UNIVERSITE DE NICE-SOPHIA ANTIPOLIS
U.F.R. FACULTE DES SCIENCES

Institut Eurécom

THESE

présentée pour obtenir le titre de
Docteur en SCIENCES (Spécialité Informatique)

par

MAZZIOTTA SANDRO

Spécification et Génération de Tests du Comportement Dynamique des Systèmes à Objets Répartis

Soutenue le 24 octobre 1997 devant le jury composé de :

- Président : **Professeur J.P. RIGAULT**
- Rapporteurs : **Professeur A. SCHAFF**
  Professeur D. SERET
- Examinateurs : **Dipl Ing. Inf ETHZ R. EBERHARDT**
  Professeur J. LABETOULLE (directeur de thèse)
  Professeur G. PUJOLLE

à 10 Heures à L'ESSI.

A ma Famille,
A mes parents Pietro et Jacqueline,
A ma soeur Sonia,
A Fanny.

Remerciements

Je remercie les membres de mon jury :

- Monsieur Jean-Paul Rigault, Professeur à l'Université de Nice-Sophia Antipolis et directeur de l'ESSI, qui m'a fait l'honneur de présider le jury.
- Madame Dominique Seret, Professeur à l'Université de Paris V, et Monsieur André Schaff, Professeur à l'Université de Nancy, d'avoir bien voulu accepter la charge de rapporteur.
- Monsieur Guy Pujolle, Professeur à l'Université de Versailles, d'avoir bien voulu accepter de participer au jury.
- Monsieur Rolf Eberhardt, Diplomé en informatique de l'ETH Zurich, non seulement d'avoir accepté de participer au jury, mais surtout pour tous les conseils pertinents prodigués au long de ces trois dernières années qui m'ont permis de mener à bien ce travail.

Je remercie Monsieur Jacques Labetoulle, Professeur à l'Institut Eurécom, de m'avoir proposé cette thèse et accueilli au sein de son équipe. Il a su à la fois diriger et conseiller mon travail tout en me laissant une grande liberté dans mes investigations et choix de réalisation. Je tiens aussi à le remercier pour nos nombreuses discussions et ses précieux conseils lors de la rédaction de ce mémoire.

Je remercie Monsieur Dominique Sidou, Docteur en Informatique de l'EPF Lausanne, pour avoir suivi et participé à mon travail ainsi qu'à la rédaction d'articles. Son amitié, son enthousiasme et son intérêt pour ce travail m'ont encouragé tout au long de ces trois années. Nos nombreuses discussions et ses multiples relectures m'ont permis d'achever ce mémoire.

Je remercie Monsieur Claude Jard, chargé de recherches au CNRS, et Monsieur Thierry Jéron, chargé de recherches à l'INRIA, tous deux membres de l'équipe PAMPA de l'IRISA, pour leur soutien décisif lors de la deuxième partie de ce travail. Je remercie Monsieur Simon Znaty, Professeur à l'ENST Bretagne et Monsieur Olivier Festor, chargé de recherches à l'INRIA pour l'aide qu'ils m'ont apportée et les discussions que nous avons eues tout au long de ces trois années.

Je remercie la compagnie Swisscom pour avoir lancé ce projet et pour son soutien financier. Je remercie enfin le personnel du centre de recherche et de développement de Swisscom avec qui j'ai eu beaucoup de plaisir à interagir.

Table des matières

Liste des tableaux

Liste des figures

"Keep it simple: as simple as possible, but no simpler."
Albert Einstein.

Chapitre 1

Introduction Générale

1.1 De l'émergence de l'Orienté Objet aux systèmes à objets répartis

La demande de la part des utilisateurs pour que les concepteurs de logiciel fournissent des produits avec des critères de qualité et de sûreté est de plus en plus grande. Ceci est principalement dû au fait que les logiciels sont de plus en plus compliqués. Cette complexité est due aux difficultés à appréhender le domaine du problème dans sa totalité, à construire des logiciels flexibles et dans la difficulté à gérer leur processus de développement.
Ces dernières années, l'ingénierie des logiciels modélisés par des objets s'est vue accepter comme une des meilleures approches pour outrepasser les difficultés inhérentes à la construction des applications complexes. De nos jours, on utilise la conception Orienté Objet (OO) dans des secteurs industriels très variés comme : les bases de données, les systèmes opératifs (operating systems), les applications de "commerce", tarification et de provision de services, ...

Parallèlement à l'émergence de l'OO, il devient de plus en plus difficile de parler d'applications informatiques sans parler de répartition, de réseaux et de communications. Ceci est principalement dû à l'amélioration des performances de communication. Les débits de transfert d'informations sont passés de quelques centaines d'octets par seconde, il y a quelques années, à plusieurs gigaoctets par seconde aujourd'hui. L'amélioration des performances et le développement des fonctionnalités qui permettent un accès aux systèmes informatiques en améliorant les interfaces utilisateurs ont fait que les nouveaux systèmes informatiques sont basés sur la répartition et l'échange des informations. Ainsi le concept qui allie à la fois l'OO et la répartition de l'information est apparu, c'est celui des systèmes à objets répartis. Les systèmes de télécommunications qui nous intéressent, n'échappent pas à cette tendance.

1.2 La validation des systèmes à objets répartis

Une étude récente montre que la phase de validation prend au moins la moitié du temps de travail d'une équipe pour produire un logiciel de qualité acceptable. Suivant des principes de conceptions plus traditionnels (pour du code séquentiel avec des langages comme C, Cobol, Fortran), les

statistiques montrent que sur 100 lignes de code, un logiciel va avoir de 1 à 3 "bugs". Ce sont des paramètres de qualité qui ne sont pas acceptables dans le contexte des applications critiques (contrôle de réacteurs nucléaires, fusée, télécommunications, ...).
Un exemple particulièrement médiatique, qui nous montre que les cycles de validation ne conduisent pas encore à des systèmes sûrs, est la récente catastrophe du lanceur Arianne 501, le 4 juin 1996. Le rapport des experts [Lyo96] montre clairement que la catastrophe résulte d'au moins quatre fautes dans le développement du système de contrôle des trajectoires. Ces fautes liées à une mauvaise réutilisation des composants informatiques se sont produites lors de la spécification (spécification incorrecte et incomplète), de la conception (modèle inadapté de fautes), lors de l'implémentation (code insuffisamment robuste) et du test (test d'intégration incomplet).

Le terme validation est un terme générique. Il regroupe différentes activités qui sont :

- la simulation fonctionnelle (ou simulation comportementale) [1] qui est une technique où un modèle exécutable est développé à partir d'une spécification formelle. Ce modèle est ensuite observé. Pour pouvoir détecter des erreurs, le modèle ne doit pas être trop complexe et doit donc faire abstraction des détails non intéressants (du point de vue fonctionnel);

- le test qui est une technique où le système final est observé et son comportement est comparé avec la spécification formelle. Le test n'est pas exhaustif dans le sens où il ne peut être utilisé que pour détecter des erreurs et en aucun cas pour montrer la correction.

- la vérification qui est une technique permettant de prouver formellement qu'une spécification (formelle) satisfait bien certaines propriétés désirées, en utilisant certaines méthodes rigoureuses.

Ces définitions ont le mérite de faire apparaître un élément important pour la suite de cette discussion : l'activité de validation a nécessairement besoin d'une spécification formelle. Celle-ci doit être :

- une définition non ambiguë, claire et compréhensible (cad qu'il n'existe qu'une seule interprétation possible de la description);

- une définition qui doit permettre l'analyse du comportement du système, si possible sans transformation intermédiaire (de la part de la personne qui spécifie) ce qui est source potentielle d'incohérences.

Le problème de la validation consiste le plus souvent à produire cette spécification formelle adaptée d'une part à ce que l'on veut formaliser et d'autre part aux systèmes que l'on se propose d'étudier.

Dans le contexte des systèmes à objets répartis, la validation est très peu étudiée. Ceci est principalement dû selon nous à l'absence de spécifications formelles. L'ensemble des recherches conduites au cours de cette thèse, contribue de manière générale à combler ce manque de spécifications formelles et d'outils de validation dans le contexte des systèmes à objets répartis.

[1] à distinguer de la simulation événementielle

1.3 Motivation Initiale

Le manque crucial de formalismes adaptés aux systèmes d'objets répartis (et donc de spécifications formelles) se manifeste particulièrement dans le RGT [TMN92a] (Réseaux de gestion des Télécommunications). C'est à la suite d'une idée originale de Rolf Eberhardt (ingénieur à Swiss Telecom) que le projet TIMS[2] a commencé en Septembre 1994. C'est d'ailleurs dans le contexte de ce projet entre Swiss Telecom PTT et l'Institut Eurécom que cette thèse s'est développée. L'idée principale du projet était de contribuer à une meilleure mise en service des applications du RGT et ce tant sur le point de la rapidité que sur le point de la sûreté.

Une mise en service plus rapide signifie qu'il faut proposer des méthodes (ou des outils) pour améliorer le délai (actuellement extrêmement long) entre la spécification (informelle) des besoins de la part de l'utilisateur et le délivrement, par le fournisseur de service, du produit final à l'utilisateur. Une mise en service de produits plus sûrs signifie qu'il faut proposer une méthode qui délivre des produits finaux avec de bonnes garanties quant à la sûreté de fonctionnement.

La seule contrainte fixée quant à la résolution du problème, était de fournir une solution (méthode, outils, formalismes, ...) simple. Il faut préciser que simple signifie : simple à utiliser et simple à mettre en oeuvre car il est illusoire d'imaginer qu'une solution simple puisse résoudre des problèmes compliqués. Nous verrons au cours de ce mémoire comment cette contrainte de simplicité s'est illustrée dans notre travail.

1.3.1 Identification de la source des problèmes

Après une étude du cycle de développement des applications du RGT, nous nous sommes aperçus que ce cycle consistait simplement en l'implémentation des systèmes directement à partir de spécifications semi-formelles. Ces spécifications sont définies comme semi-formelles car autant du point de vue statique, le modèle d'informations est bien établi (on dispose d'une syntaxe et d'une sémantique statique) autant du point de vue dynamique, le modèle d'informations se repose sur une description du comportement dynamique des objets en langue naturelle. Ainsi dans ce contexte particulier, les problèmes de spécifications informelles qui conduisent à la lenteur de la mise en service sont de deux natures :

- les différences d'interprétations dues à l'ambiguïté inhérente à l'utilisation de la langue naturelle pour la description du comportement dynamique des objets, conduit à des problèmes d'interfonctionnement. Cette incapacité à interfonctionner correctement se manifeste le plus souvent par des implémentations qui n'interopèrent pas (au niveau service) bien qu'elles soient "conformes" à leurs spécifications.

- comme nous venons de le présenter, la spécification formelle est le point d'entrée de toute activité de validation. Le manque de spécification explique pourquoi on ne dispose d'aucun outil de validation pour s'assurer de la correction de la spécification. Il apparaît impossible

[2]TIMS signifie TMN-based Information Model Simulator (ce qui donne en français : simulateur de modèles d'informations du Réseaux de Gestion des Télécommunications (RGT))

alors de produire des implémentations correctes à partir de spécifications (informelles) potentiellement fausses. Il est souvent trop tard lorsque l'on s'aperçoit de ce type d'erreurs et les effets sont catastrophiques.

Ces problèmes s'avèrent coûteux et inacceptables dans le contexte compétitif de la déréglementation des télécommunications et l'objectif de cette thèse est de contribuer à améliorer cette situation.

