Simulation de spécifications d'applications de gestion de réseaux de télécommunications

Alain Cougoulic

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N. attribué par la bibliothèque :

THÈSE

pour obtenir le grade de

DOCTEUR DE l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE
Discipline : Informatique, Communication et Système

présentée et soutenue publiquement

par

Alain COUGOULIC

le 28 juin 1999

Titre :

Simulation de spécifications d'applications de gestion de réseaux de télécommunications

Directeur de thèse :

M. Paul Jacquet

JURY

M.	Jacques Mossière	,Président
M.	Daniel Herman	,Rapporteur
M.	André Schaff	,Rapporteur
M.	Yann Rouzaud	,Examinateur
M.	Louis-Olivier Donzelle	,Examinateur
M.	Paul Jacquet	,Directeur de thèse

Table des matières

Table des figures

Chapitre 1

INTRODUCTION

Les applications de gestion de réseaux de télécommunications suivent la même évolution que toute application informatique : elles sont de plus en plus complexes, tout en offrant plus de services. La complexité de ces applications est aussi une conséquence de l'évolution des réseaux et de l'émergence des technologies de la répartition. Celles-ci permettent d'encapsuler les *hétérogénéités* des applications. La tendance actuelle est de traiter l'encapsulation comme un ensemble d'objets communicants. On utilise aussi le terme de *composants*, mais plus dans le domaine de la définition d'architecture des applications.

Au cours de la réalisation d'une application répartie, une des premières phases est la spécification des objets répartis de l'application. On utilise des langages de spécifications pour réaliser cette phase. Ils décrivent formellement (ou semi-formellement) les objets et leurs services.

La nécessité de sûreté de certaines applications impose leur validation au plus tôt dans le processus de réalisation. Dans cette thèse, nous définissons une aide à la validation des applications par simulation de leurs spécifications. Nous montrons qu'il est important que ces dernières soient décrites le plus formellement possible.

1.1 La validation de la spécification d'une application

La validation des applications peut être effectuée à plusieurs niveaux au cours du processus de réalisation d'une application (cahier des charges, spécification, conception, implantation, etc.). La validation d'une application peut être réalisée par simulation. La simulation s'applique aux spécifications, aux implantations, etc.

Dans cette thèse, nous nous intéressons à la validation de la spécification des applications, c'est-à-dire de vérifier que les besoins définis dans les cahiers des charges sont respectés, voire de mettre en évidence des incohérences de ces derniers (une fonction de l'application définie dans le cahier des charges n'est pas réalisable).

La spécification d'une application peut s'effectuer en plusieurs phases, comme par exemples la spécification fonctionnelle de l'application, de son architecture, etc. Les fonctions et les composants d'architectures sont des traductions des besoins. La validation revient à vérifier que ces fonctions et composants sont conformes aux attentes des utilisateurs et sont *réalisables*.

La tendance actuelle pour représenter des applications réparties est l'objet. Les langages de spécifications décrivent un objet à l'aide d'une classe. Une classe décrit un objet comme un ensemble de données (attributs) et de services (représentant les fonctions des applications). Actuellement, les environnements (ou plus récemment nommés *bus logiciels*[1]) permettant de traiter la répartition utilisent des langages représentant des interfaces d'objets (CORBA[2]/IDL[3] [COR93], GDMO[4] [GDM92], etc.) comme langages de spécifications. Plus généralement, dans le monde des télécommunications, d'autres types de langages de spécifications sont utilisés (graphiques, méthodes formelles, etc.). Nous présentons les divers types de langages de spécifications dans le chapitre 2.

Les fonctions d'une application peuvent être représentées par un ou plusieurs services d'objets, les composants par un ou plusieurs objets. La simulation de spécifications revient à instancier des classes d'objets et à exécuter les services des objets créés. Les résultats d'exécution permettent de vérifier que le *comportement* de l'application attendu est préservé. Dans la section suivante, nous présentons plus en détail le comportement d'une application.

1.2 Comportement d'une application

On regroupe sous le terme *comportement* l'ensemble des fonctions et caractéristiques d'une application. Dans le monde objet, le comportement est représenté par les objets et leurs interactions. On le classifie soit de :

- statique (invariant de l'application) ;
- dynamique (les modifications de l'application apportées par un service sont conformes aux fonctions attendues de l'application).

Les comportements statiques concernent les *états* des applications. Une application n'est cohérente que si son *état* courant préserve certaines valeurs caractéristiques de l'application. Un exemple d'école est celui du sac : un sac est vide ou contient des éléments mais ne peut avoir un contenu négatif. Notons que la cohérence des comportements statiques peut être assurée par un typage fort des valeurs caractéristiques de l'application.

1. technologie *middleware* [Ber93].
2. Common Object Request Broker Architecture
3. Interface Definition Language
4. Guidelines for the Definition of Managed Objetcs

Les comportements dynamiques d'une application sont représentés par les services des objets. Un service peut modifier l'*état* d'une application. Sur l'exemple du sac, le service d'ajout d'un élément modifie l'état du sac.

La spécification des comportements des applications dans le monde des télécommunications et des systèmes répartis n'est pas toujours définie dans les langages. Les comportements statiques sont souvent représentés par le typage des attributs des objets. Les comportements dynamiques ne sont pas toujours spécifiés par les langages de spécifications orientés objet (CORBA/IDL : pas de spécifications ; GDMO : utilisation de la langue naturelle).

Dans le chapitre 2, nous présentons des langages qui permettent de spécifier les comportements des applications. Ces langages ne sont pas très usités dans le monde industriel (nous en donnons les raisons dans le chapitre 2). De ce fait, nous proposons dans cette thèse une étude permettant d'intégrer une description formelle des comportements dans les langages de spécifications orientés objet.

De cette étude, nous définissons une aide à la simulation des applications qui permet de valider à partir de leurs spécifications les besoins. Cette aide n'est pas seulement fondée sur la formalisation des comportements, mais aussi sur une extension de cette dernière. En effet, nous montrons que la formalisation des comportements dynamiques ne permet pas toujours une exécution au plus haut niveau des ceux-ci.

1.3 Plan

Ce mémoire se compose de six chapitres :

- Le chapitre 2 présente les langages de spécifications d'applications de gestion utilisés dans le monde des télécommunications, et plus généralement les *modèles* de conception. Ce chapitre décrit un bilan de la représentation des comportements dans les langages de spécifications.

- Le chapitre 3 introduit la simulation d'application de gestion de réseaux de télécommunications. Nous montrons que la validation des spécifications des comportements dynamiques des applications nécessite un *traitement opérationnel* par la simulation.

- Le chapitre 4 présente tout d'abord l'extension *invariants* et *pré/post-conditions* des langages de spécifications orientés objet permettant d'intégrer des *langages d'assertions* spécifiant les comportements. Il présente une autre extension qui concerne les langages d'assertions. Elle représente une solution au *traitement opérationnel* des spécifications par la simulation.

- Le chapitre 5 présente un langage d'assertions fondé sur celui des prédicats du premier ordre et prenant en compte le monde objet. Il intègre l'extension *opérationnelle*

consacrée à la simulation. Ce chapitre introduit plus précisément la notion d'*état* dans un monde objet.

- Le chapitre 6 présente le traitement par la simulation des comportements définis à l'aide du langage d'assertions présenté dans le chapitre 5. Ce chapitre propose une sémantique opérationnelle du traitement des comportements.

- Le chapitre 7 présente QG^2S, le simulateur de spécifications Q-GDMO-GRM étendu à l'aide du langage d'assertions défini dans le chapitre 5. QG^2S intègre la sémantique opérationnelle présentée en chapitre 6.

- La conclusion présente les avantages de notre approche auprès du monde des télécommunications. Nous y présentons les perspectives de notre travail.

Chapitre 2

Spécification des applications de télécommunication

Les applications de gestion sont très largement normalisées dans le monde des télécommunications. Ces applications portent aussi bien sur la gestion des réseaux que la gestion des services de télécommunication. Un réseau de télécommunication est un réseau d'éléments connectés entre eux et s'échangeant des données. Des fonctions permettant de gérer des équipements sont nécessaires pour assurer les échanges. De plus, les réseaux étant le support des services *aux clients*, des fonctions de gestion assurant le bon déroulement des services sont aussi nécessaires.

Dans ce chapitre, nous présentons tout d'abord les modèles de gestion utilisés dans le monde des télécommunications permettant de standardiser la représentation des fonctions de gestion. Nous montrons aussi que les modèles de référence portant sur les systèmes répartis peuvent inclure les modèles de gestion des télécommunications. Nous présentons ces modèles en section 2.1.

Les modèles de gestion influencent la spécification des applications de gestion des réseaux de télécommunications, c'est-à-dire les concepts définis dans ces modèles doivent apparaître dans les spécifications des applications. Il existe divers langages de spécifications qui permettent de décrire une application selon les fonctions de gestion attendues des applications. En section 2.2, nous décrivons quelques types de langages de spécifications d'applications utilisés dans le monde des télécommunications. En section 2.3, nous effectuons une synthèse des paradigmes et concepts attendus d'un langage de spécifications et plus particulièrement pour les comportements des applications.

2.1 Modèles et spécifications des applications de gestion

Les applications de télécommunication permettent de fournir des services aux clients (qu'ils soient externes ou internes). Ces services utilisent des ressources situées sur des sites plus ou moins éloignés et hétérogènes. Les applications de télécommunication sont alors déployées sur des réseaux pouvant comporter des systèmes différents.

La diversité des systèmes rend plus complexe la mise en oeuvre d'une application. Cette diversité est tout d'abord apportée par des architectures hétérogènes (matériels divers, routeurs, commutateurs ; des protocoles de communications différents, X.25, Ethernet, TCP [1]/IP [2], etc.). Un besoin d'homogénéisation des données relatives aux équipements est nécessaire pour concevoir une application sur des systèmes importants. Par données, nous entendons les informations stockées par les équipements et qui transitent entre eux (par exemple, les numéros d'abonnés téléphoniques contenus dans les commutateurs locaux).

La gestion des données s'effectue par le biais de protocoles définis par des modèles de gestion réseaux tel que l'OSI (Open Systems Interconnection). Les applications de gestion des services sont basées sur des modèles de gestion de plus haut niveau (par exemple le modèle Gestionnaire/Agent).

Un besoin d'homogénéiser la gestion des réseaux et services de télécommunication est concrétisé par un modèle de gestion normalisé au sein de l'ITU (International Telecommunication Union) [3]. Nous allons dans un premier temps présenter en sous-section 2.1.1 ce modèle de gestion des télécommunications et son application en terme de protocoles.

L'émergence des systèmes répartis a fait naître des modèles de *gestion* (*de description*) des applications définies sur ces systèmes. Les opérateurs de télécommunication sont d'ailleurs très actifs dans la définition et la normalisation de ces modèles. En sous-section 2.1.2, nous présentons les modèles de référence de conception des applications réparties et leurs liens avec le monde des télécommunications.

2.1.1 Le modèle de gestion de télécommunications

L'ITU a standardisé la gestion des réseaux et services des télécommunications en quatre modèles distincts [ITU92b] :

- le modèle fonctionnel : il permet de définir des fonctions génériques traitant la gestion des fautes, la configuration des réseaux, la tarification des services, de la performance

1. Transmission Control Protocol
2. Internet Protocol
3. regroupement des normes ISO et CCITT du domaine des télécommunications.

des réseaux et de la sécurité. Ce modèle met en évidence des concepts génériques que l'on peut retrouver à chaque niveau (couche) des modèles de référence. Il reste à la charge des concepteurs des applications d'effectuer les choix d'implantation des fonctions génériques (quelles couches doivent supporter une fonction générique, etc.) ;

- le modèle organisationnel : il permet de définir les intervenants de l'application et leurs méthodes de transmissions et de collectes des informations. Ce modèle est en fait souvent appelé Client/Serveur [GO96] (d'un point de vue général) ou encore Gestionnaire/Agent [ITU97] (d'un point de vue des télécommunications). Le serveur (agent) gère les données et les transmet vers les clients (gestionnaires). Le client (gestionnaire) collecte les informations obtenues (ou parvenues) des serveurs (agents) et les transmet aux entités logicielles liées au client (gestionnaire) ;

- le modèle d'information : il permet la description des interfaces entre les serveurs et les ressources gérées par ces serveurs. Les interfaces permettent dans un premier temps de décrire les informations accessibles d'une ressource par les serveurs (pour une même ressource il peut exister n interfaces différentes pour m serveurs différents). Les interfaces sont aussi connues des clients (il faut fournir un système de nommage des interfaces et des objets manipulés dans un serveur) ;

- le modèle de communication : il permet de définir les protocoles de communications et l'architecture nécessaires à l'échange d'informations entre les clients et serveurs de l'application. Ce modèle comporte la définition des requêtes échangées entre clients et serveurs (un service d'une application peut être modélisé par un ensemble de requêtes échangées entre un client et des serveurs plus un traitement des résultats des requêtes au niveau du client). Les requêtes sont transmises au travers des sous-couches de protocoles des réseaux et doivent donc être compatibles avec les protocoles du réseau supportant l'application.

Les modèles de référence OSI et IAB (Internet Architecture Board) sont conformes aux quatre modèles *fonctionnel, organisationnel, information et communication* pour définir les applications de gestion au sein des réseaux. Ces quatre modèles sont supportés par les couches hautes des modèles OSI (*application, etc.*) et IAB (SNMP, Simple Network Management Protocol)[RM90]. Nous allons rapidement présenter les quatre modèles de gestion en terme de réalisations au sein des deux modèles de référence OSI et IAB.

Le modèle fonctionnel

Le modèle fonctionnel permet de mettre en évidence des fonctions génériques de gestion que doivent supporter les réseaux et donc les applications définies sur ces réseaux. L'OSI comporte des normes décrivant ces fonctions (nommées aussi SMF, *Systems Management Functions*). Les normes sont regroupées dans les séries de documents X.73x, X.74x et X.75x (la norme X.730 [ITU92d] décrit la SMF de gestion des *objets gérés*, la

norme X.733 [ITU92e] décrit la SMF de gestion des alarmes, la norme X.734 [ITU92f] décrit la SMF de gestion des événements, notifications, etc.). D'autres normes décrivent des SMF spécifiques aux réseaux.

Les SMF s'intègrent aussi dans les autres modèles (organisationnel, information et communication). Par exemple, la SMF de gestion des objets gérés [ITU92d] est décrite en prenant en compte la description des services et protocoles de communication du modèle de communication de l'OSI.

Le modèle IAB ne comporte pas de description de fonctions de gestion génériques significatives.

Le modèle organisationnel

Les modèles OSI et IAB ont retenu comme modèle organisationnel le modèle Gestionnaire/Agent [ITU97]. Nous présentons ce modèle dans la figure 2.1.

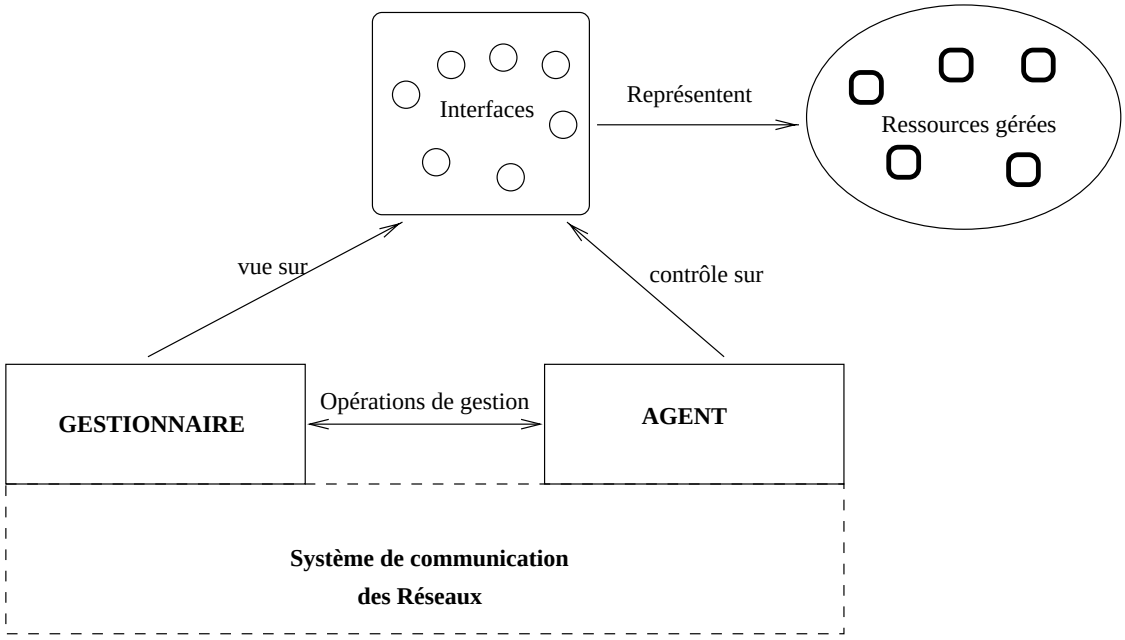


FIG. 2.1 – *Le modèle Gestionnaire/Agent*

La figure 2.1 montre une vue de l'organisation des acteurs d'une application de gestion. Une application est représentée par des interactions entre gestionnaires et agents. Des mécanismes de collecte des données (*informations*) entre gestionnaires et agents sont alors nécessaires. Dans le modèle de communication nous présentons comment les données sont collectées par le gestionnaire.

Le modèle d'information

Pour représenter les *objets gérés*, la norme du modèle OSI [GDM92] propose le langage GDMO qui permet de spécifier les interfaces des objets gérés. Le langage GDMO décrit sous forme de formulaires les interfaces. GDMO offre plusieurs types de formulaires : un formulaire pour décrire les attributs, un pour décrire un type de service, etc). Pour plus de détails sur la diversité des formulaires GDMO, une rapide présentation est effectuée dans l'annexe A.2.

Pour le modèle IAB, SNMP offre aussi une description des interfaces sous forme de formulaires [RM91]. SNMP n'offre qu'un type de formulaire d'interface (assimilable à une description d'une classe d'objet). Par rapport à GDMO, la réutilisation est moins fine. En effet, par exemple, un formulaire d'attribut peut être référencé dans différents formulaire d'objets gérés en GDMO. En SNMP, il faut redéfinir l'attribut à chaque référence.

Les langages d'interfaces permettent de représenter les attributs des objets et les services accessibles sur les objets. Les attributs représentent des valeurs plus ou moins complexes des ressources. GDMO et SNMP utilisent le standard ASN.1 (Abstract Syntax Notation One) [ASN88] pour typer les valeurs des attributs. ASN.1 permet de spécifier tout type de base (entiers de diverses longueurs, chaînes de caractères de divers formats, etc.), des séquences (tuples, structures), des ensembles et des unions. La nouvelle version d'ASN.1 [ASN94] permet de spécifier des objets. Cependant, seul GDMO prend en compte tous les types ASN.1. SNMP, pour des raisons de simplicité, n'utilise qu'une petite partie des types ASN.1.

Les interfaces spécifient les services sur les objets. Les services accessibles sur les objets gérés sont standardisés et suivent le modèle de communication de l'OSI ou de SNMP. La description d'un service est représentée par sa signature et son comportement. Dans les deux modèles de référence, les comportements sont spécifiés par la langue naturelle. Dans la figure 2.2, nous donnons un exemple de comportement d'un objet géré de classe *connectivity* ou d'un objet géré de classe *terminationPoint* du modèle d'information générique de réseau [ITU92c]. Notons ici la réutilisation du même comportement par deux classes d'objets différentes.

```
alarmSeverityAssignmentPointerPackageBehaviour BEHAVIOUR
    DEFINED AS
        "If the alarm severity assignment profile pointer is NULL,
then one of the following two choices applies when reporting alarms:
a) agent assigns the severity or b) the value 'indeterminate' is used.";
```

FIG. 2.2 – *Comportement en langage naturel.*

Le modèle de communication

Le modèle de communication permet de décrire les protocoles d'accès aux ressources au travers des services définis sur les objets gérés. Le modèle de communication de l'OSI est représenté par les normes CMIS[4]/CMIP[5] [ITU91a] [ITU91b]. Les services d'objets gérés existants sont les services de création et destruction, d'accès et de modification de valeurs d'attributs, des actions et des notifications (rapports d'événements envoyés par l'objet vers le ou les gestionnaires au travers de l'agent).

Au sien du processus de gestion de la communication des entités réseaux, la transmission des services CMIS s'appuie sur le protocole CMIP. Nous préconisons au lecteur de se référer aux documents [ITU95a], [ISO89] et [Col89] donnant une description complète du processus de gestion des services CMIS.

Les services des objets gérés dans l'approche SNMP se veulent simples et suffisants pour répondre au besoin de gestion des objets. La consultation, la modification et les notifications d'objets sont ces services. Le protocole de communication de SNMP est fondé sur les protocoles UDP[6]/IP[7].

Bilan

Dans cette sous-section, nous ne détaillons pas tous les protocoles nécessaires à la gestion dans les modèles OSI et IAB (le nommage des instances d'objets, les droits des gestionnaires et agents, etc.). Nous laissons au lecteur intéressé le soin de consulter les normes ITU relatives à l'OSI et les RFC de l'IETF (Internet Engeneering Task Force).

Les approches OSI et IAB du modèle de gestion des réseaux *fonctionnel, organisationnel, information, communication* sont très *protocolaires* (indépendantes des éléments et sites réseaux). L'évolution actuelle du monde informatique (l'internet, le matériel, etc.) permet de concevoir des systèmes et applications plus complexes et importants sur des architectures logicielles et matérielles hétérogènes. Par exemple, on peut se permettre de déplacer des parties d'applications d'un site vers un autre. La gestion *protocolaire* de tels systèmes *répartis* n'est pas toujours adéquate. Dans la sous-section 2.1.2, nous présentons les travaux actuels sur la description et la gestion des systèmes répartis.

2.1.2 Les modèles de gestion des systèmes répartis

Un standard de modélisation d'applications (systèmes) réparties est le modèle de référence ODP (Open Distributed Processing) [ODP95]. ODP fournit un vue globale de

4. Common Management Information Service
5. Common Management Information Protocol
6. User Datagram Protocol
7. Internet Protocol

la modélisation des applications réparties (une description des objectifs, des propriétés que doit préserver une application tout au long de sa description et de sa conception[8]). Nous allons tout d'abord décrire le modèle de référence ODP (RM-ODP).

Le modèle de référence ODP

Dans le modèle ODP, la notion d'application est remplaçée par la notion de système (ensemble d'applications coopérantes sur des réseaux complexes). On peut faire un parallèle entre le modèle ODP et les modèles de conception de logiciel connus (le modèle de la cascade, le modèle en spirale de Boehm, etc. [Som92]). Les phases de conception d'applications en ODP sont regroupés au sein de concepts architecturaux d'ODP définis en trois parties :

- un ensemble de *points de vue* décrivant une application ;
- un ensemble de *fonctions de gestion* (sécurité, accès aux objets, etc.) ;
- un ensemble de *transparences* sur l'application (accès aux objets, localisation des objets, persistance des objets, etc.).

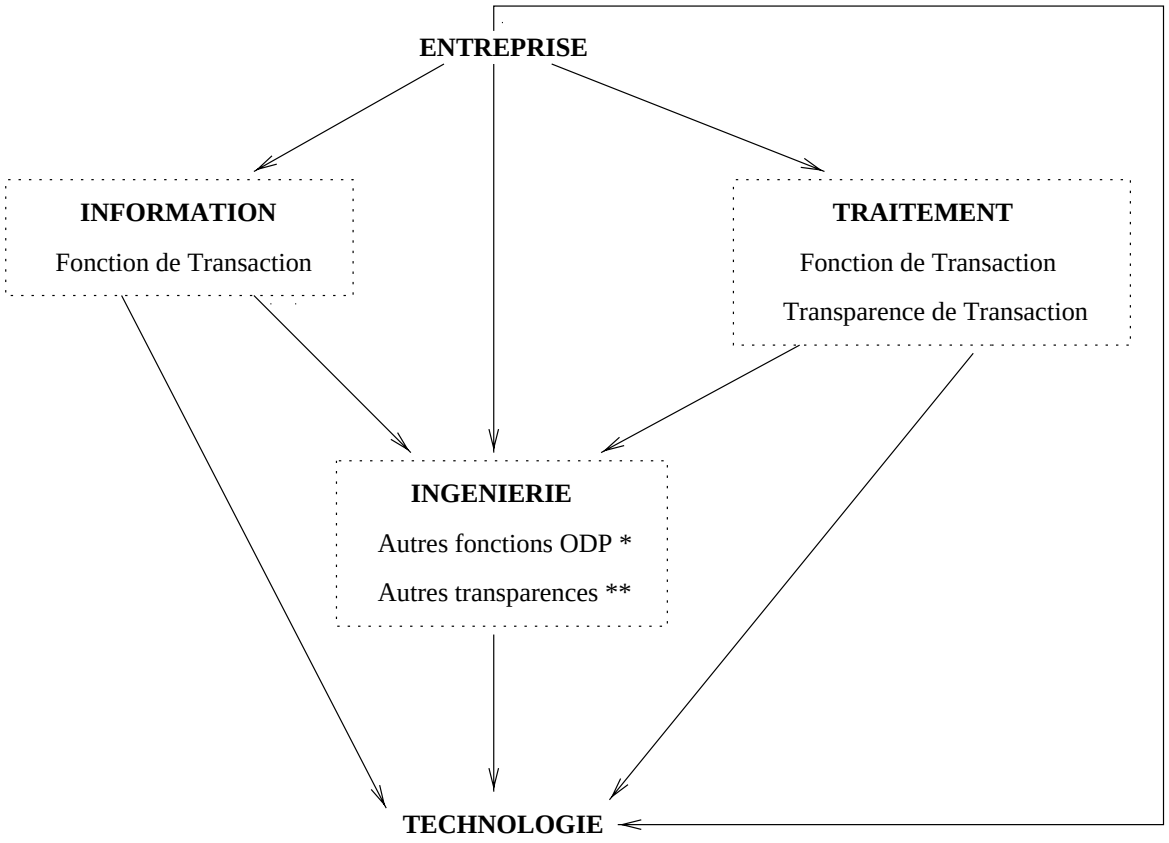
Les fonctions de gestion et les transparences sont intégrées aux points de vue. Les points de vue ODP sont :

- le point de vue entreprise qui permet de définir les objectifs, les rôles et les politiques d'un système ODP et de son environnement ;
- le point de vue information qui permet de définir l'information d'un système ODP et le traitement de cette information ;
- le point de vue traitement qui permet de donner une décomposition fonctionnelle d'un système ODP en objets interagissant par le biais d'interfaces ;
- le point de vue ingénierie qui décrit les mécanismes et fonctions nécessaires à la répartition des objets du système ;
- le point de vue technologie qui décrit les choix technologiques de réalisation et d'implantation du système.

La figure 2.3 montre les relations existantes entre les trois parties regroupant les concepts ODP. Le fait qu'il y ait des relations entre les points de vue ne consiste pas à créer un système en couches (cf. modèles OSI et IAB). Chaque point de vue est une

8. par abus de langage, on peut nommer cette description le *cahier des charges* de l'application comportant une *analyse des besoins* de l'application (performances, matériels, etc.).

abstraction du système spécifié et peut ensuite être présent dans les couches basses ou hautes des réseaux. Les fonctions de transparence et de gestion sont surtout présentes dans le point de vue ingénierie, ces fonctions traitant en fait de la répartition du système ODP. La transparence d'accès n'est pas citée dans la figure 2.3, car elle masque les structures de données et leurs invocations. Cette transparence n'est pas générique car elle dépend des diverses architectures logicielles, des langages de programmation, etc.



* Gestion des noeuds, des objets, ... - Duplication - Sécurité
** Persistance - Localisation - Relocalisation - Migration - Défaillance - Duplication

FIG. 2.3 – *Le modèle ODP*

RM-ODP et les télécommunications

Nous présentons deux applications de RM-ODP dans le monde des télécommunications. La première porte sur le rapprochement des concepts de répartition de RM-ODP et du modèle de gestion OSI (ODMA). La deuxième porte plus sur l'intégration des services de télécommunications dans RM-ODP (TINA).

ODMA (Open Distributed Management Architecture) permet de définir des objets

(composants) traitant de la répartition des applications de gestion de l'OSI. Les fonctions de gestion (SMF) définies dans les normes X.73x à X.75x sont étendues à la répartition dans le modèle de gestion du modèle ODMA. La définition des composants rend transparent la gestion de la répartition par rapport à la conception d'applications réparties.

Les travaux de l'ODMA portent principalement sur les points de vue traitement et ingénierie de RM-ODP. Dans le point de vue traitement, le but est d'intégrer le modèle Gestionnaire/Agent et de définir les composants gérants et gérés d'une application. L'intégration des services CMIS est aussi proposée. Le point de vue ingénierie d'ODMA offre le traitement des notifications du modèle OSI et des réponses multiples (*scoping*) des requêtes CMIS.

Une autre application des télécommunications du modèle de référence ODP est représentée par les travaux du consortium TINA-C (Telecommunications Information Networking Architecture Consortium). Ce consortium s'est appliqué à traiter la gestion des services de télécommunications plus que la gestion des réseaux de télécommunications. TINA est une architecture permettant d'intégrer les services comme des composants indépendants des applications.

TINA reprend les principes d'architecture d'ODP qu'elle applique au monde du RGT (Réseaux de Gestion des Télécommunications). Le RGT permet d'interconnecter les équipements de réseaux de télécommunications (routeurs, commutateurs,etc.) avec les systèmes gestionnaires (postes de travail, etc.). Le RGT décrit aussi la gestion commerciale des réseaux et services, etc.

Pour plus de détails sur ces diverses architectures de réseaux et services, nous conseillons au lecteur de se référer à [SZ97].

Les technologies *middleware*

L'émergence des modèles de gestion d'applications et de réseaux répartis a fait apparaître de nouvelles technologies nommées *middleware* (en français, *les plates-formes réparties*). Les technologies middleware offrent la gestion des applications pour les couches *session* et *présentation* du modèle OSI et une partie de la couche application. On peut citer les technologies middleware CORBA (Common Object Request Broker Architecture)[COR93] (normalisé par l'OMG[9]) et DCE (Distributed Computer Environment)[KTW93] (normalisé par l'OSF[10]) parmi les plus connues. Une autre plate-forme répartie moins connue est celle définie dans l'architecture TINA : le DPE (Distributed Processing Environment). Le DPE est défini au travers du point de vue ingénierie du modèle de référence ODP. La figure 2.4 présente une vue schématique d'une plate-forme répartie [Ber93] [GGM97].

La technologie middleware actuellement prisée est CORBA. Nous basons notre description de CORBA sur [GGM97]. Le bus logiciel CORBA permet de concevoir des ap-

9. Open Management Group
10. Open Software Foundation

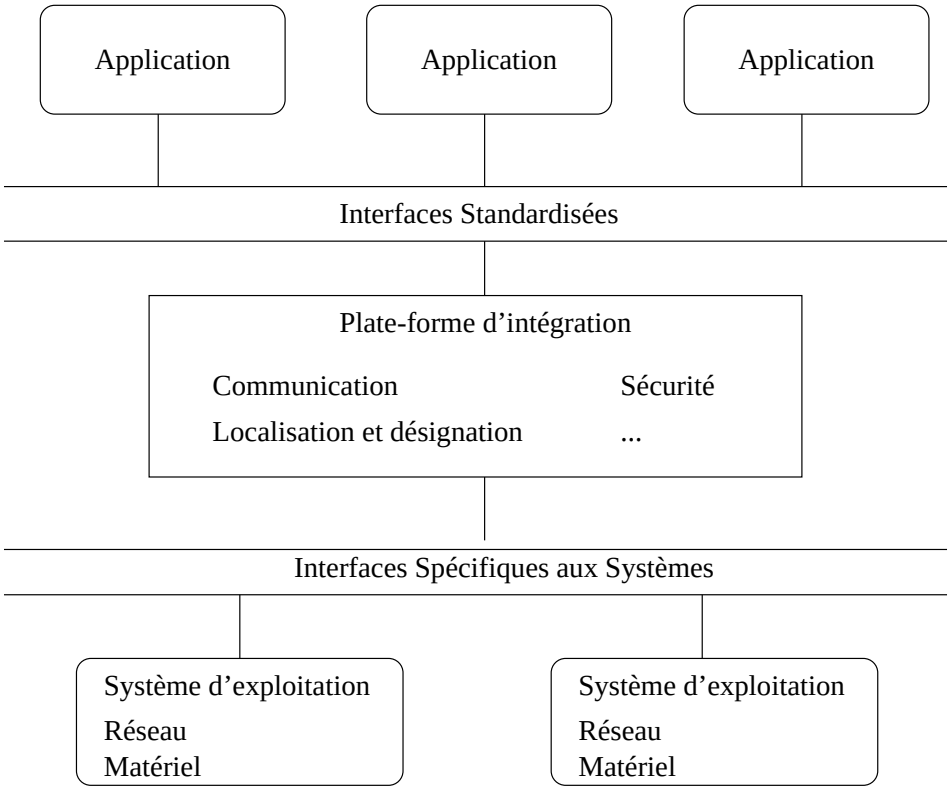


FIG. 2.4 – *Une vue schématique d'une plate-forme répartie*

plications définies à partir d'objets répartis. Ces objets sont définis sur des réseaux hétérogènes (on peut aussi définir des applications non réparties). CORBA les encapsule au moyen de *bus d'objets répartis* nommés ORB (Object Request Broker). Un ORB rend transparent les implantations des objets, les systèmes d'exploitations, les formats de données, et comporte des protocoles de transport permettant d'effectuer les communications inter-ORB ou avec d'autres systèmes. CORBA offre aussi des objets génériques de gestion tels que la création et destruction d'objets CORBA, le nommage des instances, la persistance, etc.

