (pour 'Isolated' ==> ACTTYPE = 'I')
Q contient les lignes de l'action de l'exemple 1.

Cette procédure est composée de deux phases.

- Phase 1

La première ligne est extraite et parsée pour isoler les valeurs *c*, *i*, *a*, *s*, [*cc* et *ci*] correspondant respectivement aux noms externes ACTCLAS, ACTADDR, ACTACTI, ACTSITU, ACTCCLA et ACTCADD d'identifiants respectifs 2, 3, 4, 5, 6 et 7.
Ensuite les autres lignes de l'action sont successivement extraites et parsées pour isoler, pour chaque paramètre de l'action et pour chaque attribut de l'instance d'objet, le nom externe correspondant et sa valeur associée.

Un « tableau » de variables composées, de préfixe **m.** , est construit parallèlement à ces opérations pour stocker les couples (nom externe, valeur) recensé dans l'action ainsi que le couple (ACTTYPE, type).

Pour l'action isolée de l'exemple 1, ce tableau est le suivant:

M.1 ACTTYPE I
M.2 ACTCLAS SELECT_ACC
M.3 ACTADDR 6754512
M.4 ACTACTI endOfList
M.5 ACTSITU giveClient
M.6 ACTCCLA
M.7 ACTCADD
M.8 CURSOR__E 10
M.9 CURSOR 10
M.10 RESULT 17 Element number 1 -- value 2 Element number 2 -- value 4
M.11 MYMASTER OBJECT NAME: DELQUE_ACC_D OBJECT ADDRESS:6736688
M.12 CARDISSUED 0
M.13 CREDITLIMIT 0
M.14 BALANCE 0
M.15 STATUS ENDOFLIST
M.16 ACCNUM
M.17 INDEX 4
M.18 OWNER 0
M.19 OWNERID

Remarques:

- Certaines valeurs sont « nettoyées »;
- Certaines valeurs sont des strings vides;
- Les valeurs des attributs « pointeurs » gardent la même forme que dans la trace (valeur composée + mots clés);

▪ Les lignes 'Element number ...' (ou bien la ligne 'Empty List !') qui suivent une ligne contenant le nom d'un attribut ou paramètre de type **Liste** sont concaténées dans un string représentant la « valeur » associée à ce nom d'attribut ou de paramètre (valeur composée + mots clés);
▪ Le nom de chaque attribut recensé est traduit arbitrairement en majuscules pour former le nom externe correspondant (voir 4.4.1 : conventions ADDF);
▪ Le nom de chaque paramètre recensé est traduit arbitrairement en majuscules et suffixé par le string '__E' pour former le nom externe correspondant (voir 4.4.1 : conventions ADDF).

- Phase 2

Le tableau **m.** est parcouru de manière à construire un autre tableau **audit_data.** contenant les données d'audit qui vont composer l'enregistrement NADF.

Chaque élément du tableau m. fournit un *nom_externe* et une *valeur*. L'*idendifiant* associé à *nom_externe* est déduit du tableau *identifier*.
(*identifiant = identifier.nom_externe*). La *longueur* réelle de la valeur est calculée ainsi que la longueur ajustée *long2*. La donnée d'audit correspondante est sauvée dans un élément du tableau *audit_data.* [*audit_data.identifiant = (identifant,longueur)valeur* ajustée].

Le parcours de m. permet également de calculer la longueur totale réelle et la longueur totale ajustée du futur enregistrement NADF et de construire une liste des identifiants de données recensés dans l'action traitée.

Pour l'exemple 1, ceci produit les données suivantes:

audit_data.1 (1,1)I❒
audit_data.2 (2,10)SELECT_ACC
audit_data.3 (3,7)6754512❒
audit_data.4 (4,9)endOfList❒
audit_data.5 (5,10)giveClient
audit_data.6 (6,0)
audit_data.7 (7,0)
audit_data.42 (42,2)10
audit_data.41 (41,2)10
audit_data.100 (100,59)17 Element number 1 -- value 2 Element number 2 -- value 4❒
audit_data.76 (76,56)OBJECT NAME: DELQUE_ACC_D OBJECT ADDRESS:6736688
audit_data.24 (24,1)0❒
audit_data.39 (39,1)0❒
audit_data.19 (19,1)0❒
audit_data.105 (105,9)ENDOFLIST❒
audit_data.10 (10,0)
audit_data.56 (56,1)4❒
audit_data.87 (87,1)0❒
audit_data.89 (89,0)

liste_identifiants = 1 2 3 4 5 6 7 42 41 100 76 24 39 19 105 10 56 87 89

longueur_totale = 260

long_totale4 = 260

La liste des identifiants est ensuite triée en ordre ascendant.

liste_identifiants = 1 2 3 4 5 6 7 10 19 24 39 41 42 56 76 87 89 100 105

Une fois triée, cette liste est parcourue de gauche à droite. Chaque identifiant rencontré permet de retrouver la donnée d'audit correspondante (*audit_data.identifiant*) qui est suffixée au string *nadf_record*.
Ce string, initialement vide, contient, après ce parcours, la concaténation des données d'audit en format NADF. Il est ensuite préfixé de la *longueur_totale* et éventuellement paddé à droite (alignement)

nadf_record =

[260](1,1)I❒(2,10)SELECT_ACC(3,7)6754512❒(4,9)endOfList❒(5,10)giveClient(6,0)(7,0)(10,0)(19,1)0❒(24,1)0❒(39,1)0❒(41,2)10(42,2)10(56,1)4❒(76,56)OBJECT NAME: DELQUE_ACC_D OBJECT ADDRESS:6736688(87,1)0❒(89,0)(100,59)17 Element number 1 -- value 2 Element number 2 -- value 4❒(105,9)ENDOFLIST❒

Enfin, ce string est écrit en appendice du fichier N.

4.4.3 Commentaires concernant le Format Adaptator

La construction d'un Format Adaptator de trace Oblog présente certaines singularités. En effet, alors que les Format Adaptators d'audit trails traditionnels doivent prendre en compte des formats d'enregistrements natifs particulièrement diversifiés et des noms externes en nombre relativement limité, un Format Adaptator de trace Oblog ne traite qu'un seul format d'enregistrement natif (l'événement élémentaire) mais un nombre très élevé de noms externes.

La construction du fichier ADDF d'une application Oblog ne pose aucun problème et est automatisable sur base d'une consultation du Repository. Toutefois, le fichier ADDF associé à une application Oblog doit être reconstruit lors de chaque modifications significatives des attributs de classe ou des paramètres d'action.

L'élaboration du programme de traduction est simple et découle tout naturellement de l'organisation des données d'une trace Oblog. Ce programme est réutilisable pour la traduction de trace de toute application Oblog.
Une légère perte d'information résulte d'un choix arbitraire et non figé de non traduction des indications des types élémentaires. Ces indications de types pourraient ultérieurement être traduites dans le fichier ADDF, lorsque l'outil ASAX prendra en compte les types natifs.
Remarquons toutefois que certains aspects propriétaires du format NADF (conventions d'alignement, format d'entiers pour les longueurs) peuvent entraîner certaines difficultés de portabilté du fichier NADF mais aussi obscurcir légèrement le code du programme (StoreAction Phase2).

Partie 5 : Analyse d'une Trace Oblog via ASAX

5.1 Intérêt d'une analyse de trace Oblog et pistes d'analyse

L'analyse d'une trace Oblog, et plus généralement l'analyse de la trace d'un système d'objets actifs communiquant par événements partagés, peut s'avérer intéressante pour :

- la mise au point et le contrôle de la Base de Données;
- le suivi de l'utilisation réelle de l'application;
- le support au développement, aux tests et à la maintenance de l'application;
- les mesures quantitatives du comportement d'un système.

5.1.1 La mise au point et le contrôle de la Base de données.

La détection de tous les événements touchant les objets TBL du système tracé permet de recenser toutes les opérations de mises-à-jour de la BD.
Ceci peut s'avérer intéressant pour l'optimisation et/ou pour l'évolution du schéma de la BD. Toutefois, cet intérêt est à notre sens limité car les opérations de consultation, elles aussi importantes dans cette problématique, n'apparaissent pas dans la trace puisque localisées essentiellement dans les conditions et instantiations de la spécification des objets.
Par contre, ce recensement peut s'avérer particulièrement utile lors d'incidents affectant la BD ou lors de contrôle à posteriori de la séquence et des paramètres des opérations de mises-à-jour de la BD.

5.1.2 Le suivi de l'utilisation réelle de l'application.

L'analyse d'une trace Oblog peut être utilisée pour détecter des anomalies, des infractions ou des présomptions d'infractions à la sécurité de l'application. Par exemple l'analyse d'une trace du système Monde Bancaire peut comporter des règles de suivi des comptes bancaires ayant fait l'objet de plusieurs tentatives de retrait infructueuses ou encore des comptes ayant subi de multiples retraits importants.

L'ergonomie des interfaces d'une application peut également être étudiée:
sélection des événements touchant les objets DBX, comportement de l'utilisateur en fonction de la taille de listes de sélection, détection d'erreurs de navigation, ...

L'efficacité de l'application peut également être mesurée: critères de sélection mal choisis produisant des listes de taille importantes que l'utilisateur ne parcourt que sommairement.

La mesure des temps de réponse de l'application suite à une demande utilisateur serait également intéressante (ajout d'un client, production d'une liste de comptes,..). Elle n'est toutefois pas envisageable actuellement car les événements d'une trace Oblog ne contiennent pour l'instant aucune référence temporelle.

Le suivi de l'utilisation effective de l'application peut mettre en évidence la nécessité de mettre à jour l'application (optimisation, évolution des fonctionnalités, correction de problèmes de design,...)

5.1.3 Le support au développement, aux tests et à la maintenance de l'application.

L'analyse d'une trace Oblog peut être particulièrement intéressante dans les phases de développement, de tests et d'adaptation d'une application ou d'une partie d'application.

Le suivi d'un service macroscopique complexe peut par exemple être assuré via une détection des scénarii qui le caractérisent (ajout d'un compte, retrait d'une somme d'argent via l'automate).

Le comportement d'un objet ou bien d'un sous système (unité) peut être isolé ainsi que les relations qu'il entretient avec des objets périphériques.

Le parallélisme des objets du système peut être vérifié par recouvrement de scénarii:
par exemple, retrait d'une somme sur un compte pendant l'ajout d'un nouveau client.

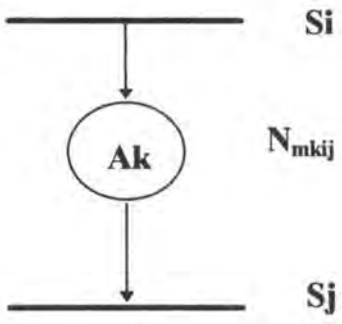
La détection d'événements ou de scénarii imprévus après filtrage peut attirer l'attention sur des erreurs ou des lacunes de spécification.

Il pourrait s'avérer également particulièrement intéressant de mettre en évidence à tout moment des conditions favorables (ou défavorables) à l'occurrence d'un événement et leurs conséquences. Ceci n'est pas possible toutefois car la trace Oblog ne contient pas d'informations sur l'évaluation des conditions.

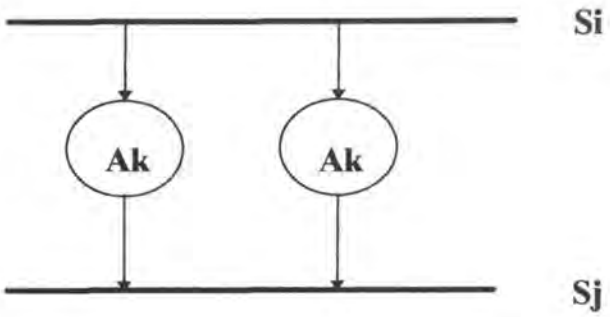
5.1.4 Les mesures quantitatives du comportement d'un système.

Illustrons divers types de mesures quantitatives du comportement d'un système réalisables à partir de l'analyse de sa trace et précisons l'intérêt qu'elles peuvent présenter.

▪ Mesure de la fréquence d'apparitions d'un événement élémentaire pour les objets de classe $CLASS_m$.