1.3.2 Vers une généralisation de la problématique

Ces problèmes se révèlent d'autant plus intéressants qu'ils ne se limitent pas qu'aux systèmes du RGT (qui ne sont en fait qu'un cas particulier de systèmes à objets répartis). Ce travail peut s'étendre à tous les systèmes à objets répartis. On peut classer dans cette catégorie : les systèmes ODMA [ODM95], TINA [TIN94b, TIN94a], CORBA [OMG96] et les très récents systèmes qui se basent sur JAVA-RMI [Jav].
Nous verrons plus tard que ce problème de spécification commun à l'ensemble des systèmes à objets répartis a déjà été identifié par les comités de standardisation des télécommunications et qu'il a motivé la création d'une nouvelle série de standards : le modèle de référence ODP.

1.4 Les contributions

La première contribution de ce mémoire réside **en la proposition d'un formalisme : le formalisme BL (Behavior Language) adapté à la description du comportement dynamique des systèmes à objets répartis.** Fournir une description formelle est déjà un résultat en soi et constitue le premier objectif du projet TIMS. Mais nous avons un second objectif qui est de pouvoir simuler le comportement dynamique de ces systèmes à l'aide de spécifications écrites dans notre formalisme grâce à un simulateur : le simulateur TIMS. Nous veillerons donc à doter BL d'une sémantique opérationnelle afin de pouvoir le rendre exécutable.
En cela, on peut voir le simulateur TIMS comme un environnement de validation (pour les spécifications BL) identique (du point de vue méthodologique) aux environnements CADP (pour le formalisme LOTOS) ou SDT et ObjectGeode (pour le formalisme SDL). L'aspect innovateur de notre formalisme est en fait sa parfaite adéquation à la problématique posée par les systèmes à objets répartis.

Nous nous sommes ensuite orientés vers la génération automatique de test qui est le deuxième problème majeur des applications du RGT. Selon [ACM+96] "l'écriture des tests coûte très cher. Les SSII facturent un jour ingénieur pour quatre tests écrits, ce qui met à plus d'un million de francs le prix de revient d'une suite de tests moyenne." Nous avons pensé, de manière intuitive, que notre formalisme de description des comportements dynamiques et son environnement de simulation, constituaient un excellent point de départ pour la génération de tests du comportement dynamique des systèmes à objets répartis. La seconde contribution de ce mémoire réside donc **en la proposition d'une méthode pour la génération automatique de tests du comportement dynamique des systèmes à objets répartis**.
La simulation à l'aide du simulateur TIMS des spécifications BL nous permet d'obtenir un graphe d'accessibilité qui est un système de transitions étiqueté particulier. C'est à partir de ce graphe

d'accessibilité, que l'on obtient grâce à un générateur de tests (dans notre cas l'outil TGV), des tests écrits dans le formalisme TTCN qui est notation standardisée pour la spécification des tests.

La troisième et dernière contribution de ce travail a consisté **en l'application du formalisme BL et des outils de simulation et de génération de tests sur les interfaces Xcoop**[3]. Cet exemple nous permet d'analyser et de critiquer notre approche sur un cas d'étude réel standardisé appartenant au domaine du RGT. Cela nous permet d'évaluer l'impact de notre travail quant à l'objectif initial de cette thèse qui est d'obtenir une meilleure mise en service de ce type d'applications.

1.5 Plan

Conformément à ce qui vient d'être exposé dans cette dernière section, le mémoire se décompose en trois parties :

- **partie I :** Spécification du comportement dynamique des systèmes à objets répartis
  - le chapitre 2 présente dans un premier temps la problématique de la spécification du comportement dynamique des systèmes à objets répartis. Dans un second temps, il est rappelé quels sont nos objectifs et on fixe à cette occasion des hypothèses de travail. On présente enfin les formalismes existants et on explique pourquoi, conformément à nos hypothèses, ces formalismes n'ont pas été retenus et pourquoi il a fallu développer notre propre formalisme;
  - le chapitre 3 débute par une présentation de BL qui est notre formalisme pour la description du comportement dynamique des systèmes à objets répartis. Nous présentons ensuite la sémantique opérationnelle du formalisme BL et l'algorithme de traitement de cette sémantique qui constitue le noyau générique d'un environnement de simulation qui nous permet d'obtenir différents types de simulations.
  - Les chapitres 2 et 3 ont fait l'objet des publications suivantes : [SME95, MS96b, EMS97]
- **partie II :** Génération de tests du comportement dynamique des systèmes à objets répartis
  - le chapitre 4 présente dans un premier temps la problématique de la génération de cas de tests du comportement dynamique des systèmes à objets répartis. On rappelle, dans un second temps, quels sont les objectifs et on fixe à cette occasion nos hypothèses de travail. On relate enfin notre travail par rapport aux approches équivalentes et on explique pourquoi une de ces approches, la génération à partir du graphe d'accessibilité, est adaptée à nos besoins;
  - le chapitre 5 présente notre approche pour la génération automatique de cas de test. Elle se base sur l'intégration d'une part de l'environnement de simulation tel qu'il aura été présenté dans la seconde partie du chapitre 3 et d'autre part d'un outil de génération de tests : TGV développé par l'équipe de recherche PAMPA de l'IRISA.

[3] les interfaces Xcoop sont les interfaces inter-operateurs pour la gestion des réseaux de transport de télécommunications de type SDH

  - Les chapitres 4 et 5 ont fait l'objet des publications suivantes : [Maz95]

- **partie III :** Application à la gestion des interfaces Xcoop

  - le chapitre 6 présente dans un premier temps le RGT (Réseaux de Gestion des Télécommunications) et la gestion de réseaux OSI qui forment le cadre de travail des interfaces Xcoop. Il présente dans un second temps l'environnement de simulation TIMS dédié à la simulation de modèles d'informations du RGT. C'est grâce à cet environnement que nous allons pouvoir valider notre travail sur le cas d'étude que constitue les interfaces Xcoop;
  - le chapitre 7 montre comment on a appliqué nos résultats à la fois pour spécifier et simuler le comportement dynamique de ce type d'interfaces et pour générer des tests. Ceci nous permet d'analyser nos résultats et donc de critiquer notre approche et d'évaluer son impact pour une meilleure mise en service de ce type d'applications.
  - Les chapitres 6 et 7 ont fait l'objet des publications suivantes : [SME95, MS96a]

- la conclusion résume ce travail et met en évidence les différentes contributions. On propose ensuite des axes de recherche qui n'ont pas pu être exploités durant l'accomplissement de cette thèse et qui constitueraient un contexte pour des travaux futurs.

Partie I

Spécification du comportement dynamique des sytèmes à objets répartis

Chapitre 2

Le comportement dynamique des systèmes à objets répartis

2.1 Introduction

La spécification du comportement dynamique des systèmes à objets répartis est un problème difficile à résoudre dans sa globalité. Ce chapitre vise principalement à introduire la problématique et à fixer un cadre de travail dans lequel ce problème de spécification devient plus facile à appréhender. Le chapitre suivant va quant à lui exposer notre formalisme pour la spécification du comportement dynamique des systèmes à objets répartis.
Ce chapitre se décompose en trois parties :

- la première partie est une introduction qui vise à fixer le cadre de notre travail en introduisant les notions de base que sont les objets, leurs comportements dynamiques et enfin les systèmes d'objets répartis. Comme nous l'avons vu dans l'introduction, notre travail de manière générale vise à améliorer l'interfonctionnement des objets répartis. Cet interfonctionnement des systèmes à objets répartis est le centre d'intérêt du modèle de référence ODP dont bien sûr nous allons nous inspirer. La présentation de ce modèle constitue donc l'essentiel de cette introduction;

- la seconde partie rappelle dans un premier temps nos objectifs (un formalisme pour la description du comportement dynamique des systèmes à objets répartis et un simulateur de ces descriptions). Dans un second temps, un certain nombre d'hypothèses de travail sont fixées. L'examen de chacune de ces hypothèses nous permet de dégager l'ensemble des propriétés que devra posséder le formalisme que nous retenons pour la description du comportement dynamique des systèmes à objets répartis;

- enfin compte-tenu de ces hypothèses et propriétés, la troisième partie de ce chapitre examine les différentes alternatives quant à la résolution de nos objectifs. L'examen de ces différentes alternatives revient en fait à explorer les formalismes de descriptions formelles déjà existants et de discuter leur faculté à posséder les propriétés identifiées dans la partie précédente.

2.2 Les systèmes à objets répartis selon ODP

Le modèle de référence ODP (RM-ODP) [RM-a] repose sur des concepts précis issus des développements récents dans le domaine des systèmes répartis, en particulier du projet Advanced Networked Systems Architecture (ANSA). Cette série de standards résulte d'un travail commun entre l'ISO et l'ITU-T. Ce travail se situe actuellement comme le modèle de référence pour l'interfonctionnement des systèmes ouverts. Il vise notamment à ce que les divers composants des applications réparties interopèrent entre eux et ce malgré l'hétérogénéité des équipements, des systèmes opératifs, des réseaux, des langages, des modèles de bases de données ou des autorités de gestion.

2.2.1 La notion de système ouvert et la notion d'interfonctionnement

Avant de présenter le modèle de référence ODP, il est important de définir la notion de système ouvert et celle d'interfonctionnement.

2.2.1.1 Les systèmes ouverts

Les systèmes ouverts sont définis comme des systèmes réactifs et extensibles tant dans le domaine spatial que dans le domaine temporel :

- Les systèmes ouverts sont réactifs car ils n'accomplissent pas un service une fois pour toute mais ils peuvent accomplir des services potentiellement à l'infini et ce à la demande.

- Ils sont extensibles dans le domaine spatial quand leurs composants n'ont pas besoin d'une connectivité fixe entre eux. Ils interagissent par envoi de messages (appel de procédure local, invocations "multi-threadés", communication distante asynchrone, ...).

- Ils sont temporellement extensibles car ils n'ont pas une configuration fixe lors de leur cycle de vie. Cette configuration évolue par ajout (suppression) de nouveaux composants ou bien par ajout (suppression) de nouvelles fonctionnalités pour un composant donné. A noter que ces modifications peuvent être dynamiques (le système est toujours en service, son arrêt n'est pas nécessaire).

Les systèmes ouverts se composent de plusieurs éléments indépendants. Chaque élément est usuellement décrit sous la forme d'un objet. C'est pourquoi les systèmes à objets répartis sont le plus souvent présentés comme une instantiation du concept plus générique de systèmes ouverts.

2.2.1.2 L'interfonctionnement

Des équipements capables de coopérer sont qualifiés d'équipements interopérables. La capacité, l'aptitude ou la propriété qu'ont ces équipements à interopérer sur un réseau, via leur module de communication (protocole), sont qualifiées d'interopérabilité.

Définition 1 *Une définition informelle de l'interopérabilité serait de dire que c'est l'aptitude qu'ont N systèmes hétérogènes conformes à un ensemble de normes référencées en fonction des prescriptions d'un même profil, à communiquer et à coopérer selon ces normes dans un environnement représentatif de la réalité.*

Nous distinguons l'interopérabilité relative à un profil où toutes les couches du modèle sont considérées, de l'interopérabilité relative à une couche particulière du profil. Un profil de communication est un ensemble de normes choisies parmi les normes de base et où quand c'est applicable, une identification claire est faite pour les classes de conformité, les sous-ensembles, les options et les paramètres de chacune des normes choisies, afin d'accomplir une fonction particulière.