Par rapport aux modèles OSI et IAB, CORBA n'offre pas de nouveaux protocoles basses couches (CORBA fait reposer les communications sur les protocoles existants comme ceux définis par les modèles OSI et IAB). Les protocoles basses couches sont encapsulés par les protocoles GIOP[11] et IIOP[12] (IIOP est une dérivation de GIOP pour TCP/IP) instanciés sur les couches transport des réseaux.

Par rapport au modèle de référence ODP, CORBA est une application de certains des concepts de ce modèle (la répartition, les transparences, etc.). Cette application fondée sur le modèle Client/Serveur est nommée OMA (Object Management Architecture). OMA

11. General Inter-ORB Protocol
12. Internet Inter-ORB Protocol

est normalisée par l'OMG.

Les technologies middleware sont actuellement un compromis *opérationnel* entre l'existant (les protocoles réseaux) et les concepts génériques décrivant les fonctions des applications. Elles mettent en oeuvre ces fonctions indépendemment des entités réseaux (à la différence des protocoles définis pour les modèles OSI et IAB).

2.1.3 Bilan

Dans cette section, nous avons présenté les modèles de gestion des applications réparties et plus particulièrement les modèles de gestion des télécommunications. Dans cette thèse nous nous intéressons au comportement des applications au cours de leur vie (exécution d'un service et conséquence sur l'application, réaction à un événement, etc.). La description des services et des événements particuliers s'effectue lors de la description des données et des interfaces entre ces données.

Les données et interfaces sont décrits dans la couche *application* des modèles OSI et IAB. Les langages de spécifications utilisés sont GDMO et SNMP/MIB. Les comportements des applications sont définis en langue naturelle (cf. figure 2.2).

Les données et interfaces sont décrits dans les points de vue *information* et *traitement* du modèle RM-ODP. Aucun langage de spécifications n'est préconisé dans ces points de vue. Par contre, les applications du modèle RM-ODP, tel TINA, définissent des langages de spécifications pour ces points de vue. Dans le chapitre 7, nous présentons le langage de spécification Q-GDMO-GRM, langage préconisé par le consortium TINA-C pour le point de vue information de TINA. Ce langage ne fournit pas de description totale des comportements (une structuration tout au plus). Cependant, on peut imaginer utiliser un langage de spécifications formalisant les comportements dans une architecture fondée sur le modèle de référence ODP.

Nous n'avons pas encore exprimé comment les comportements sont représentés par les technologies middleware. Comme pour les modèles énumérés dans les paragraphes précédents, ces technologies fournissent des langages d'interfaces. Un nom générique est donné à ces langages : *Interface Definition Language (IDL)*. Les comportements dans les IDL ne sont pas spécifiés.

Dans la section suivante, nous présentons quelques langages de spécifications utilisés dans le monde des télécommunications. Nous les décrivons en termes de notations et de descriptions des comportements.

2.2 Télécommunication et langages de spécifications

Dans le monde des télécommunications, comme dans le monde des systèmes répartis, différents langages de spécifications ont vu le jour. Suivant les applications à créer et selon les systèmes les supportant, le langage de spécifications choisi pour spécifier une application peut être différent.

On dégage deux grandes familles de langages de spécifications dans les télécommunications : les notations semi-formelles et les méthodes formelles. Nous présentons ces deux familles dans les sous-sections suivantes.

2.2.1 Les notations semi-formelles

Les notations semi-formelles se présentent sous deux aspects : graphique ou textuel. Les langages graphiques représentent les objets sous forme de boîtes et les relations sous forme de flèches entre les boîtes. Les langages textuels représentent les objets et les relations sous forme de *formulaires*.

Les langages graphiques

Des exemples de langages graphiques sont OMT[13] [RBE+91], et plus récemment UML[14] [MUL97], Classe Relation [Des96], etc. La philosophie de ces langages est de fournir des représentations graphiques des spécifications des objets d'une application et des relations entre les spécifications des objets (c'est-à-dire les *classes*). Une classe *graphique* ressemble à une *boîte* à laquelle on associe un nom. Une spécification décrite par langage graphique est un ensemble de classes graphiques reliées par des arcs représentant les relations entre elles.

Le langage UML a réuni les concepts de plusieurs méthodes de conception objet (OMT, OOD[15] et OOSE[16]) [BGV97]. UML, en plus de la description des objets et de leurs relations, permet de décrire la dynamique d'un système (comportement) à l'aide de trois diagrammes : *séquence*, *collaboration* et *activités*. Le diagramme d'*activités* permet de définir les dépendances d'*actions* entre les *acteurs* du système. En terme applicatif, il permet de définir les dépendances entre services des objets d'une application.

Cependant, UML n'offre pas de description des services des objets (en terme de contenu). Certains langages graphiques sont liés directement à des AGL (Ateliers de Génie Logiciel). Par exemples, le langage Classe Relation est intégré dans l'AGL Objecteering [SOF97] ou encore l'AGL ROSE[Rat99] pour UML. Les AGL permettent, entre

13. Object Modeling Technique
14. Unified Modeling Language
15. Object Oriented Design [BOO91]
16. Object Oriented Software Engineering [JCJ92]

autres, la traduction des classes graphiques en type correspondant au langage de programmation choisi pour l'implantation de l'application. Les comportements dynamiques sont alors définis directement dans le langage d'implantation et non dans la spécification de l'application.

Des exemples d'applications de gestion des télécommunications spécifiées à l'aide de langages graphiques sont FTMN-Alarmes en Classe-Relation [Fra97b] ou encore l'architecture fonctionnelle des réseaux de transport en OMT [Int95].

Les langages textuels

Les langages de spécifications semi-formels de forme textuelle les plus utilisés dans les télécommunications peuvent être scindés en deux groupes :

- les langages spécifiant des *objets gérés* : GDMO [17] [GDM92], SNMP [18]/MIB [19] [RM91] ;
- les langages d'interfaces (les IDL, cf. sous-section 2.1.2) : CORBA/IDL [COR93], DCE/IDL [KTW93], etc.

Les langages spécifiant des objets gérés décrivent les objets à l'aide de *formulaires*. GDMO est le langage le plus complet (ou complexe) dans ce groupe. En effet, il permet de spécifier *unitairement* les composants d'un objet (attributs et services). Par exemple, la description d'un attribut peut être réutilisée dans plusieurs descriptions d'objets gérés.

Une autre caractéristique de GDMO est le découpage du formulaire décrivant un objet en formulaires nommés *packages*. Les packages représentant un objet peuvent être obligatoires ou conditionnels. Ce découpage permet d'optimiser la réutilisation et plus précisément la spécialisation (cf. sous-section 2.3.2). En effet, deux objets peuvent être de la même classe mais ne pas avoir les mêmes attributs et services, mais ils ont un comportement global équivalent par rapport au monde qui les entoure. Pour résumer, GDMO est un langage de normalisation, il a pour vocation de *mettre tout le monde d'accord*. Nous reconnaissons que cette *vocation* n'est pas toujours simple a mettre en oeuvre, GDMO en est un exemple par la multitude de ces formulaires et concepts.

GDMO et SNMP/MIB n'offrent pas de description des relations entre classes d'objets. L'ITU a associée à GDMO la norme GRM [20] [ITU95b] qui permet une description des relations entre objets gérés.

Les IDL décrivent une application sous forme d'objets représentés par des interfaces (description des données, contenues dans l'objet, et des services, comportements de l'ob-

17. Guidelines of Description of Managed Objetcs)
18. Simple Network Management Protocol
19. Management Information Base
20. General Relationship Model

jet). La communication entre objets s'effectue en Client/Serveur, mais au contraire des autres langages semi-formels, un objet est Client et/ou Serveur.

Un autre langage de spécification semi-formel est Q-GDMO-GRM. Il définit des formulaires de représentation des objets gérés et des relations entre ces objets. Il offre un schéma pour intégrer une notation formelle pour décrire les comportements. Les autres langages textuels cités précédemment n'offre pas un schéma de représentation des comportements, sauf GDMO. Les comportements en GDMO sont définis en langue naturelle (cf. sous section 2.1.1) et encapsulés dans des formulaires. On peut déclarer un formulaire de comportement pour un attribut (comportement de l'attribut quelle que soit la classe d'objets le contenant), un service, etc. Q-GDMO-GRM comporte moins de formulaires que GDMO et les comportements sont moins complexes à construire. Nous discutons plus longuement de la différence entre Q-GDMO-GRM et GDMO dans le chapitre 7.

Des applications de gestion des télécommunications sont spécifiées à l'aide de langages semi-formels telles que la gestion des alarmes dans le modèle OSI [ITU92e], la gestion d'une pile TCP/IP [MR90][MR91], l'application de configuration de réseau OPERA [CNE96], etc.

2.2.2 Les méthodes formelles

Dans le monde des télécommunications et plus précisément de la normalisation ISO (ITU), les *méthodes formelles* permettant de spécifier des applications sont regroupées sous le nom générique de FDT (Formal Description Technique) [ISO94] [Gar89]. Par rapport aux notations semi-formelles, elles permettent de spécifier les comportements des applications. Les méthodes formelles les plus utilisées dans les télécommunications sont VDM-SL[21] [ISO96], SDL[22] [SDL93], LOTOS[23] [LOT89], Z [Spi92] et Object-Z [DKRS94].

Les méthodes formelles n'offrent pas toutes une description objet d'une application. Leurs notations sont souvent issues des langages de spécifications algébriques. Ceux-ci permettent de décrire des types de données complexes [Loe87]. Des exemples de langages algébriques sont ACT ONE [EM85], CLU [LAB+81], LARCH [GHG+93], LPG [BDE87], etc.

L'intégration des méthodes formelles dans le monde des télécommunications n'est pas aisée. Une des principales raisons est de ne pas prendre en compte les types de données échangés dans les applications de gestion, notamment les données ASN.1 [Fes94] [Kel95].

L'utilisation des méthodes formelles dans le monde industriel est de plus en plus fréquente. Des exemples de travaux sont cités et décrits dans [ABL96], [FJL97] et [LFB96]. Dans le monde des télécommunications, nous pouvons citer les spécifications des services et protocoles OSI à l'aide de FDT répertoriées dans [Gar89], ou encore la description en

21. Vienna Development Method-Standard Language
22. Standard Definition Language
23. Language Of Temporal Ordering Specifications

Z de l'architecture fonctionnelle des réseaux de transport décrite dans [Int95].

2.3 Les langages de spécifications d'interfaces d'objets gérés

La section 2.2 introduit les principaux langages de spécifications utilisés pour spécifier les applications de gestion des réseaux de télécommunications. La tendance actuelle pour les systèmes répartis et les télécommunications est de représenter sous forme d'objets les données et services d'une application. L'encapsulation objet des données apporte quelques concepts et paradigmes concernant la spécification d'applications réparties. Nous les présentons globalement dans la sous-section 2.3.1 et aux travers des langages de spécifications décrits dans la section 2.2.

Après la description de la structure des applications par les langages de spécifications, il nous faut présenter plus en détail la spécification des comportements. La simulation de spécifications doit s'appuyer sur celles-ci dans le cas de leur mise au point. Dans la sous section 2.3.2, nous présentons une description globale du contenu des comportements (références aux données, structures internes des comportements, etc.) et l'intérêt de leur formalisation dans les langages de spécifications.

2.3.1 Le monde objet

Les notations textuelles (cf. sous-section 2.2.1) ont une représentation orientée objet des applications. Les méthodes formelles ne sont pas toutes ce type de représentation (Z est ensembliste, etc.). Cependant, des extensions orientées objet de méthodes formelles ont été proposées et définies (Object-Z, OSDL, etc.).

Le monde objet permet d'intégrer des concepts généraux (interopérabilité, réutilisation, etc.) dans la spécification d'applications. Nous énumérons et décrivons les principaux concepts destinés aux télécommunications.

- *L'interopérabilité*, en général, *implique la distribution d'informations, la communication et la compréhension mutuelle entre différents domaines d'exécution* (cf. [Oa97] page 340). Dans le domaine des spécifications, l'interopérabilité est intégrée en trois étapes :
  - la description sous forme d'objets offre la *distribution d'informations* ;
  - la *communication entre les domaines d'exécution* est assurée par la description des comportements et des relations entre objets[24] ;

24. beaucoup de langage de spécifications permettent de décrire des relations de dépendances entre objets. Les relations peuvent être considérées comme des comportements particuliers.

  - la *compréhension entre les domaines d'exécution* est représentée par la définition d'un typage des données au sein du langage de spécifications.

- *La réutilisation* s'applique :

  - aux *sources d'informations* existantes (cf. [Oa97] page 341), c'est-à-dire qu'un objet peut être utilisé par plusieurs applications ;
  - aux spécifications, c'est-à-dire que la spécification d'une interface peut être utilisée par plusieurs applications.

- *La spécialisation* d'interfaces existantes (l'héritage, cas particulier de la réutilisation), i.e. un même objet peut être représenté par deux interfaces différentes dans deux applications différentes.

Les langages de spécifications doivent prendre en compte ces différents concepts. Cependant, ces concepts sont interprétés différemment suivants les langages de spécifications. Par exemple, le concepts de spécialisation de GDMO est plus général que celui de CORBA/IDL. En effet, GDMO permet à deux objets de même classe de ne pas avoir les mêmes attributs en assurant que les deux objets aient le même comportement.

Dans la sous-section suivante, nous traitons plus particulièrement la spécification des comportements des applications. La spécification des comportements doit prendre en compte les concepts objet précédemment énumérés.

2.3.2 Les comportements

Les comportements d'un objet rassemblent les services et les *invariants* de l'objet. Des services simples sont par exemple l'accès à un attribut, sa modification, etc. Les invariants représentent les propriétés que doit respecter l'objet tout au long de sa vie. Par exemple, le comportement défini dans la figure 2.2 décrit que lorsque l'attribut *alarmSeverityAssignmentProfilePointer* est de valeur *NULL* alors deux choix se présentent :

- soit l'agent assigne cet attribut avec une valeur valide (une valeur du domaine de définition de l'attribut, c'est-à-dire une valeur de type ASN.1 *PointerOrNull*) ;
- soit la valeur *indeterminate*.

La différence entre les services et les invariants porte sur la dynamique. En effet, les services permettent de décrire la dynamique de l'objet (voire de l'application). Pour schématiser cette différence, nous introduisons succinctement la notion d'*état* d'une application. La vie d'une application est représentée par un ensemble d'états (à un moment t l'application est dans l'état e, à un moment t' dans l'état e', etc.). Tout état de l'application doit respecter les invariants. Un service permet de passer d'un état à l'autre de

l'application. Ces descriptions très abstraite des comportements d'objets et de la notion d'état sont détaillées dans les chapitres 4 et 5.

Les comportements peuvent être plus complexes et concerner, par exemple, plusieurs attributs de l'objet. En allant plus loin dans la complexité, un comportement peut être la cause d'un déclenchement d'un comportement lié à une autre objet. De tels comportements peuvent être représentés au travers des relations entre objets. En effet, par le biais d'une relation, un objet à la connaissance d'un autre objet (ces relations peuvent être des relations de dépendances d'exécution de services, cf. diagramme d'activités du langage UML sous-section 2.2.1).

Dans le paragraphe précédent, nous décrivons un comportement comme une description d'un état. La spécification des comportements dans les langages de spécifications doit prendre en considération l'état d'un objet. Les langages dit d'interfaces (GDMO, IDL, etc.) n'offrent pas de description des comportements, du moins, lorsqu'ils ne sont pas associés à des langages de description de relations. Cependant, les relations ne sont qu'une partie des comportements d'une application.

Les notations semi-formelles offrent une syntaxe et une sémantique objet des fonctions d'une application, mais ne décrivent pas totalement les comportements dynamiques. Les méthodes formelles offrent une syntaxe et une sémantique décrivant de tels comportements [HB95]. La sémantique introduite permet de détecter les ambiguïtés des spécifications informelles des comportements. L'utilisation des méthodes formelles présentent d'autres intérêts, notamment la validation des spécifications par la preuve des propriétés des applications spécifiées, le raffinement prouvé, la génération de jeux de tests, etc. [GMSB96]

Chapitre 3

Simulation et applications de télécommunication

Le but de ce chapitre est de définir notre approche de la *simulation de spécifications d'applications de gestion* dans le domaine des télécommunications. La simulation est la démarche qui permet d'effectuer la validation et la vérification des applications. La validation d'une application permet d'assurer que la description des besoins (on les décrit sous formes de fonctions) *répond à l'attente des utilisateurs et aux contraintes de leur environnement* [GMSB96]. La vérification d'une application permet de *s'assurer que les descriptions successives de l'application satisfont la spécification globale* [GMSB96].

Entre validation et vérification des applications, nous avons choisi la validation. Dans la section 3.1, nous présentons notre approche de celle-ci par sa description dans le monde de la simulation. D'un point de vue général, cette dernière peut s'effectuer sur les implantations et/ou les spécifications des applications. Après un comparatif entre ces deux supports de simulation, nous présentons les concepts, que doit intégrer la simulation, nécessaires à la validation de spécifications.

Comme nous l'avons vu, les applications de gestion des réseaux de télécommunications sont décrites à l'aide de langages de spécifications divers. En section 3.2, nous présentons quelques travaux autour de la simulation d'applications définies dans certains de ces langages.

Enfin, dans la section 3.3, nous effectuons un bilan des deux chapitres 2 et 3. De ce bilan, nous présentons nos choix et objectifs de la simulation de spécifications d'applications de gestion des réseaux de télécommunications.

3.1 La simulation

Une définition globale de la simulation est la suivante [Bil87] :

Définition 1 *La simulation est une technique qui consiste à construire puis exécuter le modèle d'un système réel pour en étudier le comportement sans perturber le système réel.*

Dans le cadre de la thèse, la simulation des applications revient à construire un modèle représentant un ensemble d'appels à des services des objets de celles-ci. Ces appels peuvent être exécutés soit sur les implantations des applications (*système réel*), soit sur des *maquettes* des applications [GMSB96]. Dans la sous-section 3.1.1, nous présentons la construction d'un modèle suivant le choix d'environnement d'exécution de celui-ci.

Une maquette est une ébauche des applications définie à partir des spécifications des applications. Elle n'a pas les performances, la qualité attendue et toutes les fonctions attendues [GMSB96]. Par contre, elle permet de valider les besoins des applications au plus tôt dans le cycle de réalisation de celles-ci. Dans la sous-section 3.1.2, nous présentons la construction des maquettes à partir des spécifications.

Dans la définition 1, le but de la simulation d'une application est d'*étudier son comportement*. L'étude peut se décliner en deux phases distinctes : la *validation* et la *prédiction*[1]. La validation permet de vérifier au travers des fonctions de l'application que les besoins des utilisateurs sont respectés. Dans le cadre de la simulation d'une application de gestion des alarmes d'un réseau, une fonction à valider est par exemple l'assurance que toute alarme émise est prise en compte par un *réparateur* susceptible de résoudre le problème lié à cette alarme.

Au contraire de la validation, la prédiction permet de déduire ou de construire des comportements complexes de l'application. Dans l'exemple de la simulation d'une application de gestion d'alarmes, on peut par prédiction construire les comportements de l'application lorsqu'un *réparateur* se déconnecte subitement du réseau, c'est-à-dire comment sont traitées ces alarmes en attente, etc. La prédiction s'oriente vers des travaux de tests et principalement de génération automatique de tests. Nous conseillons le lecteur de se référer à [VA98] qui offre un état de l'art sur la génération automatique de tests. [VA98] définit une méthode d'analyse de spécifications formelles des applications permettant de prédire des comportements des applications et de construire les tests correspondants.

Dans cette thèse, nous n'effectuons pas de la prédiction de comportements d'applications mais de la validation des fonctions des applications. Dans la sous-section 3.1.3, nous présentons comment est réalisée cette validation.

1. nous rappelons que nous n'effectuons pas de vérification.

3.1.1 Modèle de simulation

Un modèle d'une application est *un ensemble d'appels à des services des objets de celle-ci*. Dans le monde de la simulation, définir les appels pertinents ou nécessaires à la validation des fonctions de l'application est l'activité de *test*. Celle-ci regroupe un ensemble de tests (un test est un appel d'un service ou une suite d'appels de services) que l'on peut classifier ainsi [Som92] :

- les tests unitaires : on teste chaque composant de l'application indépendemment des autres composants (un composant est un attribut, un service, etc.) ;
- les tests des modules : un module est par exemple un objet. On teste les liens entre attributs et services de l'objet ;
- les tests de sous-systèmes : permettent de tester les interfaces entre les objets d'une application ;
- les tests des systèmes : ils permettent de tester les relations entre acteurs d'une application (entre gestionnaires et agents, cf. sous-section 2.1.1). Les aspects fonctionnels et non fonctionnels de l'application doivent être validés par ces tests ;
- les tests d'acceptation : on teste l'application dans l'environnement de l'utilisateur final. Ces tests permettent de mettre en évidence les erreurs des besoins exprimés dans le cahier des charges de l'application.

La classification précédente met en évidence la granularité des tests et donc de la validation. Dans cette thèse, nous nous attachons à valider les fonctions des applications par rapport aux besoins exprimés. Les modèles que nous traitons sont constitués de test de systèmes et d'acceptation. Comme exemple, nous définissons informellement le test de système permettant de valider la fonction de prise en compte des alarmes émises dans le cadre de l'application de supervision d'alarmes FTMN-Alarmes [Fra97a] réalisée dans le laboratoire DES/ERA. En effet, les alarmes émises sur un réseaux de télécommunications sont diverses (conditions climatiques, matériel en panne, charge trop importante, etc.) et ont plusieurs niveaux de priorités. L'application FTMN-Alarmes permet de distribuer les alarmes émises suivant des groupes d'utilisateurs adéquats. La définition de *profils utilisateurs* permet d'assurer une distribution des alarmes vers les bons utilisateurs. Un modèle de l'application FTMN-Alarmes traitant cette fonction doit contenir un test qui à chaque déclenchement d'alarme vérifie qu'il existe au moins un utilisateur susceptible de la traiter.

Dans l'introduction de cette section, nous précisons que l'on peut exécuter un modèle d'une application soit sur ses implantations, soit sur des maquettes de celle-ci. Quel que soit l'environnement de simulation, les différents types de tests énumérés précédemment peuvent être pris en compte. Cependant, nous pouvons tout de même dégager une différence entre ces deux environnements de simulation suivant les types de tests. En effet, il est plus avantageux d'effectuer la validation des fonctions et besoins d'une application sur

des maquettes de celle-ci. Pour des raisons de coûts, la détection des erreurs fonctionnelles et de besoins sur une maquette permet la correction (mise au point) des spécifications au plus tôt dans le cycle de réalisation de l'application traitée. Ce type d'erreur peut engendrer des modifications importantes de l'architecture de l'application et donc des implantations.

D'un point de vue contraire, il est avantageux d'effectuer les tests unitaires, de modules et de sous-systèmes sur les implantations des applications. Ces tests portent sur les performances des composants de l'application et sur les communications entre ceux-ci. Les réaliser directement sur les implantations nous parait plus adéquat.

Le discours que nous tenons dans les paragraphes précédents impose l'exécution des modèles sur un environnement suivant les types de tests qu'ils représentent. Dans certains cas, il est nécessaire d'effectuer les tests de systèmes et d'acceptation sur les implantations et/ou d'effectuer les tests unitaires, de modules et de sous-systèmes sur des maquettes. Pour le premier cas, un exemple porte sur la validation de fonctions de performances. En effet, prenons une application de gestion dont les fonctions doivent vérifier les temps d'acheminements de données entre entités réseaux. La simulation de ces fonctions sur une maquette est moins crédible que sur une implantation. Cependant, ce type de validation est particulière dans le monde des télécommunications. Dans cette thèse, nous nous intéressons à la simulation sur des maquettes des applications.

Pour conclure cette sous-section, les modèles pris en compte sont construit à partir des spécifications des applications pour représenter les fonctions des applications. Les modèles décrivent ces fonctions en termes d'appels à des services d'objets. Ces appels sont ensuite exécutés sur des maquettes. Dans la sous-section 3.1.2, nous donnons une description globale de la construction de ces dernières.

3.1.2 Environnement d'exécution de la simulation

Dans la définition 1, Le modèle doit être *exécuté ... sans perturber le système réel.* Dans le cadre de la simulation sur une maquette, le système réel n'est pas perturbé. On peut aussi citer que la maquette est une représentation fictive du système réel [Smi96]. Construire une maquette d'une application est l'activité de traduction des spécifications d'une application dans un environnement dédié à la simulation. [Tou89] présente un système réel comme l'ensemble d'un objet réel et de son environnement d'évolution. Un maquette d'un système réel est alors une représentation de ce système dans un autre environnement avec la contrainte suivante: les objets définis dans ce nouvel environnement doivent préserver les comportements des objets équivalents dans les implantations de l'application.

Tous les objets d'une application ne sont pas définis dans une maquette. En effet, les maquettes sont des images des systèmes (implantations). Ces images ne sont pas obligatoirement complètes mais peuvent être partielles, i.e. la simulation peut s'appliquer à ne valider que certaines parties du système réel. Dans l'application de supervision des

alarmes FTMN-Alarmes, dans un premier temps, le test de durée du traitement de l'alarme est à effectuer sur l'implantation de l'application. D'un point de vue fonctionnel, ce test n'est pas obligatoirement une priorité dans la simulation.

Les objets d'une maquette d'une application sont déduits à partir des spécifications. Les spécifications d'une application sont souvent abstraites (cf. sections 2.2 et 2.3). Il faut donc définir une traduction concrète des spécification qui puisse être interprétée par le simulateur. De plus, pour déclencher les services des objets simulés, il faut que l'environnement d'exécution puisse interpréter des *requêtes*, images concrètes des appels aux services des objets. L'exécution des requêtes est abordée dans la sous section 3.1.3.

Les représentations concrètes des objets des applications dans une maquette ne suivent pas toujours celles des implantations. En effet, prenons le cas d'une alarme, il n'est pas nécessaire de définir explicitement la gestion des alarmes en attente de traitement par un *réparateur*. Les langages de haut niveau (fonctionnels, logiques) offrent un environnement d'exécution permettant de prendre en compte ce niveau d'abstraction. Les divers avantages de ces langages sont, par exemples, la gestion automatique de la mémoire, le contrôle de types (ces langages sont souvent typés), etc. Pour la simulation, l'utilisation de tels langages pour concevoir les objets d'une maquette permet de focaliser la simulation sur l'étude du comportement de l'application.

En conclusion des deux sous-sections précédentes, nous préconisons de concevoir les simulateurs (permettant la validation des fonctions des applications) sur des environnements associés à des langages de haut niveau. En ne rappelant pas les avantages d'un point de vue maquette, ces environnements permettent de concevoir rapidement des modèles et de les exécutés directement sur les maquettes. Notons que les modèles peuvent être aussi validés sémantiquement (contrôle dynamique des types des paramètres des appels aux services des objets, etc.). Dans la sous section 3.1.3, nous détaillons ces contrôles. Dans le chapitre 7, nous présentons un simulateur écrit dans un langage de haut niveau.

3.1.3 Étude du comportement

L'étude du comportement d'une application consiste à exécuter le modèle représentant ce comportement et à interpréter le résultat de cette exécution. La validation revient à considérer que le résultat est conforme aux besoins des utilisateurs. Elle peut intervenir au niveau de :

- l'exécution d'une requête représentant un service d'un objet simulé de l'application. On vérifie que la requête fournit le résultat attendu dans le modèle ;
- l'exécution d'un scénario de requêtes. On vérifie que l'ordonnancement des requêtes ne viole pas les besoins. La validation peut alors s'effectuer sur le modèle ;
- la préservation de la cohérence globale de l'application. On vérifie l'invariant.

Le dernier point de validation peut être illustré ainsi : dans l'application FTMN-Alarmes, on peut vérifier qu'il existe pour tout type d'alarmes un profil utilisateur correspondant. Cette validation peut s'effectuer statiquement et dynamiquement. En effet, statiquement, elle est réalisée par un contrôle de type, dynamiquement, à chaque création d'une alarme par le modèle dans la maquette, on vérifie qu'il existe un objet *réparateur* susceptible de traiter cette alarme.

Le cas dynamique précédent rejoint la validation du résultat de l'exécution d'une requête ou d'un scénario. La validation revient à signaler une erreur due à une incohérence dans la spécification des services. L'erreur considérée n'est pas obligatoirement due à une réponse négative d'une requête. Par exemple, dans l'application FTMN-Alarmes, un service doit fournir la date de fin de traitement d'une alarme. Ce service ne peut être appelé que lorsque l'alarme a été traitée. Dans un modèle, on peut effectuer un test pour vérifier qu'une alarme non traitée ne possède pas de date de fin de traitement, il suffit de faire à appel au service fournissant cette date et de vérifier qu'il ne peut être rendu. Nous avons là une exécution attendue du modèle.

La figure 3.1 schématise succinctement le processus de validation du modèle (la figure 3.1 est librement inspiré de la figure représentant *le processus de mise au point* donnée dans [Som92], page 374).

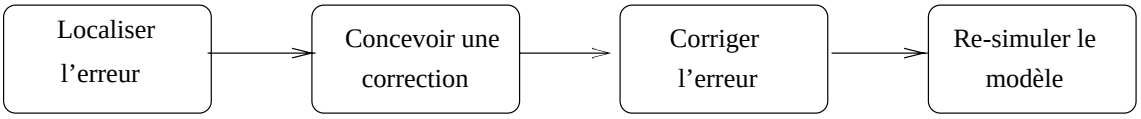


FIG. 3.1 – *Processus de correction d'erreurs*

La première étape est de *localiser l'erreur*. Elle est à la charge de l'utilisateur qui suivant les résultats des requêtes décide s'il y a une erreur, c'est-à-dire de juger si les résultats sont conformes à ses attentes. Dans le cadre de la thèse, les types d'erreurs proviennent des spécifications ou des modèles. Le simulateur doit fournir la possibilité à l'utilisateur de corriger une les spécifications et les modèles (étapes *concevoir une correction* et *corriger l'erreur* du processus de mise au point).

L'étape de conception d'une correction peut être complexe. Par exemple, l'émission d'une alarme sur le réseau est un événement dynamique que doit traiter l'application FTMN-Alarmes. Cette émission engendre le déclenchement d'autres services et d'autres événements dynamiques qui ne sont pas obligatoirement décrits dans le modèle. Le simulateur doit fournir une correction en prenant en compte ces appels et événements non signalés.

Le discours tenu dans les paragraphes précédents permet de considérer la simulation prise en compte dans cette thèse comme de la mise au point des comportements des applications. Cette mise au point est réalisée par les utilisateurs soit en modifiant les spécifications soit les modèles. Cependant, ces modifications sont satisfaisantes à l'instant de l'erreur mais peuvent ne pas l'être à un instant précédent. La dernière étape du proces-

sus de correction est de *re-simuler le modèle*. Cette étape permet de valider la correction apportée aux spécifications ou au modèle dans tout le cycle de vie de l'application. La figure 3.2 montre la reprise sur erreur au cours de la mise au point des spécifications.

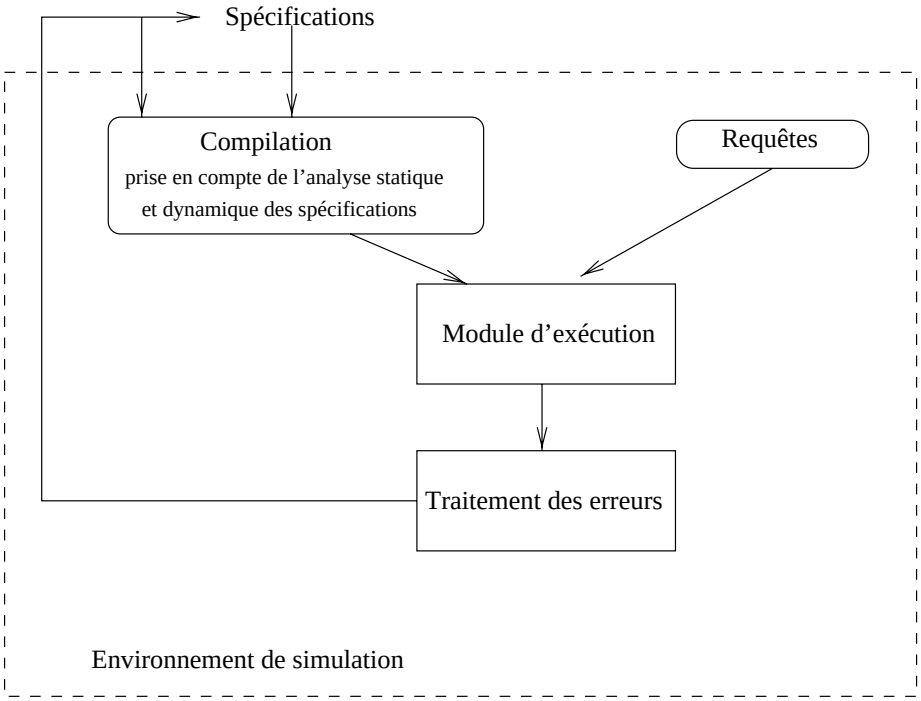


FIG. 3.2 – *Simulation de spécifications*

La figure 3.2 encapsule les différents modules constituant un simulateur dans l'*environnement de simulation*. Cette enveloppe est accessible souvent par le biais d'une IHM (Interface Homme Machine). Cette dernière permet aux utilisateurs de visualiser les spécifications, les résultats, le modèle, etc. Elle permet ainsi aux utilisateurs de remplir efficacement leur rôle d'*oracle* [GMSB96].