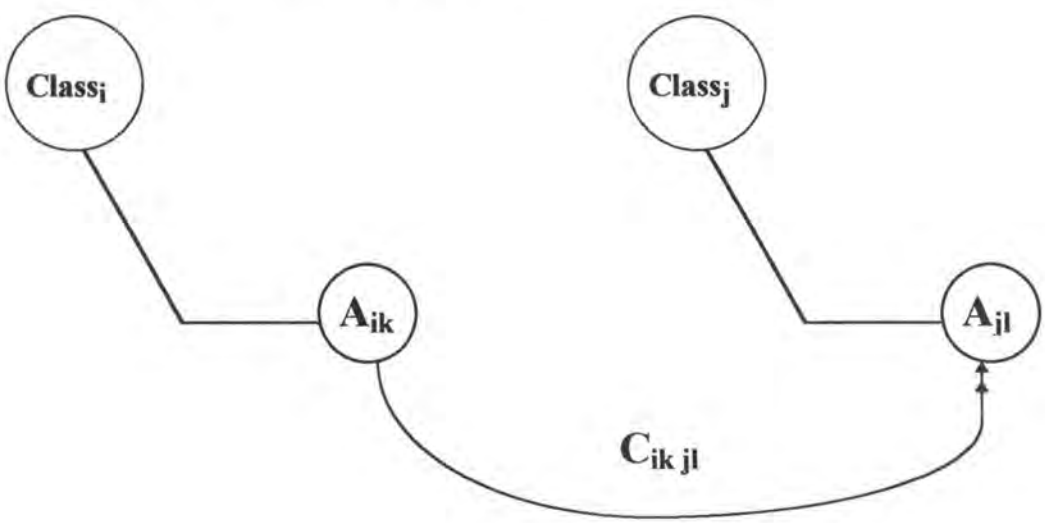
N_{mkij} représente le nombre d'événements élémentaires ayant fait transiter des objets de classe CLASS $_m$ de la situation S_i à la situation S_j via une action A_k.
Remarquons que cette mesure peut englober des événements distincts sans que le détail puisse être produit en raison d'un manque d'information dans la trace. La figure suivante illustre un tel cas de figure.



A partir de ces mesures, il est possible d'effectuer pour chaque classe d'objets une pondération des événements de son GTE (diagramme de comportement local). Cette pondération du GTE peut permettre de suggérer des adaptations de ce GTE pour optimisation. Elle peut également permettre de suspecter des erreurs ou lacunes par exemple en indiquant des événements qui n'apparaissent jamais.

Ce type de mesure peut très bien être réalisée non plus au niveau des classes mais au niveau des instances d'objets pour produire des vues plus détaillées.
Elle peut aussi être agrégée (N_{mk}) pour mesurer la charge des actions de chaque classe d'objets (quels que soit les événements élémentaires qui les ont déclenchées).

▪ Mesure de la fréquence d'apparition d'un call d'une action d'un objet de classe CLASS $_j$ par une action d'un objet de classe CLASS $_i$.



$C_{ik\,jl}$ représente le nombre de calls détectés d'une action A_{ik} d'objets de classe $CLASS_i$ vers une action A_{jl} d'objets de classe $CLASS_j$.
Remarquons que cette mesure peut ici aussi englober des calls distincts sans que le détail puisse être produit en raison d'un manque d'information dans la trace.

A partir de ces mesures, il est possible d'établir des diagrammes d'interactions directes de tout ou partie des classes d'objets du système où chaque call spécifié est pondéré par sa fréquence d'apparition dans la trace.

Il peut s'avérer également très utile de produire des diagrammes d'interactions directes agrégés (une flèche représente ici la présence d'au moins un call spécifié d'une classe vers une autre) dont les flèches sont pondérées par le nombre total de calls détectés d'une classe vers une autre (cfr. les diagrammes d'interactions directes de l'application Monde Bancaire).

Ces mesures sont particulièrement intéressantes pour illustrer la densité des interactions entre types d'objets. Illustrons deux types parmi d'autres, d'exploitation intéressante de ces mesures.

Une première exploitation consiste à appliquer des techniques de clustering à l'ensemble des classes d'objets du système. Ce clustering serait basé sur la définition d'une distance entre couple de classes. Une telle distance serait d'autant plus courte que les partenaires du couple interagissent fortement (mesure du nombre de calls entre partenaires).
Les clusters ainsi dégagés (grappes d'objets interagissant fortement entre eux mais faiblement avec les objets d'autres grappes) peuvent être comparés aux unités déjà spécifiées du système pour les évaluer.
La découpe en clusters peut également servir de base à une étude des possibilités de réalisation d'un système distribué.

Une deuxième exploitation consiste à mesurer la cohésion des unités du système, par exemple par la technique qui suit.

Considérons tout d'abord deux classes d'objets X et Y. Appelons $(CX_i)_{i=1..n}$ et $(CY_j)_{j=1..m}$ les calls spécifiés émis respectivement de ou vers la classe X et de ou vers la classe Y. Les p premiers éléments de ces deux suites sont, par convention, les calls émis de X vers Y ou de Y vers X et sont donc les mêmes dans les deux suites; nous les appellerons donc $(C_k)_{k=1..p}$. Appelons $(NX_i)_{i=1..n}$, $(NY_j)_{j=1..m}$ et $(N_k)_{k=1..p}$, les nombres d'apparitions respectifs de ces calls dans la trace.

La force relative du niveau d'interaction **dynamique** entre X et Y peut être mesurée par la formule suivante[3]:

$$\mathbf{Inter_2(X,Y)} = \frac{\sum_{k=1..p} (N_k)^2}{\left(\sum_{i=1..n} (NX_i)^2 \right)^{1/2} \left(\sum_{j=1..m} (NY_j)^2 \right)^{1/2}}$$

Etant donné un ensemble de classes d'objets $U = \{O_1, O_2, ..., Or\}$ formant une unité U, la cohésion de U est mesurée par la formule[4]:

$$\mathbf{Cohesion(U)} = \frac{\sum_S \mathbf{Inter2(Oi,Oj)}}{\sum_{q=1..r-1} q}$$

où $S = \{ (i,j) \mid i,j \in [1,r] \wedge i > j \}$

Cette métrique permet d'évaluer la cohésion des unités (des valeurs proches de 1 sont excellentes), de comparer les cohésions des différentes unités et éventuellement de modifier les unités pour en augmenter la cohésion.
Lorsque la cohésion d'une unité est mauvaise, une étude des valeurs Inter2(Oi,Oj) peut permettre d'isoler le ou les objets responsables.

Les statistiques basées sur cette métrique doivent, bien entendu, s'appuyer sur des techniques d'échantillonnage adéquates.

---

[3] Remarquons que $Inter_2(X,Y) \in [0,1]$. En particulier, si X n'interagit dynamiquement qu'avec Y, alors $Inter_2(X,Y) = 1$.

[4] Cohesion(U) représente la moyenne des forces relatives d'interaction entre chaque couple de classes de U. Cette mesure est donc également comprise dans l'intervalle [0,1].

▪ Autres mesures quantitatives plus complexes: mesure de l'apparition de scénarii complexes.
Par exemple détection de l'apparition de séquences de n actions consécutives A1, A2, .. ,An dans le cycle de vie d'objets de classe CLASS.
Ces mesures permettent d'évaluer la fréquence de certains scénarii. Elles guident une éventuelle optimisation, elles permettent de mettre le doigt sur des lacunes. Elles permettent également le suivi de l'utilisation de l'application.
L'idée de mesurer des scénarii plus complexes germe généralement d'une mesure globale préalable des scénarii simples que nous avons abordée au point précédent de cette section.

5.2 Analyse d'une trace Oblog: Problématique et Choix d'un outil

L'analyse d'une trace Oblog devrait être menée de façon efficace et intelligente. Un outil automatisé destiné à une telle analyse doit être conçu de manière adaptée aux problèmes particuliers rencontrés dans ce type d'analyse.

- L'outil doit donner les moyens à l'analyste de **détecter les scénarii très divers et éventuellement complexes** de comportement des objets d'un système Oblog.

En effet, ces objets peuvent évoluer (comportement interne: dynamique des événements isolés) indépendamment les uns des autres (objets actifs), excepté dans les moments d'interaction directe, où ils évoluent de manière synchronisée (comportement interactif: dynamique des événements partagés). Certains d'entre eux (objets DBX, objets subissant un call externe au système) peuvent évoluer en réponse à des événements externes. Un service macroscopique parmi d'autres peut dès lors réclamer la collaboration d'une collection d'objets dans le cadre de scénarii (ordonnancement d'événements partagés et/ou isolés, d'origine interne ou externe au système) relativement complexes. Hormis le comportement du système, le comportement des utilisateurs externes peut également être appréhendé au travers de scénarii faisant intervenir des objets DBX.

- L'outil doit faciliter l'analyse de **traces de taille très importante** et ce de manière efficace.

En effet, une trace Oblog peut contenir un très grand nombre d'événements concernant un grand nombre d'objets. En l'absence de mécanismes de sélection (l'analyste isole ce qui l'intéresse) et de filtrage (l'analyste rejette des éléments inutiles ou éprouvés) souples à mettre en oeuvre et efficaces, l'analyse se révèle impossible en pratique.

- L'outil doit être **réutilisable**.

Au niveau **physique**, il devrait par exemple permettre l'analyse de traces d'autres formats produites par d'autres systèmes d'objets actifs communiquant par événements partagés.
Au niveau **logique**, l'outil devrait pouvoir supporter un modèle général de trace produite par un système logique d'objets actifs communiquant par événements partagés. Un langage de haut niveau devrait permettre à l'analyste d'interroger une trace logique au moyen de questions exprimées en termes de composants abstraits: événements, actions, objets, résultats, ...
La réutilisabilité au niveau logique permet la portabilité des questions d'analyse d'un système à l'autre et la comparaison de systèmes.

A première vue, il apparaît que la problématique associée à l'analyse d'une trace Oblog est très similaire à celle rencontrée dans l'analyse des audit trails d'un système d'exploitation. L'outil ASAX a largement fait ses preuves dans ce domaine avec pour objectif l'étude et la mise en oeuvre de la sécurité du système d'exploitation [Sécurité1].

Rappelons que ASAX est un outil universel, puissant et efficace.

▪ Son caractère universel permet théoriquement l'analyse de tout fichier séquentiel via traduction par Format Adaptator de ce fichier dans le format NADF (réutilisation physique). La construction du Format Adaptator d'une trace Oblog réalisée dans ce mémoire est un exemple de plus à l'appui de cette intuition.
Une extension d'ASAX au support d'un modèle logique de détection d'intrusion (réutilisation logique) est envisagée par les auteurs du projet. Elle est aussi théoriquement envisageable pour le support d'un modèle logique de l'activité d'un système d'objets actifs communiquant par événements partagés.

▪ La puissance de ASAX s'exprime au travers du langage Russel. Ce dernier permet l'expression de critères de sélection complexes qui concernent des séquences d'enregistrements arbitrairement longues (détection de scénarii, filtrage, sélection, ...). D'autre part, ce langage est conçu pour traiter le fichier d'analyse de gauche à droite (une seule passe), ce qui est obligatoire pour l'efficacité du traitement de fichiers de grande taille.

▪ L'efficacité de ASAX est atteinte d'une part par le principe constitutif de Russel qui permet de ne traiter qu'une et une seule fois chaque enregistrement et d'autre part par des techniques d'implémentation efficaces.

ASAX semble donc **approprié comme outil support d'une analyse de trace Oblog**.

Il nous permettra de mettre en oeuvre diverses techniques d'analyse de trace:

- Filtrage de scénarii classiques, éprouvés ou fréquents pour ne conserver que l'exceptionnel, le non prévu ou les erreurs;
- Analyse statistique du comportement du système afin de déterminer des profils de comportement fréquents, de détecter une mauvaise organisation du système ou d'une partie du système, de mettre en évidence des scénarii exceptionnels;
- Sélection/détection de scénarii exceptionnels connus ou de répétitions anormalement élevées de tels scénarii et suivi des actions consécutives à la détection de tels phénomènes;
- Sélection des événements présentant de l'intérêt pour un objectif précis d'analyse.

Nous allons présenter, dans la section suivante, quelques exemples de règles Russel d'analyse d'une trace Oblog produite par l'application Monde Bancaire.

5.3 Exemples de règles Russel pour l'analyse d'une trace Oblog

Dans les sections précédantes, nous avons d'abord présenté différents domaines d'étude où l'analyse d'une trace Oblog peut s'avérer intéressante. Ensuite nous avons justifié le choix de l'outil ASAX pour supporter une telle analyse.
Nous allons maintenant présenter quelques règles Russel simples pour l'analyse d'une trace de l'application Monde Bancaire afin d'illustrer ce double propos.

5.3.1 Exemple 0 : Produire et compter les événements élémentaires de la trace

Dans cet exemple, l'objectif est d'afficher sous la forme (nom externe, valeur associée) les données d'audit contenues dans chaque enregistrement d'une trace Oblog traduite en format NADF et de compter ces enregistrements. L'intérêt est de valider le processus de traduction.

La fonction d'affichage sera implémentée en utilisant trois règles:

scan , *show_rec* et *prnt_count*

Le but de la règle *scan* est de déclencher la règle *show_rec* chaque fois qu'un enregistrement est détecté et d'incrémenter le nombre d'enregistrements (*nbre*) rencontrés à ce stade. Ce nombre est maintenu dans une variable globale de manière à être récupérable en fin de session.
Remarquons qu'il est nécessaire que la règle *scan* se perpétue explicitement pour traiter tous les enregistrements du fichier NADF.

Le but de la règle *show_rec* est d'afficher tous les couples (nom externe, valeur associée) présents dans l'enregistrement courant.

Le but de la règle *prnt_count* est d'afficher *nbre.*

La règle *scan* est déclenchée en début de session et la règle *prnt_count* est déclenchée en fin de processus.