Définition 2 *Nous désignons par le terme interfonctionnement, l'aptitude des processus d'application d'une application répartie à coopérer.*

Nous faisons cette distinction pour mettre en évidence la différence entre le niveau communication normalisé et le niveau applicatif non normalisé. La localisation de ces deux définitions dans une application distribuée est donnée par la figure 2.1.

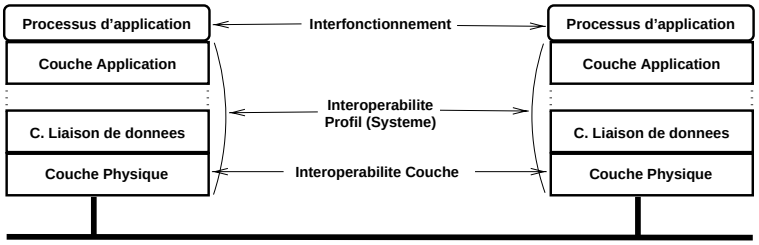

Figure 2.1: Interfonctionnement et Interopérabilité

2.2.2 Le modèle de référence RM-ODP

L'ambition de cette section n'est pas de présenter le standard dans sa globalité. Il s'agit juste de mettre en évidence les concepts qui nous intéressent afin d'éviter une lecture du standard qui s'avère laborieuse (certains points du standard ne sont qu'une énumération de définitions d'un modèle assez abstrait qui rend l'ensemble assez difficile à lire). Les points n'entrant pas directement dans le centre d'intérêt de cette thèse seront tout au plus énumérés. Cette introduction est structurée selon le plan suivant :

- la section 2.2.3 présente la partie 2 du modèle de référence [RM-b]. Cette partie définit les concepts de base : le modèle objet, l'architecture et la conformité.
- la section 2.2.4 présente la partie 3 du modèle de référence [RM-c]. Le cadre architectural ODP : les langages de point de vue, les fonctions et les transparences ODP.

La partie 1 [RM-a] de RM-ODP qui donne une vue d'ensemble du modèle de référence et la partie 4 [RM-d] de RM-ODP qui définit la sémantique architecturale ne sont pas présentées dans cette introduction.

2.2.3 Les concepts de base

Cette section présente de manière simplifiée d'une part le modèle objet et d'autre part les concepts architecturaux définis par RM-ODP. Ces concepts ne font en fait qu'introduire la section suivante (cf section 2.2.4) qui présente de manière plus précise l'architecture d'un système ODP. La conformité sera traitée plus tard dans la deuxième partie de ce mémoire quand nous parlerons du test des systèmes à objets répartis (cf section 4.3.2).

2.2.3.1 Le modèle objet

a) La notion d'objet Un objet est une entité contenant de l'information et offrant des services. La notion d'objet est complètement générique. Nous verrons plus tard que c'est l'unité de description pour tous les points de vues d'une spécification (cad toutes les abstractions d'un système à objets répartis). Un objet est caractérisé par :

- son identité :
  L'identité d'un objet permet de le distinguer de tout autre objet.

- son état :
  L'état d'un objet caractérise l'information qu'il détient à un instant donné dans le temps.

- son comportement :
  Le comportement d'un objet caractérise l'ensemble des changements d'états possibles qui peuvent l'affecter et définit l'ensemble des actions potentielles auxquelles l'objet peut prendre part.

b) La notion d'action L'ensemble des actions d'un objet est subdivisé en deux groupes :

- les actions internes;

- les actions externes ou interactions;

Une action interne se produit sans la participation de l'environnement de l'objet tandis que les interactions se produisent avec son environnement. On définit par environnement tout ce qui n'est pas l'objet lui-même.

c) La notion d'interface L'interface d'un objet définit l'ensemble des interactions possibles de cet objet, qui correspond à un comportement observable. En effet, du fait de l'encapsulation, seules les interactions sont observables. Un objet peut avoir plusieurs interfaces et leur nombre peut varier dans le temps. Celles-ci constituent donc des vues abstraites des fonctions fournies par un objet en masquant la façon dont celui-ci manipule l'information ou effectue les changements d'état.

d) L'encapsulation Les notions d'actions et d'interfaces nous amènent tout naturellement à définir le concept le plus important du modèle objet de RM-ODP : l'encapsulation.
L'encapsulation signifie, en termes simples, que les données, de même que les méthodes permettant de les manipuler, sont contenues dans un même emballage et que l'environnement ne peut directement atteindre les données associées à cet objet.

Le modèle objet de RM-ODP introduit de plus des notions comme les gabarits, le typage, le sous-typage. Elles ne sont pas précisées dans ce mémoire, le lecteur est prié de se référer à [RM-b].

2.2.3.2 Les concepts architecturaux

Cette section présente un ensemble de concepts de structuration permettant de décrire et d'organiser un système à objets répartis. Cette présentation est importante car elle participe plus tard (cf section 2.3.2) à la justification de la modélisation explicite des relations (et donc des rôles).

a) Groupe et Domaine Pour des raisons structurelles ou comportementales, un ensemble d'objets peut être organisé en groupe. Lorsque celui-ci possède un objet de contrôle, on l'appelle domaine. Ceci nous permet d'introduire des notions telles que : l'autonomie, l'autorité et le contrôle. Par exemple, dans un domaine de sécurité, les objets sont soumis à la politique de sécurité établie par l'objet "autorité de sécurité".

b) Contrat : Obligation, Permission et Prescription Lorsque des objets interagissent, la partie de comportement collectif de ces objets est régie par un contrat qui spécifie les obligations, les permissions et les interdictions s'appliquant aux objets vis-à-vis de leur environnement :

- une obligation est une prescription stipulant la nécessité d'un comportement particulier. Elle est remplie par l'occurrence du comportement prescrit;
- une permission est une prescription qui autorise un comportement. Il est donc possible que le comportement ne se produise pas;
- une interdiction est une prescription stipulant qu'un comportement particulier ne doit pas se manifester. Il est donc obligatoire que le comportement ne se produise pas.

Un contrat spécifie le (ou les rôles) des objets. C'est une spécification du comportement dynamique d'une configuration d'objets dans la mesure où les règles qu'il établit s'appliquent à une partie de comportement collectif non nécessairement permanente dans le temps.
Par définition, le comportement spécifié dans le contrat est le "comportement correct". Tout écart de compatibilité avec ce comportement correspond à une défaillance, c'est à dire à une violation du contrat. Une défaillance peut résulter d'une erreur, qui est la manifestation d'une faute dans un objet.

c) Liaison L'établissement d'un contrat entre des objets donnés résulte d'un comportement d'établissement entre ces objets, par exemple une négociation. Lorsque le contrat est établi, son existence définit un contexte contractuel. La réalisation d'un comportement d'établissement résulte

en une relation entre objets appelée relation de liaison. Puisqu'il peut avoir plusieurs rôles, un objet peut être impliqué dans plusieurs relations de liaison simultanées. Le contexte contractuel résultant d'un comportement de liaison est appelé liaison (binding).

2.2.4 Le cadre architectural

Cette section commence par une introduction sur les cinq points de vue (viewpoint) nécessaires à la description des systèmes ODP. A chaque point de vue correspond un langage de point de vue. Une attention particulière est ensuite portée sur deux de ces langages : le langage d'information et le langage de traitement. Les deux sections suivantes présentent respectivement les fonctions et les transparences à la répartition des systèmes ODP. Ces notions très importantes dans le contexte des systèmes répartis ne seront que brièvement présentées compte tenu du centre d'intérêt de notre travail.

2.2.4.1 Les points de vue

Un point de vue est une subdivision d'une spécification d'un système complexe. Il permet de concevoir et spécifier ce système à des niveaux d'abstraction différents. L'objectif principal de la subdivision est de générer des spécifications plus concises et donc moins complexes pour l'homme. Dans le contexte de RM-ODP, les cinq points de vues qui ont été définis sont les suivants :

- le point de vue entreprise décrit un système ODP en spécifiant les besoins d'un domaine d'application donné. Il est donc axé sur les rôles, les objectifs, le domaine d'application et les politiques du système.
- le point de vue information spécifie l'information gérée et manipulée ainsi que les traitements de cette information effectués par un système ODP. Il est axé sur la sémantique et le traitement de l'information.
- le point de vue traitement exprime une décomposition fonctionnelle d'un système en objets qui interagissent via des interfaces de traitement.
- le point de vue ingénierie décrit l'infrastructure qui est requise pour mettre en oeuvre une spécification de traitement. Il est axé sur les mécanismes et les fonctions qui prennent en charge l'interaction répartie des objets du système.
- le point de vue technologie est axé sur les choix technologiques de réalisation et d'implémentation du système.

Les points de vue représentent des abstractions différentes du même système. Ils ne forment pas une séquence fixe et ne doivent donc pas être assimilés à une structuration en couches. Cependant, il existe une relation entre ces différents points de vue qui est donnée dans la Figure 2.2 et qui les représente suivant une vue conception de logiciel.

Nous nous intéressons à la spécification et la simulation fonctionnelle (comportementale) des systèmes à objets répartis. Notre attention se porte donc tout naturellement sur le point de vue

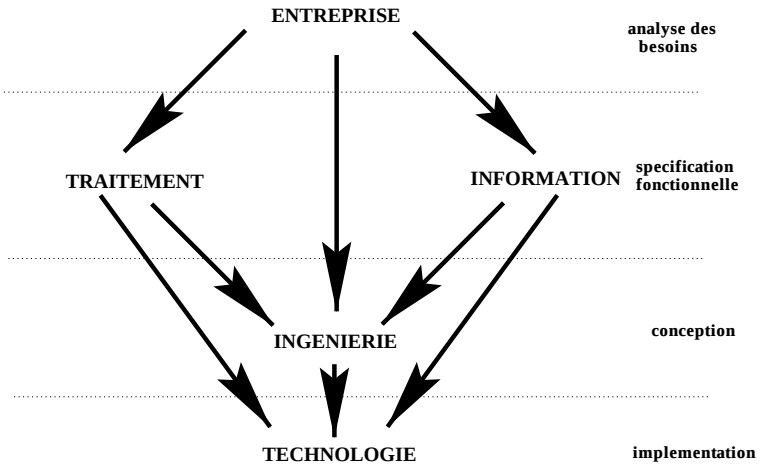

Figure 2.2: ODP selon une vue conception de logiciel

traitement (décomposition fonctionnelle du système) et sur le point de vue information (décomposition sémantique et traitement de l'information). Le point de vue d'entreprise ne donne pas d'informations pertinentes en ce qui concerne le comportement (dynamique) du système, il fixe tout au plus des paramètres statiques que l'on retrouve dans le point de vue information, modélisés sous la forme d'un objet d'information. Les points de vue d'ingénierie et de technologie précisent les détails d'implémentation de l'application répartie. Ces détails ne sont pas nécessaires à la simulation et peuvent donc être abstraits de nos spécifications.

Après avoir précisé la notion de langage de point de vue, nous allons dans les paragraphes suivants détailler les langages d'information et de traitement. Pour des informations supplémentaires sur le langage d'entreprise, d'ingénierie et de technologie le lecteur est prié de se rapporter à [RM-c].