3.2 Simulation et télécommunication

Dans la section précédente, nous décrivons globalement la mise au point de spécifications d'applications. Dans cette section, nous reprenons cette description dans le monde des télécommunications. Dans un premier temps, il nous faut étudier la relation entre la simulation et les spécifications. Les langages de spécifications sont de types différents (notations semi-formelles, méthodes formelles), la mise au point des applications n'est pas de même nature. Dans la sous-section 3.2.1, nous présentons des simulateurs définis sur des langages différents.

Pour simuler une application, il nous faut définir une maquette et construire un ou des modèles. Dans la sous-section 3.2.2, nous présentons comment les différents si-

mulateurs existants traduisent les spécifications en maquette et conçoivent les modèles. Nous montrons surtout la difficulté de valider les fonctions des applications à partir de ces modèles, et particulièrement de valider les comportements dynamiques.

Enfin, nous donnons notre point de vue sur le processus de corrections des erreurs de simulation d'applications de télécommunications. La sous-section 3.2.3 récapitule les erreurs possibles des spécifications et des modèles de ces applications. Elle propose un canevas d'actions permettant de les solutionner.

3.2.1 Simulation et langages de spécifications

Les langages de spécifications permettent de spécifier les fonctions des applications. Une fonction peut être représentée comme un ensemble de services. Cependant, suivant le langage utilisé, ceux-ci sont plus ou moins spécifiés. Les notations semi-formelles n'offrent pas de description formelle des services au contraire des méthodes formelles. Nous présentons différents outils de simulation fondés sur les deux types de langages.

Un outil de simulation portant sur les notations semi-formelles GDMO/GRM est le simulateur TIMS (TMN-based Information Model Simulator) [Maz97]. TIMS permet d'exécuter des *messages* CMIS sur des objets GDMO. Un autre travail est l'extension du simulateur CRUSADE (Communicating RUle Systems Automated Development Environment) [Esc91] au monde du RGT (Réseaux de Gestion des Télécommunications[2]) [ITU92c]. Cet outil étendu [Fes94] offre aussi la possibilité d'exécuter des requêtes CMIS sur des objets GDMO.

GDMO n'offrant pas de description formelle des services des objets, les deux outils cités dans le paragraphe précédent utilisent des spécifications formelles les décrivant (BL[3] pour TIMS, LOBSTERS[4] pour l'extension de CRUSADE).

Des outils de simulation ont été aussi développés sur des méthodes formelles tel l'outil EXPOSE [WB91]. Cet outil permet *d'animer* des spécifications formelles d'applications définies sous forme de processus concurrents, et en particulier des spécifications LOTOS. Nous introduisons ici la notion d'*animation* de spécifications. Cette notion permet de faire une différence entre des spécifications exécutables et non (ou partiellement) exécutables. En effet, au cours de la simulation de spécifications, ces dernières peuvent contenir des ambiguïtés (notion de *non-déterminisme*). Les ambiguïtés contenues dans les spécifications ne peuvent pas être traitées par la simulation sans explosion combinatoire.

Pour traiter le non-déterminisme, des solutions proposées dans la littérature sont les

2. en anglais : TMN, Telecommunication Management Network.

3. Behaviour Language, [Maz97].

4. Language for Object Behaviour Specification based on Templates and Extended Rule Systems, [Fes94].

suivantes :

- restreindre le langage de spécifications pour qu'il soit exécutable (le langage de spécifications restreint est un langage de programmation). Des travaux relatifs à cette solution sont ceux effectués sur le langage de spécifications VDM-SL. Ces travaux ont abouti à la définition du langage IPTES Meta-IV [LL91] [ELL92]. Les auteurs de ce langage montrent qu'une partie de VDM-SL est exécutable. Un exemple d'utilisation de IPTES Meta-IV est donné dans [ELA93] ;
- traduire des spécifications *non exécutables* en langage de programmation. Un exemple de traduction de schémas Z en Prolog est présenté dans [SCT96]. Cependant, les auteurs reconnaissent que toutes les caractéristiques de Z ne sont pas traduites ainsi que leurs corrections (par exemple, la vérification qu'une fonction est injective, totale, etc.).

Restreindre les langages de spécifications induit une perte d'expressivité (on n'exprime pas le non-déterminisme des spécifications). L'outil EXPOSE simule des spécifications LOTOS non restreintes. Cependant, les spécifications prises en entrée du simulateur sont non ambiguës. Pour les simulateurs TIMS et CRUSADE étendu, les spécifications des comportements dynamiques sont définis dans des langages *exécutables*. TIMS est défini en Scheme [CR91]. CRUSADE offre un environnement de simulation du langage CRS (Communicating Rule Systems) [MNM87], langage utilisé pour décrire les comportements dynamiques dans LOBSTERS.

3.2.2 Maquette et modèle

Pour simuler des spécifications d'application, il faut traduire ces spécifications pour concevoir une maquette et des modèles de cette application. Une maquette doit être le plus proche possible de la représentation de l'application qu'ont les utilisateurs. La maquette doit aussi être proche de la représentation sous forme d'objets et de relations entre ces objets.

Les méthodes formelles n'offrent pas toutes une représentation objet des applications. La validation des fonctions doit s'effectuer sur une représentation souvent *mathématique* des applications. En effet, les méthodes formelles sont définies sur des modèles mathématiques. Les maquettes sont définies sur des simulateurs intégrant ces modèles. Les langages dits de haut niveau assimilent ces modèles (les langages fonctionnels, les langages de programmation logiques). De par leur richesse, les langages de haut niveau permettent un maquettage rapide des applications.

Au contraire des méthodes formelles, les notations semi-formelles utilisées dans les télécommunications intègrent le monde objet. D'un point de vue des utilisateurs, ces notations permettent une description plus proche du résultat final des applications. Concevoir une maquette à partir de telles spécifications peut aussi s'effectuer à l'aide de langages de

haut niveau. En effet, des langages alliant le fonctionnel ou la logique et la programmation objet existent. Un des premiers langages de ce type est CLOS[5][Ste90]. Dans le chapitre 7, nous utilisons un tel langage comme langage de simulation.

Quelque soit le langage de spécifications utilisé, il faut définir des modèles permettant de valider les fonctions des applications. Ces modèles sont exécutés sur les maquettes des applications. La méthode pour les concevoir consiste à offrir un langage de requêtes ou encore une IHM permettant de construire visuellement les requêtes. Ce langage permet d'encapsuler les appels aux services des objets dans un format pouvant être interprété par le simulateur (par le module d'exécution du simulateur). Par exemple, TIMS et CRUSADE étendu prennent en entrées des requêtes CMIS. Ces requêtes correspondent aux services des objets gérés définis en GDMO.

Les modèles d'une application peuvent être gérés par les simulateurs. Des outils intéressants sont :

- un gestionnaire de fichiers contenant les modèles (un modèle par fichier). Le format des requêtes doit être traduit en appel à des méthodes interprétées par le module d'exécution ;
- un service permettant aux utilisateurs de construire et de gérer leurs requêtes. Cela impose que le simulateur soit dédié au langage de spécifications (il faut toujours construire des compilateurs spécifiques entre les langages de spécifications et les langages de simulation).

Le schéma permettant d'exécuter les requêtes sur la maquette d'une application est souvent celui utilisé dans le monde des télécommunications, le modèle Gestionnaire/Agent (cf. figure 2.1). La maquette joue le rôle de l'agent et les requêtes celui du gestionnaire.

Nous n'avons pas abordé la traduction des comportements des applications dans les langages de simulation. Dans le cas de spécifications formelles, les langages de haut niveau offrent un ensemble de concepts qui permettent une traduction. Cependant, le non-déterminisme n'est pas traité par ces langages. Dans les outils de simulation cités dans cette section, le non-déterminisme n'est pas pris en compte. Cependant, on peut considérer le non-déterminisme comme une erreur de simulation, c'est-à-dire qu'on laisse son traitement à l'utilisateur. Dans la sous-section suivante, nous développons ce choix de traitement.

3.2.3 Traitement des erreurs

Dans cette sous-section, nous considérons que les spécifications des applications respectent la sémantique statique des langages. Les problèmes de types, d'héritage, de nommage de classes, ... , sont résolus à la compilation.

5. Common Lisp Object System

Les simulateurs contiennent un processus de correction des erreurs. Nous classifions ainsi les erreurs susceptibles d'être remontées par la simulation :

- problèmes de types. Une telle erreur peut se produire lors de l'exécution d'un service. Par exemple, un paramètre de la requête n'est pas bien typé ou, au contraire, la spécification du type du paramètre du service n'est pas conforme aux attentes des utilisateurs ;
- problèmes de résultats non conformes. Les utilisateurs ont décelé un erreur de spécifications ou une erreur de modèle ;
- problèmes de non-déterminisme. La spécification d'un service demande un choix .

Le deuxième point est le coeur de la mise au point des spécifications. Le but de la simulation est de vérifier que les fonctions des applications attendues (images des besoins définis dans le cahier des charges) soient validées. Le module de traitement des erreurs reçoit de la part du module d'exécution le résultat d'exécution de chaque requête. Le module de traitement des erreurs (piloter par les utilisateurs) doit décider si une erreur de spécifications est à déceler et comment il faut la corriger (cf. sous-section 3.1.3).

Le traitement du non-déterminisme par la simulation n'est pas souvent exprimé dans la littérature. Ce traitement peut s'effectuer de deux façons :

- soit l'utilisateur choisi qu'elle branche de simulation poursuivre ;
- soit le simulateur effectue un choix aléatoire de la branche à suivre.

Un autre cas est possible : tester toutes les branches. Nous ne prenons pas ici en compte ce cas (nous ne définissons pas de contraintes permettant d'éviter les explosions combinatoires).

Cependant, pour effectuer un choix de branche, il faut déceler le non-déterminisme d'une spécification. Dans cette section, nous n'avons pas abordé la traduction des comportements dans les environnements de simulation. Dans le bilan de ce chapitre, nous définissons nos choix de spécifications pour simuler les comportements. Ces choix intègrent le non-déterminisme.

3.3 Bilan

Dans le bilan de ce chapitre, nous présentons tout d'abord nos choix de langages de spécifications pour leur traitement par la simulation. La mise au point de spécifications par la simulation impose que le simulateur soit spécifique du langage de spécifications utilisé. Le travail développé dans cette thèse ne porte pas sur un langage de spécifications

particulier mais se veut proche des langages les plus usités dans le monde des télécommunications. Les industriels spécifient leurs applications à l'aide de notations semi-formelles, notre travail porte sur ses notations, et plus précisément sur les langages d'interfaces.

Le but de cette thèse est d'offrir une méthode de simulation de la dynamique des applications dès la phase de spécification de celles-ci. Cependant, les langages d'interfaces n'offrent pas de description formelle des comportements. Pour simuler, il faut construire les comportements (cela revient à définir des implantations particulières des applications). Une solution est d'utiliser des méthodes formelles pour les décrire [Fes94]. Ces dernières mettent en évidence le non-déterminisme des applications. Elles permettent un traitement plus aisé de celui-ci.

Dans le chapitre 4, nous donnons un schéma d'intégration de notations formelles des comportements dans les langages d'interfaces. Ce schéma prend en compte la *frilosité* des industriels lorsqu'il faut utiliser des spécifications formelles. En effet, l'utilisation des méthodes formelles demande souvent un long apprentissage que les industriels ne sont pas toujours près à effectuer. Nous reconnaissons que cet investissement n'est pas toujours justifié. Aussi, le schéma proposé se veut être le plus proche des spécifications et du monde objet. Dans le chapitre 5, nous présentons la notation formelle utilisée pour décrire les comportements, celle-ci intègre les concepts objets.

Dans le cadre de la simulation, l'opérationnel des spécifications est important pour valider les fonctions des applications. L'utilisation de spécifications formelles n'enlève pas le problème du choix (non-déterminisme). Suivant les fonctions que l'utilisateur veut simuler, les spécifications peuvent être non déterministes. Le traitement du non-déterminisme est difficile à mettre en oeuvre (voir irréalisable dans certains cas sans raffinement supplémentaire). Les travaux cités dans la sous-section 3.2.1 (traduction de Z en PROLOG [SCT96], restriction du langage VDM-SL [LL91]) montrent qu'il est difficile d'animer des spécifications ambiguës.

Dans l'absolu, nous ne pouvons réduire les ambiguïtés d'une application sans en contraindre son expressivité. De plus, les industriels veulent les préserver pour des raisons de réutilisations des spécifications dans d'autres applications. Cependant, le non-déterminisme contient du déterminisme, c'est-à-dire que l'on peut, dans un cadre ensembliste, considérer que le déterminisme est un cas particulier du non-déterminisme. Nous montrons dans le chapitre 4 qu'il n'est pas possible d'isoler automatiquement le déterminisme.

L'aide à la simulation que nous proposons dans cette thèse permet de mettre en évidence des parties déterministes des comportements des applications. Ces parties sont exécutées au cours du traitement des requêtes.

Chapitre 4

Formalisation et validation des comportements

Ce chapitre présente la problématique de la simulation de la dynamique des applications à partir des spécifications de celles-ci. La dynamique est incluse dans les *comportements*. La formalisation des comportements est une aide importante pour octroyer une vue opérationnelle à la simulation (animer la dynamique des applications). Les langages de spécifications étant divers et variés (cf. section 2.2), nous proposons dans la section 4.1 une approche globale de la formalisation des comportements.

Cependant, la formalisation (la plus abstraite possible) des comportements permet de mettre en évidence les ambiguïtés des applications. L'exemple du comportement *alarmSeverityAssignmentPointerBehaviour*, décrit dans la figure 2.2, montre qu'il faut effectuer un choix de valeur pour l'attribut *alarmSeverityAssignmentProfilePointer*. Dans le cadre de la simulation, le non-déterminisme des comportements doit être pris en compte et traité d'un point de vue opérationnel. En section 4.2, nous présentons notre choix de traitement du non-déterminisme.

4.1 Formalisation des comportements

Dans le domaine des télécommunications, les langages de spécifications sont majoritairement orientés objet. La représentation sous forme d'objets d'une application permet d'intégrer plus facilement les concepts de réutilisation et d'hétérogénéité définis dans la section 2.3. Ce type de représentation permet de mettre en évidence les valeurs caractéristiques des applications (champs ou attributs des objets) et leurs services (comportements dynamiques, méthodes des objets). Entre les notations semi-formelles et les méthodes formelles, ces dernières fournissent une description formelle complète des applications. Cependant, elles ne s'intègrent pas (facilement) aux modèles normalisés de gestion (typage de données, héritage, etc.) [Fes94] [Kel95]. Cette thèse propose principalement d'étendre

les langages d'interfaces. Cependant, nous montrons que notre travail peut aussi s'appliquer à tout langage de spécifications. Dans cette section, nous présentons une méthode de formalisation des comportements dans les langages d'interfaces.

[Maz97] propose un langage *idéal* de spécifications des comportements des applications. Ce langage se veut être une synthèse des langages de spécifications et regroupe les concepts permettant de structurer et de formaliser les comportements (cf. [Maz97] section 2.4). Il est construit à partir du modèle de référence ODP. Deux de ses concepts sont intéressants à introduire dans la spécification abstraite des comportements :

- possibilité d'exprimer le non-déterminisme ;
- utilisation d'un langage d'assertions.

Le non-déterminisme est une propriété inhérente des systèmes répartis [Maz97]. En effet, les applications définies sur ces systèmes peuvent être en perpétuelle mutation, aussi le non-déterminisme permet, entre autre, l'indépendance par rapport aux implantations des applications. On doit pouvoir l'exprimer dans une spécification abstraite.

Les comportements d'une application représentent les propriétés qu'une application doit respecter au cours de sa vie. L'utilisation d'un langage d'assertions permet de définir les propriétés d'une application. Notre approche de la formalisation des comportements se veut la plus abstraite possible. Les langages d'assertions fondés sur le langage des prédicats du premier ordre répondent à cette exigence. Ils intègrent le non-déterminisme.

Ayant fait notre choix de langage pour formaliser les comportements, nous montrons comment ce choix s'intègre dans les langages d'interfaces. Nous présentons aussi sa compatibilité aux concepts objet.

4.1.1 Structuration des comportements

Les comportements représentent les aspects d'une application : *que fait elle quand ...* ou *elle doit respecter la propriété ...* ou encore *comment elle réagit pour l'événement ....* Ces aspects peuvent être classés en deux catégories :

- les comportements statiques ;
- les comportements dynamiques.

Avant de présenter ces deux catégories, nous rappelons tout d'abord que les comportements permettent de décrire les *propriétés* que doivent respecter les applications. Les propriétés d'une application peuvent se modéliser à l'aide de *la notion d'état*. L'état d'une application à un instant t est l'ensemble des *valeurs caractéristiques* de l'application. En

section 5.2, nous lui donnons une définition plus complète et *orientée objet*. Dans la suite de ce chapitre, nous le considérons comme un ensemble de couples $(variable, valeur)$, où *variable* est l'identifiant dans les spécifications d'une valeur caractéristique (aussi nommée *variable d'état*) et *valeur* sa valeur. Les comportements d'une application sont définis sur les valeurs caractéristiques de l'application et donc sur les états successifs de l'application (état à l'instant t, $t+1$, etc).

Les comportements statiques représentent les propriétés qu'une application doit préserver tout au long de son existence (tout état de l'application doit respecter ces propriétés). On les dénomme souvent *invariants de l'application* [HJ89b]. Dans un monde objet, l'invariant d'une application est l'ensemble des invariants des objets présents dans l'application.

Les comportements dynamiques représentent les services[1] des applications. Ils modifient une partie des propriétés d'une application, comme, par exemple, le changement de valeur d'un champ d'un objet de l'application ou encore la création d'un nouvel objet. En terme d'état, un service permet d'en changer (passage de l'instant t à $t+1$). On peut représenter les comportements dynamiques comme suit :

- description de l'état de l'application avant réalisation d'un service : *pré-conditions du service* [HJ89b] ;
- description de l'état de l'application après réalisation d'un service : *post-conditions du service* [HJ89b].

Un couple de pré/post-conditions permet de spécifier un service d'une application. Ce dernier peut être déclenché soit par :

- l'utilisateur. Il décide à quel moment tel service doit être exécuté (modèle de la *machine à boutons*) ;
- des événements particuliers. Par exemple, un éclair foudroyant un équipement réseau déclenche l'envoi d'une alarme vers le gestionnaire du réseau (modèle des *systèmes réactifs*).

Les événements particuliers déclenchant un service peuvent être spécifiés. Certains langages, tels GDMO+ [GDM97] et BL [Maz97], permettent de les décrire. Les comportements correspondant sont nommés *triggering conditions* (conditions de déclenchement). Dans le chapitre 7, nous présentons ces conditions au sein du langage Q-GDMO-GRM.

Le schéma *invariants, pré/post-conditions* structure les comportements indépendemment des implantations et reste proche des spécifications. Dans le chapitre 7, nous montrons son intégration dans le langage Q-GDMO-GRM.

1. les méthodes des objets des applications.

Les langages d'interfaces possèdent une sémantique. Le schéma *invariants, pré/post-conditions* doit la respecter. Dans la sous section suivante, nous montrons que ce schéma de représentation des comportements préserve la sémantique des langages, c'est-à-dire leurs concepts objet.

4.1.2 Le schéma *invariants, pré/post-conditions* et l'orienté objet

Un langage orienté objet permet de matérialiser une application à l'aide d'objets. Ces objets sont spécifiés par des *classes*. Une classe regroupe une description des attributs et des services. Avec l'objet est apparue la notion d'héritage. Elle représente la spécialisation d'une classe. Une définition de l'héritage est :

Définition 2 *Cette technique permet de remplacer un objet A par un objet B plus complexe mais ayant au moins le même comportement que A.*

Dans [Ame87] et [Ame91], une distinction est faite entre l'héritage d'objet et de comportement d'objet. Celui d'objet permet de *partager* du code ou des implantations diverses des objets. Par exemple, une classe peut hériter de tout ou d'une partie des attributs et services de ses classes mères. Un langage de spécifications illustrant l'héritage d'objet est GDMO. Une classe est représentée par un ensemble de *packages* obligatoires ou conditionnels. Une classe fille n'hérite pas obligatoirement des packages conditionnels de ses classes mères.

Le schéma *invariants et pré/post-conditions* doit préserver l'héritage de comportements des classes d'objets. [Ame87] le dénomme *sous-typage*. [Boo92] a défini l'*héritage de spécialisation* qui, à notre sens, préserve les propriétés des classes mères au sein d'une classe fille. Nous le présentons dans la figure 4.1.

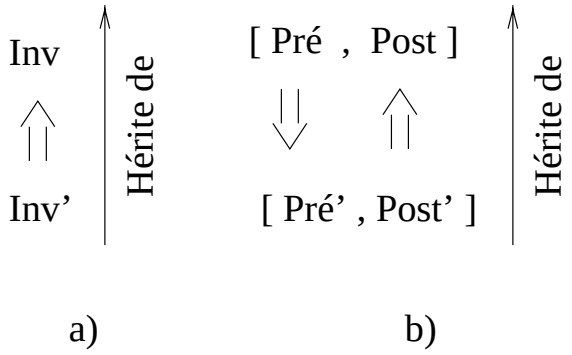


FIG. 4.1 – *Mécanisme d'héritage de spécialisation a) des invariants d'objets b) des pré/post-conditions*

Le langage EIFFEL [Mey92] offre un mécanisme de composition des invariants et des pré/post-conditions prenant en compte l'héritage de spécialisation. Ce mécanisme est

résumé par les trois expressions suivantes :

$$Inv' = Inv \wedge InvAjouté \tag{4.1}$$

et

$$Pré' = Pré \vee PréAjoutée \tag{4.2}$$

et

$$Post' = Post \wedge PostAjoutée \tag{4.3}$$

Ces trois expressions respectent les implications définies dans la figure 4.1. En effet, Inv' et $Post'$ sont plus forts ou équivalents respectivement à Inv et $Post$ car, par définition de la conjonction, $Inv \wedge InvAjouté$ et $Post \wedge PostAjoutée$ sont plus forts ou équivalents respectivement à Inv et $Post$. $Pré'$ est plus faible ou équivalente à $Pré$ car, par définition de la disjonction, $Pré \vee PréAjoutée$ est plus faible ou équivalente à $Pré$.

4.2 Validation des comportements

La simulation de spécifications revient à définir une implantation partielle des applications. Lors de la simulation, on crée des objets et on *exécute* des services sur ces objets (cf. sous-section 3.2.2). Les langages d'interfaces permettent de concevoir facilement les classes concrètes des objets à partir de leurs spécifications. Par contre, la conception des services est plus délicate.

La spécification des services est effectuée à l'aide d'un couple de pré/post-conditions définies dans un langage d'assertions quelconque[2]. La simulation se voulant opérationnelle, il faut déduire à partir des couples de pré/post-conditions une *exécution* des services des objets.

[HJ89a] montre que les spécifications ne sont pas nécessairement exécutables. [Fuc92] rétorque qu'il est préférable que les spécifications soient exécutables. Ces deux articles décrivent les comportements à l'aide de langages d'assertions fondés sur le langage des prédicats du premier ordre. [Fuc92] précise qu'un langage déclaratif est suffisant pour définir des spécifications exécutables intégrant le non-déterminisme des assertions. Pour cela, il impose des choix de représentation des données, des règles d'induction, etc. En fait, il effectue un raffinement de spécifications abstraites vers des spécifications concrètes par réécriture. Cette dernière permet de *penser autrement* le problème traité. Cependant, elle n'est pas toujours applicable.

Le travail présenté dans [Fuc92] permet de raffiner les spécifications abstraites vers des spécifications exécutables. [Mor90] montre que le raffinement des spécifications permet

2. dans la suite de cette section, les assertions sont définies à l'aide du langage des prédicats du premier ordre.

soit de tester tous les cas soit de contraindre les domaines de définitions des variables d'états pour restreindre le nombre de cas.

Dans cette thèse, nous décidons de ne pas contraindre les spécifications pour la simulation. Nous partons donc d'un langage pouvant être non exécutable (comme [HJ89a] et au contraire de [Fuc92]). Nous laissons le soin aux utilisateurs de raffiner leurs spécifications. Par contre, nous offrons la possibilité d'un traitement automatique lors de la simulation de la partie exécutable d'un service à partir de ces spécifications abstraites. Ce traitement est lié à l'étude du non-déterminisme.

4.2.1 De l'abstrait à l'opérationnel

Dans la section 4.1, nous avons mis en évidence l'intérêt de préserver le non-déterminisme dans les spécifications des applications. Nous n'en avons pas donné une définition précise. Dans le contexte de la thèse, un service est non-déterministe s'il offre, pour un même état de départ, le choix entre différents états d'arrivée. D'un point de vue mathématique, le non-déterminisme est une relation des états dans les états.

Cependant, comme nous l'avons déjà signalé, le non-déterminisme contient le déterminisme. D'un point de vue mathématique, ce dernier n'est pas une relation mais une fonction des états dans les états. Dans le contexte de la thèse, il n'y a qu'un seul état d'arrivée. Un avantage du déterminisme est de définir une partie opérationnelle au sein des spécifications. Nous le montrons à partir des assertions définissant les services des applications.

Le schéma de spécifications *invariants, pré/post-conditions* permettent de représenter les états (du moins une partie) des applications. La relation entre un état et une assertion s'effectue par le biais des variables d'états. Les variables d'états y sont représentées par des variables dites *logiques* et *libres*. En section 5.2, nous donnons une définition plus précise de la représentation des variables d'états dans les langages d'assertions. Nous rappelons simplement que ces langages sont fondés sur celui des prédicats du premier ordre, aussi, notre discours tend à se rapprocher de la logique.

Les assertions nous apporte des informations relatives aux valeurs des variables d'états. Par exemple, l'assertion [3] :

$$x = 1 \wedge 2 < y < 7 \tag{4.4}$$

précise que la variable d'état x est de valeur 1 dans l'état que cette assertion représente et celle de y est comprise dans l'ensemble $\{3, 4, 5, 6\}$ [4]. La post-condition d'un service peut ainsi offrir des informations sur les nouvelles des variables d'états après son exécution.

3. $\wedge$ représente le connecteur de conjonction, *et logique*.
4. nous considérons que x et y sont des entiers.

Dans l'assertion précédente, une seule valeur est possible pour la variable x au contraire de y. Aussi, cette assertion représente au moins quatre états (quatre valeurs sont possibles pour y), elle est non-déterministe. Par contre, lors de la simulation, on peut espérer effectuer automatiquement certaines affectations de variables d'états à partir du déterminisme de la post-condition. Dans l'exemple de l'assertion 4.4, on peut déduire l'affectation de la variable d'état x à 1.

Dans la suite de la thèse, nous nous permettons d'extrapoler le déterminisme sur les assertions (le déterminisme porte en fait sur les changements d'états des applications). Pour synthétiser, nous donnons les deux définitions suivantes :

Définition 3 *Une assertion est dite déterministe si toute variable d'état référencée est associée à au plus une valeur*[5].

Définition 4 *Une assertion est dite non-déterministe s'il existe au moins une variable d'état référencée à laquelle on peut associer plusieurs valeurs.*

Pour pouvoir effectuer des affectations, il faut pouvoir isoler la partie déterministe d'une assertion. En d'autres termes :

Existe-t-il un algorithme pour tout langage d'assertions qui permet de calculer la partie déterministe d'une assertion définie dans ce langage ?

Extraire la partie déterministe d'une assertion revient à la décomposer en deux parties et à montrer qu'une de celles-ci est déterministe. Pour effectuer ce découpage, il faut définir un *calcul* sur les assertions. Cependant ce calcul n'est pas *décidable* sur les langages fondés sur celui des prédicats du premier ordre et contenant l'arithmétique[6] [LRd93].

4.2.2 Restriction du déterminisme

Nous avons mis en évidence que le déterminisme d'une assertion permet d'associer une valeur unique à une variable d'états. Le calcul du déterminisme étant indécidable, il nous faut partir des langages pour l'exhorter. Dans ces derniers, l'association entre variables et valeurs s'effectue par le biais de prédicats. Sans imposer un langage, un prédicat standard et déterministe est l'égalité (nous n'en connaissons pas d'autre qui soit aussi présent dans la plupart des langages d'assertions).

5. une assertion peut ne pas associer de valeur à une variable, par exemple, $x = 1 \wedge x = 2$ n'associe aucune valeur à x.

6. nous considérons que les langages d'assertions utilisés pour formaliser les comportements contiennent au minimum l'arithmétique.

Le prédicat d'égalité est déterministe. Sa définition primaire est celle de l'identité[7] que nous devons en partie à Leibniz et dont voici une définition plus précise tirée de [GG91] :

Définition 5 *Soient A et B deux objets (valeurs) désignés par les références (termes) a et b dans une assertion, si A et B sont identiques (représenté par a = b) alors a et b sont interchangeables.*

Pour illustrer cette définition, nous prenons le symbole = pour représenter le prédicat d'égalité. Soient x et y des variables libres (x et y sont des variables d'états) et 6 une constante, les assertions suivantes sont équivalentes :

$$x = y \wedge y = 6 \ , \ x = 6 \wedge y = 6 \tag{4.5}$$

Comme pour le calcul du déterminisme (cf. sous-section 4.2.1), il faut isoler les assertions atomiques définies sur le prédicat d'égalité. Au contraire du premier calcul, le problème à résoudre est *syntaxique*.

Cependant, nous ne cherchons pas seulement à isoler les assertions atomiques définies sur le prédicat d'égalité mais aussi celles qui sont déterministes dans leur contexte. Prenons les exemples simples des assertions suivantes[8] :

$$x = 1 \wedge y = 2 \text{ et } x = 1 \vee x = 2 \tag{4.6}$$

La première assertion est déterministe au contraire de la deuxième. Ces exemples montrent que l'on doit aussi prendre en compte le déterminisme des connecteurs et des quantificateurs (cf. section 5.1). En effet, le connecteur $\wedge$ préserve le déterminisme de ses opérandes à la différence du connecteur $\vee$: dans la deuxième assertion de 4.6, un choix entre les valeurs 1 et 2 pour la variable x est nécessaire[9].

Cependant, toutes les assertions atomiques basées sur le prédicat d'égalité ne sont pas déterministes. En effet, $x + y = 5$ n'est pas déterministe car on peut associer des valeurs différentes à x et y. Par contre, l'assertion (où x et y sont des entiers) :

$$x + y = 5 \wedge y = 2 \tag{4.7}$$

est déterministe (x prend la valeur unique 3 et y la valeur unique 2). Pour calculer ce dernier, il faut intégrer le calcul d'un ordre partiel sur l'ensemble des variables d'états

7. dans la suite du document, nous utilisons le terme *égalité* pour signifier l'*identité*.

8. $\vee$ représente le connecteur de disjonction, *ou logique*.

9. en sous-section 6.1.4, nous donnons des exemples plus précis de connecteurs et quantificateurs déterministes et non déterministes.

référencées dans les assertions [Gal86]. Dans ce cas, l'implantation de l'algorithme final du calcul du déterminisme d'une assertion est dépendante des langages, voire les restreint. En effet, dans le cas de l'assertion 4.7, la fonction inverse de l'addition (c'est-à-dire la soustraction) doit être définie. En terme générique, cela impose que pour toute fonction d'un langage, son inverse y soit définie, c'est-à-dire que toute fonction est une bijection [Gal86].

Une solution pour préserver un calcul purement syntaxique du déterminisme d'une assertion est de l'exhiber *explicitement*. Dans la sous section, nous présentons des solutions dans ce sens.