La traduction de ces règles en Russel prend la forme suivante:

```
global nbre: integer;

rule scan;
begin
        trigger off for_current show_rec;
        trigger off for_next scan;
        nbre := nbre + 1
end;
```

```
rule show_rec;
begin
if present ACTTYPE      --> println('ACTTYPE    ', ACTTYPE)   fi;
if present ACTCLAS      --> println('ACTCLAS    ', ACTCLAS)   fi;
if present ACTADDR      --> println('ACTADDR    ', ACTADDR)   fi;
if present ACTACTI      --> println('ACTACTI    ', ACTACTI)   fi;
if present ACTSITU      --> println('ACTSITU    ', ACTSITU)   fi;
if present ACTCCLA      --> println('ACTCCLA    ', ACTCCLA)   fi;
if present ACTCADD      --> println('ACTCADD    ', ACTCADD)   fi;

...
...
...

if present ZIP          --> println('ZIP        ', ZIP)       fi;
if present ZIP__E       --> println('ZIP__E     ', ZIP__E)    fi;
println('_____')
end;

rule prnt_count;
println('___Nombre total d'événements : ', nbre ,' _____');

init_action;
begin
        nbre := 0;
        trigger off for_next scan;
        trigger off at_completion prnt_count
end.
```

Dans la suite, nous considérerons que nous disposons toujours de la règle *show_rec* (accessible comme règle externe via la clause Russel **uses**) ou encore d'une procédure externe *Display_current* équivalente mais plus efficace.

5.3.2 Exemple 1 : Suivi d'un service macroscopique

Dans cet exemple, nous allons détecter et suivre jusqu'à leur aboutissement, toutes les occurrences de la demande de service « Ajout d'un nouveau client » par un utilisateur du système (voir schéma des interactions directes de la section 2.6). Nous produirons également tous les événements concourants à ces demandes de service pour détecter d'éventuelles actions parallèles.

Un tel service peut, par exemple, être initié par un événement partagé (composé de trois événements élémentaires) qui se traduit dans la trace Oblog comme suit:

```
□□□□□
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Object SYNTAX_CLIENT     (3329776) Action start      Situation doCheck     Caller ADD_CLI_D      (3325904)
Event Attribute myMaster = OBJECT NAME: ADD_CLI_D      OBJECT ADDRESS:3325904
Event Attribute Name = STRING: 'Fatal' (len=5)
Event Attribute Street = STRING: 'Damaia Steet' (len=12)
Event Attribute Phone = STRING: '244444628' (len=9)
Event Attribute ZIP = STRING: '1500' (len=4)
Event Attribute Town = STRING: 'AMADORA CITY' (len=12)
Event Attribute CliNum = STRING: '1000' (len=4)
Event Attribute Title = STRING: 'Mr' (len=2)
Attribute myMaster = OBJECT NAME: ADD_CLI_D      OBJECT ADDRESS:3325904
Attribute Name = STRING: 'Fatal' (len=5)
Attribute Street = STRING: 'Damaia Steet' (len=12)
Attribute status = SYN_CLI_S: ERROR
Attribute ZIP = STRING: '1500' (len=4)
Attribute Town = STRING: 'AMADORA CITY' (len=12)
Attribute CliNum = STRING: '1000' (len=4)
Attribute Phone = STRING: '244444628' (len=9)
Attribute Title = STRING: 'Mr' (len=2)
--------------------------------------------------------------------------------
Object ADD_CLI_D      (3325904) Action add      Situation syntaxWait     Caller ADD_CLI_X      (3327088)
Event Attribute Phone = STRING: '244444628' (len=9)
Event Attribute CliNum = STRING: '1000' (len=4)
Event Attribute Name = STRING: 'Fatal' (len=5)
Event Attribute Street = STRING: 'Damaia Steet' (len=12)
Event Attribute myself = OBJECT NAME: ADD_CLI_D      OBJECT ADDRESS:3325904
Event Attribute myServer = OBJECT NAME: SYNTAX_CLIENT      OBJECT ADDRESS:3329776
Event Attribute Title = STRING: 'Mr' (len=2)
Event Attribute ZIP = STRING: '1500' (len=4)
Event Attribute Town = STRING: 'AMADORA CITY' (len=12)
Attribute created_msg = STRING: 'Client was created !' (len=20)
Attribute exists_msg = STRING: 'Client already exists!' (len=22)
Attribute syntaxErr_msg = STRING: 'Syntax Error !' (len=14)
Attribute status = SYN_CLI_S: ERROR
Attribute counterRef = OBJECT NAME: COUNTER      OBJECT ADDRESS:3325104
Attribute myDialog = OBJECT NAME: ADD_CLI_X      OBJECT ADDRESS:3327088
--------------------------------------------------------------------------------
Object ADD_CLI_X      (3327088) Action add      Situation waitUser
Attribute Name = STRING: 'Fatal' (len=5)
Attribute Street = STRING: 'Damaia Steet' (len=12)
Attribute message = STRING: Empty String.
Attribute ZIP = STRING: '1500' (len=4)
Attribute Town = STRING: 'AMADORA CITY' (len=12)
Attribute CliNum = STRING: '1000' (len=4)
Attribute Phone = STRING: '244444628' (len=9)
Attribute Title = STRING: 'Mr' (len=2)
Attribute myDriver = OBJECT NAME: ADD_CLI_D      OBJECT ADDRESS:3325904
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Cet événement partagé est suivi de 0, 1 ou plusieurs autres événements élémentaires précédant une clôture du service qui peut, par exemple, être détectée par l'apparition d'un deuxième événement partagé, apparaissant comme suit dans la trace Oblog:

```
□□□□□
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Object ADD_CLI_X      (3327088) Action setMessage    Situation waitUser     Caller ADD_CLI_D      (3325904)
Event Attribute message = STRING: 'Client was created !' (len=20)
Attribute Name = STRING: 'Fatal' (len=5)
Attribute Street = STRING: 'Damaia Steet' (len=12)
Attribute message = STRING: 'Client was created !' (len=20)
Attribute ZIP = STRING: '1500' (len=4)
Attribute Town = STRING: 'AMADORA CITY' (len=12)
Attribute CliNum = STRING: '1000' (len=4)
Attribute Phone = STRING: '244444628' (len=9)
Attribute Title = STRING: 'Mr' (len=2)
Attribute myDriver = OBJECT NAME: ADD_CLI_D      OBJECT ADDRESS:3325904
------------------------------------------------------------------------------
Object ADD_CLI_D      (3325904) Action createAnswer   Situation dialogWait    Caller CREATE_CLIENT     (3330896)
Event Attribute message = STRING: 'Client was created !' (len=20)
Event Attribute status = CREA_CLI_S: CREATED
Attribute created_msg = STRING: 'Client was created !' (len=20)
Attribute exists_msg = STRING: 'Client already exists!' (len=22)
Attribute syntaxErr_msg = STRING: 'Syntax Error !' (len=14)
Attribute status = SYN_CLI_S: OK
Attribute counterRef = OBJECT NAME: COUNTER      OBJECT ADDRESS:3325104
Attribute myDialog = OBJECT NAME: ADD_CLI_X      OBJECT ADDRESS:3327088
------------------------------------------------------------------------------
Object CREATE_CLIENT     (3330896) Action done      Situation ???
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

De manière plus générale, la spécification de Monde Bancaire [Oblog1] précise qu'une demande d'ajout d'un nouveau client est initiée par un événement partagé de type:

SYNTAX_CLIENT.start ←← ADD_CLI_D.add ←← ADD_CLI_X.add

D'autre part, elle peut être clôturée (réponse à la demande) par l'un ou l'autre des deux types d'événements partagés qui suivent:

ADD_CLI_X.setMessage ←← ADD_CLI_D.syntaxAnswer ←← SYNTAX_CLIENT.done

ADD_CLI_X.setMessage ←← ADD_CLI_D.createAnswer ←← CREATE_CLIENT.done

Le premier type d'événement partagé de clôture indique un refus syntaxique tandis que le deuxième type indique soit un refus d'ajout car le client proposé existe déjà, soit une acceptation de l'ajout.

L'identité de l'objet de classe ADD_CLI_X doit être identique dans l'événement initial et dans l'événement final.

La fonction de suivi est implémentée comme suit.

Une règle *detect1* se perpétue. Lorsqu'elle reconnaît un événement élémentaire **SYNTAX_CLIENT.start**, elle déclenche une règle *detect2* pour l'enregistrement suivant.
La règle *detect2* déclenche une règle *detect3* pour l'enregistrement suivant, lorsqu'elle reconnaît un événement élémentaire **ADD_CLI_D.add**. La règle *detect3* déclenche une règle *watch* pour l'enregistrement suivant, lorsqu'elle reconnaît un événement élémentaire **ADD_CLI_X.add**, en passant comme paramètre un compteur de clôture $n = 3$ et l'identité de l'instance de classe ADD_CLI_X subissant l'événement.
Chacune de ces règles affiche l'enregistrement courant.

```
uses show;

rule detect1;
if ACTCLAS = 'SYNTAX_CLIENT' and ACTTYPE = 'F'
  and ACTACTI = 'start' and ACTCCLA = 'ADD_CLI_D'
  --> begin
      println('detect1');
      trigger off for_current show_rec;
      trigger off for_next detect2;
      trigger off for_next detect1
      end;
  true
  --> trigger off for_next detect1
fi;

rule detect2;
if ACTCLAS = 'ADD_CLI_D' and ACTTYPE = 'N'
  and ACTACTI = 'add' and ACTCCLA = 'ADD_CLI_X'
  --> begin
      println('detect2');
      trigger off for_current show_rec;
      trigger off for_next detect3
      end;
  true
  --> println('detect2 failed')
fi;

rule detect3;
if ACTCLAS = 'ADD_CLI_X' and ACTTYPE = 'L' and ACTACTI = 'add'
  --> begin
      println('detect3');
      trigger off for_current show_rec;
      trigger off for_next watch(3,ACTADDR)
      end;
  true
  --> println('detect3 failed')
fi;
```

La règle *watch* reçoit en argument un compteur de clôture *n* et l'identité *xid* de l'instance de classe ADD_CLI_X à l'origine de la demande.
Si elle détecte un événement élémentaire **ADD_CLI_X.setMessage**, pour une instance d'objet *xid*, avec $n = 3$, elle se perpétue en décrémentant *n* de 1.

Si elle détecte un événement élémentaire **ADD_CLI_D.syntaxAnswer** ou **ADD_CLI_D.createAnswer**, avec $n = 2$, elle se perpétue en décrémentant n de 1.
Si elle détecte un événement élémentaire **ADD_CLI_D.syntaxAnswer** ou **ADD_CLI_D.createAnswer**, avec $n = 2$, elle se perpétue en décrémentant n de 1.
Si elle détecte un événement élémentaire **SYNTAX_CLIENT.done** ou **CREATE_CLIENT.done**, avec $n = 1$, elle ne se perpétue plus.
Dans tous les autres cas, elle se perpétue avec les arguments (3,*xid*).

A chaque activation, l'enregistrement courant est affiché.

```
rule watch(n: integer; xid: string);
if n = 3 and ACTCLAS = 'ADD_CLI_X' and ACTTYPE = 'F'
  and ACTACTI = 'setMessage' and ACTCCLA = 'ADD_CLI_D'
  --> begin
      trigger off for_current show_rec;
      trigger off for_next watch(2,xid)
      end;
  n = 2 and ACTCLAS = 'ADD_CLI_D' and ACTTYPE = 'N'
  and ( (ACTACTI = 'syntaxAnswer' and ACTCCLA = 'SYNTAX_CLIENT')
       or (ACTACTI = 'createAnswer' and ACTCCLA = 'CREATE_CLIENT'))
  --> begin
      trigger off for_current show_rec;
      trigger off for_next watch(1,xid)
      end;
  n = 1 and ( (ACTCLAS = 'SYNTAX_CLIENT' and ACTACTI = 'done')
            or (ACTCLAS = 'CREATE_CLIENT' and ACTACTI = 'done'))
  and ACTTYPE = 'L'
  --> begin
      trigger off for_current show_rec
      end;
  true
  --> begin
      trigger off for_current show_rec;
      trigger off for_next watch(3,xid)
      end
fi ;
```

Remarquons que les règles *detect1*, *detect2*, *detect3* et *watch* pourraient être rendues plus robustes à d'éventuelles modifications de la spécification du système en contrôlant les identités des instances appelantes (via un paramètre supplémentaire: identité de l'appelant) dans les actions appelantes.

La règle *detect1* est déclenchée à l'initialisation.

```
init_action;
trigger off for_next detect1.
```

5.3.3 Exemple 2 : Isoler le comportement des objets d'une classe donnée

Dans cet exemple, nous allons isoler le comportement des objets de classe WITHDRAW ainsi que les appels de ou vers ces objets.

Pour ce faire, nous allons construire deux règles génériques (*isole* et *caller*) qui nous permettront d'isoler le comportement des objets d'une classe donnée *class* quelconque ainsi que les appels de ou vers les objets de cette classe donnée.

Le but de la règle *isole*, d'argument *class*, est d'afficher toutes les actions concernant des instances d'objet de classe *class* ainsi que toutes les actions appelées par une action d'une instance d'objet de classe *class*. La règle *isole* se perpétue.