Afin d'exprimer de manière précise leur contenu, RM-ODP définit un langage pour chacun des points de vue. Il ne faut pas confondre langage et notation. C'est le sens le plus large du mot langage qui est utilisé "un ensemble de termes (un vocabulaire) et de règles sur ces termes (une grammaire) qui permet de construire des phrases". Il n'est par contre pas proposé de notation pour les langages de points de vue. Toutefois il est préconisé (mais ce n'est pas une obligation) par [RM-d], l'utilisation des techniques formelles standardisées (Lotos [LOT], Z [Spi89], SDL [IT92f] et Estelle [iso92a]) comme notation pour les langages de points de vue.

2.2.4.2 Le langage d'information

Le langage d'information d'un système ODP définit la sémantique et le traitement de cette information. Du point de vue information, le système ODP se décompose en des objets d'information. Les objets d'information représentent les unités d'information. La notion de répartition de l'information n'est pas prise en compte à ce point de vue. Le langage d'information délimite les informations (données) qu'il échange avec son environnement et il spécifie les changements d'états qui résultent des échanges d'informations avec l'environnement ou des modifications internes.

Le comportement des objets d'information La description du comportement dynamique des objets d'information est liée à la notion d'état de ces objets. Comme dans tous les systèmes distribués, il n'y a pas de notion d'horloge globale ou d'état du système. L'état du système est donné par l'ensemble des états de tous ses composants. Dans le cas des systèmes à objets répartis, c'est donc celui de l'ensemble des objets qui le compose.
Pour éviter d'énumérer l'ensemble des états possibles et les transitions associées possibles, le modèle de référence ODP suggère d'utiliser des prédicats (à la Z [Spi89]) sur le modèle d'information. Cet ensemble de prédicats est présenté sous la forme d'un schéma :

- le schéma statique décrit l'état et la structure du système à un instant donné dans le temps;
- le schéma d'invariant décrit des propriétés qui doivent toujours être vérifiées dans le temps;
- le schéma dynamique décrit comment un système peut modifier son état ou sa structure. Il décrit la transition d'un schéma statique initial vers un schéma statique final tout en vérifiant un schéma d'invariant.

2.2.4.3 Le langage de traitement

Au contraire du langage d'information, le langage de traitement adresse principalement le problème de la répartition de l'information. Du point de vue du traitement, le système ODP se décompose en des objets de traitement. Les objets de traitement représentent les unités de répartition qui peuvent (mais pas nécessairement) être réparties.

Le langage de traitement fixe des contraintes de modélisation qui assurent que les objets (de traitement) pourront être répartis de manière transparente (cad que la répartition effective des objets de traitement n'est pas un problème de modélisation un problème d'implémentation). Du point de vue du traitement, il sera fait abstraction de tous les détails concernant la localisation, la construction et les modes de communications.

Dans [RM-c], le lecteur trouvera de plus amples détails sur les concepts suivants, définis par le langage de traitement, qui ne sont ici que résumés :

- le modèle d'interaction :
  l'ensemble des interactions est subdivisé en trois groupes : les opérations, les flux et les signaux;
- les interfaces de traitement :
  elles sont décrites par une signature, un comportement et un contrat d'environnement. Il existe pour chaque type d'interaction un type d'interface;
- le modèle de liaison :
  les interactions entre interfaces de traitement nécessitent préalablement l'établissement d'une liaison entre elles. On distingue les liaisons implicites (créées implicitement par l'infrastructure) des liaisons explicites (créées explicitement à la suite d'une requête d'un objet de traitement)

Le comportement des objets de traitement Le langage de traitement ne précise pas la manière de spécifier le comportement des objets de traitement. Tout formalisme adéquat peut faire l'affaire. La description du comportement avec les éléments du langage lui même n'est pas une tâche aisée compte tenu du peu d'expressivité de ce formalisme comme le fait remarquer [NS95].
Une seconde approche pour la description du comportement des objets de traitement est de les considérer comme des systèmes ODP et d'appliquer le modèle de référence ODP récursivement sur ces objets. On utilisera principalement le modèle du point de vue information pour décrire les objets de traitement.

A noter que c'est l'approche soutenue par [Gen95] où, au point de vue de traitement, des références explicites sont faites aux schémas dynamiques spécifiés au niveau du point de vue information. Cette approche est notamment celle adoptée dans le secteur des télécommunications par TINA [TIN94b, TIN94a] et l'ITU-SG4 [G8596].

2.2.4.4 Les fonctions

Les fonctions ODP sont une collection de fonctions qui peuvent être présentes dans les systèmes ODP pour supporter, essentiellement, les besoins "génériques" du langage de traitement et du langage d'ingénierie[1]. Les principales fonctionnalités définies dans RM-ODP sont :

- les fonctions de gestion système qui incluent la gestion de noeuds, d'objets, de grappes et de capsules;
- les fonctions de coordination qui incluent les fonctions de notification d'événement, de sauvegarde (et de reprise), de désactivation (et de réactivation), de groupe, de duplication, de migration, de ramasse-miettes pour les références d'interfaces d'ingénierie et de transaction;
- les fonctions de conteneur (repository) qui incluent les fonctions de stockage, de gestion de base d'information, de relocalisation, de conteneur de types et de courtage (trading);
- les fonctions de sécurité qui incluent les fonctions de contrôle d'accès, d'audit, d'authentification, d'intégrité, de confidentialité, de non-répudiation et de gestion de clé.

2.2.4.5 Les transparences à la répartition

Le but des transparences à la répartition est de transférer les difficultés liées à la répartition effective de l'application du programmeur à l'infrastructure sous-jacente. Le modèle de référence ODP définit un ensemble de transparences à la répartition :

- la transparence à l'accès masque les différences dans la représentation des données et les mécanismes d'invocation pour permettre l'interopérabilité entre objets;
- la transparence à la défaillance masque au niveau d'un objet les défaillances et la possibilité de reprise d'autres objets (ou de lui-même) pour garantir la tolérance aux fautes;

[1] On peut les assimiler aux System Management Functions définit dans OSI Network Management

- la transparence à la localisation masque l'utilisation des informations de localisation dans l'espace lors de l'identification et de la liaison à des interfaces. Elle permet aux objets d'accéder aux interfaces sans utiliser d'information de localisation;

- la transparence à la migration masque au niveau d'un objet la capacité d'un système à modifier la localisation de cet objet. La migration est utilisée pour équilibrer la charge et réduire le temps de latence;

- la transparence à la relocalisation masque la relocalisation d'une interface vis-à-vis des autres interfaces liées à elles;

- la transparence à la duplication masque l'utilisation d'un groupe d'objets mutuellement compatibles dans leurs comportements pour prendre en charge une interface. La duplication est souvent utilisée pour augmenter les performances et la disponibilité;

- la transparence à la persistance qui masque au niveau d'un objet la désactivation et la réactivation d'autres objets (ou de lui-même). La désactivation et la réactivation sont souvent utilisées pour maintenir la persistance d'un système lorsqu'un système ne peut pas lui assurer de manière continue les fonctions de traitement, de stockage et de communication;

- la transparence aux transactions masque la coordination des activités participant à une configuration d'objets pour en assurer la cohérence.

2.2.5 Résumé

On peut résumer cette présentation de RM-ODP en fonction du centre d'intérêt de cette partie (un formalisme pour la description du comportement dynamique dans les systèmes à objets répartis) suivant trois points :

- proposition d'un modèle générique où la notion de comportement d'un objet occupe une place prépondérante.

- définition d'un cadre architectural suivant une structure de points de vue. Un des aspects intéressants de cette définition est qu'elle définit la notion de comportement d'un objet suivant le point de vue dans lequel on se situe.

- définition d'un certain nombre de transparences à la répartition.

Si l'on observe les principaux systèmes à objets répartis, le premier et le second point ne sont pas abordés tandis qu'ils prennent très souvent en compte le troisième point, la transparence à l'accès, qui peut assurer tout au plus l'interopérabilité et non l'interfonctionnement. Cette situation résume bien le problème qui a été posé dans l'introduction que l'on peut résumer par un manque de formalisme pour la description du comportement dynamique des systèmes à objets répartis. Le rappel de cette problématique est exposé dans la prochaine section et l'examen des différents formalismes existants constituent la suite de ce chapitre.

2.3 Objectifs et hypothèses de travail

2.3.1 Rappel des objectifs

Pour résoudre les problèmes évoqués lors de l'introduction (interfonctionnement dans les systèmes à objets répartis et production de spécifications correctes), nous nous sommes proposés deux objectifs :

- le premier objectif est de proposer un formalisme pour la description du comportement dynamique des systèmes à objets répartis;
- le second objectif est de fournir un simulateur adapté à ce formalisme pour effectuer des simulations comportementales et ainsi détecter les erreurs de spécification.

A première vue la tâche semble délicate (voir impossible) s'il l'on demeure à un niveau aussi peu précis. Une étude plus fine de la problématique et des solutions proposées nous a permis de dégager un certain nombre d'hypothèses de travail qui sont présentées dans la section suivante.

2.3.2 Hypothèses de travail

Les motivations pour présenter notre étude suivant des hypothèses de travail sont que d'une part ces hypothèses nous permettent de fixer un cadre de travail plus formel et d'autre part ces hypothèses nous permettent de dégager l'ensemble des propriétés que devra posséder le formalisme de description du comportement dynamique des systèmes à objets répartis. Nous le nommons pour la suite de ce mémoire de formalisme "idéal". Ces hypothèses de travail sont classées de la plus forte (propriété intrinsèque du modèle) à la plus faible (choix de modélisation liés à notre activité).

Il est important de rappeler que l'hypothèse de travail la plus forte est d'offrir une solution (un formalisme) simple. La solution doit être utilisable par un ingénieur qui n'est pas forcément un adepte des méthodes formelles et de leurs outils associés. Cette hypothèse induit que la propriété principale du formalisme "idéal" doit être la simplicité. Nous verrons plus tard comment nous avons pris en compte ce souci de simplicité.

2.3.2.1 Hypothèse 1 : Traitement du nondéterminisme

Le nondéterminisme est une propriété inhérente aux systèmes répartis. La modélisation (spécification et simulation) du comportement dynamique de ce type de système devra donc, elle aussi, prendre en compte le nondéterminisme.

Propriété 1 *P1 : Un formalisme permettant la spécification du nondéterminisme*

2.3.2.2 Hypothèse 2 : Spécifications Exécutables

Un de nos objectifs initiaux vise la simulation du comportement dynamique de systèmes à objets répartis à l'aide de spécifications décrites dans le formalisme que l'on doit proposer. Pour pouvoir l'exécuter grâce à un simulateur, il faut donc doter le formalisme de description comportementale d'une sémantique opérationnelle.

Propriété 2 *P2 : Un formalisme doté d'une sémantique opérationnelle*

Notre spécification, une fois qu'elle est munie d'une sémantique opérationnelle, devient exécutable. Cependant [HC90] rétorque que les spécifications à partir du moment où elles sont exécutables deviennent moins expressives et moins abstraites. Nous avons donc recherché dans la littérature quelles étaient les parades à ce type de problème. Conformément à ce qu'il est préconisé dans [Fuc92], notre choix s'est porté vers les méthodes déclaratives qui sont un bon compromis entre l'exécutabilité, l'expressivité et l'abstraction.

Propriété 3 *P3 : Un formalisme de type déclaratif*

2.3.2.3 Hypothèse 3 : Spécifications fonctionnelles

Les interfaces d'objets et les transparences d'accès évoquées dans RM-ODP garantissent seulement que les messages qui transitent sur le réseau entre les divers objets sont corrects. Comme nous l'avons déjà vu, on assure ainsi l'interopérabité. L'interfonctionnement nécessite bien plus que cela. Le comportement des objets (vis à vis des messages qui arrivent à leurs interfaces) doit être spécifié formellement.