4.2.3 Représentation explicite de l'aspect opérationnel

Représenter explicitement une partie animable d'une spécification permet un calcul syntaxique (à la compilation) de cette partie. La solution la plus simple est de la décrire explicitement en langage de programmation. Certaines méthodes formelles utilisent ce type de spécifications (LOTOS, SDL, BL, etc.). Une autre solution est de restreindre le langage de spécifications en un langage exécutable. Par exemple, VDM-SL IPTES-IV est un sous ensemble exécutable de VDM-SL (cf. sous-section 3.2.2). Utiliser un langage de programmation contraint l'expressivité des spécifications et impose une implantation particulière des applications.

Une autre approche est définie dans le langage de spécifications LARCH [GHG+93]. Cette approche explicite les variables d'états modifiées au cours de l'exécution d'une méthode (d'un service). LARCH décrit sous forme de pré/post-conditions (*requires* et *ensures*) les méthodes des types abstraits (LARCH est un langage algébrique). Avant la déclaration de la clause *ensures*, une clause *modifies* spécifie les variables d'états modifiées. Ainsi, on sait explicitement quelles variables d'états changent de valeurs au cours de l'exécution du service.

Ces deux méthodes pour spécifier explicitement l'aspect opérationnel mettent en évidence les deux points suivants :

- déclaration d'opérations de modifications des variables d'états ;
- déclaration des variables d'états à modifier.

Nous devons ainsi augmenter l'expression des post-conditions en intégrant les deux points précédents. Les extensions proposées ne doivent pas modifier la sémantique associée aux langages d'assertions, ceci afin de préserver une abstraction maximale. Nous dénommons une telle extension : *déterminisme explicite*. Dans le chapitre 6, nous présentons un déterminisme explicite [CD97] [Cou97].

Chapitre 5

Langages d'assertions et état d'une application

Ce chapitre permet de faire la liaison entre la représentation des comportements par des langages d'assertions et les données des applications (les objets des applications). Dans la section 4.1, nous avons montré comment les langages d'assertions peuvent être intégrés dans les langages de spécifications orientés objet. Cependant, les comportements font référence aux objets des applications. Dans la section 5.1, nous rappelons la syntaxe générale des langages d'assertions et nous donnons aussi les liens sémantiques entre eux et les objets des applications.

La simulation des comportements des applications permet de passer d'un état à l'autre (cf. section 4.2). Un état d'une application regroupe les valeurs caractéristiques des objets de l'application. Il est représenté par un ensemble de couples $(variable, valeur)$ (cf. sous-section 4.1.1). La *variable* d'un couple est une référence que l'on retrouve dans les comportements. Après une définition générale de la notion d'état, la section 5.2 décrit le lien entre les langages d'assertions et les variables d'états, ainsi que celui un état et le monde objet.

5.1 Les langages d'assertions

Les langages d'assertions sont souvent fondés sur le langage des prédicats du premier ordre. Cela ne signifie pas que ces langages respectent la logique du premier ordre, mais qu'ils utilisent le même alphabet. Pour répondre aux attentes *opérationnelles* des spécifieurs, on augmente cet alphabet pour décrire, par exemple, des itérations (nous le présentons dans la sous section 5.1.1).

Après le rappel de l'alphabet du langage des prédicats du premier ordre, nous donnons une description orientée objet de cet alphabet. En effet, les comportements mani-

pulent des objets. Nous montrons tous d'abord que ces derniers sont des variables d'états des applications. Nous donnons ensuite leurs syntaxes dans les langages d'assertions.

5.1.1 Une syntaxe des langages d'assertions

La syntaxe que nous présentons dans cette sous-section est en fait la réunion de celle *des langages du premier ordre* [GG91] [LRd93]. Ces derniers ont l'avantage de fournir un alphabet proche de la langue naturelle et des langages de programmation. Ils sont un bon intermédiaire entre ces deux classes de langages. Un tel langage permet de définir des assertions sur les relations entre données et en incorporant des fonctions sur celles-ci. Ces fonctions peuvent être par exemple celles de l'arithmétique. Un langage de premier ordre est constitué d'un ensemble fini de fonctions.

Un langage du premier ordre a pour alphabet :

- un ensemble de connecteurs : symboles permettant d'unir des assertions logiques entre elles ;
- un ensemble de variables : symboles représentant des valeurs quelconques ;
- les quantificateurs $\forall$ et $\exists$;
- un ensemble de prédicats : symboles de relations prenant valeur dans l'ensemble de vérité {vrai, faux} et d'arité non nulle ;
- un ensemble de fonctions : symboles représentant des valeurs constantes, les fonctions d'arité nulle représentent une et une seule valeur et sont nommées constantes d'individus.

De cet alphabet, on infère des *termes* et des *formules*. Nous donnons la grammaire suivante des termes :

terme	::=	var | fonct
fonc	::=	const_individu | symb_fonc(terme (, terme)*)
const_individu	::=	1 | "chaîne de caractère" | etc.
symb_fonc	::=	+ | random | etc.

où var représente une variable. Nous présentons sa grammaire dans la sous section suivante. Notons que dans la suite de cette thèse, nous utilisons les notations infixées (2+3, etc). Nous signalons aussi que nous ne prenons pas en compte dans les grammaires

de cette section les priorités entre les fonctions (+ plus prioritaires que − par exemple). Cette information n'apporte rien à notre sujet.

La grammaire des formules est la suivante :

$$
\begin{array}{lll}
\text{formule} & ::= & \text{prédicat(terme (, terme)}^*\text{)} \mid \text{f\_complexe} \\
\text{prédicat} & ::= & = \mid < \mid \text{etc.} \\
\text{f\_complexe} & ::= & \text{connecteur(formule (, formule)}^*\text{)} \\
 & & \mid \forall \text{ var formule} \\
 & & \mid \exists \text{ var formule} \\
\text{connecteur} & ::= & \wedge \mid \vee \mid \text{etc.}
\end{array}
$$

Les remarques effectuées sur les fonctions (notations infixées, etc) sont aussi valides pour les prédicats et connecteurs.

Pour décrire le langage des prédicats du premier ordre, nous avons repris les termes logiques. Dans la suite, nous remplaçons le terme *formule* par le terme *assertion*. Nous faisons cette distinction pour la raison suivante : l'intégration du monde objet au monde logique (et vice versa) ne préserve pas obligatoirement (voire pas du tout) la sémantique apportée par le monde logique. Dans cette thèse, nous utilisons l'alphabet des langages du premier ordre pour faciliter l'expression des fonctions des applications, par contre, la *logique* n'est pas toujours respectée dans le sens théorique du terme.

Pour des besoins opérationnels, on augmente l'alphabet du langage des prédicats du premier ordre pour incorporer, par exemple, la notion d'itération. Au contraire des langages du premier ordre, les langages d'assertions traités dans ce document portent sur des domaines finis de valeurs et supportent la notion de type des valeurs. Les quantifications universelle et existentielle peuvent être considérées comme des boucles sur des ensembles finis. Nous modifions la règle de grammaire f_complexe en remplaçant les deux dernières alternatives par :

$$\forall \text{ var} \in \text{terme} \mid \text{formule et } \exists \text{ var} \in \text{terme} \mid \text{formule}$$

où le symbole $\in$ représente le prédicat d'appartenance.

Dans la règle précédente, nous avons introduit les notions d'ensemble et d'itération sur un ensemble. Ces notions nous permettent d'introduire l'utilisation de types complexes tels que les tableaux ou les listes qui offrent la possibilité de représenter des collections de valeurs. Cette extension offre la possibilité d'associer ces dernières aux attributs des objets et ainsi de pouvoir *itérer* sur celles-ci au sein des spécifications. Cependant, il nous faut définir une syntaxe pour décrire des valeurs de ce type. La notation "_[_]" représente

l'accès à un élément d'une collection. Par exemple, l'expression $f[5]$ s'interprète comme l'accès au cinquième élément de la collection représentée par la variable d'état f. Nous ne traitons pas les collections multiples ($f[5][3]...$).

Le langage global d'assertions que nous décrivons dans cette sous section est, d'un point de vue logique, d'ordre supérieur. Dans la suite de cette thèse, nous n'effectuons pas de preuves de spécifications mais de la validation de spécifications par simulation. Nous n'utilisons donc de tels langages que pour faciliter l'expression des propriétés des applications de gestion des réseaux de télécommunications.

5.1.2 Langages d'assertions et objet

Dans cette sous-section, nous montrons comment un langage d'assertions peut représenter les objets d'une application. Dans la sous-section 4.2.2, nous avons utilisé les symboles x et y (équation 4.5) pour représenter des *objets*. Or, la valeur représentée par un objet est l'ensemble des valeurs représentées par ses attributs. Pour représenter individuellement leurs valeurs, il faut fournir un opérateur (une fonction) permettant d'accéder à chacune d'elle.

Les langages orientés objet utilisent souvent les deux représentations infixées suivantes de l'opérateur d'accès : "_._" et "$\_ \rightarrow \_$". Nous choisissons la notation "_._".

Dans quel ensemble représentatif de l'alphabet des langages d'assertions l'opérateur "_._" appartient-il?

On peut le classer soit dans l'ensemble :

- des variables d'états (une variable d'état peut être $o.a$) ;
- des fonctions du langage d'assertions (cf. sous-section 5.1.1).

L'opérateur "_._" s'interprète comme une fonction à deux éléments que l'on peut décrire ainsi :

$$\_.\_ : OBJECT \times IDENT \mapsto ANY \tag{5.1}$$

où $OBJECT$ est le type générique des objets, $IDENT$ est le type des identificateurs des attributs d'objets et ANY est le type représentant tout type de données (classe d'objet, entier, réel, etc.).

La solution la plus évidente est de considérer l'opérateur "_._" comme une fonction. Cependant, par définition des fonctions, on ne peut considérer cet opérateur ainsi. En

effet, les valeurs des attributs d'objets sont la partie variable des objets. Pour un objet o donné et son attribut a, la valeur $o.a$ peut être différente à deux instants distincts, c'est-à-dire pour une même entrée de l'opérateur "_._" il existe deux ou plusieurs sorties. La *fonction* "_._" n'est pas constante (d'un point de vue logique, la théorie du langage est alors modifiée à chaque nouvelle valeur ajoutée). Notons que la *fonction* "_[_]", définie dans la sous-section 5.1.1, n'est pas constante pour les mêmes raisons.

La seule solution restante est de considérer l'expression $o.a$ comme une variable d'état. Dans un langage d'assertions, les variables permettent de représenter une valeur *quelconque*[1]. Dans notre cas, les variables d'états sont dépendantes du contexte dans lequel elles sont déclarées. Par exemple, dans l'assertion :

$$x + 1 = 4 \tag{5.2}$$

x doit prendre valeur dans l'ensemble des entiers (si l'on considère que l'opérateur + est l'addition entre deux entiers). Par contre, x peut prendre n'importe quelle valeur dans l'ensemble des entiers à tout instant (l'assertion 5.2 n'est pas obligatoirement valide).

Dans les langages d'assertions, les variables peuvent être liées ou non aux quantificateurs. Deux catégories de variables sont à prendre en compte, les variables *liées* et *libres*. Soit la définition suivante [LRd93] :

Définition 6 *Une variable x est liée dans une formule (assertion) F si elle est sous la portée d'un quantificateur, sinon elle est dite libre dans la formule (assertion) F.*

Une variable $o.a$ ne peut être une variable liée. Une variable liée à un quantificateur représente un ensemble de valeurs. C'est une factorisation des propriétés que doivent respecter ces valeurs. D'un point de vue sémantique, la durée de vie d'une telle variable est celle de l'exécution de la quantification. Une variable d'états à une durée de vie équivalente (dans l'absolu) à celle de l'application.

En résumé, nous donnons la grammaire des variables suivante :

$$\text{var} \quad ::= \quad \textbf{SYMB} \mid \text{var\_libre}$$

où **SYMB** représente une suite de caractère. Il représente aussi dans cette règle une variable liée. Dans la sous-section suivante nous donnons la description des variables libres.

5.1.3 Description des variables libres *objet*

Les variables libres représentent des valeurs d'attributs d'objets. Dans la sous-section 5.1.1, nous avons introduit l'accès à des collections de valeurs contenues dans des attributs

1. en logique pure.

(notation "_[_]"). Nous considérons que chaque élément d'une collection représentée par un attribut est une variable libre.

Un autre point que nous n'avons pas encore abordé concernant l'opérateur "_._" est la possibilité de le composer. Prenons le cas d'un objet o possédant un attribut a_1. Supposons que la valeur liée à l'attribut a_1 soit un objet possédant un attribut a_2, alors il faut donc considérer que le terme $(o.a_1).a_2$ peut être une variable libre.

Le paragraphe précédent présente la composition de l'opérateur "_._" *par les attributs*. On pourrait aussi composer *par les objets*. Prenons l'exemple du terme $o_2.(o_1.a)$. Son interprétation est la suivante : il représente la valeur de l'attribut x de l'objet o_2, où la valeur de l'attribut a de o_1 représente le label de x. Pour des raisons de simplifications, nous ne considérons pas la composition de l'opérateur "_._" par les objets mais seulement par les attributs.

Nous représentons la syntaxe des variables libres par la grammaire suivante :

var_libre ::= objet.**SYMB** **{**[terme**}**]

objet ::= **SYMB** | var_libre

Dans la sous section 2.3.2, nous avons montré l'importance des relations entre objets. A partir d'un objet, on peut accéder à un autre objet. Les langages de spécifications existants offrent diverses méthodes pour spécifier les relations. Par exemple, :

- on internalise les relations dans les objets, c'est-à-dire qu'à chaque relation on associe un attribut de chaque objet concerné contenant la liste des autres acteurs de la relation ;
- on crée un objet relation qui met en correspondance les objets en relation.

Dans ces deux cas, l'accès aux objets en relations revient à accéder à un attribut.

5.2 L'*état* d'une application

Dans la sous-section 4.1.1, nous avons introduit succinctement la notion d'état. Nous donnons la définition d'un état que l'on trouve dans la littérature. La notion d'état est introduite comme suit dans [Mor90] (les termes en gras dans cette définition ne sont pas d'origine, nous avons reformulé cette définition avec le vocabulaire de ce document) :

Définition 7 *L'état d'une* **application** *est une fonction des noms vers les valeurs. Les noms sont appelés* **variables d'états** *et les valeurs sont définies dans les* **domaines de valeurs de l'application**.

Nous allons lier cette définition d'un état aux langages d'assertions. En effet, le schéma *invariants, pré/post-conditions* permet de décrire la dynamique d'une application, c'est-à-dire l'évolution de son état. Les langages décrivent donc aussi l'état d'une application. Nous allons introduire la notion d'*évaluation d'une assertion dans un état.*

Pour lier un état et une assertion, il faut lier les termes *variables d'états* (cf. définition 7) et *variables libres* (cf. sous-section 5.1.3). Rappelons que les assertions sont introduites pour formaliser les invariants et pré/post-conditions représentant les comportements. La simulation des comportements revient donc à les évaluer. Leurs évaluations sont effectuées à partir de l'état courant de l'application.

L'évaluation d'une assertion revient à remplacer les variables libres par des valeurs. Ces dernières sont accessibles au travers de l'état par le biais des variables d'états. Il faut donc fournir une description des variables d'états compatible avec celle des variables libres définie dans la sous-section 5.1.2.

Une variable libre (simple) est représentée par $o.a$ où o est une description d'un objet et a est l'identifiant d'un attribut de l'objet. On peut alors représenter une variable d'état comme un couple $(objet, attribut)$. L'*attribut* représente l'identifiant d'un attribut. Par contre, il est plus complexe de représenter le terme $objet$. Trois problèmes s'offrent à nous :

- le nommage des objets ;
- la notion de collection ;
- la composition de l'opérateur "_._".

Le nommage des objets

Lors de la simulation d'une application (comme lors de son exécution), on instancie des classes d'objets. Ces objets communiquent entre eux par services interposés. Pour accéder à ces objets pendant l'exécution d'une application, il faut offrir un service de nommage. Dans la section 2.1, nous avons décrit des modèles de gestion d'applications de télécommunications (OSI, IAB, ODP, etc.). Tous ces modèles préconisent l'utilisation d'un service de nommage des instances d'une application.

Un exemple d'un tel service est celui proposé par le modèle OSI [ITU92a]. Son but est d'assurer un nom unique dans l'application pour chaque objet. Pour cela, on conçoit un *arbre de contenance*, c'est-à-dire qu'un objet est lié à un et un seul objet racine. La construction de cet arbre s'effectue par le biais d'attributs ad hoc. La norme [Int89] décrit le principe du nommage d'une instance. Chaque instance possède un attribut de nommage (RDN[2]) et un attribut contenant une liste de *couples* $(attribut, valeur)$. Cet attribut liste les RDN de ses pères dans l'arbre de contenance, plus son propre RDN. Cet attribut est

2. Relative Distinguished Name

appelé DN (Distinguished Name) de l'objet. A la création d'un objet, la valuation des attributs de nommage doit respecter l'unicité des noms d'objets dans l'application.

Ce service de nommage est lourd (mais sûr) à mettre en place. Dans le cadre de cette thèse, nous ne l"avons pas implanté. Les objets, manipulés dans le simulateur présenté dans le chapitre 7, sont accessibles par des chaînes de caractères. Ces chaînes sont les clés d'accès aux objets référencés dans une base interne à l'environnement d'exploitation du simulateur.

En résumé, une variable d'état peut être représentée par un couple $(objet, attribut)$ où $objet$ est une chaîne de caractères représentant le nom d'une instance.

Les attributs représentant une collection

En sous-section 5.1.2, nous avons défini que la valeur d'un attribut peut être une collection de valeur. De plus, en sous-section 5.1.3, nous précisons qu'une variable libre peut représenter un élément d'un attribut de type collection. Une variable d'état peut l'être aussi.

Nous proposons d'associer à l'état d'une application et à la notation "_[_]" une fonction (ou notation) qui permet d'accéder à un élément d'une collection. En sous-section 6.2.2, nous présentons une telle notation.

La composition de l'opérateur "_._"

La possibilité de composer l'opérateur "_._" dans une variable libre (cf. sous-section 5.1.3) impose un traitement particulier de l'évaluation des variables libres composées à partir de l'état courant. La technique proposée dans cette thèse est la même que celle du traitement des attributs représentant des collections. Nous définissons une fonction d'accès définie sur l'état. Cette fonction prend en entrée le couple $(objet, attribut)$ et rend sa valeur dans l'état.

Pour traiter la composition, cette fonction est récursive. Dans le cas où l'$objet$ en entrée est un objet composé $(o.a)$, on effectue un appel récursif à la fonction sur cet objet composé. Dans la sous section 6.2.2, nous présentons une fonction qui permet d'accéder à un élément d'un état.

Chapitre 6

Un déterminisme explicite, le :=

Le déterminisme explicite que nous présentons est une extension des langages d'assertions. Cette extension est représentée par la définition d'un nouveau prédicat représenté par le symbole :=. La sémantique de ce nouveau prédicat reprend celle du prédicat d'égalité en la contraignant. Dans la section 6.1, nous présentons la sémantique du prédicat :=.

Le traitement du déterminisme permet de rendre opérationnelle une partie des comportements au niveau de la spécification. Dans la section 6.2, nous présentons la sémantique opérationnelle du prédicat :=. Elle est la description de l'algorithme de traitement du déterminisme explicite.

Le traitement sémantique du déterminisme explicite ne prend en compte qu'une partie des post-conditions. La partie non déterministe doit cependant être traitée par la simulation. Dans la section 6.3, nous montrons comment utiliser les résultats de algorithme présenté dans la section 6.2 pour simuler le non-déterminisme des post-conditions.

6.1 Le prédicat :=, une restriction du prédicat d'égalité

Le déterminisme explicite d'une assertion doit permettre d'associer à une séquence de n variables d'états une séquence de n valeurs (respect du déterminisme). Ne voulant pas définir un nouveau symbole de prédicat, nous préférons étendre l'existant à l'aide de prédicats standards. De plus, pour ne pas perturber les utilisateurs, nous nous sommes imposé la contrainte de ne pas modifier la sémantique des prédicats standards choisis. Un nouveau prédicat possède une sémantique restreinte du prédicat standard.

Pour cette section, nous avons fait le choix dans nombre de cas de ne pas considérer le monde objet. Cette omission volontaire s'applique sur la représentation des variables

d'états. En sous-section 5.1.2, nous avons donné leur règle de construction ($o.a$, $(o.a).b$, etc.). Au lieu de les représenter ainsi, nous les représentons par des lettres (x, y, etc). Cela simplifie la lecture et les notations. Cependant, lorsque la sémantique touche à la représentation objet des variables d'états, nous revenons explicitement à celle-ci.

Dans un cadre déterministe, nous nous intéressons à l'association d'une et une seule valeur à une variable d'états. Un sous ensemble des assertions atomiques définies sur le prédicat d'égalité permet d'associer à une variable d'états une valeur unique. Cet ensemble est constitué des assertions du type :

$$x = E \text{ et } E = x \tag{6.1}$$

où x est une variable d'états et E est une expression.

Nous ne pouvons pas nous satisfaire de ce sous ensemble pour exprimer le déterminisme explicite. En effet, prenons l'assertion $x = y$ (x et y sont des variables d'états) : cette assertion apporte l'information que x et y ont la même valeur, et non celle précisant la modification de l'une de ces variables (est-ce x ou y qui est modifiée, ou les deux?). Dans notre choix de ne pas calculer le déterminisme et *a fortiori* une partie précise de celui-ci, il nous parait important d'isoler explicitement les assertions qui modifie qu'une variable d'états. La solution que nous avons adoptée est le prédicat :=.

6.1.1 Le prédicat :=

Le symbole de prédicat déterministe que nous avons choisi est celui défini dans [GDM97], le := (on retrouve dans ce symbole le symbole traditionnel d'égalité =). Sur l'exemple de l'assertion $x = y$, où x et y sont des variables d'états, il faut choisir soit x ou soit y comme variable à modifier. L'introduction du symbole := permet de faire un choix sur la valeur à affecter. Le sens de modification va de la gauche vers la droite. Par exemple, l'assertion $x := y$ s'interprète :

x prend pour valeur la valeur de y.

Les assertions atomiques basées sur le := sont de la forme générale :

$$x := E \text{ où } x \text{ est une variable d'états et } E \text{ une expression.} \tag{6.2}$$

Cependant, pour respecter le déterminisme, il nous faut définir une sémantique plus contraignante pour le prédicat :=. Dans le schéma de spécifications des comportements

invariants, pré/post-conditions, la spécification de changement d'état est effectuée par un couple de pré/post-conditions. Nous rappelons que la pré-condition d'un service spécifie l'état avant réalisation du service et la post-condition d'un service spécifie l'état après réalisation du service. Les modifications de valeurs d'états sont alors spécifiées dans la post-condition du service. Les assertions atomiques définies sur le prédicat := ne doivent apparaître que dans cette dernière.

La modification d'une valeur d'une variable d'états impose que l'on doit pouvoir différencier son ancienne valeur de sa nouvelle. Notons que la distinction entre ancienne et nouvelle valeur n'est nécessaire que dans les post-conditions.

Pour différencier une ancienne valeur d'une variable d'états de sa nouvelle valeur, le langage Z [Spi92] propose d'annoter les variables d'états représentant les nouvelles valeurs à l'aide d'une apostrophe "′" (x'). Au contraire, Le langage EIFFEL [Mey92] rajoute la fonction **old** d'arité un qui prend en paramètre une variable et rend en résultat l'ancienne valeur de la variable. Dans ce document, nous avons choisi l'annotation à la Z. Dans la suite du document, nous nommons cette annotation une décoration (*variable décorée*).

L'introduction des notions d'ancienne et de nouvelle valeur apporte une sémantique plus rigoureuse au prédicat :=. Pour définir une sémantique homogène, son opérande gauche est une variable d'états décorée. La forme générale 6.2 devient alors :

$$x' := E \tag{6.3}$$

Une contrainte supplémentaire est d'interdire la référence de variables d'état décorées dans l'opérande droit du prédicat :=. En effet prenons l'exemple de l'assertion :

$$x' := y' + 1 \wedge y' > 4 \tag{6.4}$$

De par la définition du prédicat :=, cette assertion n'est pas sémantiquement correcte. Le choix de la nouvelle valeur de x n'est pas unique car celle de y n'est pas unique. Dans ce cas, il faut vérifier la dépendance entre variables d'états décorées et en déduire si possible un ordre d'évaluation. Cependant, ce calcul d'ordre n'est pas toujours syntaxique. En effet, si l'on considère que y est de type entier et que l'on joint par conjonction l'assertion $y' < 2$ à l'assertion 6.4, le calcul de l'ordre passe par l'évaluation de la nouvelle assertion[1]. Dans cette thèse, nous décidons d'interdire la référence de variables d'état décorées dans l'opérande droit du prédicat := pour ne pas introduire un calcul non syntaxique du déterminisme explicite.

1. l'évaluation de $y' < 2$ et de $y' > 4$ permet de déduire que $y' = 3$.

En résumé, nous donnons la grammaire de assertions atomiques fondées sur le prédicat :=:

$$\text{prédicat\_det} \quad ::= \quad \text{var\_libre\_décorée} := \text{terme}$$

Nous donnons la grammaire de var_libre_décorée dans les sous section suivantes. Nous n'exprimons pas dans cette grammaire qu'aucune référence à des variables libres décorées ne doit apparaître dans les termes en opérandes droits de :=. Cette vérification doit être effectuée au cours de la compilation des comportements. Nous en discutons dans le chapitre 7.

6.1.2 Sémantique générale des variables d'états décorées

Dans la sous-section précédente, nous n'avons pas défini complètement la sémantique des variables d'états décorées. En effet, elles peuvent aussi apparaître sous la portée de prédicats autres que le prédicat := et aussi sous la portée de fonctions (+, −, etc.). L'assertion suivante :

$$x' + 2 > y \tag{6.5}$$

doit être interprétée ainsi :

la nouvelle valeur de x augmentée de deux est strictement supérieure à l'ancienne valeur de y.

La décoration permet de fournir une information sur les variables d'états modifiées par le service. Dans l'absolu, si une variable d'états n'apparaît pas décorée dans la post-condition liée au service, cela ne signifie pas qu'elle n'a pas été modifiée. Cependant, au cours de la simulation, il nous faut fixer une valeur pour toute variable d'états, aussi nous considérons que celles, qui n'apparaissent pas décorées dans la post-condition du service simulée, préservent leurs anciennes valeurs dans le nouvel état [2].

6.1.3 Décoration des variables d'états dans le monde objet

Dans les sous-sections précédentes, nous avons utilisé une représentation simplifiée des variables d'états. Cependant, nous avons introduit une notation supplémentaire dans les langages d'assertions : la décoration des variables d'états. Il nous faut l'appliquer à la représentation objet des variables d'états introduite dans la sous-section 5.1.2.

2. cette sémantique est différente de celle de Z.

Une variable d'états simple est $o.a$ et représente la valeur de l'attribut a de l'objet o. Notre choix est de décorer l'objet comme suit :

$$o^{'}.a \tag{6.6}$$

Dans le cas des variables d'états plus complexes, la décoration doit porter sur l'objet comportant l'attribut modifié, comme suit :

$$((o.a).b)^{'}.c \tag{6.7}$$

Cependant, on peut considérer que la décoration puisse apparaître plusieurs fois en opérande gauche du prédicat :=. Prenons l'exemple de la variable d'états suivante :

$$(o^{'}.a)^{'}.b \tag{6.8}$$

Son interprétation est :

on modifie l'attribut b de la nouvelle valeur de l'attribut a de l'objet o.

La valeur de l'attribut a de o est un nouvel objet. Il faut vérifier dans la post-condition la présence d'une assertion du type $o^{'}.a := E$. Dans la sous section 6.1.1, nous avons interdit la référence de variables d'état décorées en opérande droit du prédicat := pour éviter d'effectuer un calcul d'ordre non syntaxique. La calcul d'ordre pour déterminer la présence d'une assertion du type $o^{'}.a := E$ est syntaxique. Cependant, pour des raisons de simplification du calcul sémantique, nous décidons qu'une seule décoration doit apparaître dans une variable d'état et porter sur l'attribut le plus à droite (cf. la variable 6.7). Cette restriction ne concerne que l'opérande gauche de :=. Elle ne concerne pas les autres variables libres.

Une variable peut représenter un élément d'une collection de valeurs. Une telle variable est $o.a[c]$. Sa décoration est alors $o^{'}.a[c]$. Elle s'interprète ainsi :

la nouvelle valeur de l'élément c de la nouvelle valeur de l'attribut a de l'objet o.

Dans le cas des collections, une partie de la valeur de l'attribut de type collection peut être modifiée par un service (cette partie est explicitement représentée par la décoration des variables d'états dans la post-condition). Pour la partie de la collection non décorée dans la post-condition, les anciennes valeurs des éléments de cette partie sont préservées (cf. sous-section 6.1.2).

Dans $o^{'}.a[c]$, c peut être un terme complexe ($x+3+y$, etc.). Il peut faire référence à des variables d'états. Pour les mêmes raisons citées dans la sous-section 6.1.1 portant sur la dépendances des variables décorées, nous interdisons les références de celles-ci dans c.

Nous avons la nouvelle grammaire des variables libres suivante :

var_libre	::=	objet.**SYMB** **{**[terme**}**] | var_libre_décorée
objet	::=	**SYMB** | var_libre
var_libre_décorée	::=	objet'.**SYMB** **{**[terme**}**]

6.1.4 Sémantique compositionnelle du prédicat :=

Dans la sous-section 6.1.1, nous avons défini la sémantique du prédicat := pour les assertions atomiques basées sur ce prédicat. Il nous reste à préciser sa sémantique dans les assertions non atomiques.

Une assertion est une suite d'assertions atomiques liées par des connecteurs et quantificateurs. Le prédicat := doit être utilisé dans un contexte déterministe, aussi une assertion atomique basée sur le prédicat := doit apparaître dans un contexte déterministe, et en particulier en opérande de connecteurs et quantificateurs déterministes.

Les connecteurs et quantificateurs sont différents suivant les langages d'assertions. Nous définissons la sémantique compositionnelle du prédicat := sur la définition générale des langages d'assertions définie dans la sous-section 5.1.1. La règle grammaticale connecteur devient :

$$\text{connecteur} ::= \quad \wedge \text{ (et logique)} \mid \vee \text{ (ou logique)} \mid \neg \text{ (négation)}$$

Nous allons étudier le déterminisme de chacun des connecteurs et quantificateurs. Dans les paragraphes suivants, nous introduisons d'autres notations standards des langages d'assertions qui nous paraissent intéressantes pour intégrer le déterminisme explicite.

Le connecteur $\wedge$

Le connecteur $\wedge$ préserve par définition le déterminisme de ses opérandes. Les deux opérandes du connecteur doivent être évalués à vrai pour valider la propriété d'une assertion conjonctive. Une assertion atomique basée sur le prédicat := est utilisable en opérande du connecteur $\wedge$. Les assertions suivantes sont valides (syntaxiquement) :

- $x' := 1 \wedge y' := 6$
- $x' := 1 \wedge y = 6$
- $x' := 1 \wedge x' := 6$

La dernière assertion est syntaxiquement correcte, mais est évaluée à faux. Nous montrons dans la section 6.3 comment nous évitons de modifier les valeurs d'états dans le cas d'une évaluation à faux d'une post-condition.

Le connecteur $\vee$

Au contraire du connecteur $\wedge$, le connecteur $\vee$ est non-déterministe par définition. Une assertion disjonctive est évaluée à vrai si au moins un de ces opérandes est évalué à vrai. Ce problème du choix est évidemment non-déterministe. L'assertion suivante nous donne un exemple de ce qui n'est pas autorisé d'écrire et aussi un exemple de non-déterminisme :

$$x' := 1 \vee x' := 2 \tag{6.9}$$

Dans cette assertion, la nouvelle valeur de la variable d'états est soit 1 ou 2, ce qui est en contradiction avec la définition du déterminisme explicite qui n'offre qu'une valeur possible à une même variable. Une assertion atomique basée sur le prédicat := n'est pas définissable en opérande du connecteur $\vee$.

Cependant, un cas particulier de disjonction est intéressant du point de vue du déterminisme. Dans nombre de langages d'assertions, la conditionnelle est souvent présente. Elle est représentée souvent par les termes syntaxiques *if, then, else* (*si, alors, sinon* en français). Sa définition logique est :

$$if\ c\ then\ a\ else\ b\ \rightarrow\ (c \wedge a) \vee (\neg c \wedge b) \tag{6.10}$$

Nous rajoutons cette notation dans le langage d'assertions introduit dans cette section.

La conditionnelle peut être déterministe car le choix s'effectue sur une condition c. Reprenons l'assertion 6.9 en l'écrivant sous la forme d'une conditionnelle :

$$if\ c\ then\ x' := 1\ else\ x' := 2 \tag{6.11}$$

Pour que cette assertion soit explicitement déterministe, il faut que, lors de l'exécution, la condition c soit résolue. La condition c ne doit pas contenir de référence à des variables d'états décorées.