Lorsque cette règle constate qu'une action portant sur une instance d'objet de classe *class* est appelée, elle déclenche pour l'enregistrement suivant la règle *caller* en lui fournissant l'identité de l'instance appelante (Ce paramètre suffit à identifier l'action appelante car une instance d'objet ne peut exécuter qu'une et une seule action dans le cadre d'un événement partagé).

Le but de la règle *caller*, d'argument *oid*, est d'afficher le contenu de l'enregistrement courant s'il concerne une instance d'objet d'identité *oid*. Sinon, la règle se perpétue avec le même argument. Si l'enregistrement courant concerne un nouvel événement (isolé ou partagé) que celui ayant donné lieu à son déclenchement, la règle *caller* produit un message d'erreur.

```
uses show;
rule isole(class: string);
begin
if ACTCLAS = class
  --> begin
      if ACTTYPE = 'F' or ACTTYPE = 'N'
        --> trigger off for_next caller(ACTCADD) fi;
      trigger off for_current show_rec
      end;
  (ACTTYPE = 'F' or ACTTYPE = 'N') and ACTCCLA = class
  --> trigger off for_current show_rec
fi;
trigger off for_next isole(class)
end
```

```
rule caller(oid: string);
if ACTTYPE = 'I' or ACTTYPE = 'F'
  --> println('Error. Caller not found');
  ACTADDR = oid
  --> trigger off for_current show_rec;
  true
  --> trigger off for_next caller(oid)
fi;

init_action;
trigger off for_next isole('WITHDRAW').
```

La règle *isole* est déclenchée à l'initialisation avec le paramètre 'WITHDRAW'

5.3.4 Exemple 3 : Détecter des tentatives de retrait d'argent suspectes

La spécification de l'application Monde Bancaire [Oblog1] nous précise que toutes les tentatives de retrait d'argent d'un compte aboutissent à la création d'un objet transitoire de classe WITHDRAW qui contrôle la tentative. Avant d'exécuter son action de mort, un tel objet exécute une et une seule des trois actions suivantes: *withdraw*, *unknowACC* ou *overLimit* (voir le GTE de la classe WITHDRAW en annexe C) l'amenant dans la situation *done*. Ces actions sont représentatives du résultat de la tentative (respectivement retrait fructueux, refus pour compte inconnu et refus pour dépassement de limite).
Un attribut appelé *AccNum*, initialisé par l'opération de création de l'objet précise le numéro du compte concerné.

Illustrons un retrait fructueux tel qu'il apparaît dans la trace Oblog:

```
Object WITHDRAW        (6766496) Action withdraw      Situation done
Attribute status = DRAWSTATUS: OK
Attribute AccNum = STRING: '21' (len=2)
Attribute amount = 10000 (INT)
Attribute myMaster = OBJECT NAME: ATM_D          OBJECT ADDRESS:6753200
Attribute myAccount = 10 (INT)
```

Dans cet exemple 3, nous allons produire des messages d'avertissement lorsque $m > 1$ tentatives de retrait d'argent pour un même compte ont été successivement refusées pour dépassement de limite.

Pour réaliser cet objectif, nous utiliserons deux règles:

overlimit et *count_over*

Le but de la règle *overlimit*, d'argument *m*, est de détecter toute tentative de retrait refusée pour dépassement de limite et de déclencher dans ce cas, la règle *count_over* qui va surveiller les résultats des tentatives de retraits ultérieures concernant le même numéro de compte (arguments : *m*-1 et ACCNUM).

Le but de la règle *count_over* est d'afficher un message d'avertissement lorsque *m* tentatives de retrait pour le compte *num* ont successivement avorté pour dépassement de limite.
Cette règle s'arrête si elle rencontre un retrait fructueux ou un refus pour compte inconnu pour le numéro de compte *num.*

Ces règles s'expriment comme suit dans le langage Russel:

```
rule overlimit(m: integer);
if ACTCLAS = 'WITHDRAW' and ACTACTI = 'overLimit'
  --> begin
      trigger off for_next count_over(m-1,ACCNUM);
      trigger off for_next overlimit(m)
      end;
  true
  --> trigger off for_next overlimit(m)
fi;

rule count_over(m: integer; num: string);
if ACTCLAS = 'WITHDRAW' and ACCNUM = num
  and ACTSITU = 'done'
  --> if ACTACTI  = 'withdraw' or ACTACTI = 'unknownACC'
          --> skip;
          m > 1
          --> trigger off for_next count_over(m-1,num);
          true
          --> println(m,' essais hors limites: compte ',num)
      fi;
  true
  --> trigger off for_next count_over(m,num)
fi;

init_action;
trigger off for_next overlimit(3).
```

La règle *overlimit* est ici déclenchée initialement avec *m* = 3.

5.3.5 Exemple 4 : Suivi de l'utilisation de la fonction « sélection de clients »

L'application Monde Bancaire propose à l'employé du guichet une fonction de sélection de listes de clients basée sur des critères divers. Cette fonction permet à l'utilisateur de prendre des renseignements sur les clients et/ou de supprimer des clients sélectionnés.
Nous nous proposons dans cet exemple simple d'appréhender le comportement des utilisateurs de cette fonction face à des listes de taille importante.

Les demandes de gestion d'une liste de clients sont toutes soumises à des instances d'objet de classe SELECT_CLIENT [Oblog1]. L' analyse du comportement de ces objets « esclaves » permet également de suivre le comportement de leur « maître » (l'utilisateur) dans la fonction de sélection (voir le GTE de la classe SELECT_CLIENT en annexe C).

Dans l'exemple 4, notre objectif concret sera d'afficher les tailles de toutes les listes clients qui ont été entièrement parcourues et de suivre le comportement de l'utilisateur dans la fonction de sélection lorsque la liste des clients sélectionnés est importante.

Cet objectif sera poursuivi au moyen de trois règles: *endoflist, biglist et look_at.*

La règle *endoflist* affiche, chaque fois qu'elle détecte un « événement » SELECT_CLIENT.*endOfList* (significatif d'une liste clients entièrement parcourue), la valeur de l'attribut INDEX de l'instance concernée (représentant dans ce cas le nombre d'éléments de la liste)

```
rule endoflist;
if ACTACTI = 'endOfList' and ACTCLAS = 'SELECT_CLIENT'
  --> begin
      println('liste entierement parcourue, taille ',INDEX);
      trigger off for_next endoflist
      end;
  true
  --> trigger off for_next endoflist
fi;
```

La règle *biglist*, d'argument $n > 1$, détecte tous les événements de type SELECT_CLIENT.*select* concernant des instances dont la valeur de l'attribut *result* comprend au moins n éléments. Dans ces cas, elle déclenche la règle *look_at*, pour l'enregistrement suivant en lui confiant l'identité de l'instance d'objet à observer.
Précisons que l'événement SELECT_CLIENT.*select* n'apparaît qu'une seule fois dans le cycle de vie de toute instance de classe SELECT_CLIENT, consécutivement à sa création. A l'issue de l'action *select*, l'attribut *result* contient une liste de clients sélectionnés.

La règle *look_at*, d'argument *oid*, affiche tous les événements élémentaires, de l'instance d'objet de classe SELECT_CLIENT et d'identité *oid*, qu'elle rencontre. Elle se perpétue tant qu'elle n'a pas détecté la mort (ACTACTI = '???') de cette instance, c'est à dire la fin de la fonction de sélection.

```
rule biglist(n: integer);
if ACTCLAS = 'SELECT_CLIENT' and ACTACTI = 'setCursor'
  and Taille(RESULT) >= n and ACTSITU = 'choice'
  --> begin
      trigger off for_next look_at(ACTADDR);
      trigger off for_next biglist(n)
      end;
  true
  -->
  trigger off for_next biglist(n)
fi;

rule look_at(oid: string);
if ACTCLAS = 'SELECT_CLIENT' and ACTADDR = oid
  --> if ACTSITU = '???'
        --> trigger off for_current show_rec;
        true
        --> begin
            trigger off for_current show_rec;
            trigger off for_next look_at(oid)
            end;
      fi;
  true
  --> trigger off for_next look_at(oid)
fi;

init_action;
begin
trigger off for_next endoflist;
trigger off for_next biglist(30)
end.
```

A l'initialisation, les règles *endoflist* et *biglist* sont déclenchées.

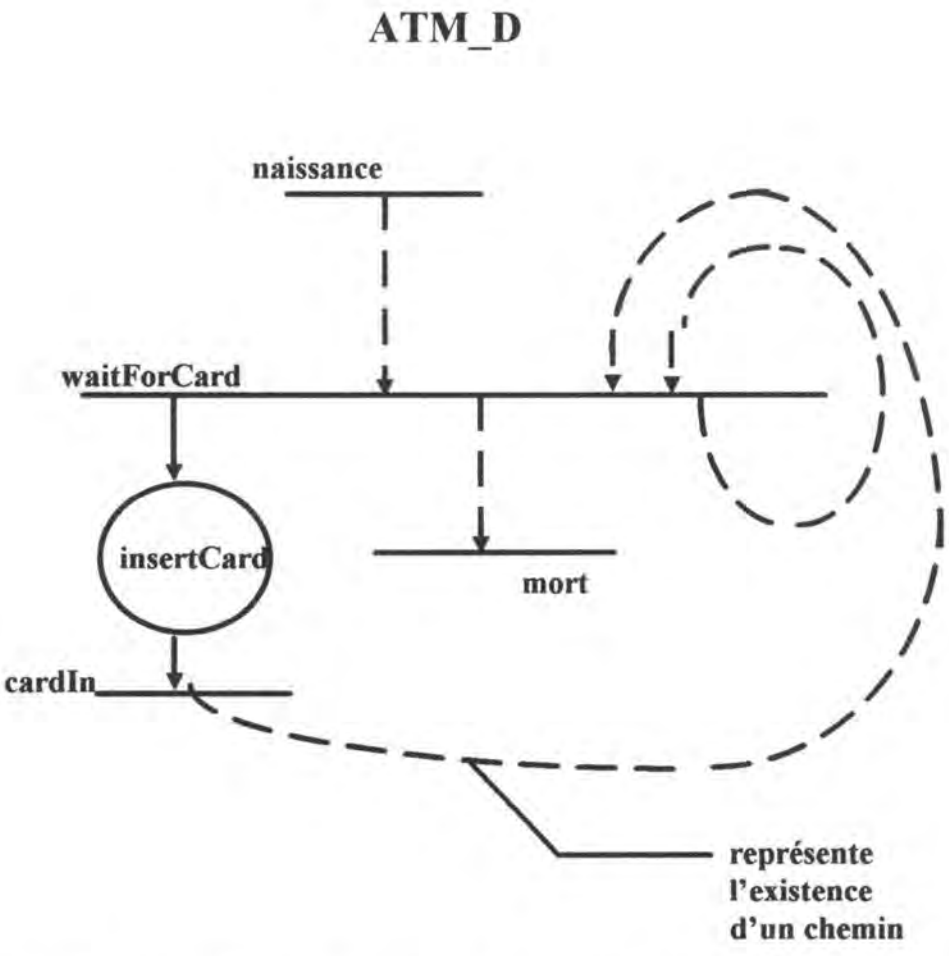
Remarquons que *Taille* est une fonction préprogrammée qui calcule le nombre d'éléments contenus dans la « liste » *result*.

5.3.6 Exemple 5 : Utilisation des codes personnels pour les retraits d'argent

Dans l'application Monde Bancaire, tous les retraits d'argent sont effectués via une interface émulant le comportement d'un automate. Chaque retrait, du point de vue de l'utilisateur, est précédé de l'insertion d'une carte identifiant le compte concerné, suivie de l'introduction d'un code personnel.

Dans cet exemple, nous allons produire un message d'avertissement à chaque couple, détecté dans la trace, d'introductions « consécutives » d'un mauvais numéro d'identification personnel pour un même compte.

Pour réaliser cette fonction, nous étudierons le comportement des instances d'objet de classe ATM_D qui gérent la logique du dialogue utilisateur/automate.
Pour aider à la compréhension, les points importants du GTE de la classe ATM_D (voir annexe C) sont illustrés dans le schéma suivant:



Une tentative de retrait d'argent (associée à une carte valide) peut être caractérisée par le déclenchement d'une action *insertCard* agissant sur une instance d'objet de classe ATM_D. Après exécution de cette action, l'instance d'objet se trouve dans la situation *cardIn*.

L'instance d'objet ATM_D évolue ensuite de la situation *cardIn* vers la situation *waitForCard* en empruntant un des chemins possibles entre ces deux situations.
Un tel chemin peut ou non passer par une et une seule action *wrongPin*, indiquant la détection d'un mauvais code personnel.
L'action *wrongPin* n'apparaît pas dans les autres types de chemins représentés en pointillés sur le schéma.
L'action *insertCard* garnit l'attribut ACCNUM identifiant le compte concerné par la tentative de retrait.
Revenue dans la situation *waitForCard*, l'instance d'objet ATM_D peut évoluer dans un chemin caractéristique d'une tentative de retrait (via *insertCard*), mais aussi dans un chemin d'un autre type.