La spécification (formelle) de l'interfonctionnement de ces systèmes doit faire apparaître les propriétés fonctionnelles de ces systèmes. Spécifier les propriétés fonctionnelles signifie qu'il ne faut surtout pas modéliser l'application dans sa version finale. Par exemple, les informations liées à l'implémentation ainsi que les étapes successives de "raffinement" entre la spécification et l'implémentation ne doivent pas apparaître dans les propriétés fonctionnelles. Il est donc tout naturel de nous situer au niveau des spécifications fonctionnelles de ce type de systèmes. Comme nous l'avons vu dans la section 2.2.4.1, dans le contexte de RM-ODP, les spécifications fonctionnelles se situent à la fois au niveau du point de vue information et du point de vue de traitement .

Propriété 4 *P4 : Un formalisme adapté à la description du comportement d'un objet au niveau du point de vue information*

Propriété 5 *P5 : Un formalisme adapté à la description du comportement d'un objet au niveau du point de vue traitement*

De plus, en nous situant à ces points de vue de RM-ODP (information et traitement), nous spécifions des propriétés sur un modèle générique de système à objets répartis. Les détails faisant perdre cet aspect générique sont spécifiés au niveau des points de vue ingénierie et technologie. Nous désirons conserver cette généricité de manière à pouvoir appliquer notre travail dans le contexte du RGT (motiviation initiale) mais aussi dans tout autre système à objets répartis : CORBA, TINA, ... Cette conservation de la généricité implique que d'une part il ne faut pas faire de références à une technologie d'objets répartis particulière et que d'autre part le modèle d'objet doit lui aussi être le plus générique possible (indépendant de tout langage de programmation par exemple).

Propriété 6 *P6 : Un formalisme indépendant de toute technologie de distribution des objets*

2.3.2.4 Hypothèse 4 : Modélisation de la dynamique

Traiter le comportement dynamique d'un objet séparé n'a pas de sens. Il faut plutôt raisonner sur un groupe d'objets. Du point de vue des applications existantes, ceci se matérialise dans le contexte de la gestion OSI par la notion d'"Ensemble" défini par le NMF [NMF92] ou dans le contexte de CORBA par la notion de "Distributed Object Framework" défini par l'OMG [OMG96]. Tous deux identifient comme unité de base d'une spécification, un groupe d'objets en relation. Ce groupe d'objets est aussi appelé configuration d'objets.

De la notion de relation à celle de rôle, il n'y a qu'un petit pas à franchir. Comme nous l'avons vu dans 2.2.3.2, RM-ODP définit la notion de groupe et de contrat. Le contrat spécifie le (ou les) rôle des objets. Il est défini comme une spécification du comportement dynamique d'une configuration d'objets dans la mesure où les règles qu'il établit s'appliquent à une partie de comportement collectif non nécessairement permanente dans le temps. A partir du comportement d'un objet, il est possible d'identifier des rôles de l'objet par extraction d'un sous-ensemble du comportement total de cet objet. Considérer un objet en fonction de son rôle revient alors à s'intéresser à un sous-ensemble nommé de ses actions en faisant abstraction de ses autres actions. Un objet peut avoir plusieurs rôles à un instant donné et peut avoir des rôles différents selon les moments.

Cette tendance quant à l'utilisation des relations se manifeste depuis quelques temps dans la littérature [Bap95, KR94, A.C93] et se matérialise au niveau de la standardisation du RGT par le GRM (Generic Relationship Model) [GRM]. Le concept de relation est maintenant largement utilisé dans le domaine de la conception des systèmes d'information. Principalement il existe deux modèles :

- le modèle Entités-Relation défini par Chen [Che76];
- le modèle Object Role Models (ORM) défini par Nijsssen et son équipe à Control Data au Pays-Bas. NIAM [VvB82] est l'utilisation la plus célèbre du modèle ORM.

a) Une formalisation basée sur les relations explicites Il faut donc être capable de refléter d'une part les dépendances entre les objets (configuration) et d'autre part les interactions entre les objets (comportement dynamique). Bien que dans RM-ODP, le modèle de relation (et donc de rôle) ne soit pas préconisé, l'expression de ces relations nous apparaît essentiel. C'est pourquoi le formalisme devra rendre explicite les relations entre objet.

Propriété 7 *P7 : Un formalisme doté d'outils de spécification des relations et des rôles*

b) Une formalisation basée sur les assertions Il faut se donner un moyen de contrôler ces comportements dynamiques entre objets (modélisés par des relations). Pour cela, nous proposons d'utiliser des assertions en suivant les directives de [Kil93] qui préconise leur utilisation dès lors que l'on utilise un modèle de relation et que l'on veut s'assurer de la cohérence de l'information répartie (via les relations) sur les différents objets . L'idée des assertions est de donner à l'utilisateur la possibilité de spécifier des propriétés qui vont être vérifiées durant l'exécution des simulations. Elles servent notamment à assurer la correction du système dans des cas particuliers d'exécutions

compliquées où l'utilisateur peut avoir du mal à contrôler la dynamique du système (nondéterminisme, ajout incrémental de nouveaux comportements qui influent sur ceux déjà spécifiés de manière non prévue par l'utilisateur, ...).

Propriété 8 *P8 : Un formalisme permettant la spécification d'assertions*

2.3.2.5 Hypothèse (limitation) 5 : Pas d'aspect temporel

Une hypothèse pour simplifier notre travail réside en le fait que nous faisons abstraction de la variable de temps dans nos spécifications comportementales. Certes ceci restreint le spectre des propriétés que nous pouvons spécifier (notamment propriétés liées à la performance du système et/ou aux aspects temps réel) mais l'ajout de ce type de paramètre complique la problématique de manière trop importante. Cette simplification nous permet d'obtenir un formalisme plus simple pour la spécification des propriétés comportementales classiques ce qui demeure un des objectifs initiaux de ce travail.

2.3.3 Résumé

Notre étude nous a permis de dégager un ensemble de propriétés que nous appellerons pour la suite de ce mémoire : les propriétés du formalisme "idéal".

- P1 : Un formalisme permettant le nondéterminisme;
- P2 : Un formalisme doté d'une sémantique opérationnelle;
- P3 : Un formalisme de type déclaratif;
- P4 : Un formalisme adapté à la description du comportement d'un objet au niveau du point de vue information;
- P5 : Un formalisme adapté à la description du comportement d'un objet au niveau du point de vue traitement;
- P6 : Un formalisme indépendant de toute technologie de distribution des objets;
- P7 : Un formalisme doté d'outils de spécification des relations et des rôles;
- P8 : Un formalisme doté d'assertions.

2.4 Examen des différentes alternatives pour la formalisation du comportement dynamique des systèmes à objets répartis

Maintenant que nous sommes munis d'un ensemble de propriétés que le formalisme doit posséder (les propriétés du formalisme "idéal"), nous allons examiner les différentes alternatives pour la formalisation du comportement dynamique des systèmes à objets répartis. Il s'agit en fait d'examiner

les formalismes existants et de vérifier qu'ils possèdent bien l'ensemble des propriétés. Il existe un grand nombre de méthodes formelles. Aussi nous proposons comme stratégie de parcours d'examiner les deux grandes classes de formalismes adaptées à la formalisation des systèmes répartis et d'étudier ensuite leurs extensions objets. On aura ainsi les deux grandes familles de spécification aptes à représenter le comportement dynamique des systèmes à objets répartis. Plus de références sur des approches qui concernent la description du comportement des objets (mais pas forcément répartis) sont rassemblées dans [KMS96].

2.4.1 Les formalismes existants pour la spécification formelle des systèmes répartis

Ces formalismes peuvent se diviser en deux classes :

1. les formalismes basés sur les transitions d'états;
2. les formalismes basés sur l'approche axiomatique.

2.4.2 Modèles de transition d'états

Un système réparti est défini par un ensemble d'entités concurrentes qui interagissent entre elles en échangeant des messages. Ce type de comportement peut être modélisé par des systèmes de transitions où les différentes phases du système réparti constituent les états du système. L'état d'une entité change à chaque occurrence d'un événement (une valeur du signal d'entrée). Un événement peut avoir un effet si certaines conditions sont vérifiées par le système. Le passage d'un état à un autre est appelé transition. Les entités peuvent être modélisées sous forme d'un système de transitions. Les modèles de transition d'états les plus connus sont les automates à états finis mais on peut citer les algèbres de processus (pour LOTOS [LOT]) ou les réseaux de Petri [Rei85].

Les automates à états finis (FSM Finite State Machine)

Un automate à états finis est un quadruplet (S, T, α, β) où :

- S est un ensemble fini d'états;
- T est un ensemble de transitions;
- α et β sont deux applications de T dans S qui à toute transition t de T associent les deux états $\alpha(t)$ et $\beta(t)$ qui sont respectivement l'origine et le but de la transition t.

Les FSM peuvent être représentés de plusieurs façons, les plus courantes sont les diagrammes de transitions et les tables de décision. L'échange de données se fait par messages à travers des canaux de communication. Ces canaux peuvent être modélisés par des files d'attentes de type "FIFO" (First in First out pour premier entré premier sorti). Les FSM permettent de traiter des objets abstraits (messages). Ces objets peuvent être enfilés ou reçus d'une file d'attente. La seule

synchronisation qui existe pour ces modèles est celle qui est imposée par les règles d'accès aux files. Le principal objectif de ces modèles est de décrire les relations de synchronisation entre les entités d'un système réparti. Leurs principaux défauts sont la difficulté de représenter les structures de données et les évolutions liées aux changements de leurs valeurs, ainsi que la perte de clarté des descriptions lorsque les schémas des systèmes répartis deviennent grands. Les FSM communicants s'avèrent donc limités et insuffisants pour la spécification et la vérification des systèmes répartis. Pour corriger les défauts de ces modèles, deux extensions ont été faites sur le modèle de base :

1. premièrement, on a introduit la notion de variables. Ces dernières sont des symboles qui réfèrent à des valeurs dans un domaine donné;

2. deuxièmement, un ensemble d'opérations est associé aux domaines. Ces opérations peuvent être arithmétiques ou logiques.

Les automates à états finis étendus (EFSM Extended Finite State Machine) Dans ce modèle, chaque entité est caractérisée par un ensemble de variables qui représente l'état de l'automate. Ces variables sont regroupées dans un vecteur de variables appelé vecteur d'état. On pose V_i le vecteur d'états de l'entité i et $v_{ij} \in V_i$ un élément du vecteur (la variable d'état v_{ij} prend une valeur dans un domaine D_{ij}). Un canal C_k reliant deux entités est représenté par une variable d'état z_k. Cette dernière représente la séquence de messages contenus dans le canal. L'état global du système réparti est donc spécifié par le vecteur suivant : $(V_1, V_2, \ldots, V_n, z_1, z_2, \ldots, z_m)$. Ce vecteur représente un système réparti à n entités et à m canaux de transmission. L'état initial du système réparti est donné par la valeur initiale de chaque variable d'état appartenant au vecteur; quant aux canaux, ils sont initialement vides. La valeur de ces variables d'états est changée à chaque occurrence d'un événement. Chaque entité P_i réagit à un ensemble d'événements. Ces événements agissent uniquement sur le vecteur d'état V_i et sur les canaux accessibles à partir de P_i. Les EFSM peuvent être exprimés sous des formes syntaxiques proches des langages impératifs. On obtient ainsi les langages comme SDL [IT92f], ESTELLE [iso92a].