Le connecteur $\neg$

Comme le connecteur $\vee$, le connecteur $\neg$ est non-déterministe par définition. En effet, l'assertion :

$$\neg(x' := 1) \tag{6.12}$$

exprime que la nouvelle valeur de x peut être toute valeur de son ensemble de définition sauf la valeur 1. Nous sommes alors dans un cas de choix qui est contraire à la définition du prédicat :=. L'assertion 6.12 n'est pas valide, et dans le cas général une assertion atomique basée sur le prédicat := ne peut être opérande du connecteur $\neg$.

Un cas particulier de types de données rend déterministe le connecteur $\neg$. Ce sont ceux dont l'ensemble de définition est de cardinalité deux. Le type booléen est un exemple de ces types de données (ensemble de définition $\{vrai, faux\}$). L'assertion atomique suivante ne pose alors pas de problème de choix sur la nouvelle valeur de la variable d'états x :

$$\neg(x' := vrai) \text{ la nouvelle valeur de } x \text{ est } faux \tag{6.13}$$

Cependant, considérer ces types de données impose un calcul des données, c'est-à-dire effectuer un calcul non syntaxique de la partie explicitement déterministe des assertions. Nous ne nous imposons pas un tel calcul, aussi l'assertion 6.13 n'est pas valide.

Le quantificateur $\forall$

La quantification universelle permet de donner une propriété commune à un ensemble d'éléments, elle est donc déterministe. Le prédicat := peut donc être dans la portée d'un quantificateur $\forall$. L'assertion suivante est valide :

$$\forall i \in t | f'[i] := 1 \tag{6.14}$$

Pour effectuer l'exécution d'une assertion universellement quantifiée, l'ensemble t de 6.14 doit être connu. Dans l'optique d'un calcul syntaxique du déterminisme explicite, t ne doit pas référencer de variables décorées. Voici quelques exemples d'assertions (où {...} est la constante de fonctions représentant un ensemble) :

- $\forall i \in \{1, 2, 3\} | f'[i] := i + 3$ est valide ;
- $\forall i \in \{1, y' + 7\} | f'[i + 1] := 5$ n'est pas valide car le deuxième élément de $\{1, y' + 7\}$ contient une variable décorée ;
- $\forall i \in \{1, y + 7\} | f'[i + 1] := 5$ est valide.

Le quantificateur $\exists$

La quantification existentielle précise qu'il existe au moins une valeur telle que l'assertion en portée de la quantification soit vraie. Cette définition n'est pas déterministe car plusieurs valeurs de l'ensemble d'*itération* peuvent valider l'assertion (nous ne prenons pas en considération le cas d'un ensemble d'itération contenant un seul élément pour les mêmes raisons que le cas particulier défini pour le connecteur $\neg$). D'après la sémantique du quantificateur $\exists$, une seule valeur est suffisante. Aussi, il faut effectuer un choix parmi les valeurs possibles. Une assertion atomique basée sur le prédicat := ne peut apparaître dans la portée d'un quantificateur existentiel. L'assertion suivante n'est pas valide :

$$\exists i \in t | f'[i] := 1 \tag{6.15}$$

En résumé, nous donnons la grammaire des assertions explicitement déterministe suivante :

formule_det	::=	prédicat_det | f_det_complexe
f_det_complexe	::=	formule $\wedge$ formule | **if** formule **then** formule **else** formule | $\forall$ var $\in$ terme "|" formule

On rajoute dans la règle grammaticale de formule l'alternative formule_det. Les contraintes relatives à la sémantique de :=ne sont pas toutes reprises dans cette simple grammaire. Dans le chapitre 7, nous montrons qu'elles sont prises en compte au cours de la compilation des comportements.

6.2 Sémantique opérationnelle du prédicat :=

La sémantique opérationnelle nous permet de définir un algorithme de traitement du déterminisme explicite. L'algorithme décrit dans cette sous-section est *syntaxique*, c'est-à-dire que le calcul n'est basé que sur la syntaxe du langage. Les contraintes calculées par cet algorithme sont décrites dans la section 6.1.

Dans un premier temps, nous allons montrer que le calcul de la partie explicitement déterministe peut s'effectuer syntaxiquement. Nous présentons ensuite cette sémantique.

6.2.1 Principes du calcul

Le déterminisme explicite permet de déduire les nouvelles valeurs à associer à un ensemble de variables d'états. Le calcul syntaxique effectué sur les assertions doit permettre d'exhiber ces nouvelles valeurs. Cependant, une variable d'état décorée peut, dans une même assertion (post-condition), être référencée dans la partie ne comportant pas de déterminisme explicite. Prenons comme exemple :

$$y' = x' + 2 \wedge y' := 4 \wedge x' := 2 \tag{6.16}$$

Les assertions atomiques $x' := 2$ et $y' := 4$ permettent de valider $y' = x' + 2$. Il nous parait donc évident que le calcul des assertions atomiques basées sur le prédicat := doit être effectué avant l'évaluation du reste de l'assertion.

En logique, l'ordre d'évaluation n'est pas important. Nous n'imposons pas une méthode d'écriture des assertions contenant du déterminisme explicite (si le sens de calcul est de la gauche vers la droite, sens de lecture, nous n'obligeons pas le spécifieur à définir au préalable les assertions atomiques basées sur le prédicat := avant les autres assertions). Le déterminisme explicite défini dans ce document permet, sans contrainte d'écriture des assertions, d'évaluer en premier les formules atomiques explicitement déterministes.

Dans la sous-section 6.2.2, nous montrons comment ce calcul des assertions atomiques basées sur le prédicat := est effectué au préalable. Mais pour que ce calcul soit possible, il nous faut montrer que l'on peut isoler les assertions atomiques basées sur le prédicat :=.

Le prédicat := peut apparaître dans une conjonction, une quantification universelle ou une conditionnelle, c'est-à-dire dans une assertion non atomique. Nous considérons ici les assertions dont les occurrences de variables d'états non décorées sont résolues, ainsi que les tautologies induites par ces résolutions.

Pour isoler les assertions atomiques basées sur le prédicat := d'une assertion complexe, nous pouvons lui appliquer les calculs de forme normale et de formule prénexe [LRd93]. En effet, ces calculs permet de définir pour toute assertion son équivalente du type $\forall x \forall y \exists z\ f$, avec f ne contenant pas de quantificateur (calcul prénexe). Il suffit ensuite de déduire de f sa forme normale. Les assertions atomiques basées sur le prédicat := sont alors isolées du reste de l'assertion.

6.2.2 Sémantique opérationnelle

La sémantique opérationnelle que nous allons décrire dans cette sous-section est définie sur des assertions contenant du déterminisme explicite et respectant les contraintes

définies dans la section 6.1. Ces contraintes peuvent être vérifiées lors de l'analyse syntaxique des post-conditions (cf. chapitre 7).

Cette sémantique permet de calculer les nouvelles valeurs des variables d'états apparaissant à gauche du prédicat := ; elle permet aussi de résoudre un maximum d'occurrences de variables, décorées ou non, dans la post-condition traitée. Pour résoudre les occurrences non décorées d'une post-condition, il suffit de se référer à l'état avant l'exécution du service lié à la post-condition.

Le point d'entrée de notre sémantique opérationnelle est la fonction $\mathcal{E}$ qui a pour signature :

$$(assertion \times état) \rightarrow (état \times assertion)$$

où le paramètre *assertion* est la post-condition à analyser, le paramètre *état* est l'état avant exécution et le résultat $(état \times assertion)$ est le couple constitué de l'état après analyse de la partie déterministe de la post-condition et du reste de la post-condition (appelée *assertion résiduelle*).

Dans la sous-section 6.2.1, nous avons montré que le fait de calculer les assertions atomiques basées sur le prédicat := permet de résoudre certaines assertions. La fonction sémantique $\mathcal{E}$ doit d'abord traiter le déterminisme explicite puis la résolution d'occurrences décorées. Nous donnons la règle suivante pour $\mathcal{E}$:

$$\mathcal{E}(e,s) \quad = \quad \mathcal{R}(s, \mathcal{S}(e, s, (\mathbf{InitEtat}(s), \mathbf{vrai})))$$

La fonction sémantique $\mathcal{S}$ permet de calculer les assertions atomiques basées sur le prédicat :=. $\mathcal{S}$ calcule les nouvelles valeurs des variables d'état décorées en partie gauche du prédicat :=.

La fonction sémantique $\mathcal{R}$ permet la résolution d'occurrences décorées et non décorées sur la post-condition résiduelle donnée en résultat de $\mathcal{S}$. En sous-section 6.1.2, nous avons précisé qu'il est important de donner une valeur à toute variable d'état dans le nouvel état. $\mathcal{R}$ permet de définir les variables d'états non décorés et de leur affecter leur ancienne valeur comme nouvelle valeur. Pour les variables décorées mais non explicitement affectées, le traitement est donné dans la section 6.3.

La fonction **InitEtat** permet de construire l'état après exécution à partir de l'état avant exécution. Nous avons deux types d'éléments dans un état : les variables d'état ayant des collections comme valeurs et les variables d'état ayant des valeurs non collection. Pour distinguer ces deux types de valeurs, nous représentons les valeurs collection ainsi :

$$[v_1, ..., v_n] \tag{6.17}$$

La fonction **InitEtat** prend en paramètre l'état s passé en paramètre de $\mathcal{E}$, et rend une copie de cet état dont on a modifié les valeurs des couples comme suit [3] :

- Pour tout couple $((o,a),v)$ on remplace v par $\perp$.
- Pour tout couple $((o,a),[v_1,...,v_n])$ on remplace chaque v_i par $\perp$.

où $\perp$ est la valeur *non définie*. Soit l'exemple suivant :

$$\begin{aligned}&\textbf{InitEtat}(\{...,((o_i,a_j),v_k),((o_u,a_s),[v_1,...,v_n]),...\})\\&\quad=\{...,((o_i,a_j),\perp),((o_u,a_s),[\perp,...,\perp]),...\}\end{aligned} \tag{6.18}$$

La fonction sémantique $\mathcal{S}$

$\mathcal{S}$ a pour signature :

$$(assertion \times état \times (état \times assertion)) \rightarrow (état \times assertion)$$

$\mathcal{S}$ prend en paramètres une assertion logique e, l'état avant exécution s et le résultat intermédiaire de l'évaluation de la post-condition initiale (s_{int},e_{int}), c'est-à-dire l'accumulateur de l'évaluation récursive de la post-condition e. $\mathcal{S}$ rend un couple (s_r,e_r).

A l'appel de $\mathcal{S}$ dans $\mathcal{E}$, le résultat intermédiaire est initialisé par le couple (**InitEtat**(s),**vrai**). Avant l'évaluation de la post-condition initiale, on ne connaît pas encore les variables d'état susceptibles de changer de valeur. Aussi, toute variable d'état du premier état intermédiaire de l'évaluation d'une post-condition par $\mathcal{S}$ est de valeur $\perp$.

Dans la sous-section 6.2.1, nous avons montré que les assertions explicitement déterministes sont liées conjonctivement aux autres assertions. Pour le calcul de la post-condition résiduelle d'une post-condition évaluée par $\mathcal{S}$, il suffit d'effectuer une conjonction des assertions non explicitement déterministes. De ce fait, la première post-condition résiduelle intermédiaire de $\mathcal{S}$ est **vrai**.

Nous avons la règle suivante pour $\mathcal{S}$:

$$\begin{aligned}\mathcal{S}(e,s,(s_{int},e_{int})) \quad = \quad &\textbf{si DetExp?}(e)\\&\textbf{alors } \mathcal{S}'(e,s,(s_{int},e_{int}))\\&\textbf{sinon } (s_{int},e_{int} \wedge e)\end{aligned}$$

Où **DetExp?** permet de vérifier si au moins une assertion atomique explicitement déterministe est présente dans la post-condition à calculer.

3. nous reprenons la définition de l'état donné dans la sous-section 5.1.2.

La fonction sémantique $\mathcal{S}'$ et la fonction sémantique $\mathcal{S}$ s'appellent mutuellement. $\mathcal{S}'$ permet de traiter les assertions explicitement déterministes et $\mathcal{S}$ permet d'arrêter le traitement récursif lorsque l'assertion n'est plus explicitement déterministe. $\mathcal{S}'$ est de même signature que la fonction $\mathcal{S}$. La seule différence entre les deux descriptions des paramètres et résultats est le rajout de ces définitions pour le résultat (s_r,e_r) :

- s_r est l'état intermédiaire s_{int} où certaines variables d'état ont changé leur valeur $\perp$ contre une valeur calculée au cours de l'évaluation de e ;
- e_r est la conjonction de e_{int} et de l'*assertion* résiduelle de e calculée au cours de l'évaluation de e.

Avant de présenter les règles pour $\mathcal{S}'$, nous donnons quelques indications sur la syntaxe utilisée dans les règles suivantes :

- accès à une valeur d'un élément d'un état s :
  - $s(o,a)$ est l'accès à la valeur v du couple $((o,a),v)$ contenu dans l'état s ;
  - $s(o,a)[c]$ est l'accès à la valeur v_c du couple $((o,a),[v_1,...,v_c,...,v_n])$ contenu dans l'état s ;
- E_i représente le $i$ème élément de l'ensemble E ;
- substitution d'une valeur d'un état s :
  - $s[(o,a)/v]$ est la substitution de la valeur associée à la clé (o,a) par v dans l'état s ;
  - $s[((o,a),c)/v]$ est la substitution de la valeur associée à l'élément c de la valeur associée à la clé (o,a) par v dans l'état s ;
- $e_1[i/e_2]$ est l'assertion obtenue à partir de e_1 en substituant toutes les occurrences libres de la variable liée i par e_2 ;
- **Eval**(e,s) permet de définir la valeur de l'assertion e en remplaçant les occurrences des variables d'état de e par leurs valeurs associées dans l'état s.

L'état en paramètre de la fonction **Eval** ne doit pas contenir de valeur $\perp$. Cette contrainte est nécessaire pour traiter le déterminisme explicite. Elle est respectée dans les règles suivantes, par le fait que seul l'état avant exécution est mis en paramètre de la fonction **Eval**. De plus, on peut noter que la fonction **Eval** doit faire appel à la fonction d'accès aux éléments de l'état (implantation dans le simulateur de la fonction d'accès aux attributs d'objets).

Nous allons aussi introduire les fonctions d'accès aux éléments des états. Dans la sous-section 5.1.2, nous avons présenté la syntaxe des variables d'état. Nous autorisons

la composition de l'opérateur "_._". Une fonction d'accès récursive définie sur l'état est nécessaire. Nous allons tout d'abord définir la fonction **KeyAcc** qui permet d'accéder aux *clés* d'un état. L'état est un ensemble d'éléments du type $((o,a),v)$. Les clés d'un état est l'ensemble des couples (o,a), premiers éléments des couples $((o,a),v)$ de l'état. La règle sémantique de la fonction **KeyAcc** est la suivante :

$$\begin{array}{lcl} \mathbf{KeyAcc}(o,a,s) & = & \mathbf{si}\ o = (o_i.a_i) \\ & & \quad \mathbf{alors}\ (\mathbf{AccAnc}(o_i,a_i,s),a) \\ & & \quad \mathbf{sinon}\ (o,a) \end{array}$$

où la fonction **AccAnc** permet d'accéder aux valeurs définies dans l'état avant exécution du service de l'application. La règle sémantique de **AccAnc** est :

$$\begin{array}{lcl} \mathbf{AccAnc}(o,a,s) & = & \mathbf{cas}\ o \\ & & \quad (o_i.a_i) : s(\mathbf{AccAnc}(o_i,a_i,s),a) \\ & & \quad (o_i.a_i[c_i]) : s(s(\mathbf{KeyAcc}(o_i,a_i,s))[\mathbf{Eval}(c_i,s)],a) \\ & & \quad \mathbf{sinon}\ s(o,a) \\ & & \mathbf{fcas} \end{array}$$

La fonction **KeyAcc** ne s'applique que sur les états précédents les exécutions de services. En effet, en sous-section 6.1.3, nous imposons qu'une seule décoration apparaisse dans une variable d'état et porte sur l'attribut le plus à droite. Nous définissons donc la fonction **Acc** qui permet d'accéder à l'état après exécution du service de l'application. La règle sémantique associée à **Acc** est la suivante :

$$\mathbf{Acc}(o,a,s,s') = s'(\mathbf{KeyAcc}(o,a,s))$$

Les fonctions **KeyAcc**, **AccAnc** et **Acc** rendent $\perp$ lorsqu'un des objets *calculés* n'existe pas.

Grâce aux fonctions **Acc** et **KeyAcc**, nous avons les règles suivantes de $\mathcal{S}'$ pour les assertions atomiques basées sur le prédicat := :

$$
\begin{array}{lcl}
\mathcal{S}'(o'.a := e,s,(s_{int},e_{int})) & = & \textbf{soit} \\
 & & \quad \gamma \leftarrow \textbf{Acc}(o,a,s,s_{int}) \\
 & & \quad \alpha \leftarrow \textbf{Eval}(e,s) \\
 & & \textbf{dans} \\
 & & \quad \textbf{cas } \gamma \textbf{ et } \alpha \\
 & & \quad\quad [v_1,...,v_n] \textbf{ et } [w_1,...,w_n] : \\
 & & \quad\quad\quad \textbf{s'il existe un } i \textbf{ dans } \{1,...,n\} \\
 & & \quad\quad\quad\quad\quad \textbf{tel que } v_i \neq \perp \textbf{ et } v_i \neq w_i \\
 & & \quad\quad\quad\quad \textbf{alors } (s_{int},e_{int} \wedge \textbf{faux}) \\
 & & \quad\quad\quad\quad \textbf{sinon } (s_{int}[\textbf{KeyAcc}(o,a,s)/\alpha],e_{int}) \\
 & & \quad\quad v \textbf{ et } w : \\
 & & \quad\quad\quad \textbf{si } v \neq \perp \textbf{ et } v \neq w \\
 & & \quad\quad\quad\quad\quad \textbf{alors } (s_{int},e_{int} \wedge \textbf{faux}) \\
 & & \quad\quad\quad\quad\quad \textbf{sinon } (s_{int}[\textbf{KeyAcc}(o,a,s)/\alpha],e_{int}) \\
 & & \textbf{fcas}
\end{array}
$$

Cette règle sémantique permet de mettre en évidence que l'on peut modifier une valeur de type collection. Dans ce cas, on ne teste pas seulement sur la valeur mais sur toutes les valeurs de la collection. Rappelons qu'une variable d'état est aussi une valeur de collection. La règle sémantique associée à ce type de variable est la suivante :

$$
\begin{array}{lcl}
\mathcal{S}'(o'.a[c] := e,s,(s_{int},e_{int})) & = & \textbf{si } s_{int}(\textbf{KeyAcc}(o,a,s))[\textbf{Eval}(c,s)] \neq \perp \\
 & & \quad \textbf{et } \textbf{Eval}(e,s) \neq s_{int}(\textbf{KeyAcc}(o,a,s))[\textbf{Eval}(c,s)] \\
 & & \quad \textbf{alors } (s_{int},e_{int} \wedge \textbf{faux}) \\
 & & \quad \textbf{sinon } (s_{int}[(\textbf{KeyAcc}(o,a,s),\textbf{Eval}(c,s))/\textbf{Eval}(e,s)] \\
 & & \quad\quad ,e_{int})
\end{array}
$$

D'un point de vue général, dans les règles sémantiques précédentes, on teste tout d'abord si la variable d'état définie en partie gauche possède une valeur non $\perp$ dans l'état intermédiaire. Deux cas peuvent se présenter :

- une autre assertion atomique basée sur le prédicat := a été traitée précédemment et a fourni une valeur à cette variable. La post-condition $o'.a := 6 \wedge o'.a := 3$ en est un exemple ;
- la mise en évidence du phénomène de l'*aliasing* (références multiples) [WLB97]. En effet, dans le monde de l'objet, il est fréquent de référencer à l'aide de variables différentes des données. Prenons le cas deux objets o_1 et o_2 possédant des attributs a_1 et a_2 référençant le même objet o. L'objet o possède un attribut d de type entier. La post-condition $(o_1.a_1)'.d := 4 \wedge (o_2.a_2)'.d := 6$ montre la double référence de l'attribut d de o. La fonction **Acc** permet de traiter l'aliasing.

Dans le cas où l'on veut affecter une deuxième nouvelle valeur à une variable d'état, cela signifie que l'assertion doit rendre la valeur **faux** à son évaluation. L'assertion résiduelle intermédiaire rendue est **faux**. Dans le cas contraire, on poursuit l'évaluation en affectant dans l'état la nouvelle valeur de la variable d'état traitée.

Les règles de $\mathcal{S}'$ pour les assertions complexes et explicitement déterministes sont :

$$\begin{aligned}
\mathcal{S}'(e_1 \wedge e_2, s, (s_{int}, e_{int})) &= \mathcal{S}(e_2, s, \mathcal{S}(e_1, s, (s_{int}, e_{int}))) \\
\mathcal{S}'(\text{if } c \text{ then } e_1 \text{ else } e_2, s, (s_{int}, e_{int})) &= \textbf{si } \textbf{Eval}(c, s) = \textbf{vrai} \\
&\quad \textbf{alors } \mathcal{S}(e_1, s, (s_{int}, e_{int})) \\
&\quad \textbf{sinon } \mathcal{S}(e_2, s, (s_{int}, e_{int})) \\
\mathcal{S}'(\forall\, i \in e_1 \mid e_2, s, (s_{int}, e_{int})) &= \mathcal{S}(e_2[i/E_n], s, \ldots \\
&\quad \mathcal{S}(e_2[i/E_2], s, \\
&\quad\quad \mathcal{S}(e_2[i/E_1], s, (s_{int}, e_{int})))\ldots)
\end{aligned}$$

où $E =$ **Eval** (e_1, s) et n est le nombre d'éléments de E.

Nous donnons en exemple le traitement de la post-condition e :

$$o_1'.a := (o_2.b).c + 6 \wedge o_2'.d < 2 \wedge (o_2.b)'.c := 3 \wedge o_1'.c > 1 \tag{6.19}$$

et l'état γ avant exécution associée à e :

$$\{\ldots, ((o_1, a), 4), ((o_1, c), 2), \ldots, ((o_2, b), o_1), ((o_2, d), 3), \ldots\} \tag{6.20}$$

Le traitement par la fonction sémantique $\mathcal{S}$ est :

$$\begin{aligned}
&\mathcal{S}(e, \gamma, (\{\ldots, ((o_1, a), \bot), ((o_1, c), \bot), \ldots, ((o_2, b), \bot), ((o_2, d), \bot), \ldots\}, \textbf{vrai})) = \\
&(\{\ldots, ((o_1, a), 8), ((o_1, c), 3), \ldots, ((o_2, b), \bot), ((o_2, d), \bot), \ldots\}, o_2'.d < 2 \wedge o_1'.c > 1)
\end{aligned} \tag{6.21}$$

La fonction sémantique $\mathcal{R}$

Le but de cette fonction sémantique est de fournir en résultat une assertion ne contenant que des occurrences de variables décorées dont leurs valeurs dans l'état résultat est $\bot$. ce dernier est celui fourni en résultat de l'appel à la fonction sémantique $\mathcal{S}$ dont on a résolu les occurrences non décorées et les occurrences décorées ayant une valeur dans l'état résultat de $\mathcal{S}$. La signature de $\mathcal{R}$ est :

$$(\mathit{état} \times (\mathit{état} \times \mathit{assertion})) \rightarrow (\mathit{état} \times \mathit{assertion})$$

Le premier paramètre est l'état avant exécution, le second paramètre est le résultat de l'appel à la fonction sémantique $\mathcal{S}$. La règle définissant $\mathcal{R}$ est :

$$\mathcal{R}(s,(s_r,e)) \quad = \quad \mathcal{R}_{état}(\mathcal{R}_{expr}(e,s,s_r,\{\}),s,s_r)$$

La fonction sémantique $\mathcal{R}_{état}$ permet le calcul de l'état final. Cette fonction associe leurs anciennes valeurs aux variables non décorées dans la post-condition en paramètre.

La fonction sémantique $\mathcal{R}_{expr}$ permet le calcul de la post-condition finale. Cette fonction remplace les variables décorées de la post-condition en paramètre par leurs valeurs dans l'état intermédiaire (si cette valeur n'est pas $\perp$). Elle remplace aussi les variables non décorées de la post-condition en paramètre par leurs valeurs dans l'état avant exécution.

Lors du calcul de la post-condition finale ($\mathcal{R}_{expr}$), on effectue une accumulation (quatrième paramètre de $\mathcal{R}_{expr}$) des variables d'état décorées et non valuées dans l'état résultat de $\mathcal{S}$. La valeur initiale de ce paramètre est l'ensemble vide ($\{\}$). L'accumulation de ces variables d'état permet de définir l'ensemble **Ind** des variables décorées non résolues par $\mathcal{S}$. A l'aide de l'ensemble **Ind**, la fonction ($\mathcal{R}_{état}$) permet de préserver dans l'état final :

- les valeurs $\perp$ des variables décorées non explicitement valuées dans la post-condition. Ces variables appartiennent à l'ensemble **Ind** ;
- les anciennes valeurs des variables non décorées et non exprimées dans la post-condition. Une telle variable a pour valeur $\perp$ dans l'état intermédiaire et n'appartient pas à l'ensemble **Ind**.

Par exemple, dans l'état résultat θ :

$$\{...,((o_1,a),8),((o_1,c),3),...,((o_2,b),\perp),((o_2,d),\perp),...\} \tag{6.22}$$

de l'appel 6.21, on associera la valeur o_1 à (o_2,b) (ancienne valeur de (o_2,b), cf. 6.20), mais on n'associera aucune valeur à (o_2,d).

La fonction sémantique $\mathcal{R}_{expr}$

$\mathcal{R}_{expr}$ a pour signature :

$$(assertion \times état \times état \times ensemble) \rightarrow (assertion \times ensemble)$$

Le premier paramètre est la post-condition résiduelle résultat de l'appel à la fonction sémantique $\mathcal{S}$, le second paramètre est l'état avant exécution, le troisième paramètre est l'état résultat de l'appel à $\mathcal{S}$ et le quatrième est l'accumulateur des variables d'état décorées et non explicitement déterminées.

Une particularité de la fonction $\mathcal{R}_{expr}$ par rapport à la fonction $\mathcal{S}'$ est que $\mathcal{R}_{expr}$ doit analyser toutes les assertions (prendre en compte tous les connecteurs, quantificateurs et constantes de fonctions des langages d'assertions). Les règles définissant $\mathcal{R}_{expr}$ sont :

- *variables d'état non décorées*

$$\mathcal{R}_{expr}(o.a,s,s_r,c) \quad = \quad (\mathbf{AccAnc}(o,a,s),c)$$

$$\begin{array}{lll} \mathcal{R}_{expr}(o.a[i],s,s_r,c) & = & \textbf{soit} \\ & & \quad (ind,c_1) \leftarrow \mathcal{R}_{expr}(i,s,s_r,\{\}) \\ & & \textbf{dans} \\ & & \quad (s(\mathbf{keyAcc}(o,a,s))[\text{ind}],c \cup c_1) \end{array}$$

On remplace dans la post-condition résiduelle les variables d'état non décorées par leurs valeurs dans l'état précédent l'exécution du service, i.e. par leurs anciennes valeurs. Dans la deuxième règle sémantique, nous traitons l'indice i par la fonction $\mathcal{R}_{expr}$. En effet, i peut contenir des variables dont on peut associer leurs valeurs ainsi que des variables décorées non explicitement déterminées que l'on regroupe dans l'ensemble c_1. Les règles sémantiques suivantes montrent la construction des éléments des accumulateurs c et c_1.

- *variables d'état décorées*

$$\begin{array}{lll} \mathcal{R}_{expr}(o'.a,s,s_r,c) & = & \textbf{si } \mathbf{Acc}(o,a,s,s_r) \neq \perp \\ & & \quad \textbf{alors } (\mathbf{Acc}(o,a,s,s_r),c) \\ & & \quad \textbf{sinon } (o'.a,c \cup \{\mathbf{KeyAcc}(o,a,s)\}) \end{array}$$

$$\begin{array}{lll} \mathcal{R}_{expr}(o'.a[i],s,s_r,c) & = & \textbf{soit} \\ & & \quad (ind,c_1) \leftarrow \mathcal{R}_{expr}(i,s,s_r,\{\}) \\ & & \textbf{dans} \\ & & \quad \textbf{si } s_r(\mathbf{keyAcc}(o,a,s)[\text{ind}] \neq \perp \\ & & \quad\quad \textbf{alors } (s_r(\mathbf{keyAcc}(o,a,s)[\text{ind}],c) \\ & & \quad\quad \textbf{sinon } (o'.a[ind],c \cup \{(\mathbf{KeyAcc}(o,a,s),ind)\}) \end{array}$$

Deux cas se présentent lors du traitement par $\mathcal{R}_{expr}$ d'une variable d'état décorée :

- soit la variable a été explicitement évaluée par $\mathcal{S}$. On remplace son occurrence dans la post-condition résiduelle par sa nouvelle valeur ;
- soit la variable n'a pas été explicitement évaluée par $\mathcal{S}$. On rajoute à l'accumulateur de variables décorées non résolues par $\mathcal{S}$ cette variable. La post-condition résiduelle n'est pas modifiée.

Dans les règles précédentes, l'appel de la fonction **KeyAcc** s'effectue sur s, l'état avant exécution. En effet, en sous-section 6.1.3, nous avons imposé qu'une seule décoration peut apparaître dans une variable d'état et qu'elle porte sur l'attribut le plus à droite de la variable. Le calcul de la clé est alors garanti car l'état s ne comporte pas de valeur $\perp$.

- *Constantes d'individus (0, 1, "chaîne", etc.) et variables liées*

$$\mathcal{R}_{expr}(x,s,s_r,c) \quad = \quad (x,c)$$

- *Fonctions (+, −, etc.)*

$$\begin{array}{rcl} \mathcal{R}_{expr}(f(p_1,...,p_n),s,s_r,c) & = & \textbf{soit} \\ & & \quad (v_1,c_1) \leftarrow \mathcal{R}_{expr}(p_1,s,s_r,\{\}) \\ & & \quad ... \\ & & \quad (v_n,c_n) \leftarrow \mathcal{R}_{expr}(p_n,s,s_r,\{\}) \\ & & \textbf{dans} \\ & & \quad (f(v_1,...,v_n),c \cup c_1 \cup ... \cup c_n) \end{array}$$

- *Prédicats et connecteurs*[4]

$$\begin{array}{rcl} \mathcal{R}_{expr}(O(p_1,...,p_n),s,s_r,c) & = & \textbf{soit} \\ & & \quad (v_1,c_1) \leftarrow \mathcal{R}_{expr}(p_1,s,s_r,\{\}) \\ & & \quad ... \\ & & \quad (v_n,c_n) \leftarrow \mathcal{R}_{expr}(p_n,s,s_r,\{\}) \\ & & \textbf{dans} \\ & & \quad (O(v_1,...,v_n),c \cup c_1 \cup ... \cup c_n) \end{array}$$

- *Quantificateurs*

$$\begin{array}{rcl} \mathcal{R}_{expr}(Qx \in E|F,s,s_r,c) & = & \textbf{soit} \\ & & \quad (e,c_1) \leftarrow \mathcal{R}_{expr}(E,s,s_r,\{\}) \\ & & \quad (f,c_2) \leftarrow \mathcal{R}_{expr}(F,s,s_r,\{\}) \\ & & \textbf{dans} \\ & & \quad (Qx \in e|f,c \cup c_2) \end{array}$$

L'accumulateur résultat ne prend pas en compte l'ensemble c_1 car l'ensemble E ne doit pas contenir de variables d'état décorées (cf. sous-section 5.1.2).

Pour conclure la description de $\mathcal{R}_{expr}$, nous reprenons l'exemple de la post-condition 6.19. Nous avons l'appel suivant à la fonction $\mathcal{R}_{expr}$:

4. la notation *if then else* peut être interprétée comme un connecteur à trois paramètres.

$$\mathcal{R}_{expr}(o_2'.d < 2 \wedge o_1'.c > 1, \gamma, \theta, \{\}) =$$

$$(o_2'.d < 2 \wedge 3 > 1, \{(o_2, d)\}) \tag{6.23}$$

La fonction sémantique $\mathcal{R}_{état}$

La fonction $\mathcal{R}_{expr}$ fournit la post-condition résiduelle ne contenant pas de déterminisme explicite. Elle fournit aussi l'ensemble des variables d'état susceptibles d'être modifiées par le service mais non explicitement déterminées. A ce niveau de l'algorithme, l'état après exécution du service ne contient que des valeurs définies pour les variables d'état explicitement déterminées et des valeurs non définies pour les autres variables d'états. Dans la sous-section 6.1.2, nous considérons que les variables d'états non décorées dans la post-condition initiale préservent leurs anciennes valeurs. La fonction $\mathcal{R}_{état}$ effectue cette préservation.