Pour implémenter la fonction d'avertissement, nous créons trois règles: *wrongpin*, *scanobj* et *scanacc*.

La règle *wrongpin* détecte tout événement de type ATM_D.*wrongPin*. A chaque détection, elle déclenche la règle *scanobj* pour l'enregistrement suivant, en lui confiant l'identité de l'instance d'objet concernée et le numéro de compte attribut de cette instance.

La règle *scanobj*, d'arguments *oid* et *num*, se perpétue tant qu'elle ne détecte pas le retour de l'instance *oid* dans la situation *waitForCard*. Dans ce dernier cas, elle déclenche pour l'enregistrement suivant la règle *scanacc* en lui passant pour argument le numéro *num* du compte à surveiller.

La règle *scanacc*, d'argument *no*, se perpétue tant qu'elle ne détecte pas l'un des deux contextes courants qui suivent:
- un événement ATM_D.*insertCard* (nouvelle tentative de retrait) concernant une instance d'objet dont l'attribut ACCNUM a une valeur différente de *no* (autre compte);
- un événement ATM_D.*wrongPin* concernant une instance d'objet dont l'attribut ACCNUM a une valeur égale à *no*.
Dans le deuxième contexte, elle affiche un message d'avertissement pour le compte *no*.

La traduction de ces règles en Russel est la suivante:

```
rule wrongpin;
if ACTCLAS = 'ATM_D' and ACTACTI = 'wrongPin'
  --> begin
      trigger off for_next scanobj(ACTADDR,ACCNUM);
      trigger off for_next wrongpin
      end;
  true
  --> trigger off for_next wrongpin
fi;
```

```
rule scanobj(oid,num: string);
if ACTSITU = 'waitForCard' and ACTADDR = oid
  --> trigger off for_next scanacc(num);
  true
  --> trigger off for_next scanobj(oid,num)
fi;

rule scanacc(no: string);
if ACTCLAS = 'ATM_D' and ACCNUM != no
  and ACTSITU = 'cardIn' and ACTACTI = 'insertCard'
  --> skip;
  ACTCLAS = 'ATM_D' and ACCNUM = no
  and ACTACTI = 'wrongPin'
  --> println ('Couple de PIN invalides; compte=',no);
  true
  --> trigger off for_next scanacc(no)
fi;

init_action;
trigger off for_next wrongpin.
```

A l'initialisation, la règle *wrongpin* est déclenchée.

5.3.7 Exemple 6 : Sélection des événements touchant des objets DBX

Dans cet exemple, l'objectif est de visualiser rétrospectivement toutes les interactions des utilisateurs avec le système afin de servir de base à une étude des contacts entre le système informatique et sa périphérie.

Cette fonction de visualisation est assurée par une règle *dbx*.

La règle *dbx* affiche le contenu (noms externes, valeurs correspondantes) de tous les enregistrements associés à des événements élémentaires portant sur des instances d'objet de type DBX (« dialog representation »).

La règle *dbx* est déclenchée à l'initialisation.

```
uses show;

rule dbx;
if ACTCLAS = 'ATM_X' or ACTCLAS = 'ADD_CLI_X'
  or ACTCLAS = 'DELQUE_CLI_X' or ACTCLAS = 'ADD_ACC_X'
  or ACTCLAS = 'DELQUE_ACC_X'
  --> begin
      trigger off for_current show_rec;
      trigger off for_next dbx
      end;
  true
  --> trigger off for_next dbx
fi;

init_action;
trigger off for_next dbx.
```

5.3.8 Exemple 7 : Statistiques sur les interactions directes

Nous avons éclairé dans la section 5.1, l'intérêt des mesures quantitatives (à différents niveaux d'agrégation) des interactions directes entre les acteurs d'un système d'objets actifs.

Dans cet exemple, nous nous proposons d'illustrer comment mesurer les fréquences d'apparition de calls entre actions et entre classes.
Ces fréquences permettront notamment de pondérer les « flèches » des diagrammes d'interactions directes détaillés et agrégés (voir section 2.6).

Un appel d'action peut se matérialiser comme suit dans la trace Oblog:

```
Object ADD_CLI_D        (3325904) Action exit       Situation exit       Caller ADD_CLI_X       (3327088)
Attribute created_msg = STRING: 'Client was created !' (len=20)
Attribute exists_msg = STRING: 'Client already exists!' (len=22)
Attribute syntaxErr_msg = STRING: 'Syntax Error !' (len=14)
Attribute status = SYN_CLI_S: OK
Attribute counterRef = OBJECT NAME: COUNTER          OBJECT ADDRESS:3325104
Attribute myDialog = OBJECT NAME: ADD_CLI_X          OBJECT ADDRESS:3327088
```

La détection de l'action appelée qui précède nous permettra d'incrémenter un compteur mesurant la fréquence des appels des instances de classe ADD_CLI_X vers des instances de classe ADD_CLI_D. L'incrémentation de ce compteur sera réalisée par une procédure prédéfinie *increment_interaction* invoquée ici comme suit:

increment_interaction('ADD_CLI_D' , 'ADD_CLI_X')

Les compteurs gérés par cette procédure sont conservés dans des zones de mémoire permanentes à la session ASAX.

Remarquons que l'action appelée que nous venons d'illustrer ne contient pas de données d'audit permettant de déterminer le nom de l'action appelante. Nous devrons donc examiner les enregistrements suivants (dans le cadre du même événement partagé) pour retrouver ce nom, sur base de l'identité connue (3327088) de l'instance appelante.

Supposons que l'action appelante se matérialise comme suit:

```
Object ADD_CLI_X        (3327088) Action exit        Situation waitUser
Attribute Name = STRING: 'Zeippen' (len=7)
Attribute Street = STRING: 'Piramid Street' (len=14)
Attribute message = STRING: Empty String
Attribute ZIP = STRING: '2795' (len=4)
Attribute Town = STRING: 'L_A' (len=3)
Attribute CliNum = STRING: '1001' (len=4)
Attribute Phone = STRING: '4199994' (len=7)
Attribute Title = STRING: 'Mr' (len=2)
Attribute myDriver = OBJECT NAME: ADD_CLI_D        OBJECT ADDRESS:3325904
```

Nous pouvons alors incrémenter un compteur mesurant les fréquences d'appels de ADD_CLI_X.*exit* vers ADD_CLI_D.*exit* de la manière suivante:

increment_call('ADD_CLI_D' , exit , 'ADD_CLI_X' , exit)

où *increment_call* est également une procédure préprogrammée.

La fonction de mesure sera implémentée grâce à trois règles: *detectcall*, *searchcaller* et *stat_calls*.

La règle *detectcall* détecte toutes les actions appelées. Chaque fois qu'une telle action est isolée, *detectcall* incrémente un compteur externe *cpti*(classe appelée, classe appelante) via la procédure *increment_interaction* et déclenche pour l'enregistrement suivant, la règle *search_caller* en lui confiant le nom de classe et le nom de l'action appelée ainsi que l'identité de l'instance appelante.

La règle *searchcaller*, d'arguments *c*, *a* et *ci*, se perpétue tant qu'elle ne reconnaît pas un des contextes courants suivants:

- le début d'un nouvel événement (partagé ou isolé), auquel cas elle émet un message d'erreur (l'action appelante n'a pu être détectée);

- l'instance courante est d'identité *c* (détection de l'action appelante).

Dans ce dernier contexte, *searchcaller* incrémente un compteur externe
cptc(*c* , *a* , classe courante , action courante) au moyen de la procédure *increment_call*.

La règle *stat_calls* invoque deux procédures préprogrammées *produce_stat_calls* et *produce_stat_interactions* qui affichent les contenus des compteurs gérés respectivement par *increment_call* et *increment_interaction.*

La règle *detectcall* est déclenchée en début de session et la règle *stat_calls* en fin de session.

La traduction Russel de ces règles est la suivante:

```
rule detectcall;
if ACTTYPE = 'F' or ACTTYPE = 'N'
  --> begin
      increment_interaction(ACTCLAS,ACTCCLA);
      trigger off for_next searchcaller(ACTADDR,ACTACTI,ACTCADD);
      trigger off for_next detectcall
      end;
  true
  --> trigger off for_next detectcall
fi;

rule searchcaller(c,a,ci: string);
if ACTTYPE = 'F' or ACTTYPE = 'I'
  --> println('Error');
  ACTADDR = ci
  --> increment_call(c,a,ACTCLAS,ACTACTI);
  true
  --> trigger off for_next searchcaller(c,a,ci)
fi;

rule stat_calls;
begin
produce_stat_calls;
produce_stat_interactions
end;

init_action;
begin
trigger off for_next detectcall;
trigger off at_completion stat_calls
end.
```

5.3.9 Exemple 8 : Statistiques sur les événements élémentaires (transitions)

Cet exemple illustre comment mesurer les fréquences d'apparition de transitions et les fréquences d'apparition d'actions de chaque classe d'objets.
Ces fréquences permettront notamment de pondérer respectivement les actions d'un GTE (vue détaillée) ou d'un Declaration Diagram (vue agrégée).

La mesure des fréquences d'apparitions des actions d'une classe ne pose aucune difficulté.

Pour mesurer les fréquences d'apparitions des transitions d'une classe, nous procéderons de la manière suivante.

Une fois détectée une action quelconque concernant une instance d'objet de classe *c* amenée dans une situation *si* par cette action, nous rechercherons l'action suivante de cette instance dans la trace. Une fois détectée cette deuxième action, dont le nom est *a* et qui a conduit l'instance dans la situation *sf*, nous incrémenterons un compteur
cptt(*c* , *si* , *a* , *sf*) mesurant la fréquence d'apparition des transitions des objets de classe *c*, de la situation *si* à la situation *sf* via l'action *a*.

La fonction de mesure de cet exemple sera implémentée par trois règles: *actionscan*, *findtransition* et *stat_transitions*.

La règle *actionscan* incrémente, pour chaque action de la trace, un compteur externe *cpta*(classe courante, action courante) grâce à une procédure préprogrammée *increment_action*. Elle déclenche également, à chaque action détectée, une règle *findtransition* pour l'enregistrement suivant, en lui confiant l'identité de l'instance courante et la situation courante.

La règle *findtransition*, d'argument *i* et *s*, se perpétue tant qu'elle n'a pas détecté une action concernant une instance d'identité *i*. Dans ce dernier cas, elle incrémente un compteur externe *cptt*(classe courante, *s* , action courante , situation courante) au moyen de la procédure préprogrammée *increment_transition*.

La règle *stat_transitions* invoque deux procédures préprogrammées *produce_stat_transitions* et *produce_stat_actions* qui affichent les contenus des compteurs gérés respectivement par *increment_transition* et *increment_action*.

La règle *actionscan* est déclenchée en début de session et la règle *stat_transitions* en fin de session.

La traduction Russel de ces règles est la suivante:

```
rule actionscan;
begin
increment_action(ACTCLAS,ACTACTI);
trigger off for_next findtransition(ACTADDR,ACTSITU);
trigger off for_next actionscan
end;

rule findtransition(i,s: string);
if ACTADDR = i
  --> increment_transition(ACTCLAS,s,ACTACTI,ACTSITU);
  true
  --> trigger of for_next findtransition(i,s)
fi;

rule stat_transitions;
begin
produce_stat_transitions;
produce_stat_actions
end;

init_action;
begin
trigger off for_next actionscan;
trigger off at_completion stat_transitions
end.
```

5.3.10 Idées d'exemples complémentaires

Sans produire les règles correspondantes, citons encore quelques idées d'exemples complémentaires:

- Isoler le comportement des objets d'une unité spécifique et les actions d'appel vers ces objets (focalisation sur l'étude d'une unité et de son interface);

- Produire les événements participants des événements partagés comptant plus de n>=2 participants;

- A chaque détection d'événement élémentaire, produire la liste de toutes les instances d'objets en vie pour analyse des recouvrements de cycles de vie;

- Détecter des attributs pointeurs qui référencent des objets morts;

- Détecter des mouvements importants sur certains comptes (en fréquence ou taille) et suivre l'activité de ces comptes « louches ».

5.4 Commentaires

Des domaines d'intérêt très diversifiés peuvent guider une analyse de la trace d'une application Oblog (et plus généralement, d'un système d'objets actifs). Nous avons proposé des embryons de démarche d'analyse dans quelques uns de ces domaines.

Les qualités de ASAX ont été évaluées au regard des types de problèmes (taille de la trace, complexité et diversité des scénarii) que posent l'analyse d'une trace Oblog.

Quelques exemples ont éclairé la puissance de Russel comme langage d'analyse d'une trace Oblog (elle même étant très représentative comme modèle de trace d'un système d'objets actifs communiquant par partage d'événements). Ils illustrent également la diversité des domaines d'intérêt qui peuvent guider une telle analyse.