2.4.3 L'approche axiomatique

Cette approche consiste en une modélisation mathématique des propriétés des systèmes répartis en utilisant des techniques comme :

- les types de données abstraits [B81];
- les techniques logiques comme Z [Spi89] ou bien la logique temporelle comme TLA [Lam91].

Les types de données abstraits

Les types de données abstraits définissent les opérations que l'on peut effectuer sur une instance du type indépendamment de son implantation. La définition d'un type est axiomatique si les opérations définies sur ce type satisfont des axiomes.

Ainsi, définir un type de données abstrait par la méthode algébrique-axiomatique, c'est définir une théorie du premier ordre composée :

- d'un langage défini par des opérateurs, des variables, une syntaxe et un typage des opérateurs;
- d'un ensemble d'axiomes présentés sous forme d'équations;
- d'un ensemble de règles d'inférence pour la substitution et le remplacement d'égalité.

Un système réparti peut être vu comme une structure de données complexe décrite par un type abstrait dont les états sont définis par les valeurs des données initiales (état initial) et des données induites par l'application des opérateurs (états successeurs). Ainsi, la définition d'un système réparti en type abstrait commence par identifier l'ensemble des sélecteurs et des constructeurs nécessaires à sa description et ensuite par décrire le comportement du système réparti par des axiomes qui expriment les transformations du contenu des sélecteurs lors de l'application des constructeurs. Il est possible d'associer des règles d'induction pour effectuer des preuves.

L'approche axiomatique offre la possibilité d'abstraction mathématique dans la description des systèmes répartis ce qui permet de :

- paramètriser les spécifications définies;
- exprimer des mécanismes complexes;
- prouver des propriétés sur le système réparti grâce à leur définition axiomatique et au système d'aide à la preuve offert par des démonstrateurs de théorèmes.

Cependant, on peut trouver de nombreux inconvénients à ces méthodes, les principaux étant la nécessité d'avoir des connaissances théoriques approfondies afin de guider le démonstrateur de preuves. De plus, l'architecture de base des systèmes répartis se trouve totalement noyée dans des aspects secondaires (grand nombre de détails à prouver avant d'arriver au résultat).

2.4.4 Discussion en fonction des propriétés du formalisme "idéal"

Pour la clarté de cette discussion, nous adoptons la terminologie issue de [JKS91] qui qualifie les formalismes basés sur les transitions d'états par POM (Process Oriented Model) et les formalismes basés sur l'approche axiomatique par DOM (Data Oriented Model) . Avant d'entamer cette discussion, il est important de préciser comment POM et DOM intègrent la notion d'objet.

2.4.4.1 Les extensions Orientées Objet (OO)

Dans le contexte de POM, par exemple dans le cas des automates à états finis, on affecte un objet à un automate. A noter que pour intégrer la notion de donnée, ce sont forcément les EFSM qui sont considérés. Ce type d'extension permet d'assez bien maîtriser les aspects "dynamiques" des systèmes OO. On ne peut pas, par contre, modéliser la dynamique des configurations (reconfiguration) car ce modèle se repose sur des configurations fixes.
Les aspects "réutilisation" de l'OO tels que l'héritage, l'agrégation, ... ne sont pas du tout pris en compte.

Dans le contexte de DOM, par exemple pour les types de données abstraits, on affecte un objet à un type. Ce type d'extension permet de bien maîtriser l'aspect "réutilisation", c'est par contre la modélisation des aspects dynamiques qui pose des problèmes.

Comme nous allons le voir, cette dualité entre POM et DOM va se refléter dans l'examen des propriétés qui va suivre.

2.4.4.2 Examen des Propriétés

a) P1 (formalisme permettant le nondéterminisme) POM et DOM sont suffisamment abstraits pour permettre la spécification du nondéterminisme.

b) P2 (formalisme doté d'une sémantique opérationnelle) Pour POM, du fait de l'importance donnée au contrôle, il existe quasiment toujours une sémantique opérationnelle. En ce qui concerne DOM, c'est très peu souvent le cas. Les exemples d'intégration de sémantique opérationnelle sont le plus souvent effectués dans le contexte des spécifications algébriques [BG94, BBG96].

c) P3 (formalisme de type déclaratif) POM est peu souvent déclaratif, alors que c'est souvent un propriété de base de DOM.

d) P4 (formalisme adapté à la description du comportement d'un objet au niveau du point de vue information) Si l'on fait référence à ce qui a déjà été présenté dans la section 2.2.4.2 et qui est donc préconisé par RM-ODP (cf [RM-d]), POM n'est pas adapté à représenter des comportements d'objets au niveau information. Par contre, DOM est particulièrement adapté à ce type de spécifications.
Le langage Z [Spi89] (ou ses extensions orientées objets : Zest [CW93], ObjectZ [CDD+89]) est d'ailleurs cité dans la partie 4 de RM-ODP [RM-d] pour faire office de notation pour le langage d'information de RM-ODP . Z est basé sur la théorie des ensembles et les prédicats du premier ordre. Les schémas du langage Z sont particulièrement bien adaptés à la description des différents schémas nécessaires à la description du comportement des objets d'information.

e) P5 (formalisme adapté à la description du comportement d'un objet au niveau du point de vue traitement) Si l'on fait référence à ce qui a déjà été présenté dans la section 2.2.4.3 et qui est donc préconisé par RM-ODP (cf [RM-d]), DOM est particulièrement adapté à représenter des comportements dynamiques d'objets au niveau traitement. Par contre, ce n'est pas du tout le cas de DOM.
Les techniques de descriptions formelles issues du domaine des systèmes répartis comme LOTOS [LOT], SDL [IT92f], Estelle [iso92a] sont quant à elles proposées pour la formalisation du comportement des objets de traitement par [RM-d]. Là encore, c'est l'utilisation des extensions orientées objets comme MT-LOTOS pour LOTOS ou OSDL (SDL'96) pour SDL qui est envisagée.

f) P6 (formalisme indépendant de toute technologie de distribution des objets) Les deux classes de formalismes (DOM et POM) sont suffisamment abstraites pour remplir cette propriété.

g) P7 (formalisme doté d'outils de spécification des relations et des rôles) La conclusion est la même que pour P4, POM n'est pas du tout adapté alors que c'est le cas pour DOM.

h) P8 (formalisme doté d'assertions) La conclusion est la même que pour P4. La spécification des assertions n'est pas immédiate dans le contexte de POM, alors que ce n'est pas du tout délicat d'intégrer des assertions qui se révèlent être des propriétés comme les autres dans DOM. Leur utilisation de manière dynamique devient un problème de sémantique opérationnelle.

	POM	DOM
P1	*	
P2		*
P3	*	*
P4		*
P5	*	
P5	*	
P6	*	
P7	*	
P8	*	*

Tableau 2.1: Tableau récapitulatif de l'examen des propriétés du formalisme "idéal" en fonction de la classe d'appartenance des formalismes existants

Le Tableau 2.1 nous informe d'une part que parmi l'ensemble des formalismes existants, il n'existe pas de formalisme qui possède toutes les propriétés du formalisme "idéal". A partir de cette constatation, il existe deux stratégies possibles :

1. soit on affaiblit le formalisme "idéal" en lui retirant des propriétés;
2. soit on propose une autre approche (un nouveau formalisme).

C'est la seconde alternative que nous avons choisie car nous n'étions pas contraints à réutiliser un formalisme existant. Là encore trois stratégies sont possibles :

1. la première consiste à regrouper ces deux classes formalismes;
2. la seconde consiste à partir d'un formalisme existant à l'équiper des mécanismes manquants jusqu'à ce qu'il possède toutes les propriétés;
3. la troisième consiste à construire complètement notre propre formalisme.

La première alternative est délicate à réaliser car elle consiste à mélanger en fait des formalismes de haut niveau (DOM) avec des formalismes bas niveau (POM). Ce n'est pas impossible et cela a été réalisé avec le langage COOPN/2 [BBG96]. COOPN/2 (Concurrent Object-Oriented Petri Net/2) est un langage de spécification formel adapté au développement de systèmes à objets concurrents et répartis. Ce langage est basé sur les spécifications algébriques et les réseaux de Pétri.

La seconde alternative nous semble encore plus difficile à réaliser étant donné les différences intrinsèques entre DOM et POM.

C'est la troisième alternative que nous avons choisie car elle nous semble la plus appropriée à nos objectifs. De plus, nous allons nous efforcer de fournir un formalisme qui remplit les propriétés sans forcément nous reposer sur des formalismes existants qui sont soit trop haut niveau (DOM), soit trop bas niveau (POM) pour nos besoins.

C'est en fait en ne conservant que les propriétés de chacune des classes de formalismes existants et en construisant notre propre formalisme que nous pouvons rendre le résultat simple à spécifier et surtout à utiliser. DOM et POM ont été construits avec des besoins qui sont différents des nôtres (on peut citer la preuve principalement pour DOM et la génération automatique de code et la génération de tests en plus des simulations pour POM). Ces deux classes de formalismes sont en fait trop puissantes alors que nous allons proposer un formalisme dédié à la simulation du comportement dynamique des systèmes à objets répartis.

2.4.5 Résumé

Cette section a d'abord présenté les deux principales classes de formalismes existants dans le contexte des systèmes répartis : les formalismes basés sur les transitions d'états et ceux basés sur une approche axiomatique. Nous avons ensuite examiné l'adéquation entre ces deux classes de formalismes et les propriétés du formalisme "idéal". Cet examen nous informe qu'il n'existe pas de formalismes existants remplissant toutes les propriétés. C'est pourquoi nous comptons proposer notre propre formalisme. Notre propre formalisme outre le fait qu'il possède l'ensemble des propriétés du formalisme "idéal" sera situé à un niveau intermédiaire entre les formalismes haut niveau (des formalismes basés sur une approche axiomatique) et les formalismes bas niveau (des formalismes basés sur les transitions d'états).

2.5 Conclusion

La première partie de ce chapitre était une introduction à la problématique : le comportement dynamique des systèmes à objets répartis. Pour cela, nous nous sommes inspirés du modèle de référence ODP qui est un standard qui vise à l'interfonctionnement des systèmes ouverts.

A partir d'un rappel de nos objectifs initiaux et du modèle ODP, la seconde partie de ce chapitre a consisté en l'énoncé d'un ensemble de propriétés que le formalisme "idéal" doit posséder pour permettre la description du comportement dynamique des systèmes à objets répartis.

Enfin la troisième partie de ce chapitre a consisté en l'étude des différents formalismes existants (plus précisément en l'étude en fonction de leurs classes d'appartenance). Cette étude a ensuite débouché sur un examen de l'adéquation entre ces classes de formalismes et les propriétés du formalisme "idéal". Cet examen nous a permis de déduire que parmi l'ensemble des formalismes existants, aucun ne correspond à notre attente, ce qui justifie la création de notre propre formalisme. La construction de ce formalisme est tout de même nettement plus facile dès lors que nous avons identifié les propriétés nécessaires et suffisantes. Le chapitre suivant présente notre formalisme BL (Behavior Language pour langage de comportement) et représente ainsi notre propre vision du

formalisme "idéal" pour la représentation du comportement dynamique dans les systèmes à objets répartis.