$\mathcal{R}_{état}$ a pour signature :

$$((assertion \times ensemble) \times état \times état) \rightarrow (assertion \times état)$$

Le premier paramètre est le résultat de l'appel à la fonction sémantique $\mathcal{R}_{expr}$, le second paramètre est l'état avant exécution et le troisième paramètre est l'état résultat de l'appel à $\mathcal{S}$. La règle définissant $\mathcal{R}_{état}$ est :

$$
\begin{array}{lcl}
\mathcal{R}_{état}((e,c),s,s_r) & = & \textbf{soit} \\
 & & \quad E_1 \leftarrow \{((o,a),v) | ((o,a),v) \in s_r \wedge v \neq \bot\} \\
 & & \quad E_2 \leftarrow \{((o,a),\bot) | (o,a) \in c\} \\
 & & \quad E_3 \leftarrow \{((o,a),v) | ((o,a),v) \in s \wedge ((o,a),v) \notin E_1 \cup E_2\}) \\
 & & \quad E_4 \leftarrow \{((o,a),[v_1,...,v_n]) | \\
 & & \qquad\qquad \textbf{soit} \\
 & & \qquad\qquad\quad [u_1,...,u_n] \leftarrow s(o,a) \\
 & & \qquad\qquad\quad [w_1,...,w_n] \leftarrow s_r(o,a) \\
 & & \qquad\qquad \textbf{dans} \\
 & & \qquad\qquad\quad \textbf{pour tout } i \textbf{ dans } \{1,...,n\} \textbf{ faire} \\
 & & \qquad\qquad\qquad \textbf{si } w_i = \bot \\
 & & \qquad\qquad\qquad\quad \textbf{alors } v_i = u_1 \\
 & & \qquad\qquad\qquad\quad \textbf{sinon } v_i = w_i\} \\
 & & \quad E_5 \leftarrow \{((o,a),[v_1,...,v_n]) | ((o,a),[v_1,...,v_n]) \in E_4 \wedge \\
 & & \qquad\qquad \textbf{pour tout } ((o,a),i) \textbf{ dans } c \textbf{ faire} \\
 & & \qquad\qquad\quad v_i = \bot\} \\
 & & \textbf{dans} \\
 & & \quad (e, E_1 \cup E_2 \cup E_3 \cup E_5)
\end{array}
$$

L'ensemble E_1 contient les variables d'états explicitement déterminées et leurs valeurs. Elles ne sont pas de type collection. L'ensemble E_1 est calculé à partir de l'état résultat de l'appel à la fonction $\mathcal{S}$.

L'ensemble E_2 contient les variables d'états décorées de la post-condition mais non explicitement déterminées auxquelles on associe la valeur non définie $\bot$. Elles ne sont pas de type collection.

L'ensemble E_3 contient les variables non décorées et non déclarées dans la post-condition initiale associées de leurs anciennes valeurs.

L'ensemble E_5 contient toutes les variables de type collection dont on a affecté individuellement les valeurs de chaque élément. L'état après le traitement de la partie explicitement déterministe de la post-condition est alors la réunion de E_1, E_2, E_3 et E_5.

Nous reprenons l'exemple de la post-condition 6.19 et nous appliquons le résultat de l'appel la fonction sémantique $\mathcal{R}_{expr}$ 6.23 comme premier paramètre à l'appel suivant de la fonction $\mathcal{R}_{état}$:

$$\mathcal{R}_{état}((o_2'.d < 2 \wedge 3 > 1, \{(o_2,d)\}), \gamma, \theta) =$$

$$(o_2'.d < 2 \wedge 3 > 1, \{..., ((o_1,a),8), ((o_1,c),3), ..., ((o_2,b),o_1), ((o_2,d),\bot), ...\}) \qquad (6.24)$$

Le résultat de $\mathcal{R}_{état}$ fournit le résultat final de l'analyse sémantique opérationnelle des post-conditions explicitement déterministes. Cette sémantique suggère un algorithme de traitement de ces post-conditions. Ces fonctions sémantiques sont utilisées au chapitre 7 dans le cadre de la simulation.

6.3 Simulation du non déterminisme

Nous avons défini le déterminisme explicite comme une solution à la description d'une partie opérationnelle des comportements. Lors de la simulation des post-conditions, il permet de modifier l'état simulé de l'application. Le traitement de la partie non explicitement déterministe d'une post-condition par la simulation repose sur le traitement des variables d'états modifiées indépendamment des assertions atomiques basées sur le prédicat :=.

Après le traitement par la fonction sémantique $\mathcal{E}$ d'une post-condition, la post-condition restante ne contient plus d'assertions atomiques explicitement déterministes. Ces assertions atomiques sont soit remplacées par $faux$ soit ne sont pas remplacées. La fonction sémantique $\mathcal{R}$ permet de résoudre les références des variables d'états à l'aide des anciennes et nouvelles valeurs. Les seules occurrences non résolues sont celles des variables d'états portant sur de nouvelles valeurs et non explicitement déterministes (n'apparaissant pas en opérande gauche du prédicat := et présentes dans les assertions atomiques non explicitement déterministes).

La simulation passe d'un état à l'autre d'une application. Une post-condition permet de valider son nouvel état après *exécution* d'un comportement. Pour l'instant, la fonction sémantique $\mathcal{E}$ rend un état partiellement défini, comme le montre l'exemple 6.24 en rendant l'état résultat :

$$\{..., ((o_1, a), 8), ((o_1, c), 3), ..., ((o_2, b), o_1), ((o_2, d), \perp), ...\} \tag{6.25}$$

L'entrée (o_2, d) n'est pas définie (valeur $\perp$). Les spécifications ne peuvent pas être validées sans la définition de telles variables. Une technique utilisée dans le monde du test (cf. section 3.1) pour valider une série de tests est l'*oracle* [GMSB96]. Notre simulation n'est pas basée sur le test (cf. section 3.2), mais nous allons montrer que le concept d'oracle permet de définir les variables d'états non explicitement déterministes.

Dans le monde du test, l'oracle permet de valider ou d'invalider une série de tests d'une application en se référant à un ensemble de spécifications ou à un prototype [GMSB96]. Notre approche de la simulation ne nous offre pas d'implantation de l'application et, de plus, les *tests* s'effectuent sur les spécifications elles-mêmes. Pour valider

entièrement une post-condition, l'oracle doit fournir un ensemble de valeurs pour les variables d'états non explicitement déterministes satisfaisant la post-condition.

Dans le monde du test, l'oracle est un programme. L'oracle validant des spécifications doit intégrer le non déterminisme (fournir des valeurs aux variables non déterministes). Nous avons montré l'indécidabilité d'un calcul (programme) sur les spécifications prises en compte dans ce document (cf. section 4.2). Le seul que nous ayons est donc l'utilisateur (spécifieur) lui-même.

Chapitre 7

Le simulateur QG^{2}S

Ce chapitre présente une mise en oeuvre de la simulation de spécifications d'applications de gestion de réseaux de télécommunications. Les spécifications simulées contiennent une description des comportements des applications à l'aide d'un langage d'assertions contenant du déterminisme explicite. Cette mise en oeuvre est représentée par le simulateur de spécifications Q-GDMO-GRM (Quasi-GDMO-GRM)[1].

Le langage Q-GDMO-GRM permet de décrire des applications de gestion de réseaux de télécommunications dans le point de vue *information* des concepts ODP (cf. sous-section 2.1.1). Ce choix du langage de spécifications Q-GDMO-GRM est motivé par les raisons suivantes :

- Q-GDMO-GRM s'inscrit dans les récents travaux autour de la modélisation des applications de gestions de réseaux de télécommunications. En effet, Q-GDMO-GRM est préconisé par le consortium TINA-C[2] pour modéliser la vue information du modèle ODP ;
- Q-GDMO-GRM intègre les principaux concepts des langages de spécifications cités dans la sous-section 2.3.1 (réutilisation, etc.) ;
- Q-GDMO-GRM offre une structuration permettant d'introduire le schéma *invariants, pré/post-conditions* défini dans la sous-section 4.1.1 ;
- Q-GDMO-GRM permet d'intégrer facilement un langage d'assertions pour décrire les comportements.

En section 7.1, nous décrivons succinctement le langage Q-GDMO-GRM et le langage d'assertions que nous avons choisi pour spécifier les comportements dans Q-GDMO-GRM. Nous donnons en section 7.2 une description générale du simulateur QG^{2}S s'appliquant à des spécifications Q-GDMO-GRM.

1. qui répond au doux nom de QG^{2}S.
2. Telecommunications Information Networking Architecture Consortium

En section 7.3, nous montrons comment nous intégrons le déterminisme explicite défini au chapitre 6 dans les spécifications Q-GDMO-GRM et dans le simulateur QG^2S. Nous définissons en détail la mise en oeuvre de la sémantique opérationnelle du prédicat := définie en section 6.2. Nous décrivons aussi comment nous avons mis en oeuvre le traitement du := et le traitement de la post-condition résiduelle à l'aide d'un oracle (cf. section 6.3).

Le simulateur QG^2S est écrit en ILOG TALK [ILO96b] et utilise la librairie POWER CLASSES [ILO96a]. Ce langage de programmation est une extension d'un langage fonctionnel (langage à la Lisp) et utilise la définition de classes d'objets. De plus, il intègre un MOP (Meta-Object Protocol) [KRB91] à la CLOS [Ste90] qui permet de définir des méta-classes (une classe est instance d'une classe). Dans ce chapitre, nous nous référons à la mise en oeuvre de QG^2S lorsque des fonctions ILOG TALK particulières permettent d'exprimer plus facilement les concepts de simulation définis dans les chapitres précédents.

7.1 Q-GDMO-GRM et langage d'assertions

7.1.1 Le langage de spécifications Q-GDMO-GRM

Le langage Q-GDMO-GRM est le langage de spécifications de la vue information du modèle TINA (Telecommunications Information Networking Architecture) [Tel95]. Le consortium TINA-C se fonde sur le modèle ODP pour spécifier et construire une application de télécommunications. Une description complète de Q-GDMO-GRM est donnée dans l'annexe A. Dans la suite de cette sous-section, nous décrivons rapidement Q-GDMO-GRM.

Q-GDMO-GRM est un langage de *formulaires* définissant des types d'objets gérés (formulaire OBJECT TYPE) et des relations entre objets (formulaires RELATIONSHIP TYPE, GENERIC RELATIONSHIP TYPE et ROLE BINDING). La figure 7.1 donne la spécification d'un objet *Alarme* de l'application FTMN-Alarmes[3]. Les formulaires de relations sont des formulaires d'objets comportant des *rôles*. Un rôle spécifie un type d'objet entrant dans la relation ainsi que la cardinalité du rôle (le nombre d'objet de ce type pouvant être en relation).

Un formulaire Q-GDMO-GRM est constitués d'attributs (`compteurRepetition`, `etatAcquitement`, etc.) et de services (`terminer`, `get-dateAcquitement`, etc.) sur l'objet spécifié. Les attributs Q-GDMO-GRM sont associés à un type (étiquette `PERMITTED VALUES`). Le langage de type préconisé dans Q-GDMO-GRM est l'ASN.1 [ASN94].

Des propriétés sont attachées aux attributs d'un objet par le biais des options `INITIAL VALUE`, `GET`, REPLACE, `GET-REPLACE`, ADD, REMOVE et ADD-REMOVE. L'option `INITIAL VALUE` permet de donner une valeur initiale à l'attribut. Les autres op-

3. la classe *Alarme* n'est pas totalement spécifiée dans cette figure.

```
Alarme OBJECT TYPE
       CHARACTERIZED BY pac PACKAGE
        BEHAVIOUR beh BEHAVIOUR DEFINED AS !
create-Alarme(n:TINT,cP:String,pS:String,...):(e:Alarme)
        POSTCONDS: e.numero := n and e.causeProbable := cP and ... ;;
get-dateAcquitement():(r:Date)
        PRECONDS: etatAcquitement = 0;;
acquitter()
        PRECONDS: etatTerminaison = 1 ;
        POSTCONDS: etatAcquitement' := 0 ;;
terminer()
        POSTCONDS: etatTerminaison' := 0;
... ! ;
        ATTRIBUTES
                numero
                        PERMITTED VALUES: TINT
                        GET-REPLACE,
                etatTerminaison
                        PERMITTED VALUES: Etat
                        INITIAL VALUE: 1
                        GET,
                dateAcquitement
                        PERMITTED VALUES: Date
                        GET,
                etatAcquitement
                        PERMITTED VALUES: EtatAcq
                        INITIAL VALUE: 1
                        GET,
                ...;
       ACTIONS
                acquitter(),
                terminer(),
                lancerGestionTI(),
                localiser();
        NOTIFICATIONS ;;
REGISTERED AS { fTMN 3 } ;
```

FIG. 7.1 – *Spécification Q-GDMO-GRM d'un objet Alarme de FTMN-Alarmes*

tions définissent l'existence de certains services de l'objet concernant l'attribut. Nous les présentons avec ceux des objets Q-GDMO-GRM :

- création et destruction de l'objet (spécifications dans la partie comportement des formulaires) ;

- consultation et modification d'un attribut de l'objet, un attribut peut être consulté ou modifié si et seulement si les options `GET` et `REPLACE` sont spécifiées dans le formulaire Q-GDMO-GRM (cf. figure 7.1) ;
- ajout ou retrait d'un élément d'un attribut de l'objet (dans le cas où l'attribut est de type ensemble), ces services sont autorisés si et seulement si les options `ADD` et `REMOVE` sont spécifiées dans le formulaire Q-GDMO-GRM (cf. figure 7.1) ;
- actions particulières de l'objet (méthode classique des langages à objets), la signature d'une action est spécifiée sous l'étiquette `ACTIONS` du formulaire, le comportement d'une action est spécifiée dans la partie comportement du formulaire ;
- notifications d'événements émis par l'objet, la signature d'une notification est spécifiée sous l'étiquette `NOTIFICATIONS` du formulaire, le comportement d'une notification est spécifiée dans la partie comportement du formulaire.

Une relation en Q-GDMO-GRM est traitée comme un objet à part entière, des services équivalents aux objets sont définissables ainsi que des services propres aux relations (cf. annexe A) :

- création et destruction d'un objet relation (spécifications dans la partie comportements des formulaires) ;
- création et destruction d'un rôle d'un objet relation (spécifications dans la partie comportements des formulaires).

La figure 7.1 montre que les comportements sont définis séparément de la spécification structurelle des objets Q-GDMO-GRM (les comportements sont encadrés par `BEHAVIOUR... ! ... !;`). Entre les points d'exclamation, la description suit une syntaxe précise. En effet, Q-GDMO-GRM structure la description des comportements par le biais d'étiquettes et de signature de services.

Le schéma classique de description des comportements dans Q-GDMO-GRM est le suivant. Un premier comportement (non spécifié dans la classe *Alarme*) est l'invariant de la classe (préfixé par l'étiquette `INVARIANTS:`). Ensuite, chaque service est spécifié avec sa signature et son comportement représenté par un couple de pré/post-conditions (étiquettes `PRECONDS:` et `POSTCONDS:`).

Dans les paragraphes précédents, nous avons énuméré les services possibles d'un objet. Tous ces services n'ont pas toujours de comportements particuliers. Prenons l'exemple de la consultation de l'attribut `problemeSpecifique` d'un objet *Alarme*. Aucune précondition ou post-condition de consultation ne sont nécessaires (vrai pour les deux conditions). Le comportement de cette méthode n'est pas spécifié dans la classe *Alarme*.

Par contre, le comportement de la méthode de consultation de l'attribut `dateAcquittement` est spécifié. En effet, pour obtenir la date d'acquittement d'une alarme,

il faut que cette alarme ait été acquittée. Son comportement est décrit par le service `get-dateAcquittement`.

En Q-GDMO-GRM, certains services (autres que ceux d'actions et de notifications) ont des identifiants et signatures imposées (cas du service `get-dateAcquittement`, au contraire du service `acquitter`). Les identifiants de ces services suivent les règles de syntaxe :

- création d'un objet, d'une relation ou d'un rôle : `create-`{identifiant de la classe ou du rôle} (ex : `create-Alarme`) ;
- destruction d'un objet, d'une relation ou d'un rôle : `delete-`{identifiant de la classe ou du rôle} (ex : `delete-Alarme`) ;
- services d'accès aux attributs : {identifiant du service (`get`, `add`, etc.)}-{identifiant de l'attribut} (ex : `get-dateAcquittement`).

Les signatures de ces services suivent les règles suivantes :

- création d'un objet, d'une relation ou d'un rôle : le premier paramètre de sortie est l'objet, la relation créé ou le rôle ajouté à la relation ;
- destruction d'un objet, d'une relation ou d'un rôle : le premier paramètre d'entrée est l'objet, la relation à détruire ou le rôle à retirer de la relation ;
- services d'accès aux attributs : un seul paramètre pour ces services. Pour le service de consultation, ce paramètre est en sortie et représente la valeur de l'attribut. Pour les autres services, ce paramètre est en entrée et représente la nouvelle valeur de l'attribut dans le cas d'une modification de l'attribut, d'un élément à rajouter ou à retirer pour les services `add-x` et `remove-x`.

Cette structuration est imposée par Q-GDMO-GRM, elle est décrite dans la première partie de l'annexe B.1. En revanche, [TIN95] n'impose pas un langage particulier pour décrire les comportements. Nous introduisons un langage d'assertions permettant de spécifier les comportements. Nous en discutons en sous-section 7.1.2.

7.1.2 Formalisation des comportements de Q-GDMO-GRM

Dans le chapitre 5, nous avons présenté un langage d'assertions fondé sur la syntaxe du langage des prédicats du premier ordre et augmenté du prédicat :=. Dans le chapitre 6, nous avons défini sa sémantique opérationnelle. Nous reprenons ce langage pour formaliser les comportements de Q-GDMO-GRM. Nous remplaçons seulement les symboles des

connecteurs et quantificateurs définis dans les chapitres 5 et 6 par leurs noms en anglais (`and` pour $\wedge$, `forall` pour $\forall$, etc.). En annexe B.2, nous donnons sa grammaire.

Les données prises en compte dans le langage d'assertions sont des données ASN.1. Il incorpore des opérateurs permettant de manipulées les données ASN.1 (les opérateurs arithmétiques, les opérateurs traitant les chaînes de caractères, etc.).

Des constantes de fonctions (autres que les opérateurs cités auparavant) peuvent apparaître dans les spécifications. Ainsi, dans l'exemple suivant :

```
POSTCONDS: r := RetraitElement(x,eaq)
```

l'appel à la constante de fonctions `RetraitElement` est syntaxiquement valide. Cependant, lors de la simulation, cette fonction doit être implantée.

Une autre particularité du langage d'assertions est de ne pas déclarer l'objet sur lequel on exécute les services dans les spécifications des comportements. Par exemple, dans la classe Alarme spécifiée dans la figure 7.1, le comportement du service `acquitter` contient des références aux attributs `etatTerminaison` et `etatAcquitement` sans les préfixer par un nom d'objet (la figure 7.2 rappel se comportement). Cette facilité d'écriture a une incidence sur la description des variables d'états définie en sous-section 5.1.2. En effet, une variable d'état peut être un simple identifiant (même description qu'une variable liée). Le simulateur intègre (à la compilation) la reconnaissance de ces variables ainsi que la nouvelle description de *nouvelle valeur de variable* (cf. sous-section 6.1.1) de telles variables d'états (un exemple de cette description est donnée dans la post-condition du service `acquitter`, cf. figure 7.2).

```
acquitter()
        PRECONDS: etatTerminaison = 1 ;
        POSTCONDS: etatAcquittement' := 0 ;;
```

FIG. 7.2 – *Service acquitter d'un objet Alarme*

7.2 Description de QG^{2}S

Dans la figure 3.2 définie dans la section 3.2, nous avons décrit les modules permettant d'effectuer la mise au point de spécifications. Dans cette section, nous décrivons le simulateur QG^{2}S à l'aide de ces différents modules.

Pour valider statiquement des spécifications, il suffit de vérifier leur syntaxe par rapport à celle du langage et de vérifier qu'elle respectent la sémantique statique de celuici. Nous détaillons le module de compilation des spécifications Q-GDMO-GRM dans la sous-section 7.2.1.

Le but de la validation dynamique des spécifications Q-GDMO-GRM est de vérifier que le comportement d'une application est respecté au cours du déclenchement de services (requêtes). Une requête est par exemple une création d'objet, une destruction d'objet, une modification d'un attribut d'un objet, etc. Nous détaillons les requêtes dans la sous-section 7.2.2.

Les requêtes permettent de déclencher les services des objets implantés dans le simulateur. Elles sont donc traduites en appels aux services des objets. Ces appels sont interprétés par QG^2S. Dans la sous-section 7.2.3, nous présentons le traitement des requêtes de l'application simulée, c'est-à-dire l'accès aux objets, l'accès aux comportements, etc.

L'utilisateur du simulateur doit pouvoir à tout moment contrôler l'application simulée. QG^2S offre un ensemble d'outils de visualisation et de modifications de ses spécifications, de visualisation de son état (ses objets et leurs contenus) et un gestionnaire des scénarios de requêtes. En sous-section 7.2.4, nous présentons ces outils au travers des IHM de QG^2S.

7.2.1 Compilation des spécifications Q-GDMO-GRM

Le module *compilation* du simulateur QG^2S permet l'analyse de la syntaxe des spécifications Q-GDMO-GRM (composée d'une analyse lexicale et d'une analyse syntaxique) et permet l'analyse de la sémantique statique des spécifications. La figure 7.3 montre le schéma de principe du module de compilation.

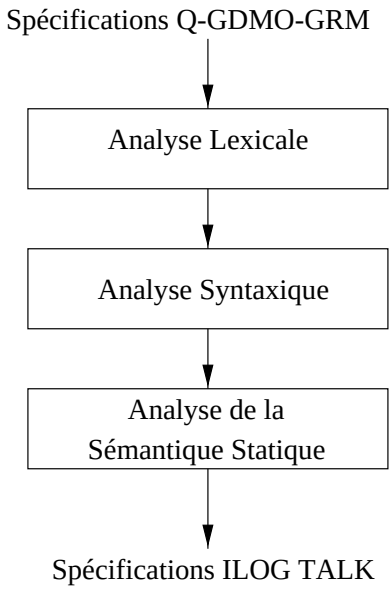


FIG. 7.3 – *Module de compilation de QG^2S*

Analyse lexicale et syntaxique

Les formulaires Q-GDMO-GRM sont traduits en classes ILOG TALK. Au sein du simulateur QG^2S, un objet d'une application est en fait une instance d'une classe ILOG TALK. Les attributs des formulaires sont traduits en attributs de classes. Au contraire des attributs, les services ne sont pas codés comme des méthodes de classes (ILOG TALK définit des méthodes génériques), mais comme des *propriétés* de la classe de l'objet. Chaque classe C est instance d'une classe $Meta - C$ (notion de méta-classe). On sauvegarde dans les champs de l'*objet* C les signatures ainsi que les comportements des services du formulaire Q-GDMO-GRM.

Le choix de sauvegarder les comportements dans les instances de classes permet :

- de limiter le nombre de création de méthodes (gain aussi au niveau de la gestion des méthodes). De plus, on ne crée la fonction d'un comportement qu'au moment de son utilisation ;
- d'effectuer de l'héritage dynamique. On ne duplique pas le code des méthodes mères. Cela permet une modification dynamique des comportements d'une classe sans modification des classes filles. Cette modification doit être contrôlée (cf. sous section 7.2.4).

Sur l'exemple du formulaire `Alarme` (cf. figure 7.1), ce formulaire est traduit en classe TALK de même nom (Alarme) d'attributs *numero, etatTerminaison,* etc. Les services sont sauvegardés dans la méta-classe de la classe Alarme. Les champs *actions, notifications,* etc. de l'objet classe Alarme contiennent les traductions TALK des comportements.

La traduction des comportements est effectuée en code ILOG TALK. Par exemple, la pré-condition du service `acquitter` présentée dans la figure 7.2 devient :

```
(equal (slot-value ob 'etatTerminaison) 1)
```

Lors de l'appel au service `acquitter` d'un objet A de classe Alarme, le simulateur QG^2S construit la fonction *anonyme*[4] prenant en entrée le paramètre `ob` de type Alarme et contenant le code précédemment cité. QG^2S interprète cette nouvelle fonction en prenant en entrée l'objet A et en rendant le résultat t (vrai) si le champ `etatTerminaison` est de valeur `1`, () sinon.

Pour traduire des formulaires Q-GDMO-GRM en TALK, nous avons utilisé l'outil de génération d'analyseurs lexicaux et syntaxiques PCCTS (Purdue Compiler-Compiler Tool Set) [Par96]. PCCTS permet de les concevoir à l'aide de LEX et YACC, les générateurs d'analyseurs lexicaux et syntaxiques de référence[5]. PCCTS génère des analyseurs en C,

4. une fonction anonyme en ILOG TALK ne possède pas d'identifiant mais peut être référencée par des variables.

5. PCCTS intègre et améliore ces deux générateurs.

C++ et JAVA. L'environnement ILOG TALK intègre facilement des modules définis en C ou C++. Nous avons généré en C le compilateur de spécifications Q-GDMO-GRM de QG^2S.

Q-GDMO-GRM utilise le standard ASN.1 pour représenter les types et données des attributs et paramètres des services des objets. Pour l'instant, tous les types ASN.1 ne sont pas supportés par QG^2S. Nous donnons en annexe C les types et données ASN.1 supportés par QG^2S.

Analyse sémantique

L'analyse sémantique est soit statique soit dynamique. La sémantique dynamique est validée par le simulateur lors des tests des services des applications. La sémantique statique peut être effectuée lors de la compilation des spécifications.

Le module d'analyse sémantique défini dans le schéma 7.3 permet de vérifier la sémantique statique du langage Q-GDMO-GRM, c'est-à-dire :

- le typage des attributs et des paramètres des services ;
- la double déclaration d'un formulaire ;
- etc.

La sémantique statique du langage Q-GDMO-GRM est définie formellement en Z dans [CJ97]. Le simulateur QG^2S l'intègre. Nous ne décrivons pas précisément dans ce document ce traitement, nous notons simplement les points d'implantations suivants de l'analyse de la sémantique statique de Q-GDMO-GRM dans QG^2S :

- le typage des attributs et paramètres de services définis dans les comportements est vérifier dynamiquement à l'exécution des services. Par exemple, si l'on veut affecter une valeur non conforme à un attribut, l'interpréteur rend une erreur de type ILOG TALK ;
- l'héritage des formulaires est vérifié à la compilation mais la composition des comportements (cf. sous-section 4.1.2) est effectuée au cours des appels aux services des objets.

7.2.2 Les requêtes

Le langage Q-GDMO-GRM ne fournit pas un langage de requêtes (encapsulation des services). Cependant, Q-GDMO-GRM est un regroupement plus ou moins fidèle des

langages GDMO et GRM (cf. section 2.2). Nous donnons les correspondances entre Q-GDMO-GRM, GDMO et GRM en annexe A.2. Le langage des requêtes associé à GDMO et GRM est fourni par la norme CMIS. Nous avons donc repris ce langage de requêtes pour effectuer les appels aux services des objets dans QG^2S.

Les requêtes sont tout d'abord traduites en *macros* ILOG TALK. Dans QG^2S, ces dernières permettent de factoriser les appels de services par types (création, destruction, consultation, modification, etc.). Pour chaque type de requêtes, une macro est définie. Elle permet de traiter dynamiquement l'héritage, les appels aux comportements, etc.

Dans la sous-section 7.2.3, nous présentons en détail le contenu des macros. L'intérêt majeur de les utiliser est la factorisation des services des objets. Elles permettent de spécifier un type de service de manière *générique* et à chaque appel d'un service de ce type de construire et d'interpréter ce service dynamiquement (au cours de l'appel). Par exemple, la figure 7.4 présente deux appels de macros de création d'un objet *Alarme* et d'un objet *Ressource* de l'application FTMN-Alarmes. Les données en paramètre sont de types ASN.1. Elles sont dynamiquement compilées au cours des appels aux macros.

```
(CREATE alarme1 Alarme 12 "Alimentation coupee"
                       "Perturbation climatique" 3 3 2 "indeterminee"
                       "reparer" [ jour 25 , mois 12 , annee 97 ]
                       1 "pas de commentaire")

(CREATE ressource1 Ressource "MT25" "res1" "CODE45" "Brest")
```

FIG. 7.4 – *Appels de macros TALK pour la création d'un objet Alarme et d'un objet Ressource.*

Dans la sous-section 7.2.4, nous présentons l'IHM permettant de construire les requêtes. Notons que l'on peut définir dans un fichier les requêtes avec la syntaxe partiellement définie dans la figure 7.4. Cependant, l'IHM de création des requêtes est plus conviviale pour effectuer cette tâche.

7.2.3 Module de Traitement

Le module de traitement permet d'exécuter les services de l'application simulée sur les instances ILOG TALK des objets de l'application. La figure 7.5 présente le module de traitement. Nous présentons au préalable l'environnement de simulation puis le traitement de l'exécution des services d'une application dans cet environnement.

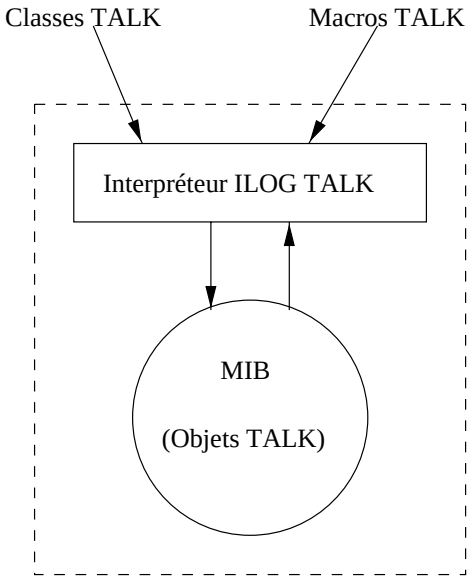


FIG. 7.5 – *Description du module de traitement de QG^2S*

Environnement de simulation

L'environnement de simulation de QG^2S contient :

- les classes ILOG TALK, traductions de formulaires Q-GDMO-GRM ;
- un référentiel d'instances ILOG TALK des objets de l'application simulée. Les clés d'entrée de ce référentiel sont les noms des objets. Dans la figure 7.4, les paramètres `alarme1` et `ressource1` sont des noms d'objets ;
- l'interpréteur ILOG TALK qui permet d'évaluer les appels de macros.

Notons que dans la figure 7.5, le référentiel d'objets est représenté par l'ensemble *MIB*. Nous avons donné ce nom à cet ensemble en référence au monde des télécommunications. La MIB (Management Information Base) [ITU92b] représentent les objets d'une application de gestion de télécommunications[6].

Traitement de la dynamique

Le traitement d'une requête passe d'abord par son analyse. Une requête porte sur un objet ou une relation, il faut alors vérifier que cet objet ou cette relation existe (sauf dans le cas d'une création). Ensuite, il faut vérifier si la requête est sémantiquement correcte,

6. Le terme *MIB* est souvent surchargé car on peut aussi l'utiliser pour dénommer l'ensemble des spécifications d'une application.

c'est-à-dire qu'elle respecte la spécification de l'objet ou de la relation. Par exemple, il est possible d'effectuer une consultation de l'attribut d'un objet si et seulement si la propriété GET est attribuée à l'attribut.

Chaque type de service (cf. sous-section 7.2.2) a un algorithme de traitement de la sémantique dynamique. Ces algorithmes sont codés par la macro correspondant au type de service. Les différents algorithmes ont la trame suivante[7] :

1. vérification de l'existence de l'objet concerné par la requête (sauf pour les requêtes de création d'objet et de relation), l'identifiant de cet objet est le premier paramètre de la requête ;
2. traitement dynamique de l'héritage des comportements. Notons ici qu'un objet peut être considéré comme un objet d'une de ces classes mères (concept de l'héritage). Les requêtes ont pour deuxième paramètre l'identifiant d'une classe. La classe considérée est soit la classe de l'objet référencé en premier paramètre de la requête soit une de ces classes mères. Dans le cas d'une classe mère, les comportements pris en compte sont ceux de la classe mère ;
3. vérification de l'invariant (cf. précisions à la fin de cette énumération) ;
4. vérification de la pré-condition du service de l'objet représenté par la requête (pré-condition complète, i.e. la pré-condition de la classe courante plus celles des classes mères si la méthode y est spécifiée) ;
5. prise en compte du déterminisme explicite (cf. section 7.3) sur la MIB ;
6. vérification de la post-condition du service de l'objet représenté par la requête (post-condition complète, i.e. la post-condition de la classe courante plus celles des classes mères si la méthode y est spécifiée) ;
7. si la post-condition n'est pas vérifiée, retour à l'état précédent l'exécution de la requête. Cette étape est assurée par le service de persistance d'objets de ILOG TALK ;
8. si la post-condition est vérifiée, vérification de l'invariant (cf. précisions à la fin de cette énumération) ;
9. si l'invariant n'est pas vérifié, retour à l'état précédent l'exécution de la requête ;
10. si l'invariant est vérifié, traitement par l'*oracle* des variables d'états décorées mais non explicitement déterminées (cf. section 6.3).

L'invariant vérifié est celui de l'application, c'est-à-dire tous les invariants d'objets de l'application. QG^2S vérifie donc chacun d'eux. En effet, au cours de l'exécution d'un service d'un objet, d'autres objets peuvent être modifiés.