Remarquons toutefois qu'il s'avérera nécessaire d'approfondir, d'organiser et de formaliser les ébauches de démarche d'analyse que nous avons évoquées.
Remarquons également que ces démarches, bien qu'intéressantes, ont une portée limitée.
En effet, elles ne permettent d'étudier que les vies effectivement réalisées d'un système et non l'ensemble de ses vies possibles pas plus que l'ensemble des besoins non satisfaits par ce système.
Enfin, précisons que nos exemples ne portent pour l'instant que sur des traces produites par un système opérationnel dérivé d'une spécification terminale ne comportant que des objets de bas niveau.

Cela dit, nous pouvons évoquer certains points susceptibles à la fois d'augmenter l'intérêt d'une analyse de trace et de simplifier encore l'écriture des règles d'analyse.

- Une référence temporelle associée à chaque événement élémentaire permettrait notamment de suivre les temps de réponses des services offerts par un ou plusieurs objets.

- L'introduction dans la trace du contexte complet (conditions et instantiations) des événements rendrait possible:

  le suivi des consultations de la Base de Données de manière à donner des indications sur son évolution et son éventuelle optimisation;

  l'étude des conditions favorables (ou défavorables) au déclenchement de certains événements et les événements effectivement déclenchés par ces conditions;

  la production de diagrammes pondérés des interactions induites pour évaluer l'architecture de l'application.

- Un nouveau champ précisant, dans la trace, le nom de l'action appelante éventuelle, bien que non essentiel, serait de nature à simplifier certaines règles Russel (voir Exemple 1).

- Le support de structures de données plus riches dans le langage Russel serait également de nature à simplifier l'écriture des règles Russel, particulièrement dans le domaine des mesures quantitatives (voir exemples 7 et 8).

- Enfin, une prise en compte plus explicite de la notion d'événement (notamment en leur donnant un nom) dans le modèle Oblog et dans la trace permettrait d'éviter des ambiguïtés dans les mesures quantitatives (notamment dans le cas où plusieurs événements peuvent déclencher une même action à partir d'une même situation initiale et aboutissant à une même situation finale). Nous reviendrons sur ce point dans la section 6.2 de ce mémoire (Améliorations, extensions et nouvelles perspectives).

Partie 6 : CONCLUSION ET EXTENSIONS

6.1 Conclusion

- L'objectif concret de ce mémoire consistait à dégager à la fois l'intérêt et la faisabilité de l'analyse, grâce à l'outil Oblog, de la trace d'un système dérivé d'une spécification Oblog. Nous estimons avoir raisonnablement atteint cet objectif.

En effet, nous avons construit sans difficultés un programme Format Adaptator capable de traduire toute trace Oblog, sans perte d'information[5], dans une trace de format normalisé NADF.
L'équivalence informationnelle des deux représentations pourrait d'ailleurs être illustrée, à défaut d'être démontrée, en reconstituant, grâce au langage Russel, une trace Oblog dans son format natif, à partir de sa traduction NADF.
Une première évaluation de la trace Oblog nous a conduit à proposer un enrichissement de son contenu, indispensable pour isoler le comportement des instances d'objets du système opérationnel Oblog. Cet enrichissement s'est avéré techniquement possible.
Sous cette forme enrichie, la trace Oblog, bien que perfectible, se révèle tout à fait satisfaisante pour décrire la vie d'un système opérationnel Oblog.

Ensuite, sans être exhaustif, nous avons éclairé l'intérêt d'une analyse de trace Oblog dans des domaines variés: la mise au point et le contrôle de la Base de Données; le suivi de l'utilisation réelle de l'application; le support au développement, aux tests et à la maintenance de l'application; les mesures quantitatives du comportement d'un système.
A cette occasion, nous avons proposé, pour chacun de ces domaines, des pistes, des conseils et/ou des suggestions techniques pour guider la démarche d'analyse.
Remarquons que notre raisonnement est, dans ce cadre, loin d'être figé. Il constitue seulement un point de départ à la mise au point de démarches organisées et formalisées d'analyse de trace Oblog dans un ou plusieurs domaines d'intérêt.

Enfin, à la fois la puissance d'ASAX comme outil de support à l'analyse de trace et la variété des domaines d'intérêt d'une telle analyse, ont été illustrées par des exemples de règles Russel construites pour interroger la trace d'une application Oblog typique.
Une deuxième évaluation de la trace nous a amené à formuler des propositions d'enrichissement de la trace de manière à accentuer l'intérêt de l'analyse mais aussi à encore simplifier les requêtes Russel. Pour ce dernier point, il s'avère également qu'une évolution du langage Russel pour supporter la définition de structures de données complexes serait décisive.

---

[5] Ceci constitue une particularité du Format Adaptator de trace Oblog.

- D'autre part, nous croyons, au-delà de l'objectif concret, avoir contribué à dégager l'intérêt et la faisabilité de l'analyse, grâce à ASAX, de toute trace produite, plus généralement, par un système opérationnel bâti sur un modèle d'objets actifs communiquant par événements partagés.

En effet,

(i) le modèle Oblog est particulièrement représentatif de cette classe de modèles;

(ii) la trace Oblog est très proche de ce que l'on pourrait appeler un « modèle logique de description de la vie d'un système d'objets actifs communiquant par événements partagés »;

(iii) les qualités d'universalité, de puissance et d'efficacité d'ASAX nous permettent de croire qu'il peut supporter l'analyse de la trace d'un système informatique quelconque;

(iv) les domaines d'intérêt susceptibles de guider une analyse de trace Oblog et les démarches et techniques d'analyse associées sont, à notre sens, « réutilisables » globalement pour analyser le comportement d'un système d'objets actifs communiquant par événements partagés.

- Remarquons enfin que l'investissement que nous avons consacré à ce travail s'avérerait bien peu productif si le modèle d'objets actifs communiquant par événements partagés auquel nous nous sommes particulièrement intéressé, au travers du modèle Oblog, ne présentait que peu d'intérêt pour l'informaticien.
Nous sommes toutefois convaincus que ce modèle connaîtra un grand succès.

En effet, souple et expressif, il peut être utilisé dans toutes les étapes du cycle de développement. Unificateur, il intègre les vues données/traitements/comportements d'un Système d'Information.
La stratégie de développement qui lui est associée propose de construire un Système d'Information en raffinant progressivement la spécification sans discontinuités de langage de manière à augmenter la productivité du processus de développement d'une application.

L'approche théorique Oblog est particulièrement représentative de cette démarche de conception.
L'outil Oblog Case qui supporte cette approche offre des avantages indéniables: notamment, il permet l'édition, le maintien et la documentation de spécifications essentiellement diagrammatiques et il permet de générer un système opérationnel à partir des spécifications.
Certaines limitations et restrictions, par rapport au modèle théorique Oblog, ne lui permettent toutefois pas encore, à ce jour, de supporter les étapes hautes de spécifications, ni le prototypage de spécifications intermédiaires.

Rien n'interdit de penser que l'expérience acquise dans ce travail ne puisse être réutilisée pour analyser la trace d'un prototype construit grâce à une version ultérieure plus puissante de l'outil Oblog Case.

6.2 Améliorations, extensions et nouvelles perspectives

● Ce travail doit avant tout être considéré comme une expérience.
Il doit encore progresser sur le plan du formalisme, de la généricité et de l'organisation des démarches d'analyse de trace.
Dans ce sens, il doit être considéré plutôt comme un point de départ que comme un raisonnement figé.

● D'autre part, certaines améliorations et extensions sont d'ores et déjà envisageables:

(i) Le format NADF, présenté comme universel, apparaît toutefois propriétaire à certains égards. En effet, les longueurs de champs de données et les longueurs d'enregistrements sont dépendantes du format d'entiers de la machine qui exécute le programme Format Adaptator. Les conventions d'alignement sont également propriétaires.
Ceci peut poser problème lorsque l'évaluateur de règles Russel tourne sur une machine distincte de celle qui exécute le Format Adaptator.
Il nous semble possible pour gagner en universalité, de construire des programmes Format Adaptators produisant des fichiers traduits dans un format réellement portable (que nous pourrions appeler PNADF), débarrassé des contraintes d'alignement et où les longueurs s'exprimeraient dans un format d'entier universel (par exemple celui utilisé par la couche Présentation du modèle OSI).
Un programme paramètrable appelé Collector, front-end de l'outil ASAX, effectuerait alors la traduction de la trace PNADF dans un format NADF propre à l'environnement d'évaluation.

(ii) L'analyse on-line de la trace d'une application Oblog devrait être également envisagée, tenant également compte du fait que traduction et évaluation peuvent s'exécuter sur des plates-formes distinctes.

(iii) La trace Oblog peut, à notre sens, être encore enrichie sans beaucoup d'investissement pour intégrer deux champs nouveaux: le nom de l'action appelante pour les éventuelles actions appelées et une référence temporelle pour tous les événements de la trace. Ceci afin à la fois de simplifier les règles d'analyse et de permettre des analyses de temps de réponse.

(iv) Une étude similaire à celle menée dans ce travail pourrait être appliquée à l'analyse de traces produites par un système opérationnel dérivé d'un autre modèle d'objets actifs communiquant par événements partagés de manière à élargir le champ d'expérience, favoriser les comparaisons et préciser la portée de la démarche.

● Enfin, de nouvelles perspectives et extensions peuvent être éclairées parallèlement à l'évolution des outils respectifs et de la trace.

- Concernant ASAX:

(i) Les possibilités d'analyse d'une trace seront très largement développées lorsque le langage Russel supportera des types de données complexes. Nous pensons particulièrement à l'élaboration des mesures quantitatives réalisables à partir d'une analyse de trace.

(ii) Les auteurs du projet ASAX envisagent de développer une interface conviviale permettant à l'utilisateur d'exprimer des requêtes de haut niveau sans posséder d'expertise Russel. Cette interface serait bien entendu utilisable pour l'analyse de la trace d'un système d'objets actifs communiquant par événements partagés.

(iii) Ces auteurs envisagent également de faire évoluer l'outil pour permettre, dans le domaine de la sécurité des systèmes d'exploitation, à l'officier de sécurité de formuler des requêtes en termes des composants abstraits d'un modèle logique de sécurité.
Pourquoi, dès lors, ne pas envisager dans l'avenir, le support par ASAX de requêtes adressées à un modèle logique d'exécution d'un système d'objets actifs communiquant par événements partagés.

(iv) Enfin, le projet ASAX prévoit la construction d'une interface aidant l'utilisateur à décrire la structure d'une trace native de manière à générer automatiquement le Format Adaptator associé à ce format de trace.

- Concernant Oblog Case:

(i) En évolution constante, cet outil vient notamment d'intégrer dans sa version 1.2 les primitives sémantiques importantes que sont l'association et la spécialisation.
Nous sommes persuadés qu'il évoluera encore:

- en supportant le constructeur de réification et en favorisant le prototypage;
- en soulevant les actuelles limites et restrictions pesant sur les effets d'action et les objets persistants.

Nous serons alors en mesure d'analyser, à la fois la trace d'un système opérationnel terminal, mais aussi la trace d'un prototype correspondant à un niveau intermédiaire de raffinement de spécifications.

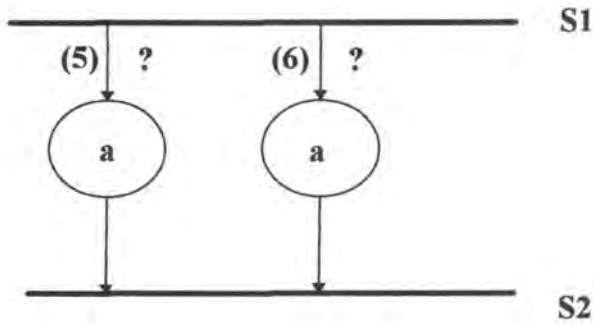
(ii) Le modèle Oblog gagnerait en clarté et en puissance, en dégageant explicitement le concept actuellement dilué d'événement. La trace, enrichie des noms d'événements en deviendrait également plus intéressante.

Illustrons sur un exemple l'intérêt du concept.

Soit a une action de la classe d'objet X
b une action de la classe d'objet Y

La partie de spécification suivante:

? (5) Cond1
? (6) not Cond1



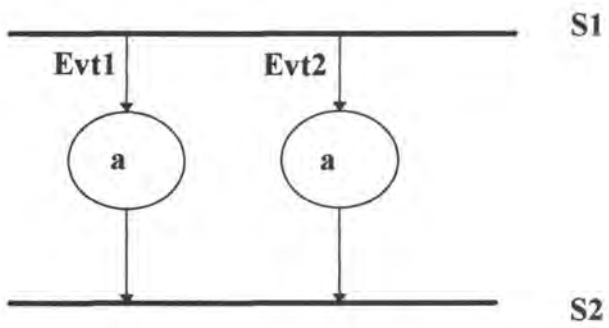
avec

X.a calls Y.b
Condition: Cond1

pourrait être mieux exprimée de la manière suivante:

Evt1
Condition: Cond1

Evt2
Condition: not Cond1



avec Evt1 calls Y.Evt3

- Concernant la trace Oblog:

(i) Bien que le contexte d'un événement élémentaire de la trace se retrouve en partie dans les valeurs des paramètres de l'action déclenchée par cet événement, il serait souhaitable toutefois de disposer également à la fois du texte des conditions et instantiations attachées à l'événement et de l'évaluation de leurs composants.
Ceci s'avère notamment indispensable pour étudier les interactions induites (queries) entre objets au travers de l'analyse de la trace.