Chapitre 3

Le formalisme de description des comportements BL

3.1 Introduction

Le rappel de nos objectifs ainsi que l'énoncé de nos hypothèses de travail nous ont permis de dégager un ensemble de propriétés que le formalisme "idéal" doit posséder. A la suite de l'examen de l'adéquation entre les différents formalismes existants et cet ensemble de propriétés, nous avons justifié la motivation pour la création de notre propre formalisme : BL (Behavior Language pour langage de comportement).

Ce chapitre se décompose en trois parties :

- La première partie de ce chapitre va présenter les principes de base de BL. Cette présentation va bien sûr enchaîner par un examen de l'adéquation entre BL et les propriétés du formalisme "idéal".

- La description du formalisme BL constitue la seconde partie de ce chapitre. Plutôt que de présenter la grammaire du langage (qui pourrait rendre la lecture ennuyeuse), nous adoptons une présentation didactique où chaque élément du langage est introduit tour à tour. L'objectif de cette présentation est que le lecteur soit capable à la fin de la présentation de spécifier des comportements à l'aide de BL. Cela explique pourquoi nous nous contentons de donner une version pseudo–formelle du langage. La grammaire du langage se trouve en annexe A. Enfin, le lecteur pourra trouver des exemples de comportements dans la partie III de ce mémoire au moment de l'application de ce travail sur les interfaces Xcoop.

- On présente ensuite dans la troisième et dernière partie de ce chapitre le noyau du simulateur TIMS. En fait nous présentons plus particulièrement quelques éléments de sémantique opérationnelle. Cette présentation n'est pas une contribution de cette thèse. C'est un résumé de la thèse de Dominique Sidou [Sid97]. Il est important de présenter quelques éléments de la sémantique opérationnelle car d'une part, elle constitue une des propriétés du formalisme "idéal" et d'autre part, les simulations TIMS (et donc le simulateur) seront réutilisées dans

la partie II de ce mémoire pour le test du comportement dynamique des systèmes à objets répartis.

3.2 Principes de base de BL

On peut résumer notre approche par l'intégration de deux concepts :

- un modèle de comportement dynamique inspiré des règles actives ECA;
- une modélisation des systèmes à objets répartis uniquement aux niveaux information et traitement de RM-ODP ainsi que la description du comportement des objets de traitement par des objets d'information comme cela a déjà été présenté dans la section 2.2.4.3.

3.2.1 Les règles actives ECA

De manière générale et informelle, le comportement dynamique d'un objet se caractérise par d'une part l'ensemble des changements d'états possibles qui peuvent l'affecter et d'autre part par l'ensemble des actions potentielles auxquelles l'objet peut prendre part. Un objet étant encapsulé, ses changements d'états ne peuvent survenir qu'à la suite d'une action interne ou d'une interaction avec son environnement. Décrire le comportement d'un objet revient donc à décrire les actions qui :

- d'une part évoquent des modifications sur son état;
- d'autre part évoquent à leur tour des actions sur d'autres objets.

Pour formaliser un peu plus notre modèle de comportement dynamique, nous nous sommes inspirés du concept de règles actives qui a été développé dans le contexte des systèmes de bases de données actives (BDA) [HW92]. Il est intéressant de noter que ces règles héritent elles mêmes de concepts anciens issus des systèmes de production de règles définis dans le domaine de l'intelligence artificielle. Une base de données active est une base de données traditionnelle qui est capable de réagir, avec ou sans l'intervention de l'utilisateur, à des modifications dans l'état de la base de données ou de son environnement. Cette capacité est due à l'intégration de règles de production (les règles actives) dans la base de données. Ces règles sont déclenchées par des transitions d'état, ensuite elles évaluent des prédicats qui portent sur l'état de la base et exécutent les actions associées. Les règles de production permettent la définition d'opérations de base de données qui sont exécutées automatiquement quand certains événements ont lieu ou quand des conditions sont satisfaites.

Parmi l'ensemble des règles actives qui ont été définies par la communauté des BDA, celles dont le format correspond le mieux à notre notion de comportement sont les règles actives ECA : **E**vénements, **C**onditions, **A**ctions.

LORSQUE	< **E**vénement(s) >
SI	< **C**ondition(s) >
ALORS	< **A**ction(s) >

Tableau 3.1: Format de base des règles actives ECA

Ce format (donné par le Tableau 3.1) nous donne un cadre de base sur lequel nous allons donc définir notre formalisme de comportement. L'**E**vénement spécifie le fait qui doit se réaliser pour que le comportement soit activé. Plus précisément pour que la **C**ondition soit évaluée et l'**A**ction exécutée en conséquence, si et seulement si la **C**ondition est vérifiée.

L'avantage à établir une équivalence entre une règle active ECA et notre notion de comportement dynamique est que cela nous permet de bénéficier de toutes les études accomplies dans le premier domaine et de les adapter à notre modèle. Outre le fait que nous nous servons du modèle des règles ECA comme format de base de comportement dynamique, le gain essentiel sera de baser notre sémantique opérationnelle sur la sémantique d'exécution de ces systèmes de règles actives.

3.2.2 Modélisation des systèmes à objets répartis

Conformément à la discussion du chapitre précédent et suivant la méthodologie proposée par RM-ODP, notre modélisation fonctionnelle des systèmes à objets répartis ne comprend que des objets de traitement et des objets d'information :

- Les objets de traitement (CO : Computational Object) représentent l'unité de distribution dans notre modèle. Que cet objet soit effectivement réparti et sa répartition effective ne nous intéresse pas du point de vue fonctionnel.
- Les objets d'information représentent l'unité d'information dans notre modèle. Un objet d'information se subdivise en deux types d'objets :
  - Les objets d'information (IO : Information Object);
  - Les objets de relation (RO : Relationship Object).

  La distinction entre les deux types est simple. Lorsqu'un objet d'information veut interagir avec un autre objet d'information, il faut au préalable mettre les deux objets en relation. Cette mise en relation est matérialisée explicitement par l'objet de relation.

La Figure 3.1 donne une vue fonctionnelle d'un système à objets répartis.

3.2.2.1 La représentation du comportement dynamique

Pour la représentation du comportement dynamique de ces objets (traitement et information), voici les règles que nous adoptons :

- description du comportement au niveau traitement :
  Un objet de traitement est une composition d'objets d'information. En faisant cette hypothèse, nous pouvons proposer que le comportement d'un objet de traitement soit représenté

par des interactions entre des objets d'information. Cette hypothèse a déjà été discutée dans la section 2.2.4.3 du chapitre précédent. Une implication très forte est que tous les comportements deviennent des comportements au niveau information.

- description du comportement au niveau information :
  Si l'on reprend la définition du comportement des objets d'information donnée dans la section 2.2.4.2 du chapitre précédent, cette description s'effectue suivant trois schémas :

  – un schéma statique qui représente l'état et la structure du système.

  – un schéma d'invariant qui permet de spécifier des propriétés que le système doit toujours vérifier.

  – un schéma dynamique qui représente la transition d'un schéma statique initial à un schéma statique final tout en vérifiant les schémas d'invariant.

  L'équivalent d'une règle ECA dans notre modèle de comportement fait office du schéma dynamique défini dans RM-ODP. Dans notre modélisation des systèmes à objets répartis, c'est le conteneur d'instances *IR* (Instance Repository) qui va jouer le rôle de schéma statique. Plutôt que d'identifier des états globaux IR_{init} et IR_{fin}, c'est lorsqu'un événement (au sens règles ECA du terme) survient sur l'*IR* et si et seulement si des conditions (au sens règles ECA du terme) sont réunies sur *IR* que nous passons de IR_{init} à IR_{fin} suite à une série d'actions (au sens règles ECA du terme). Le schéma d'invariant sera quant à lui instancié plus tard dans la présentation de BL avec les assertions 3.3.2.1.

3.2.2.2 Les messages dans notre modèle

Le mode communication entre les objets étant l'envoi de messages asynchrones; ceux-ci occupent une place importante dans notre modèle de comportement.

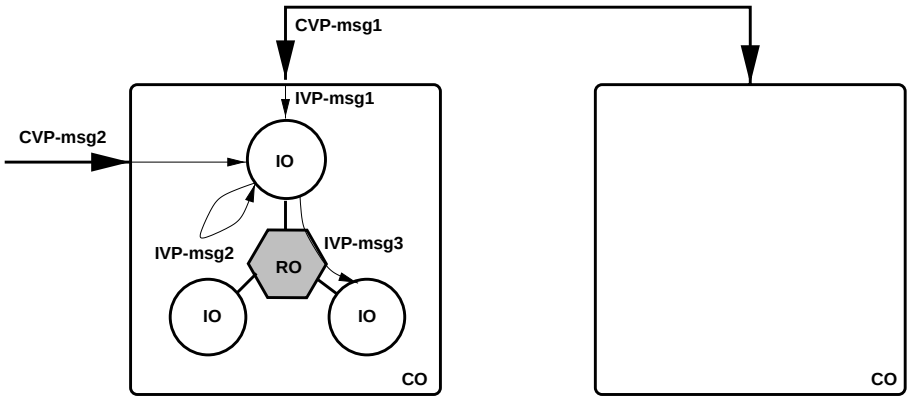

Figure 3.1: Les messages dans un système à objets répartis

a) Les messages CVP Les objets de traitement envoient des messages de traitement : les messages CVP (Computational View Point). On distingue toutefois deux types d'interaction :

- les messages qui vont d'un objet de traitement à un autre (CVP-msg1 dans la figure 3.1);
- les messages qui vont ou proviennent de l'extérieur du système (CVP-msg2 dans la figure 3.1).

Chaque message de traitement est traduit en un ou plusieurs messages d'information (que le message de traitement corresponde à une interaction avec l'environnement ou bien avec un autre objet).

Il a été choisi d'écrire des messages CVP dédiés à des technologies particulières de distribution des objets plutôt que d'écrire des messages CVP génériques. Ce choix se justifie par le fait qu'il faudra ensuite de toute manière spécialiser les messages CVP dans la technologie de distribution cible si les objets CVP sont effectivement répartis.

De plus ce choix se justifie aussi par la différence des protocoles (et des services) entre les différentes technologies de distribution. Par exemple nous avons examiné la gestion de réseaux OSI et CORBA :

- dans le cas de la gestion de réseaux OSI, on dispose de six services (create, delete, set, get, action et notification);
- alors que dans le cas de CORBA, on ne dispose que de deux services (l'envoi d'un message c2req et la réception d'un message c2rep).

Le tableau des messages CVP dédié à la gestion de réseaux OSI est donné dans la section 6.3.2.

b) Les messages IVP Les messages d'information sont appelés les messages IVP (Information View Point). On distingue trois types de message IVP :

- ceux qui résultent d'une traduction d'un message de traitement (IVP-msg1 dans la figure 3.1);
- ceux qui correspondent à des modifications internes de l'objet (IVP-msg2 dans la figure 3.1);
- ceux qui correspondent à des interactions entre objets d'informations et qui sont supportés par les RO (Relationship Object) (IVP-msg3 dans la figure 3.1).