7. une relation est considérée comme un objet dans cette énumération.

La vérification de l'invariant est effectuée deux fois : avant celle de la pré-condition et après celle de post-condition. Cette vérification est historique car le simulateur fonctionne actuellement de manière synchrone (une requête est traitée à la fois), c'est à dire qu'il est suffisant de l'effectuer après celle de la post-condition.

Le simulateur QG^{2}S propose de traiter les variables d'états décorées mais non explicitement déterminées en faisant appel à l'utilisateur. Dans la sous-section 7.3.2, nous présentons plus en détails la résolution des valeurs de ces variables. Lorsque l'oracle a fourni toutes les valeurs de ces variables, QG^{2}S reprend la trame de l'algorithme de traitement du service à l'étape 7, c'est-à-dire qu'il vérifie la post-condition du service et l'invariant de l'application avec ces nouvelles valeurs.

7.2.4 L'IHM de QG^{2}S

L'IHM de QG^{2}S permet à l'utilisateur d'avoir une vue de l'application simulée et d'interagir sur la simulation (activer des requêtes, fournir de nouvelles spécifications et jouer le rôle d'oracle). La mise en oeuvre de l'IHM s'appuie sur la librairie C++ ILOG VIEWS [ILO95]. L'intégration des classes d'objets ILOG VIEWS dans l'interpréteur ILOG TALK y est assurée par une librairie de fonctions TALK correspondant aux méthodes des classes ILOG VIEWS standards.

Visualisation des objets

QG^{2}S permet la visualisation des objets de la MIB ainsi que de leurs états. La fenêtre d'entrée de QG^{2}S offre un ensemble de menus permettant de gérer la simulation et un espace de visualisation des objets de l'application simulée. La figure 7.6 donne une vue de la simulation de l'application FTMN-Alarmes. La vue d'une application est constituée de la représentation des objets par des rectangles rouges (foncés sur la figure) et des objets de relations par des rectangles verts (clairs sur la figure). Les objets de relations sont liés aux objets en relations par des arrêtes *labelisées* par les noms des rôles (exemples de labels : `Origine`, `Historique`).

Le simulateur permet de visualiser l'état d'un objet en sélectionnant sa représentation dans la fenêtre d'entrée de QG^{2}S (cf. figure 7.6). Il permet aussi de visualiser sa spécification Q-GDMO-GRM dans le fichier de spécifications d'origine. Un sous menu de QG^{2}S permet de rechercher tout formulaire Q-GDMO-GRM à partir de son identifiant.

Le simulateur permet de sauvegarder une application simulée, c'est-à-dire l'ensemble des états de l'application de la première à la dernière requête au moment de la sauvegarde. Le service de persistance de ILOG TALK permet de sauvegarder des objets dans un fichier. Lorsque l'on veut reprendre la simulation d'une application, il suffit de charger ce fichier, c'est-à-dire les classes ILOG TALK correspondant aux formulaires Q-GDMO-GRM de l'application et les scénarios de requêtes liés à l'application.

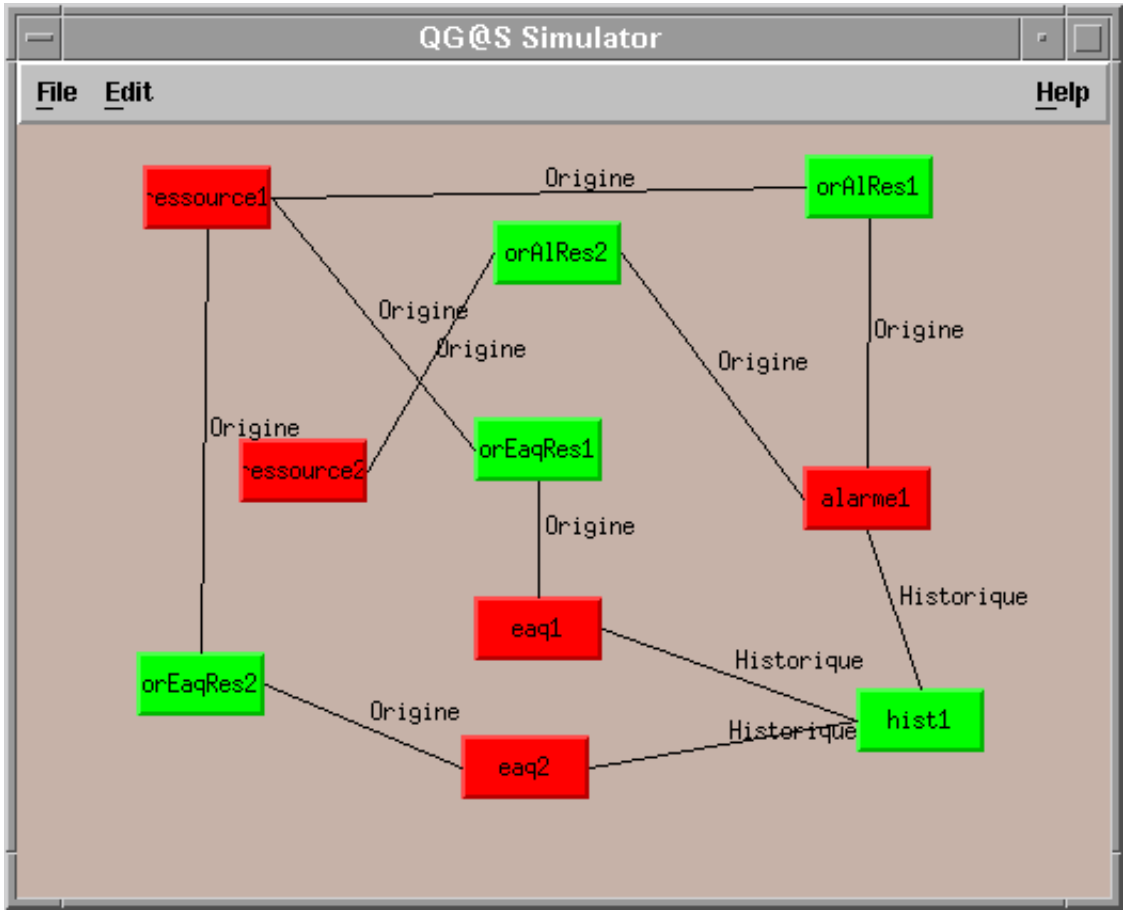


FIG. 7.6 – *Fenêtre de visualisation des objets d'une application*

Cependant, notons que dans le cadre d'une modification d'une spécification il est préférable de recommencer toute la simulation. En effet, des états antérieurs peuvent être faux par la modification apportée. Pour des raisons de cohérence, nous conseillons de re-simuler l'application courante traitée par QG²S. Pour des applications importantes, re-simuler est une contrainte lourde, nous pouvons imaginer la définition de *points de reprise*. QG²S n'intègre pas une telle définition.

Gestion des spécifications Q-GDMO-GRM et des scénarios de requêtes

Par le menu *File* de la fenêtre principale de QG²S (cf. figure 7.6), on peut à tout moment insérer de nouvelles spécifications Q-GDMO-GRM. Ce menu fait appel au module de compilation de QG²S (cf. sous-section 7.2.1).

Pour gérer les requêtes et les scénarios, QG²S offre une IHM permettant de les éditer, de les modifier et de les créer (il existe une fenêtre IHM spécifique pour chaque type de requête). Il est possible d'avoir plusieurs scénarios valides pour une même simulation. De

plus, on peut les composer (intégration d'un scénario dans un autre).

Les scénarios sont sauvegardés dans des fichiers. Nous conseillons de les construire à partir de l'IHM, des facilités sont offertes pour concevoir les requêtes le plus rapidement possible (par exemple, pour une requête de consultation d'un attribut d'un objet, le choix de l'objet donne accès à sa classe et à la liste de ses attributs que l'on peut consulter, etc.). La figure 7.7 donne le scénario permettant la construction des objets représentés graphiquement dans la figure 7.6.

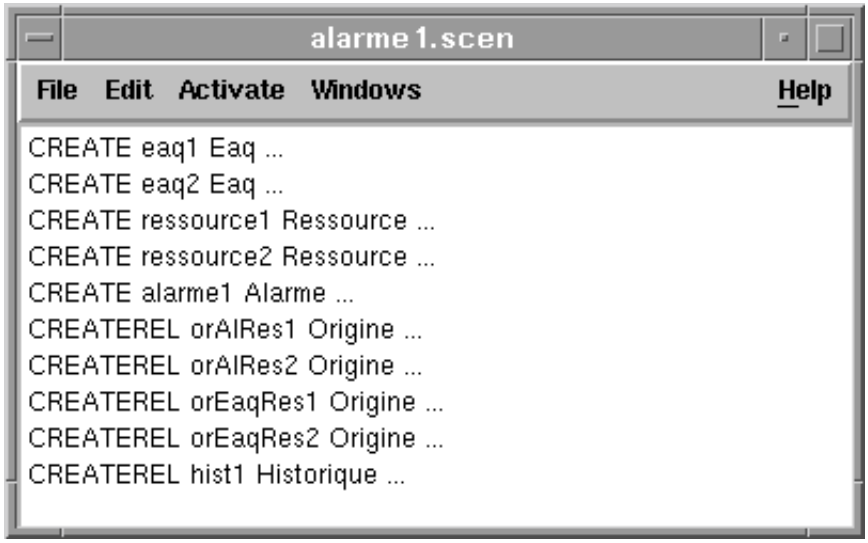


FIG. 7.7 – *Fenêtre de visualisation d'un scénario de QG^2S*

La fenêtre de visualisation d'un scénario offre un menu permettant d'exécuter ses requêtes (menu *Activate*). Trois choix d'exécution sont proposés :

- la requête courante (requête sélectionnée dans le scénario) ;
- pas à pas (requête par requête) ;
- tout le scénario.

L'exécution la requête courante revient à faire l'appel à la macro correspondant au type de requête (`CREATE`, etc) avec les paramètres de la requête (`eaq1`, etc). Cet appel est interprété et le résultat est répercuté au niveau de l'IHM du simulateur QG^2S. Par exemple, la création d'un objet modifie la MIB mais aussi rajoute la représentation graphique de l'objet dans la fenêtre principale de QG^2S.

7.3 Le déterminisme explicite et QG^2S

Le langage d'assertions que nous avons introduit dans Q-GDMO-GRM (cf. sous section 7.1.2) permet d'introduire des assertions atomiques définies sur le prédicat :=. Un exemple de l'utilisation du prédicat := dans les spécifications Q-GDMO-GRM est donné dans le comportement de la classe *Alarme* (cf. figure 7.1).

Le déterminisme explicite n'apparaît que dans les post-conditions. La compilation des formulaires Q-GDMO-GRM vérifie qu'aucunes occurrences des symboles := et $'$ n'apparaissent dans les invariants et pré-conditions des services des objets.

Le traitement par le simulateur QG^2S des post-conditions contenant du déterminisme explicite s'effectue d'abord à la compilation des spécifications Q-GDMO-GRM. Le traitement des contraintes liées au déterminisme explicite est décrit par la sémantique opérationnelle définie dans le chapitre 6. Une partie de la sémantique opérationnelle est traitée lors de la compilation des post-conditions :

- on vérifie que les contraintes de l'utilisation du := (cf. section 6.1) sont respectées ;
- un code ILOG TALK particulier est généré pour une post-condition :
  - une partie du code permet *d'exécuter* la partie explicitement déterministe de la post-condition ;
  - l'autre partie contient le code de la post-condition résiduelle.

Dans les sous-sections suivantes, nous décrivons l'implantation dans le simulateur QG^2S de la fonction sémantique $\mathcal{E}$ décrite dans la sous-section 6.2.2, c'est-à-dire celles de $\mathcal{S}$ et $\mathcal{R}$.

7.3.1 Mise en oeuvre de $\mathcal{S}$

La mise en oeuvre de la sémantique opérationnelle représentée par la fonction $\mathcal{S}$ revient à implanter les fonctions **DetExp?**, **InitEtat**, **Eval** et les fonctions d'affectations des nouvelles valeurs dans l'état de l'application.

- **DetExp?** permet de définir si une post-condition contient du déterminisme explicite. Lors de la compilation, on isole les assertions atomiques explicitement déterminisme, c'est-à-dire que chaque post-condition TALK est un couple, le premier élément contient l'explicite, le second le reste ;
- **InitEtat** permet de créer le nouvel état de l'application simulée et valorise à $\perp$ ses valeurs de variables. Le service de persistance ILOG TALK permet de décider à

quel moment une application exécutée sur l'environnement change d'état. Ce service offre la possibilité de gérer son graphe d'états. Le passage d'un état à un autre est réalisé à chaque appel d'un nouveau service. Nous n'en créons donc pas un nouveau comme le préconise la fonction **InitEtat**, mais nous passons d'un état à l'autre par le biais du service de persistance ;

- **Eval** permet d'évaluer les nouvelles valeurs des variables d'états explicitement déterminées, les valeurs des indices de collection, les valeurs des ensembles d'itérations associés aux assertions quantifiées universellement et les valeurs des conditions des conditionnelles. Cette fonction porte sur des assertions traduites en code ILOG TALK dans QG²S. Le rôle de la fonction **Eval** revient à interpréter ce code ;
- les affectations dues au déterminisme explicite sont effectuées par les fonctions d'affectation d'attributs d'objets ILOG TALK (les types ensembles et séquences ASN.1 sont traduits classes ILOG TALK).

L'utilisation du service de persistance ne permet pas de déceler au préalable l'*aliasing* d'une post-condition (cf. sous-section 6.2.2). En effet, les affectations des variables d'états explicitement déterminées sont réalisées directement sur l'état courant du simulateur, c'est-à-dire qu'aucune valeur d'une variable d'état n'ait de valeur $\perp$. Nous reprenons l'exemple des deux objets o_1 et o_2 possédant des attributs a_1 et a_2 référençant le même objet o. Cet objet possède un attribut d de type entier. L'assertion :

$$(o_1.a_1)'.d := 4 \wedge (o_2.a_2)'.d := 6 \tag{7.1}$$

montre la double référence de l'attribut d de o.

Dans la fonction sémantique $\mathcal{S}'$, lors de l'évaluation de $(o_2.a_2)'.d := 6$, on teste si la valeur de $(o_2.a_2)'.d$ est $\perp$. Si ce test est faux, l'*aliasing* est décelé. Dans QG²S, ce test n'est pas possible. Pour y remédier, nous détectons l'*aliasing* à postiori. Chaque assertion atomique explicitement déterminée de la post-condition est traduite dans la post-condition résiduelle en remplaçant le prédicat := par le prédicat d'égalité. Par exemple, le post-condition résiduelle de la post-condition 7.1 est :

$$(o_1.a_1)'.d = 4 \wedge (o_2.a_2)'.d = 6 \tag{7.2}$$

L'évaluation de l'assertion résiduelle 7.2 rend le résultat faux, le résultat attendu.

Un autre détail de mise en oeuvre dû à l'utilisation du système de persistance de ILOG TALK porte sur l'implantation des fonctions **Eval** et **AccAnc** (fonction d'accès dans l'état avant exécution d'un service). Dans ces fonctions, l'accès à des anciennes valeurs de variables d'états (variables non décorées) requièrent l'utilisation des fonctions de persistance d'accès à l'ancien état.

7.3.2 Mise en oeuvre de $\mathcal{R}$

La fonction sémantique $\mathcal{R}$ permet de :

- valuer les variables d'états non décorées dans la post-condition ou n'y apparaissant qu'à l'aide de leurs anciennes valeurs ;
- fournir la post-condition résiduelle ne contenant que des variables décorées et non explicitement déterminées.

Le système de persistance de ILOG TALK ne permet pas d'implanter telle quelle la fonction sémantique $\mathcal{R}$. La valorisation des variables d'états non décorées est obtenue sans modifier les valeurs des variables. En effet, avant l'exécution du service, le nouvel état est une copie conforme de l'ancien état.

Dans la post-condition résiduelle, les variables décorées et explicitement déterminées y apparaissent aussi (cf. sous-section 7.3.1). Pour différencier si une variable d'état décorée a été explicitement déterminée ou non, le système de persistance offre un service permettant de définir si une valeur de l'état courant a été modifiée ou non par rapport à l'état précédent. Le simulateur traite donc une variable décorée x suivant les deux cas suivants :

- x est signalée modifiée dans l'état courant, l'occurence x' (nouvelle valeur de x) de la post-condition résiduelle est remplacée par la valeur de x dans l'état courant ;
- x n'est pas signalée modifiée dans l'état courant, une fenêtre IHM est lancée et demande à l'utilisateur de fournir une nouvelle valeur à x ou de préserver l'ancienne (rôle de l'oracle).

7.3.3 Les variables d'états de Q-GDMO-GRM

Dans cette sous-section, nous présentons la syntaxe et la sémantique des variables d'états de Q-GDMO-GRM. Dans la sous-section 5.1.3, nous donnons une description des variables d'état dans un monde objet. Une variable d'état est du type *o.a*. Dans le comportement d'un objet Q-GDMO-GRM, les occurrences des attributs de l'objet ne sont pas précédés de la référence du nom de l'objet. Dans l'exemple du service `acquitter` 7.2 du formulaire `Alarme`, les attributs `etatTerminaison` et `etatAcquittement` ne sont précédés du nom l'objet courant. La décoration porte alors directement sur l'attribut (`etatAcquittement'`).

Les services des objets ont des paramètres. Ils peuvent être d'entrée et de sortie. Ils sont considérés comme des variables d'états (ils peuvent référencés des objets de l'applications). Leur sémantique est la suivante :

- les paramètres de sortie ne peuvent être référencés que dans les post-conditions. Par

contre, ils ne sont pas décorés, mais peuvent apparaître dans la partie gauche du prédicat :=. Un exemple est donné dans la post-condition du service de création de la classe Alarme (cf. figure 7.1) ;

- les paramètres d'entrée peuvent apparaître dans la pré-condition et la post-condition du service. Ils sont considérés comme de sortie. Ils peuvent apparaître décorés dans la post-condition du service et en partie gauche du prédicat :=.

Chapitre 8

CONCLUSION

La simulation de spécifications présentée dans cette thèse permet de valider les spécifications d'une application avant d'en commencer le développement. La difficulté principale de ce type de simulation réside dans la formalisation de la dynamique des applications. Dans cette thèse, nous nous sommes attachés à étudier les travaux existants sur la formalisation des comportements des applications de gestion des réseaux de télécommunications.

De cette étude, nous avons tout d'abord montrer l'intérêt de la formalisation des comportements pour leur simulation. Cependant, la spécification formelle des comportements n'est pas toujours suffisante. Elle n'est pas toujours exécutable. Dans cette thèse, nous avons défini une aide à la simulation de spécifications basée sur l'extension *opérationnelle* de la spécification formelle des comportements des applications. Dans la section 8.1, nous concluons sur les objectifs et apports de notre approche.

Actuellement, nous l'avons implanté sur le langage de spécification Q-GDMO-GRM. Une des perspectives de notre travail est de l'implanter pour d'autres langages de spécifications. Dans la section 8.2, nous présentons les diverses perspectives d'application et d'amélioration de notre approche.

8.1 Apports

Les langages de spécifications actuels sont divers. Dans le domaine des télécommunications, les langages majoritairement utilisés pour spécifier les applications sont des langages semi-formels (UML, GDMO, CORBA/IDL, etc.). Quelques méthodes formelles ont été standardisées pour spécifier les comportements des applications (VDM-SL, LOTOS, etc.). Cependant, l'utilisation des méthodes formelles par les industriels est en générale frileuse. Dans cette thèse, nous avons tenté de leur offrir une solution intermédiaire en intégrant des spécifications formelles de la dynamique des applications dans les langages

semi-formels.

Les langages de spécifications semi-formels permettent de fournir une structuration des applications sous formes d'objets communicants (pour la plupart des langages de spécifications). Les comportements, quand ils sont spécifiés, le sont en langue naturelle. Dans cette thèse, nous avons montré qu'il est possible de formaliser les comportements dans le cadre de ces langages. Notre approche est de formaliser les comportements à l'aide du schéma classique *invariants, pré/post-conditions*. Nous avons défini un langage d'assertions classique (prenant source dans l'alphabet du langage des prédicats du premier ordre) pour définir les invariants et pré/post-conditions.

La simulation permet de valider les besoins et fonctions des applications avant leur mise en service. Au cours de la réalisation des applications, plus tôt cette validation est effectuée, plus le coût de correction des besoins et fonctions est moindre. Pour cela, une solution est d'exprimer formellement la sémantique d'une application et d'en vérifier la cohérence. Les méthodes formelles offrent des notations logico-mathématiques permettant d'exprimer ainsi la sémantique d'une application. Par raffinement de ces spécifications, on obtient des implantations respectant cette sémantique.

Des méthodes formelles actuelles permettent d'accompagner les différentes phases de raffinement des spécifications. Un exemple est la méthode B [Abr96]. Le but de cette méthode est d'utiliser une seule notation pour définir les diverses phases de réalisation d'une application. Les spécifications d'une phase sont un raffinement de la précédente et elles doivent respecter des obligations de preuves.

Les concepts définis dans la méthode B remettent en cause la vue qu'on encore bon nombre d'industriels du processus de réalisation des applications à l'aide de méthodes formelles. Au contraire des méthodes traditionnelles, les méthodes formelles n'offrent pas d'implantations partielles (de prototypes) des applications (notons qu'elles réduisent considérablement la phase de *debug*, tests opérationnels). Le processus de réalisation en B d'une application permet de raffiner au plus tôt certaines parties de l'application rendant ainsi possible des tests opérationnels.

Ce que nous proposons dans cette thèse est une extension des spécifications formelles permettant de distinguer le déterminisme des spécifications lors de la simulation. Nous pensons que ce travail est un intermédiaire entre les méthodes formelles à la B et la pratique courante de mise au point des implantations par les industriels. En effet, le simulateur QG^2S permet aux utilisateurs de raffiner *opérationnellement* leurs applications dans les spécifications et non dans les implantations. Par contre, les *obligations de preuves* du raffinement sont effectuées par les utilisateurs.

Pour conclure, notre approche de la validation d'application de gestion des réseaux de télécommunication offre un environnement de validation proche de ceux des méthodes formelles, mais sans prétention d'offrir un système de preuves. Par contre, elle s'intègre dans les langages de spécifications les plus utilisés actuellement pour spécifier les applications de gestion. De plus, son apprentissage est simple car proche des descriptions des

fonctions des applications.

8.2 Perspectives

Nous avons proposé dans cette thèse l'utilisation de langages d'assertions dans les langages dit d'interfaces. Nous l'avons appliqué à Q-GDMO-GRM. Une perspective intéressante est de montrer que l'on peut aussi en enrichir d'autres.

- Le travail précurseur de cette thèse portait sur le langage GDMO. L'amendement [GDM97] définit une extension syntaxique de GDMO à la spécification des comportements. Cette extension suit la structuration des comportements proposé dans cette thèse et dans Q-GDMO-GRM (i.e. la définition sous forme du schéma *invariants, pré/post-conditions*). Le langage d'assertions proposé est proche du langage d'assertions que nous proposons dans cette thèse.
- Une perspective intéressante est de proposer une extension du langage d'interfaces CORBA/IDL permettant de spécifier les comportements des services. En effet, d'un point de vue fonctionnel, CORBA/IDL est moins riche que Q-GDMO-GRM car il n'offre pas de description des relations entre objets et utilise un langage de types plus simple que ASN.1. L'extension de CORBA/IDL revient à intégrer le schéma *invariants, pré/post-conditions* et de définir un langage d'assertions intégrant des données du type CORBA/IDL. La figure 8.1 montre un exemple d'extension de la description d'interfaces de CORBA/IDL au travers de la spécification d'une Alarme de l'application FTMN-Alarmes. L'extension possible est définie entre guillemets.

La figure 8.1 propose une extension de CORBA/IDL à la formalisation des comportements et le déterminisme explicite préconisés dans cette thèse. Une perspective de notre travail est de montrer que l'on peut construire un simulateur de spécifications CORBA/IDL permettant de valider les interfaces CORBA et utilisant des compilateurs de CORBA/IDL existants.

L'intégration du prédicat := dans les spécifications permet de simuler une partie de la dynamique des applications. Cette partie est exécutable (traduction du := en ILOG TALK). Une perspective de notre travail est de la raffiner automatiquement vers des langages de programmation. Nous rejoignons ainsi le raffinement en B qui permet de définir, à partir d'un modèle abstrait d'une application, un modèle exécutable.

Pour enrichir la partie exécutable des spécifications à l'aide du prédicat :=, une autre perspective est de réduire les contraintes d'intégration du := dans les assertions. Dans le chapitre 6, nous interdisons la référence de variables décorées en opérande droit du prédicat :=. Cette contrainte permet de ne pas rajouter un calcul de dépendance des variables décorées. Une étude intéressante porte sur l'intégration d'un algorithme d'ordonnancement dans la sémantique opérationnelle définie en section 6.2.

```
interface Alarme {

attribute long Numero ;
attribute Etat_Terminaison EtatTerminaison ;
attribute Date DateAcquittement ;
attribute etatAcquittement EtatAcquittement;
...;

void creer_alarme(in long n, in string cP, in string pS, ...)
"post-condition: Numero := n and ... ";

void acquitter()
"pre-condition: EtatTerminaison = 1;
 post-condition:  EtatAcquittement' := 0 ";

void terminer( )
"post-condition:  EtatTerminaison' := 0 ";

...;
}
```

FIG. 8.1 – *Un exemple d'extension possible de CORBA/IDL au travers de l'interface Alarme*

Glossaire

– **AGL** Atelier de Génie Logiciel

– **ASN.1** Abstract Syntax Notation One

– **CCITT** Comité Consultatif International Télégraphique et Téléphonique

– **CMIP** Common Management Information Protocol

– **CMIS** Common Management Information Service

– **CORBA** Common Object Request Broker Architecture

– **DCE** Distributed Computer Environment

– **DPE** Distributed Processing Environment

– **FDT** Formal Description Technique

– **GDMO** Guidelines for the Definition of Managed Objetcs

– **GIOP** General Inter-ORB Protocol

– **GRM** General Relationship Model

– **IAB** Internet Architecture Board

– **IDL** Interface Definition Language

– **IETF** Internet Engeneering Task Force

– **IIOP** Internet Inter-ORB Protocol

– **IP** Internet Protocol

– **ISO** International Standardization Organization

– **ITU** International Telecommunication Union

– **LOTOS** Language Of Temporal Ordering Specifications

- **Q-GDMO-GRM** Quasi-GDMO-GRM
- **QG^{2}S** Q-GDMO-GRM Simulator
- **MIB** Management Information Base
- **ODMA** Open Distributed Management Architecture
- **ODP** Open Distributed Processing
- **OMA** Object Management Architecture
- **OMG** Open Management Group
- **OMT** Object Modeling Technique
- **OOD** Object Oriented Design
- **OOSE** Object Oriented Software Engineering
- **ORB** Object Request Broker
- **OSF** Open Software Foundation
- **OSI** Open Systems Interconnection
- **RGT** Réseaux de Gestion des Télécommunications
- **RM-ODP** Reference Manual-Open Distributed Processing
- **SDL** Standard Definition Language
- **SMF** Systems Management Functions
- **SNMP** Simple Network Management Protocol
- **TCP** Transmission Control Protocol
- **TINA-C** Telecommunications Information Networking Architecture Consortium
- **UDP** User Datagram Protocol
- **UML** Unified Modeling Language
- **VDM-SL** Vienna Development Method-Standard Language

Bibliographie

[ABL96] J.R. Ed. Abrial, E. Ed. Borger, and H. Ed. Langmaack. *Formal methods for industrial applications : specifying and programming the steam boiler control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Abr96] J-R Abrial. *The B-Book*. Cambridge, University Press, 1996.

[Ame87] P. America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP'87: European Conference on Object-Oriented Programming, Paris, France, June 15-17*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer, 1987.

[Ame91] P. America. Designing an object-oriented programming language with behavioural subtyping. In J.W. Bakker (de), W.P. Roever (de), and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, Rex School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[ASN88] International Telecommunication Union. *Open Systems Interconnection - General Specification of Abstract Syntax Notation One (ASN.1), Recommendation X.208*, 1988.

[ASN94] International Telecommunication Union. *Data NetWorks and Open system Communications, OSI Networking and System Aspects - Abstract Syntax Notation One (ASN.1), Information Technology - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation, Recommendation X.680*, 1994.

[BDE87] D. Bert, P. Drabik, and R. Echahed. Lpg: A generic, logic and functional programming language. In Guy Vidal-Naquet (Martin Wirsing) Franz-Josef Brandenburg, editor, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1987.

[Ber93] P.A. Bernstein. Middleware: An architecture for distributed system services. In *Technical Report No 93/6 of Digital Equipment Corporation Cambridge Research Lab*, march 1993.

[BGV97] M. BOUZEGHOUB, G. GARDARIN, and P. VALDURIEZ. *Les OBJETS, Edition revue et augmentée*. Eyrolles, 1997.

[Bil87] W.E. Biles. Introduction to simulation. In *Proceedings of 1987 Winter Simulation Conference*, 1987.

[BOO91] G. BOOCH. *Object Oriented Design with Application.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Boo92] G. Booch. *Conception orientée objets et applications.* Addison-Wesley France, 1992.

[CD97] A. Cougoulic and L.O. Donzelle. Spécification de comportements d'objets répartis et simulation. In *Acts of NOuvelles TEchnologies de la REpartition, pages 19-33, November 4-6, 1997, Pau, France*, 1997.

[CJ97] A. Cougoulic and P. Jacquet. *Analyse de la Sémantique Statique de Q-GDMO-GRM,* CNET Scientific Report, november 1997.

[CNE96] CNET. *OPERA, Modèle d'analyse*, 1996.

[Col89] W. Collings. Osi management service elements, protocols and application layer structure. In B. Meandzija and J. Wescost, editors, *Proc. of the IFIP TC6/WG6.6 1st. Int. Symp. on Integrated Network Management*, pages 119–131. Elsiever Science Publixhers B.V.(North-Holland), 1989.

[COR93] Object Management Group. *The Common Object Request Broker : Architecture and Specification, Rev. 1.2.*, 1993.

[Cou97] A. Cougoulic. A extension of logical formalisms to simulate system behaviors. In *Acts of the workshop Foundation of Composant-Based of Systems in ESEC'97, pages 71-80, vol. I, september 26, 1997, Zurich, Suisse*, 1997.

[CR91] W. Clinger and J. Rees. Revised 4, report on the algorithmic language scheme. In *ACM Lisp Pointers, 4(3)*, 1991.

[Des96] P. Desfray. *Modélisation par Objets : la fin de la programmation.* Masson, 1996.

[DKRS94] R. Duke, P. King, G. Rose, and G. Smith. Object-z : a specification language advocated for the description of standards. Technical report, Technical Report, SVRC, University of Queensland, 45, 1994.

[ELA93] R. Elmström, P.B. Lassen, and M. Andersen. An executable subset of vdm-sl in an sa/rt framework. In *Real-Time Systems*, volume 5, pages 197–211. Kluwer Academic Publishers, 1993.

[ELL92] R. Elmström, P.B. Lassen, and P.G. Larsen. Making specifications executable-using iptes-iv. In *Proceedings of Euromicro'92*, pages 521–528, 1992.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification - 1 : equations and initial semantics.* EATCS monographs on theoretical computer science. Springer-Verlag, 1985.

[Esc91] W. Eschebach. *Interpretative Ausfuehrung kommunizierender Regelsysteme.* PhD thesis, University of Kaiserslautern, Germany, 1991.

[Fes94] O. Festor. *Formalisation du comportement des objets gérés dans le cadre du modèle OSI.* PhD thesis, Université Henri Poincaré - Nancy I, 1994.

[FJL97] J. Ed. Fitzgerald, C.B. Ed. Jones, and P. Ed. Lucas. *FME'97: Industrial applications and strengthened foundations of formal methods : proceedings*, volume 1313 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[Fra97a] France Télécom/Branche Dévelopement/CNET/DES/ERA. *Document Projet, Spécifications fonctionnelles*, janvier 1997.

[Fra97b] France Télécom/Branche Dévelopement/CNET/DES/ERA. *FTMN-alarmes, Modèle d'analyse (structurel, opératoire, dynamique) DOSSIER DE SPECIFICATION*, mars 1997.

[Fuc92] E.N. Fuchs. Specifications are (preferably) executable. *Software Engineering*, pages 323–334, September 1992.

[Gal86] J.H. Gallier. *Logic for computer science : foundations of automatic theorem proving.* Harper and Row computer science and technology series, 1986.

[Gar89] H. Garavel. *Compilation et Vérification de Programmes LOTOS.* PhD thesis, Université Joseph Fourier - Grenoble I, 1989.

[GDM92] International Telecommunication Union. *Structure of Management Information: Guidelines for the Definition of Managed Objects, X.722*, 1992.

[GDM97] International Telecommunication Union. *Structure of Management Information: Guidelines for the Definition of Managed Objects, Draft Amendment 4, X.722*, 1997.