(ii) Il nous semble également intéressant d'étudier comment les relations d'association et de spécification (et ultérieurement de réification) pourraient être traduites dans la trace et d'évaluer le parti que l'on pourrait en tirer dans une analyse de trace.

(iii) Finalement, pourquoi ne pas envisager l'idée que le système opérationnel Oblog étudié produise une trace directement en format NADF (ou PNADF) et que l'analyse de cette trace soit réalisée grâce à l'outil ASAX au travers d'une interface conviviale.

Bibliographie

[ASAX1]
Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji and Isabelle Mathieu
Preliminary report on Advanced Security Audit Trail Analysis on uniX
Technical Report,Institut d'Informatique, university of Namur, March 1993

[ASAX2]
Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji and Isabelle Mathieu
ASAX: Software Architecture and Rule-Based Language for Universal Audit Trail Analysis
In Proceedings of the third European Symposium on Research in Security
(ESORICS 92)
Lectures Notes in Computer Science, Toulouse, november 1992
Springer-Verslag

[ASAX3]
Naji Habra, Baudouin Le Charlier and Abdelaziz Mounji
ASAX: Specification of some extensions to the Analyzer
Technical Report, Institut d'Informatique, university of Namur, September 1994

[ASAX4]
Abdelaziz Mounji
User Guide for Implementig NADF Adaptors
Technical Report, Institut d'Informatique, university of Namur, January 1995

[Cowlishaw1]
Michael F. Cowlishaw
The REXX Language: A Practical Approach to Programming
Prentice Hall, Englewood Cliffs, 1990

[Denning1]
D.E. Denning
An Intrusion-Detection Model
IEEE Transactions on Software Engineering, Vol. 13, No 2, February 1987

[DOD1]
The Orange Book
Department of Defense, NCSC, National Computer Security Centre
DOD 5200.28-STD, December 1985

[IBM1]
IBM Systems Application Architecture. Common Programming Interface.
Procedures Language Level 2 Reference
IBM Publication: SC24-5549-0

[Meyer1]
Bertrand Meyer, Claude Baudouin
Méthodes de Programmation (troisième édition)
Editions Eyrolles, 1984

[Oblog1]
OBLOG Case V1.0 - User's Guide
ESDI SA, Lisbon, April 1993

[Parnas1]
D.L. Parnas
On the Criteria to be Used in Decomposing Systems into Modules
Communications of the ACM, Vol 15, No 12, December 1972

[Rolland1]
Colette Rolland et Corinne Cauvet
Modélisation conceptuelle orientée Objet
INRIA, BD3, Lyon, 1991

[Sécurité1]
- Naji Habra, Baudouin Le Charlier, Abdelaziz Mounji and Denis Zampunieris
Distributed Audit Trail Analysis. In Proceedings of the Internet Society Symposium on Network and Distributed System Security (ISOC 95)
San Diego, California, February 1995
IEEE
- Baudouin Le Charlier, Abdelaziz Mounji and Morton Swimmer
Dynamic detection and classification of computer viruses using general behaviour patterns
Technical Report, Institut d'Informatique, university of Namur, July 1995
- Haulotte V.
Etude des virus informatiques et utilisation d'un langage d'analyse d'audit trail pour leur détection
Mémoire de fin d'études, Institut d'Informatique, université de Namur, juin 1993
- Guedira Abdelmounaim
Analyse des fichiers de sécurité sur réseau de stations SUN
Mémoire de fin d'études, Institut d'Informatique, université de Namur, 1993

[Sernadas1]
Amilcar Sernadas, José Felix Costa and Cristina Sernadas
Object Specification Through Diagrams: OBLOG Approach
INESC, Lisbon, 1994

[Sernadas2]
Amilcar Sernadas, José Felix Costa and Cristina Sernadas
Formal Specifications
INESC-DMIST, June 1992

ANNEXES

A) Le langage REXX

Le langage REXX a été spécifié par M.F. Cowlishaw des laboratoires IBM UK [Cowlishaw1].
REXX est typiquement un langage interprété, qui s'est développé initialement dans l'environnement VM (Virtual Machine) IBM. Langage procédural, il a été conçu pour être plus proche de l'humain que les langages classiques de troisième génération: pas de déclarations de variables, évaluation contextuelle des opérations, pas de pointeurs ni de tableaux, grande souplesse de debugging, richesse des opérateurs de transformations de données, ...

Son succès auprès des utilisateurs a conduit IBM à l'intégrer comme langage clé dans sa Systems Application Architecture [IBM1]. Actuellement, des implémentations de REXX sont disponibles dans une grande diversité d'environnements: MVS, VM, OS/2, AS/400, DOS, UNIX, VMS,... Dans beaucoup de ces environnements sont également disponibles des compilateurs REXX.
La spécification de M.F. Cowlishaw est la base acceptée de la future définition standard du langage REXX ANSI X3J18.

Dans le cadre de ce travail, nous avons choisi d'écrire le programme Format Adaptator de trace Oblog dans ce langage, à la fois pour sa puissance dans le parsing des symboles de la trace mais aussi pour son expressivité et sa concision.

B) Eléments de REXX pour la lecture du programme Format Adaptator (exemples)

- **Exemples de termes**

'C''est le beau temps'	constante alphanumérique
3.14159	constante numérique
mois	variable
temps.mois	variable composée

- **Exemples d'expressions**

mois	variable		
j + 7	addition		
'étouf' 'fant'	concaténation 1	⟹	'étouf fant'
'étouf' || 'fant'	concaténation 2	⟹	'étouffant'
47 // 13	reste div. entiére	⟹	8

j > 2	comparaison	
(j > 2) | (a ^= b)	ou logique	
(j > 2) & (a ^= b)	et logique	
Length('étouffant')	fonction	==> 9

- **Instruction d'assignation**

```
mois = 'Juillet'
temps.mois = 'étouffant'
```

- **Instruction say**

say mois	affiche la valeur de la variable mois
say 'Résultat' 3 + 4	affiche : Résultat 7

- **Utilisation des variables composées**

Les variables composées fournissent l'équivalent des tableaux.

Par exemple, le programme:

```
m.1 = 'Napoléon'
m.2 = 'Toto'
m.3 = 'est plus fort que'
say m.1 m.2 m.3
```

affiche Napoléon est plus fort que Toto

Par exemple, le programme:

```
temps.Juillet = 'étouffant'
temps.Aout = 'lourd'
mois = 'Aout'
say temps.mois
```

affiche lourd

- **Instructions composées, répétitives et conditionnelles**

Ces instructions ont une syntaxe relativement classique. Nous n'en donnerons pas d'illustration, si ce n'est dans le programme Format Adaptator.

- **Instruction queue**

  queue expression

  L'exécution de cette instruction provoque l'évaluation de *expression*. La valeur résultante est enfilée [Meyer1] dans une File « système » de type FIFO , appelée external data queue.

- **Instruction exit**

  L'instruction *exit* provoque la fin de l'exécution du programme

- **Instruction parse**

  Cette instruction est l'une des plus puissantes du REXX. Nous n'illustrerons toutefois que les éléments de son fonctionnement utiles à la compréhension du programme Format Adaptor.

  parse source_identification template

  L'instruction parse divise un string source en composants et assigne ces composants à des variables en respectant les directives d'un patron template.

  source_identification
  indique l'origine du string source à parser. Il peut notamment être un des mots clés suivants:

  ARG
  dans ce cas, le string source est l'argument de la procédure ou de la fonction dans laquelle est exécutée l'instruction parse;

  PULL
  dans ce cas, le string source est obtenu en extrayant le premier élément (FIFO) de la File Système (external data queue); si cette file est vide, un prompt console invite l'utilisateur à introduire le string source;

  VAR variable_name
  dans ce cas, le string source est la valeur de la variable *variable_name*.

  template
  spécifie comment découper le string source et comment assigner ses composants à des variables. Il est composé d'une succession de noms de variables et de délimiteurs. Ces délimiteurs peuvent notamment être des espaces, des littéraux ou des points.

  Illustrons la méthode de parsing sur différents exemples.

(i) **parse pull line**

le premier élément de la File système est extrait et assigné à la variable line

Soit v = 'This is the text wich , I think, is scanned'
u = 'short text'

(ii) **parse var v v1 v2 v3**

est équivalent à

v1 = 'This'
v2 = 'is'
v3 = 'the text wich , I think, is scanned'

(iii) **parse var u v1 v2 v3**

est équivalent à

v1 = 'short'
v2 = 'text'
v3 = ''

(iv) **parse var v v1 . v3**

est équivalent à

v1 = 'This'
v3 = 'the text wich , I think, is scanned'

(v) **parse var v v1 ',' v2**

est équivalent à

v1 = 'This is the text wich '
v2 = ' I think, is scanned'

(vi) **parse var v v1 '(' v2**

est équivalent à

v1 = 'This is the text wich , I think, is scanned'
v2 = ''

Soit v = 'This is the text wich , I think, is scanned'

(vii) **parse var v v1 v2 v3 ',' v4 ',' v5 .**

est équivalent à

v1 = 'This'
v2 = 'is'
v3 = 'the text wich '
v4 = ' I think'
v5 = ' is'

Soit v =

'Object COUNTER (3325104) Action create Situation started Caller ROOT (3319488)'

(vii) **parse var v . c '(' i ')' . a . s . cc '(' ci ')'**

est équivalent à

c = 'COUNTER '
i = '3325104'
a = 'create'
s = 'started'
cc = 'ROOT '
ci = '3319488'

Soit v =

'Event Attribute myMaster = OBJECT NAME: ROOT OBJECT ADDRESS:3319488'

(viii) **parse var v mot1 mot2 mot3 mot4 reste**

est équivalent à

mot1 = 'Event'
mot2 = 'Attribute'
mot3 = 'myMaster'
mot4 = '='
reste = 'OBJECT NAME: ROOT OBJECT ADDRESS:3319488'

- **Schéma des appels de procédures et des appels de fonctions**

```
...
...
call procedure_name argument1
...
x = function_name(argument2)
...
...

procedure_name: procedure expose list_variables1
parse arg argt1
...
...
return
...
function_name: procedure expose list_variables2
parse arg argt2
...
...
return resultat
...
```

Remarque:

Toutes les variables sont locales dans les fonctions et procédures sauf les éventuelles variables « exposées » (list_variables1/2) qui peuvent être lues et modifiées dans la procédure/fonction appelante par la procédure/fonction appelée.
Par exemple,

StoreAction: procedure **expose N identifier.**

indique que la procédure StoreAction peut lire et écrire la variable **N** et toutes les variables composées dont le nom est préfixé par **identifier.** dans la procédure appelante.

- **quelques fonctions built-in (exemples : outputs = function_name(inputs))**

rc = **Stream**(N,'Command','OPEN CREATE')

Stream crée un nouveau fichier dont le nom externe est contenu dans la variable N et l'ouvre en écriture.
rc est le statut de l'opération.

rc = **Stream**(N,'Command','WRITE', nadf_first_record)

Stream écrit le contenu de la variable nadf_first_record en appendice du fichier dont le nom externe est contenu dans la variable N.
rc reçoit le statut de l'opération.

x = **Strip**(' a b c ')

Strip supprime les éventuels blancs en préfixe et en suffixe de son argument et retourne la valeur ainsi obtenue dans x (x = 'a b c')

result = **D2C**(129 , 4)

D2C retourne dans result un string de 4 caractères correspondant à la représentation hexadécimale 00000081 (129 en binaire)

x = **Left**('abc def', 5)

Left retourne dans x les 5 premiers caractères de 'abc def' à partir de la gauche (x = 'abc d')
remarque : Left('abc def', 9) = 'abc def '

x = **Right**('abc def', 5)

Right retourne dans x les 5 derniers caractères de 'abc def' à partir de la droite (x = 'c def')
remarque : Right('abc def', 9) = ' abc def'

n = **Queued**()

Queued retourne dans n le nombre d'éléments présents dans la File système (external data queue)

x = **Word**('I think, therefore I am.' , 2)

Word retourne dans x le deuxième mot délimité par des blancs du premier argument (x = 'think,')
remarque: Word(' ') = ''

n = **Words**('I think, therefore I am.')

Words retourne dans n le nombre de mots délimités par des blancs dans le premier argument (n = 5)
remarque: Words(' ') = 0

x = **Subword**('Les bijoux de la Castafiore',2,3)

Subword retourne dans x le sous-string commençant au deuxième mot du premier argument et qui contient au moins trois mots consécutifs
(x = 'bijoux de la')
remarque: Subword('short text ',2,3) = 'text'

x = **Subword**('Les bijoux de la Castafiore',2)

Subword retourne dans x le sous-string commençant au deuxième mot du premier argument
(x = 'bijoux de la Castafiore')
remarque: Subword('short text ',5) = ''

x = **Delword**('Les bijoux de la Castafiore',2,3)

Delword retourne dans x le string obtenu en retirant du premier argument le sous-string commençant au deuxième mot et qui contient au moins trois mots consécutifs.
(x = 'Les Castafiore')

n = **Length**('12345678')

Length retourne dans n la longueur du string argument (n = 8)

x = **Translate**('abCD12')

Translate retourne dans x son argument traduit en majuscules (x = 'ABCD12')

x = **Linein**(N)

Linein lit une ligne du fichier texte dont le nom externe est contenu dans N et retourne cette ligne dans x

n = **Lines**(N)

Lines retourne dans n : 0 si le pointeur associé au fichier texte dont le nom externe est contenu dans N pointe vers la fin de ce fichier , sinon 1.