[GG91] P. Gochet and P. Gribomont. *Logique, méthodes pour l'informatique fondamentale*, volume 1. HERMES, 1991.

[GGM97] J.M. Geib, C. Gransart, and P. Merle. *Corba, Des concepts à la pratique.* InterEditions, 1997.

[GHG+93] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[GMSB96] M.C. Gaudel, B. Marre, F. Schlienger, and G. Bernot. *Précis de génie logiciel*, chapter 3. MASSON, 1996.

[GO96] G. Gardarin and Gardarin O. *Le Client-Serveur.* Eyrolles, 1996.

[HB95] M.G. Hinchey and J.P. Bowen. Applications of formals methods faq. In *Applications of Formals Methods*. Prentice Hall International series in computer science, 1995.

[HJ89a] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering*, 4(6):330–338, November 1989.

[HJ89b] C.A.R. Hoare and C.B. Jones. *Essays in Computer Science*. Prentice Hall International, 1989.

[ILO95] ILOG S.A. *ILOG VIEWS, Reference Manual, Version 2.2*, 1995.

[ILO96a] ILOG S.A. *ILOG POWER CLASSES, Object Services Manual, Version 1.4*, 1996.

[ILO96b] ILOG S.A. *ILOG TALK, Overview, Version 3.2*, 1996.

[Int89] International Standardization Organisation. *ISO/IEC 7498-3: Information processing systems - Open Systems Interconnection - Basic Reference Model-Part 3: Naming and adressing*, 1989.

[Int95] International Telecommunication Union. *Generic functional architecture of transport networks, Recommendation G.805*, 1995.

[ISO89] International Standardization Organisation. *ISO/IEC 9072-1: Information processing systems – Text communication – Remote Operations – Part 1: Model, notation and service definition*, 1989.

[ISO94] ISO/IEC JTC 1/SC 21/WG 4 N8088. *Working Draft on the Use of FDTs for the Specification of the Behaviour of Managed Objects*, 1994.

[ISO96] International Standardization Organization. *ISO/IEC 13817-1:1996, Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, 1996.

[ITU91a] International Telecommunication Union. *Data Communication Networks: Open Systems Interconnection (OSI); Management, Common Management Information Protocol Specification for CCITT Applications, Recommendation X.711*, 1991.

[ITU91b] International Telecommunication Union. *Data Communication Networks: Open Systems Interconnection (OSI); Management, Common Management Information Services Definition for CCITT Applications, Recommendation X.710*, 1991.

[ITU92a] International Telecommunication Union. *Data Communication Networks, Information Technology - Open Systems Interconnection - Structure of Management Information: Management Information Model, Recommendation X.720*, 1992.

[ITU92b] International Telecommunication Union. *Data Communication Networks, Management Framework for Open Systems Interconnection (OSI) for CCITT Applications, Recommendation X.700*, 1992.

[ITU92c] International Telecommunication Union. *Generic Network Information Model, Recommendation M.3100*, 1992.

[ITU92d] International Telecommunication Union. *Information Technology - Open Systems Interconnection - Systems Management: Object Management Function, Recommendation X.730*, 1992.

[ITU92e] International Telecommunication Union. *Information Technology - Open Systems Interconnection - Systems Management: Alarm Reporting Funtion, Recommendation X.733*, 1992.

[ITU92f] International Telecommunication Union. *Information Technology - Open Systems Interconnection - Systems Management: Event Management Function, Recommendation X.734*, 1992.

[ITU95a] International Telecommunication Union. *Information technology - Open Systems Interconnection - Service definition for the association control service element Common text with ISO/IEC, Recommendation X.217*, 1995.

[ITU95b] International Telecommunication Union. *Information Technology - Open Systems Interconnection - Structures of Management Information: General Relationship Model, Recommendation X.725*, 1995.

[ITU97] International Telecommunication Union. *Information technology - Open Systems Interconnection - Systems management overview, Recommendation X.701*, 1997.

[JCJ92] I. Jacobson, M. Christerson, and G. Jonson, P. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[Kel95] J. Keller. An extension of gdmo for formalizing managed objects behaviour. In *8th IFIP TC6 International Conference on Formal Description Techniques (FORTE'95), October 17-20, 1995, Montreal, Canada.*, 1995.

[KRB91] G. Kiczales, J. Rivières(des), and D.G. Bobrow. *The Art of the Metaobject Protocol.* The MIT press, 1991.

[KTW93] N. Kincl, D. Thompson, and R. Webber. *Adapting DME to Manage DCE-based Services, RFC 22.0.* The Open Group, 1993.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[LFB96] P.G. Larsen, J. Fitzgerald, and T. Brookes. Applying formal specification in industry. *IEEE Software*, pages 48–56, May 1996.

[LL91] P.G. Larsen and P.B. Lassen. An executable subset of meta-iv with loose specification. In *VDM'91 Symposium, Springer-Verlag*, 1991.

[Loe87] J. Loeckx. Algorithmic specifications : A constructive specification method for abstract data types. In *ACM Transactions on Programming Languages and Systems, Vol.9, No. 4, October 1987, Pages 646-685*, 1987.

[LOT89] International Standardization Organization. *ISO 8807:1989 Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[LRd93] R. Lassaigne and M. Rougemont (de). *Logique et fondements de l'informatique : logique du 1er ordre, calculabilité et lambda-calcul*, chapter 8. Traité des nouvelles technologies. Série informatique. Hermes, 1993.

[Maz97] S. Mazziotta. *Spécification et Génération de Tests du Comportement Dynamique des Systèmes à Objets Répartis.* PhD thesis, Université de Nice-Sophia Antipolis, UFR Faculté des Sciences, Institut Eurécom, 1997.

[Mey92] B. Meyer. *EIFFEL, The Language.* Object-Oriented. Prentice Hall, 1992.

[MNM87] L.F. Mackert and I.B. Neumeier-Mackert. Communicating rule systems. In H. Rudin and C.H. West, editors, *In Proceedings on 7th. International Symposium on Protocol Specification, Testing and Verification, North-Holland*, pages 77–88, 1987.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[MR90] K. MacCloghrie and M.T. Rose. *Management Information Base for Network Management of TCP/IP-based Internets, RFC 1156.* Internet Engeneering Task Force, 1990.

[MR91] K. MacCloghrie and M.T. Rose. *Management Information Base for Network Management of TCP/IP-based Internets: MIB-II, RFC 1213.* Internet Engeneering Task Force, 1991.

[MUL97] P.A. MULLER. *Modelisation objet avec UML.* Eyrolles, 1997.

[Oa97] C. Oussalah and alii. *Ingenierie objet : concepts et techniques.* InterEditions, 1997.

[ODP95] International Telecommunication Union. *Reference Model of Open Distributed Processing*, 1995.

[Par96] T.J. Parr. *Language Translation Using PCCTS and C++, a Reference Guide.* Automata Publishing Company, 1996.

[Rat99] Rational. *http://www.rational.com/products/rose/index.jtmpl*, 1999.

[RBE+91] J. RUMBAUGH, M. BLAHA, F. EDDY, W. PREMERLANI, and W. LORENSEN. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[RM90] M.T. Rose and K. MacCloghrie. *Structure and Identification of Management Information for TCP/IP-based Internets, RFC 1155.* Internet Engeneering Task Force, 1990.

[RM91] M.T. Rose and K. MacCloghrie. *Concise MIB Definitions, RFC 1212.* Internet Engeneering Task Force, 1991.

[SCT96] L. Sterling, P. Ciancarini, and T. Turnidge. On the animation of "not executable" specifications by prolog. *International Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87, 1996.

[SDL93] International Telecommunication Union. *CCITT specification and description language (SDL), Recommandation Z.100*, 1993.

[Smi96] R.D. Smith. Simulation: A gentle introduction. *ACM. SIGSIM (A QUATERLY PUBLICATION OF SIGSIM THE SPECIAL INTEREST GROUP ON SIMULATION DIGEST), SIMULATION DIGEST*, 26(2):14–24, 1996.

[SOF97] SOFTEAM Object Engineering. *Guide Utilisateur Objecteering 4 : Introduction*, 1997.

[Som92] I. Sommerville. *Le Génie Logiciel.* Addison-Wesley, traduit de l'anglais par J.M. André, 1992.

[Spi92] J.M. Spivey. *The Z Notation.* Prentice Hall, 1992.

[Ste90] G. Steele. *Common Lisp: The Language, Second Edition.* Digital press, 1990.

[SZ97] N. Simoni and S. Znaty. *Gestion de réseau et de service, similitude des concepts, spécificité des solutions.* InterEditions, 1997.

[Tel95] Telecommunications Information Networking Architecture Consortium. *Overall Concepts and Principles of TINA Version: 1.0*, 1995.

[TIN95] Telecommunications Information Networking Architecture Consortium. *Information Modelling Concepts, Version: 2.0*, 1995.

[Tou89] L. Toutain. *SAMSON: un simulateur pour systèmes répartis et temps-réel.* PhD thesis, Université de Le Havre, 1989.

[VA98] L. Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels.* PhD thesis, Université de Rennes I, 1998.

[WB91] A.C. Winstanley and D.W. Bustard. Expose: an animation tool for process-oriented specifications. *Software Engineering Journal*, pages 463–475, november 1991.

[WLB97] T. Wahls, G.T. Leavens, and A.L. Baker. Executing formal specifications with constraint satisfaction. Technical report, Department of Computer Science Iowa State University, TR number 97-12, august 1997.

Annexe A

Grammaire du langage Q-GDMO-GRM

Dans cette annexe, nous présentons la grammaire du langage Q-GDMO-GRM. Q-GDMO-GRM est un langage de spécifications fondé sur les langages GDMO [GDM92] et GRM [ITU95b]. Cependant, Q-GDMO-GRM ne reprend pas totalement les concepts de ses deux pères. Nous montrons ces différences de concepts dans la section A.2.

La grammaire que nous proposons dans cette annexe est différente de celle présentée dans le document [TIN95], chapitre 6. Dans la section A.1, nous discutons de cette différence. Dans la section A.3, nous présentons la grammaire du formulaire OBJECT TYPE. Dans la section A.4, nous présentons la grammaire des formulaires de relations.

A.1 Pourquoi une nouvelle grammaire ?

Dans cette section, nous donnons nos arguments sur notre choix de grammaire pour le langage Q-GDMO-GRM. Ces arguments portent sur le non respect de la part des auteurs du langage Q-GDMO-GRM de la grammaire abstraite décrivant le langage de spécification type attendu pour la vue information dans TINA. Cette grammaire abstraite est fournie dans le document [TIN95] au chapitre 5 (juste avant le chapitre décrivant Q-GDMO-GRM).

La raison invoquée justifiant la différence entre le langage Q-GDMO-GRM proposé et la grammaire abstraite attendue est la simplification du langage. Cependant, une simplification d'un langage ne doit pas signifier la perte d'expressivité de celui-ci, ce qui est le cas dans celle proposée dans [TIN95].

Le manque d'expressivité porte sur la définition des formulaires définissant les rela-

tions. Dans la grammaire abstraite, ceux-ci sont au nombre de trois :

- formulaire de relation ;
- formulaire de relation générique ;
- formulaire liant des rôles[1] à une relation générique.

Dans la grammaire décrivant Q-GDMO-GRM, seuls les formulaires définissant les relations génériques et leurs liens avec les rôles sont retranscrits. Ainsi, pour définir une relation entre un ensemble d'objets de types précis, il faut déclarer au minimum deux formulaires, au lieu d'un seul dans le cadre de l'utilisation de la grammaire abstraite. Ceci ne remet en cause l'expressivité du langage, au contraire de cette constation :

Le langage Q-GDMO-GRM ne permet pas d'exprimer totalement la notion de généricité des relations, et plus particulièrement de leurs comportements.

Une relation générique permet de définir des comportements et attributs communs à un certain nombre de relations. Les formulaires liant des rôles à cette relation générique permettent de définir (ou de retrouver) ses filles. Ainsi, dans la relation générique, on définit les comportements communs des relations et, dans les formulaires de liens, on définit une spécialisation de ceux-ci. Dans Q-GDMO-GRM, cette spécialisation n'est plus possible, d'où une perte d'expressivité.

Nous avons donc décider de proposer une autre grammaire des formulaires de relations. Elle découle de la grammaire abstraite proposée au chapitre 5 du document [TIN95].

A.2 Comparaison entre Q-GDMO-GRM et GDMO/GRM

A.2.1 Comparaison entre Q-GDMO-GRM et GDMO

GDMO est le langage de spécification des objets gérés utilisé dans le modèle OSI. GDMO définit sous forme de formulaires les différentes entités composant un objet géré. Une spécification GDMO d'un objet géré (formulaire nommé MANAGED OBJECT CLASS) est composée de références à des *packages* qui peuvent être obligatoires ou conditionnels. Un package est un formulaire composé des formulaires :

- *behaviours*, définissant les comportements de l'objet ;

1. un rôle représente un type d'objet lié par la relation.

- *attributs*, définissant les attributs de l'objet ;
- *attribut groups*, définissant des ensembles d'attributs et donc des comportements communs de l'objet ;
- *actions*, définissant les actions que peut subir l'objet ;
- *notifications*, définissant les notifications que peut transmettre l'objet ;
- *parameters*, définissant les paramètres d'erreurs, d'actions et de notifications de l'objet.

Nombre de ceux-ci font référence à d'autres formulaires. Cette *débauche* de formulaires montre la granularité forte de la réutilisation des spécifications en GDMO (n'oublions pas de rajouter la notion d'héritage entre MANAGED OBJECT CLASS). En effet, un formulaire d'un attribut peut être référencé par plusieurs packages qui eux peuvent être référencés dans plusieurs MANAGED OBJECT CLASS.

Un plus de cette granularité est la possibilité de définir des comportements globaux. Par exemple, dans le cas d'un attribut de type entier, on peut préciser que quelque soit l'objet le contenant, sa valeur ne peut dépasser 100.

Le langage Q-GDMO-GRM n'est pas aussi granulaire que GDMO. En effet, la réutilisation de spécification ne s'effectue que par le biais de l'héritage des formulaires OBJECT TYPE (équivalent au MANAGED OBJECT CLASS). La notion de package est préservé avec les restrictions suivantes :

- déclaration d'un et d'un seul package dans un OBJECT TYPE [2] ;
- pas de notion de packages conditionnels.

Avec de telles restrictions, nous nous demandons s'il est encore viable de préserver la notion de packages dans Q-GDMO-GRM. Les autres formulaires de GDMO ne sont pas repris. Q-GDMO-GRM définit localement les attributs, les actions, les notifications et les paramètres liés à la classe d'objet.

A.2.2 Comparaison entre Q-GDMO-GRM et GRM

GRM est un langage de spécifications des relations entre objets gérés. Comme GDMO, GRM définit des formulaires pour représenter les relations qui sont :

- *relationship class*, décrivant la relation, c'est-à-dire les types de objets, les méthodes pour lier un objet à la relation, et celles de création et de destruction de la relation ;

2. cette déclaration est locale et de plus le package ne peut pas être réutilisé

– *relationship mapping*, décrivant la traduction des méthodes d'une relation en requêtes CMIS.

Q-GDMO-GRM offre plus d'expressivité que GRM. En effet, en Q-GDMO-GRM, on peut définir des relations génériques (cf. section A.1) au contraire de GRM. De plus, il est possible de définir des actions, notifications, attributs portant sur l'objet relation. En GRM, on ne définit que les méthodes de gestion de la relation.

Une autre différence entre Q-GDMO-GRM et GRM est la définition de la cardinalité des rôles. En effet, en GRM, la cardinalité d'un rôle définit le nombre d'objets référencés par celui-ci d'une même instance de relation. En Q-GDMO-GRM, à chaque rôle d'une instance de relation est associé un et un seul objet. De plus, la cardinalité d'un rôle est un ensemble précisant les nombres possibles de liens que peuvent avoir les autres objets de la relation. La figure A.1 présente une relation entre deux types d'objets A et B, dont les cardinalités sont CardA = {1,2} et CardB = {1}.

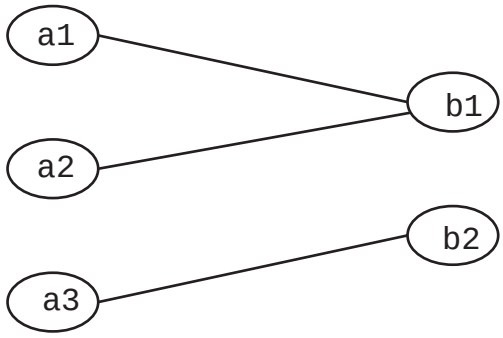


FIG. A.1 – *Exemple d'une relation entre des objets de types A et B*

Cette définition de la cardinalité nous paraît bien singulière. Autant, dans une relation binaire, elle ne pose aucun problème, autant, elle est déroutante pour des relations n-aire. Prenons le cas simple d'une relation unaire, la définition de la cardinalité est alors inutile.

Nous nous proposons de changer la signification de la cardinalité des rôles. On inverse l'application de la contrainte, c'est-à-dire que la cardinalité d'un rôle porte sur lui. Dans l'exemple précédent CardA devient CardB et vice versa.

A.3 Grammaire du formulaire OBJECT TYPE

Dans la grammaire suivante (et celle définie en A.4), les expressions regulières définissant les divers labels contenus dans un formulaire OBJECT TYPE doivent respecter

ces conditions :

- *LblType* représente un label de formulaire, il est constitué de lettres, de chiffres et de traits d'union ne se suivant pas. Il débuter par une majuscule.
- *LblString* représente un label d'un attribut, d'une opération, etc. Sa constitution est la même que celle de *LblType*. Par contre, il doit débuter par une minuscule.
- *ASN1TYPE* représente un label d'un type ASN1. Sa représentation est équivalente à celle de *LblType*.

Dans la grammaire suivante, quatre règles sont définies dans les autres annexes. Ces règles sont :

- *behaviourGrammar* : règle d'entrée pour l'analyse d'un comportement (cf. annexe B) ;
- *valueGrammar* : règle d'entrée pour l'analyse d'une valeur, celle-ci est de type ASN1 (cf. annexe C) ;
- *objectIdentifierValue* : règle permettant de lire un identifiant de formulaire (cf. annexe C) ;
- *setValue* : règle permettant de lire un ensemble de valeur (cf. annexe C).

```
objectType      ::=  LblType OBJECT TYPE
                         {derivedFrom}
                         tempBody ;
                         registration ;

derivedFrom     ::=  DERIVED FROM LblType (, LblType)* ;

tempBody        ::=  CHARACTERIZED BY LblString PACKAGE
                         BEHAVIOUR LblString
                           BEHAVIOUR DEFINED AS
                             ! behaviourGrammar ! ;
                         ATTRIBUTES
                             {attributes} ;
                         ACTIONS
                             {aCtions} ;
                         NOTIFICATIONS
                             {notifications} ;

attributes      ::=  attDecl ( , attDecl)*

attDecl         ::=  LblString {permitVal} {initVal}
                           {getReplace} {addRemove}
```

permitVal ::= **PERMITTED VALUES:** LblType

initVal ::= **INITIAL VALUE:** value

getReplace ::= **GET** | **REPLACE** | **GETREPLACE**

addRemove ::= **ADD** | **REMOVE** | **ADDREMOVE**

aCtions ::= opSign (, opSign)*

notifications ::= opSign (, opSign)*

opSign ::= LblString ({varTypes}) { : (varTypes) }

varTypes ::= varType (, varType)*

varType ::= LblString : LblType

registration ::= **REGISTERED AS** objectIdentifierValue

A.4 Grammaire des formulaires de relation

A.4.1 Grammaire du formulaire RELATIONSHIP TYPE

Soit la grammaire suivante :

relationType ::= LblType **RELATIONSHIP TYPE**
{derivedFrom}
tempBody
role$^+$;
registration ;

role ::= **ROLE** LblString
RELATED TYPE LblType {cardinality} ;

cardinality ::= **ROLE CARDINALITY CONSTRAINT** valueCard

valueCard ::= NUM { .. NUM } | **{** NUM$^+$ **}**

A.4.2 Grammaire du formulaire RELATIONSHIP GENERIC TYPE

Soit la grammaire suivante :

genRelationType:= LblType **GENERIC RELATIONSHIP TYPE**
{derivedFrom}
tempBody
genRole^{+} ;
registration ;

genRole ::= **ROLE** LblString {cardinality} ;

A.4.3 Grammaire du formulaire ROLE BINDING

Soit la grammaire suivante :

roleBinding ::= LblType **ROLE BINDING**
RELATIONSHIP TYPE LblType ;
tempBody
role^{+} ;
registration ;

Annexe B

Grammaire des comportements Q-GDMO-GRM

Dans cette annexe, nous associons à la grammaire structurelle des comportements Q-GDMO-GRM celle définissant les assertions. Elle est proche de l'alphabet du langage des prédicats du premier ordre et plus précisément du langage d'assertions défini dans les chapitres 5 et 6.

Il faut noter que les constantes de fonctions définies dans cette grammaire permettent d'effectuer des opérations sur des données ASN.1. Nous donnons en annexe C les types ASN.1 supportés par le simulateur QG^2S.

Certaines règles de grammaire et expressions régulières sont déjà définies dans l'annexe A.3.

B.1 Grammaire de la structuration des comportements de Q-GDMO-GRM

Soit la grammaire suivante :

behaviourGrammar	::=	{**COMMENTS:** String ; } { **INVARIANT:** predicate ; } {obRelConstraints} (roleConstraints)* (operations)*
predicate	::=	expressionGrammar

obRelConstraints ::= **create**-LblType (opSign prePost ;
{**delete**-LblType (opSign prePost ;
| **delete**-LblType (opSign prePost ;

prePost ::= {**PRECONDS**: assertion ; }
{**POSTCONDS**: assertion ; }

roleConstraints ::= **create**-LblString (opSignC prePost ;
{**delete**-LblString (opSignC prePost ;
| **delete**-LblString (opSignC prePost ;

operations ::= LblString (; attributeMethodes* | (operationSpec)
| attributeMethodes+

attributeMethodes ::= get | replace | add | rEmove

get ::= **get**-LblString ( ) : (LblString : LblType)
prePost ;

replace ::= **replace**-LblString (LblString : LblType) : ( )
prePost ;

add ::= **add**-LblString (LblString : LblType) : ( )
prePost ;

rEmove ::= **remove**-LblString (LblString : LblType) : ( )
prePost ;

operationSpec ::= opSignC
prePost
{**TRIGGERINGCONDS**: assertion ; } ;

opSign ::= {varTypes}) {: ({varTypes}) }

B.2 Grammaire des assertions

Soit la grammaire suivante :

assertion ::= ((assertion)
| predicate
| **not** assertion
| **forall** LblString **in** expression | assertion
| **exists** LblString **in** expression | assertion
| **if** assertion **then** assertion **else** assertion
) binaireOp assertion

binnaireOp	::=	**and** | **or**
predicate	::=	expression predicateOp expression
predicateOp	::=	= | <= | >= | < | > | :=
expression	::=	(expression) | exprVal {binaire exprVal}
binaire	::=	+ | − | / | *
exprVal	::=	constFonct | exprValObjet | value

où *value* est le point d'entrée de lagrammaire des valeurs ASN.1 (cf. annexe C.2).

constFonct	::=	LblType ({expression (, expression)*})
exprValObjet	::=	LblString {'} . LblString | (valObjet) {'} . LblString
valObjet	::=	LblString . LblString | (valObjet) . LblString

Annexe C

Types et valeurs ASN.1 intégrés par QG^{2}S

Dans cette annexe, nous précisons quels types et valeurs ASN.1 [ASN94] sont intégrés dans le simulateur QG^{2}S. Tout le typage ASN.1 n'est pas accepté par le simulateur. Par exemple, le sous-typage n'est pas traité (on peut le lire mais on ne le prend en compte).

L'ASN.1 pris en compte est l'ASN1 1990 en tenant compte des modifications apparaissant dans l'ASN.1 1994. Par exemple, l'accès à une valeur d'un *choice* est distingué d'une *sequence*.

Dans les termes définis dans les comportements des spécifications Q-GDMO-GRM, des valeurs ASN.1 sont utilisées. Un des problèmes majeur d'ASN.1 est l'obligation d'utiliser un label pour référencer une valeur. Celui-ci est défini dans un module ASN.1 et est associé à la valeur et à son type. Cela est obligatoire pour différencier un ensemble d'une séquence, car ces deux types ont une syntaxe équivalente. Nous trouvons cette pratique trop lourde et décidons de changer la grammaire ASN.1 en conséquence :

- les types *set* et *sequenceOf* sont remplaçés respectivement par *sequence* et *setOf* (*setOf* devient *set*) ;
- un type et une valeur *sequence* sont délimités par [et] au lieu de { et }.

C.1 Grammaire des types ASN.1

Soit la grammaire de la définition d'un module ASN.1 :

moduleDefinition ::= LblType { **{** (NUM | LblString { (NUM) }) $^+$ **}**}
DEFINITIONS { tagDefault} ::=
BEGIN
{**EXPORTS**{ symbolList} ;}
{**IMPORTS** (symbolsFromModule)* ;}
(typeAssignment | valueAssignment)*
END

tagDefault ::= (**EXPLICIT**|**IMPLICIT**|**AUTOMATIC**) **TAGS**

symbolList ::= symbol (, symbol)*

symbol ::= LblString | LblType

symbolsFromModule ::= symbolList **FROM**
LblType { **{** (NUM | LblString { (NUM) }) $^+$ **}**}

typeAssignment ::= LblType ::= type

valueAssignment ::= LblString type ::= value

Soit la grammaire de la définition des types ASN.1 :

type ::= bitStringType | octetStringType
| booleanType | characterStringType
| choiceType | embeddedPDVType
| enumeratedType | externalType
| integerType | nullType
| objectIdentifierType | realType
| sequenceType | setOfType
| taggedType | definedType

bitStringType ::= **BIT STRING** { **{** namedBitList bf } }

namedBitList ::= LblString ((NUM | definedValue))
(, LblString ((NUM | definedValue)))*

octetStringType ::= **OCTET STRING**

booleanType ::= **BOOLEAN**

characterStringType ::= **BMPString** | **GeneralString**
| **GraphicString** | **IA5String**
| **ISO646String** | **NumericString**
| **PrintableString** | **TeletexString**
| **T61String** | **UniversalString**
| **VideotexString** | **VisibleString**
| **CHARACTER STRING**

choiceType ::= **CHOICE {** namedType (, namedType)*bf }

embeddedPDVType::= **EMBEDDED PDV**

enumeratedType ::= **ENUMERATED**
{ LblString { (({ - } NUM | definedValue)) }
(, LblString{ (({ - } NUM | definedValue))})***}**

externalType ::= **EXTERNAL**

integerType ::= **INTEGER** { **{** namedNumber (, namedNumber)***}** }

namedNumber ::= LblString (({ - } NUM | definedValue))

nullType ::= **NULL**

objectIdentifierType::= **OBJECT IDENTIFIER**

realType ::= **REAL**

sequenceType ::= **SEQUENCE {**{ componentType (, componentType)*}**}**

componentType ::= namedType{ (**OPTIONAL** | **DEFAULT** value)}
| **COMPONENTS OF** type

setOfType ::= **SET OF** type

taggedType ::= [{ (**UNIVERSAL** | **APPLICATION** | **PRIVATE**)}
(NUM | definedValue)]
{ (**IMPLICIT** | **EXPLICIT**)} type

namedType ::= LblString type

definedType ::= LblType{. LblType}

Nous notons que le type sélection et le sous typage ne sont pas pris en compte dans cette grammaire.

C.2 Grammaire des valeurs ASN.1

Soit la grammaire suivante :

```
value                ::= bitOctetStringValue | booleanValue
                         | characterStringValue | choiceValue
                         | embeddedPDVValue | enumeratedValue
                         | externalValue | integerValue
                         | nullValue | objectIdentifierValue
                         | realValue | sequenceValue
                         | setValue

bitOctetStringValue::= BSTRING | HSTRING
                         | {{ LblString (, LblString)*}}

booleanValue         ::= TRUE | FALSE

characterStringValue:= CSTRING
                         | { ( (CSTRING | definedValue)
                                 (, (CSTRING | definedValue))*
                             | NUM , NUM{ , NUM , NUM} ) }
                         | [ identification (
                             syntaxes: [ abstract objectIdentifierValue ,
                                         transfert objectIdentifierValue ]
                             | syntax: objectIdentifierValue
                             | presentationcontextid: integerValue
                             | contextnegociation:
                               [ presentationcontextid integerValue ,
                                 transfertsyntax objectIdentifierValue ]
                             | transfertsyntax: objectIdentifierValue
                             | fixed: NULL ) ,
                           stringvalue (
                             notation: NULL
                             | encoded: bitOctetStringValue ) ]

choiceValue          ::= LblString: value

embeddedPDVValue::= [ identification (
                             syntaxes: [ abstract objectIdentifierValue ,
                                         transfert objectIdentifierValue ]
                             | syntax: objectIdentifierValue
                             | presentationcontextid: integerValue
                             | contextnegociation:
                               [ presentationcontextid integerValue ,
                                 transfertsyntax objectIdentifierValue ]
                             | transfertsyntax: objectIdentifierValue
                             | fixed: NULL ) ,
                           datavalue (
                             notation: NULL
                             | encoded: bitOctetStringValue ) ]
```

enumeratedValue ::= LblString

externalValue ::= [**identification** (
syntax: objectIdentifierValue
| **presentationcontextid:** integerValue
| **contextnegociation:**
[**presentationcontextid** integerValue ,
transfertsyntax objectIdentifierValue]) ,
datavaluedescriptor NULL,
datavalue (
notation: NULL
| **encoded:** bitOctetStringValue)]

integerValue ::= LblString | { -} NUM

nullValue ::= NULL

objectIdentifierValue ::= **{**{ definedValue}
(NUM | LblString{ (NUM)} | definedValue)$^{+}$**}**

realValue ::= **0** | sequenceValue
| **PLUS-INFINITY** | **MINUS-INFINITY**

sequenceValue ::= [{ LblString value (, LblString value)*}]

setValue ::= **{**{ value (, value)*}**}**

definedValue ::= LblString | LblType.LblString

Résumé : La complexité des réseaux de télécommunications et des applications qui les gérent a conduit à la définition de plusieurs modèles de gestion tels OSI, ODP, etc. La simulation des spécifications des applications permet de valider et de mettre au point leurs fonctions et leurs architectures avant la phase d'implantation. Dans cette thèse, nous nous sommes intéressés plus particulièrement à la validation de la dynamique des applications (egalement appelée *comportement*), et donc à la modélisation de cette dernière. La technique de formalisation des comportements que nous avons étudiée permet d'étendre les langages de spécifications existants en intégrant le schéma classique d'invariants et de couples de pré/post-conditions définis par un langage d'assertions simple.
La simulation des spécifications d'une application doit permettre de définir ses états successifs. Cependant, l'ambiguïté des assertions peut engendrer une explosion combinatoire dans le test de toutes les successions d'états possibles. Le but de cette thèse n'est pas de traiter les ambiguïtés mais de mettre en évidence les parties non ambiguës des assertions ceci afin de faciliter la simulation. Pour cela, nous avons introduit dans les langages d'assertions un prédicat intégrant du *déterminisme explicite*.
Nous avons mis en oeuvre notre solution dans le langage Q-GDMO-GRM, langage de spécifications du point de vue information du modèle TINA (application du modèle ODP pour les télécommunications). Nous avons développé le simulateur QG^2S prenant en compte la sémantique opérationnelle relative au déterminisme explicite présenté dans cette thèse.

Title : Simulation of management applications specifications of telecommunication networks

Abstract : The complexity of telecommunication networks and their management applications has introduced some management models as OSI, ODP, etc. The simulation of application specifications allows to validate and to finalize their functions and theirs achitectures before implementation. In this thesis, our interest gets on the dynamic application validation (so spelling *behaviors*), and so to modelize its. The behaviors formal description which we have studied allows to stretch existents specifications languages out to the classical *invariants and pre/post-conditions* schema, where a simple assertion language defines conditions.
The simulation of an application must define its successifs states. Morever, the ambiguous assertions can cause a combinatory explosion when test application states succession. The thesis goal is not to resolve ambiguities but to display the unambiguous parts of assertions to make simulation easier. For that, we introduce an *explicite determinism* predicat in assertion languages. We implement our solution in Q-GDMO-GRM language, specification language of the information point of vue of TINA model (telecommunication application of ODP model). The QG^2S simulator implements the operational semantic related to explicite determinism presented in this thesis.

Discipline : Information, Communication et Système

Mots-clés : Simulation, modèles de gestion des réseaux, langages d'interfaces et de spécifications, comportements.

Laboratoire LSR (Logiciel, Systèmes et Réseaux) de l'IMAG (Institut de Mathèmatiques Appliquées de Grenoble) domicilié au Domaine Universitaire, 681 rue de la Passerelle 38042 Saint Martin d'Hères Cedex,
en collaboration avec le laboratoire FT/BD/DvSI/DES/ERA (Exploitation du Réseaux d'Accès) domicilié au CNET, Centre Norbert Segard, 28 chemin du Vieux Chêne, 38243 Meylan Cedex.