- **remarque concernant les types de données**

Le string de caractères est le seul type implicite en REXX. Ceci n'exclut pas les traitements numériques par conversion contextuelle.

Exemple:

```
nombre = 5678                    (ou bien: nombre = '5678')
first = Left( nombre , 1 )
say  first                       ==> affiche 5
produit = nombre + first
say produit                      ==> affiche 5683
bizarre = 'abc' - produit        ==> erreur !!!
```

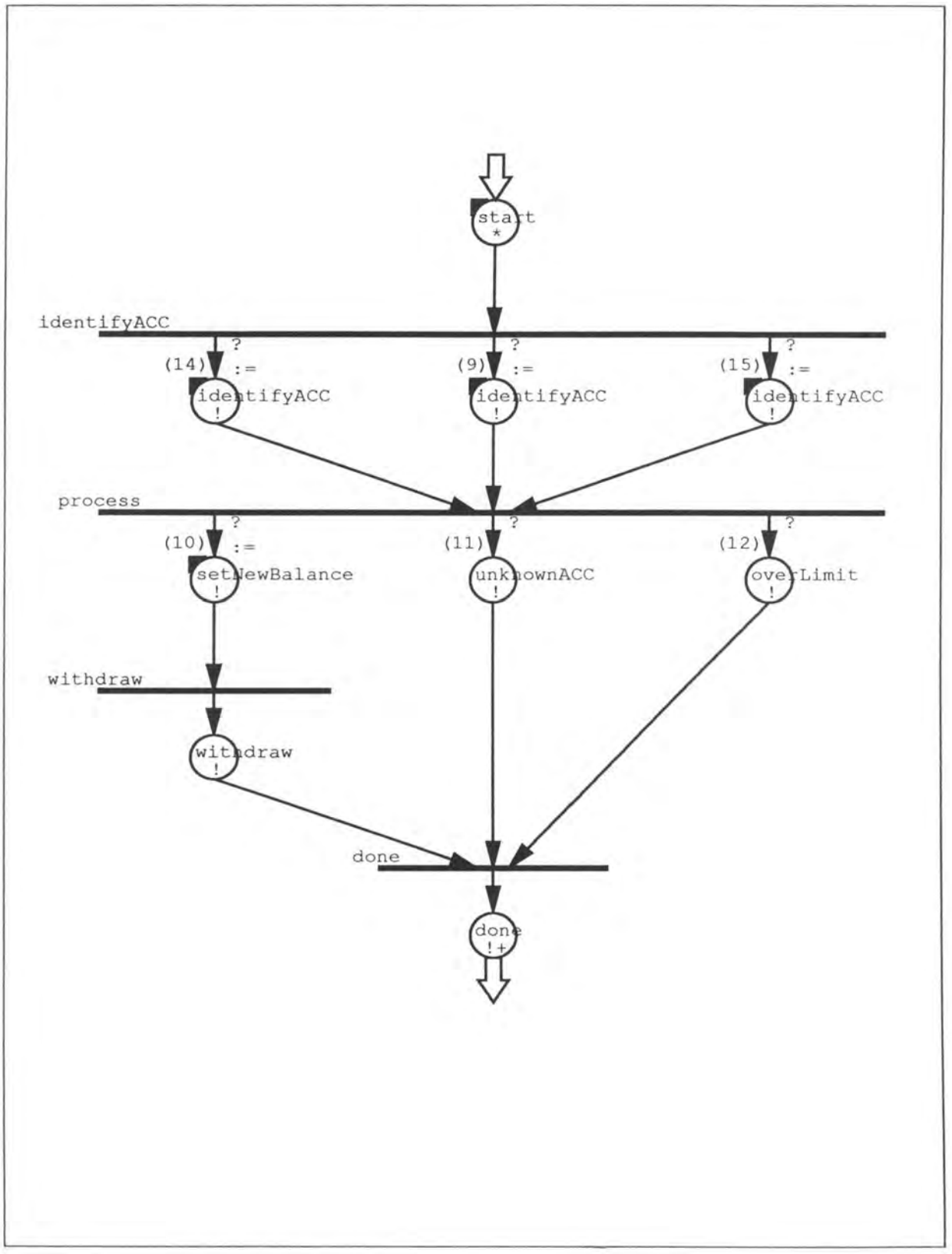
C) Quelques éléments complémentaires de la spécification de Monde Bancaire

Les trois pages suivantes présentent les GTE respectifs des classes d'objets :

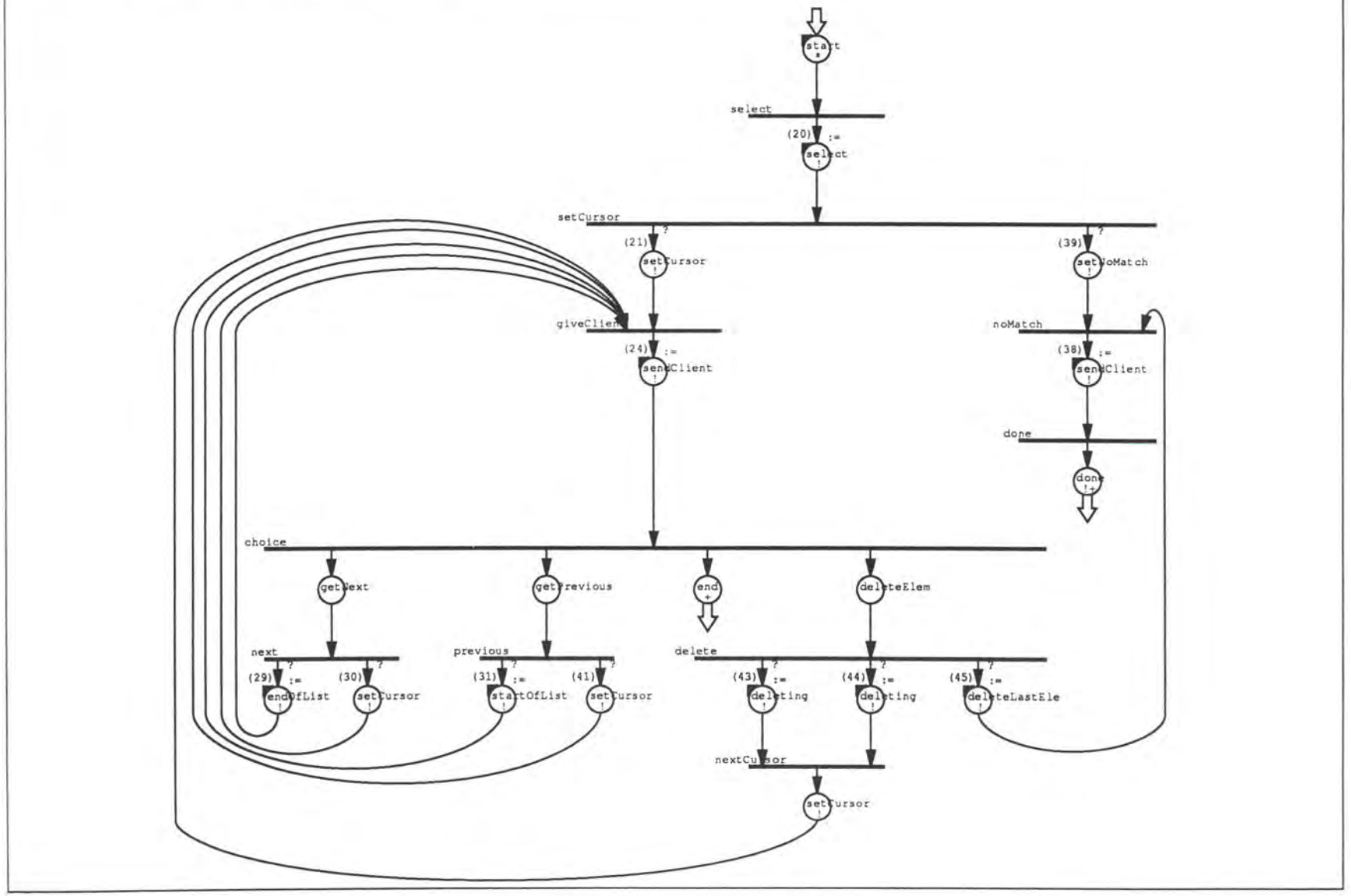
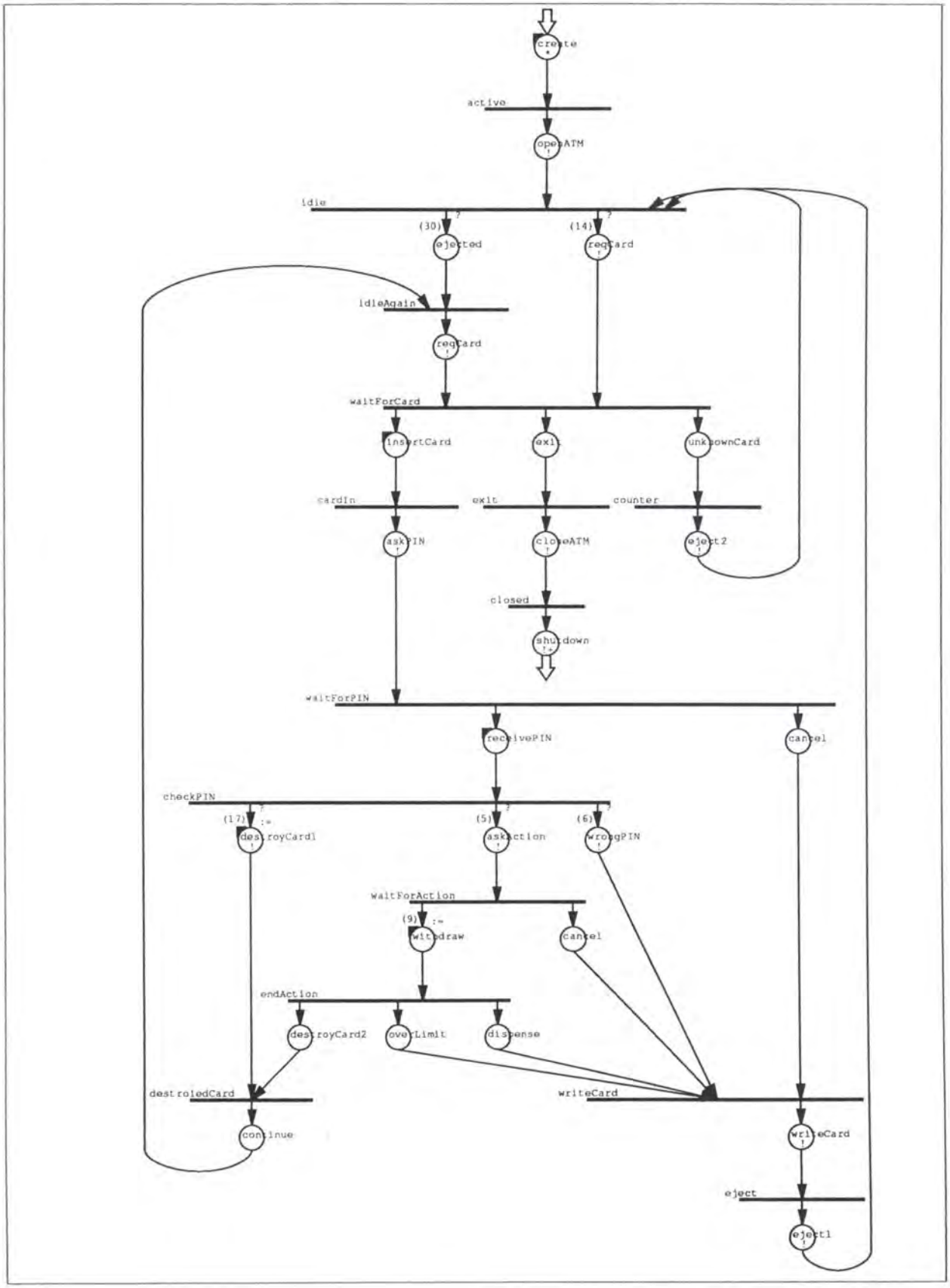
- WITHDRAW
- SELECT_CLIENT
- ATM_D

tels qu'ils apparaissent dans la spécification de l'application Monde Bancaire [Oblog1].

Behavior Diagram: ACCOUNT\WITHDRAW



Behavior Diagram: CLIENT\SELECT_CLIENT



Behavior Diagram: ATM_X\ATM_D