(*ivpmsg* ⇔*create*	*inst*	*inits*)	
(*ivpmsg* ⇔*delete*	*inst*)		
(*ivpmsg* ⇔*set*	*inst*	*attr*	*val*)
(*ivpmsg* ⇔*add*	*inst*	*attr*	*val*)
(*ivpmsg* ⇔*remove*	*inst*	*attr*	*val*)
(*ivpmsg* ⇔*update*	*inst*	*attr*	*fct*)
(*ivpmsg* ⇔*establish*	*ri*	*role* ⇔*inits*	*inits*)
(*ivpmsg* ⇔*terminate*	*ri*)		
(*ivpmsg* ⇔*bind*	*ri*	*inst*	*role*)

(*ivpmsg*⇔*unbind* *ri* *inst* *role*)

Tableau 3.2: Liste des messages IVP

Le Tableau 3.2 montre comment on crée par exemple une instance d'objet d'information *inst* par l'appel de la primitive *ivpmsg*⇔*create*. La valeur initiale des attributs est donnée par la liste des attributs *inits*. De la même manière, on joint dynamiquement à une instance de la relation *ri*, une instance d'objet d'information *inst* qui remplit le rôle *role* par appel de la primitive *ivpmsg*⇔*bind*.

3.2.3 Adéquation entre les principes de base de BL et les propriétés du formalisme "idéal"

Comme nous venons de le voir, la réutilisation des règles ECA est extrêmement importante car elle nous permet de décrire nos comportements dynamiques de manière déclarative et ainsi de remplir la propriété P3 (formalisme de type déclaratif).

L'intégration des concepts de RM-ODP, quant à eux, nous garantit le respect de la propriété P6 (formalisme indépendant de toute technologie (objets)).

Le mélange des deux concepts quant à lui nous permet de remplir les propriétés P4 (formalisme adapté à la description du comportement d'un objet au niveau du point de vue information), P5 (formalisme adapté à la description du comportement d'un objet au niveau du point de vue traitement), P7 (formalisme doté d'outils de spécification des relations et des rôles).

La majeure partie des propriétés sont d'ores et déjà recouvertes ce qui justifie et motive nos choix pour les principes de base de BL. Cependant, il reste des propriétés du formalisme idéal à intégrer.

Le traitement des propriétés manquantes

Du fait que le modèle de comportement de BL et celui des règles actives ECA soient équivalents, on peut imaginer que de la même manière, nous allons pouvoir réutiliser la sémantique opérationnelle de ces règles dans notre modèle de comportement et ainsi remplir la propriété P2 (formalisme doté d'une sémantique opérationnelle).

Il faudra ensuite permettre la spécification d'assertions et leur prise en compte lors de l'exécution (dans la sémantique opérationnelle) pour ainsi remplir la propriété P8 (formalisme doté d'assertions). A noter qu'à cette occasion, notre approche pourra être qualifiée d'encore plus conforme à ODP avec sa notion de schéma d'invariant.

De la même manière, il faudra permettre la spécification du nondéterminisme et sa prise en compte lors de l'exécution (dans la sémantique opérationnelle) pour ainsi remplir la propriété P1 (formalisme permettant le nondéterminisme). On peut d'ores et déjà noter que cette prise en compte ne sera pas très délicate à réaliser étant donné la nature déclarative de notre modèle.

3.2.4 Résumé

Cette section a présenté dans sa première partie les principes de bases du formalisme BL. Ceux-ci sont basés d'une part sur la réutilisation de concepts issus des bases de données actives : les règles

actives ECA et ils sont basés d'autre part sur la réutilisation de concepts de modélisation issus de RM-ODP : modélisation au niveau du point de vue information et traitement, composition des objets de traitements par des objets d'information.

La seconde partie de cette section a ensuite consisté à examiner l'adéquation entre ces principes de bases (de BL) et les propriétés du formalisme "idéal". L'adéquation étant plutôt bonne, les principes de base vont former l'ossature du formalisme BL. De plus cet examen de l'adéquation nous instruit aussi sur quels types de mécanismes il va falloir équiper cette ossature pour que BL remplisse toutes les propriétés du formalisme "idéal".

La construction de l'ossature ainsi que l'ajout des divers mécanismes sont présentés dans la section suivante qui présente le formalisme BL.

3.3 Description du formalisme

Comme cela a été proposé dans l'introduction (afin de ne pas égarer le lecteur avec des détails de syntaxe) nous utiliserons durant cette description une représentation pseudo-formelle des comportements dans BL. De plus, pour plus de clarté, chaque comportement est au préalable donné en prose. Des exemples de comportement sont donnés dans la partie III. La grammaire du formalisme est donnée sous sa forme BNF dans l'annexe A.

3.3.1 L'ossature du langage BL

L'ossature du langage BL résulte de la traduction des règles ECA auxquelles on ajoute la notion d'étiquette.

3.3.1.1 La traduction de la règle ECA

Nous avons vu dans la section 3.2.1 que notre format de comportement est basé sur les règles ECA. Nous allons donc commencer par définir chacun des composants (Evénement, Condition et Action) dans BL. Une règle active ECA se traduit par un comportement dont le format est exprimé par le tuple caractéristique suivant :

$$< P, \quad G, \quad C >$$

où respectivement

- *P* est la **Portée** du comportement. C'est l'équivalent de la clause événement dans une règle ECA. C'est la clause qui permet de caractériser l'événement qui va activer le comportement. Cet événement est nommé le message déclencheur. Cette clause permet de définir ce message déclencheur, mais aussi de définir le contexte de déclenchement que l'on définit comme le couple (relation, rôle) dans lequel l'objet pour lequel on spécifie un comportement participe. Ce contexte de déclenchement détermine au niveau sémantique opérationnelle un BEC (Behavior Execution Context) qui sera précisé et exploité plus tard dans la section 3.4.

- *G* est la **Garde** du comportement. C'est l'équivalent de la clause condition dans une règle ECA. C'est un mécanisme classique qui permet de spécifier des conditions qui doivent être remplies afin d'autoriser l'exécution du comportement. La Garde est une expression booléenne.
- *C* est le **Corps** du comportement. C'est l'équivalent de la clause action dans une règle ECA. C'est la clause où l'on spécifie le comportement dynamique.

3.3.1.2 L'Etiquette

Le premier élément que nous ajoutons à notre modèle de comportement est une étiquette. Elle est unique et nous permet d'une part de différencier le comportement qu'elle étiquette des autres comportements dans un fichier de spécification et d'autre part d'identifier (tracer) ce comportement lors de son exécution dans une simulation.

E étant l'**Etiquette** du comportement, le tuple exprimant un comportement BL devient :

$$< E, \quad P, \quad G, \quad C >$$

3.3.1.3 Exemples de comportements simples

Nous allons commencer par présenter quelques comportements simples. Ces exemples ne sont là que pour instancier les concepts simples.

a) Un comportement vraiment simple

Comportement 1 *"comportement-simple" : Lorsque l'objet A reçoit le message O et si son état est actif, alors l'état de l'objet A devient inactif.*

La Figure 3.2 représente schématiquement ce comportement.

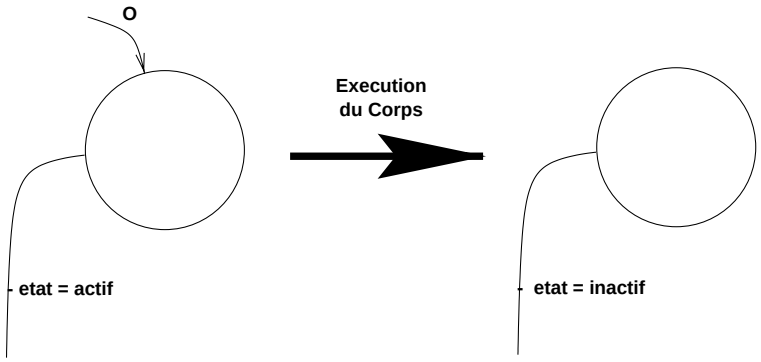

Figure 3.2: Un comportement simple

La transformation en notation pseudo-formelle nous donne :

$$
\begin{array}{rc}
< E : & \text{"comportement-simple"}, \\
P : & \text{A} \Leftarrow \text{O}, \\
G : & \text{A.etat} = \text{actif}, \\
C : & \text{A.etat} \leftarrow \text{inactif} >
\end{array}
$$

A noter que l'attribut état reflète ici l'état opérationnel de la ressource qu'il modélise et non l'état de l'objet lui même qui ne serait d'ailleurs être caractérisé par un unique attribut. A noter aussi que par abus de langage, nous dirons objet A au lieu de objet de la classe A.

Le premier comportement (certainement le plus simple possible) illustre la modélisation de l'effet d'un message sur un objet. Le résultat est une modification de l'IR (le conteneur d'instance).

b) Un second comportement simple

Comportement 2 *"second-comportement-simple" : Lorsque l'objet A participant à la relation R dans le rôle r1 reçoit le message O et si son état est inactif, alors l'objet B avec qui il est en relation (via R) et qui a le rôle r2 devient lui aussi inactif.*

La Figure 3.3 représente schématiquement ce comportement.

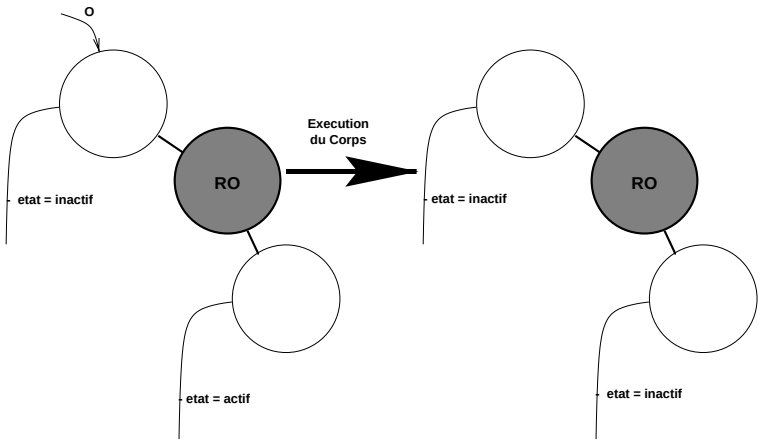

Figure 3.3: Un second comportement simple

La transformation en notation pseudo-formelle nous donne :

$$
\begin{array}{rc}
< E : & \text{"second-comportement-simple"}, \\
P : & [\text{R,r1}] \Leftarrow \text{O}, \\
G : & [\text{R,r1}].\text{etat} = \text{inactif}, \\
C : & [\text{R,r2}].\text{etat} \leftarrow \text{inactif} >
\end{array}
$$

Cet exemple bien qu'il soit toujours très simple illustre la modélisation basée sur les relations car ici il y a interaction (et en fait envoi de message) entre deux objets. Ce qu'il est intéressant de

noter, c'est que dans le notation pseudo-formelle, les objets A et B sont caractérisés par le couple (relation, rôle) dans lequel ils participent.

3.3.1.4 Comportement déclenché/non-déclenché

Nous venons de voir deux exemples de comportement déclenchés. On parle de comportement déclenché car au moment de la spécification, on connaît l'événement "déclencheur". Ce n'est pas toujours le cas. Il faut bien comprendre que tous les comportements ont des déclencheurs, mais il se peut que le comportement soit déclenchable par plusieurs messages déclencheurs différents. Afin d'éviter l'énumération de toutes ces possibilités, nous avons introduit la notion de comportements non-déclenchés. C'est un mécanisme de spécification paresseuse qui se traduit par le fait qu'il est possible de ne pas remplir la clause **Portée**. La clause **Garde** sera alors l'élément unique qui va décider de l'activation du comportement. On peut résumer en donnant l'équation suivante :

$P = nil \Leftrightarrow P = \forall messages.$

Comportement 3 *"comportement-non-declenché" : Si l'état de l'objet A est actif et si son compteur est égal à 0 alors incrémenter son compteur et rendre l'état de l'objet inactif.*

La Figure 3.4 représente schématiquement ce comportement.

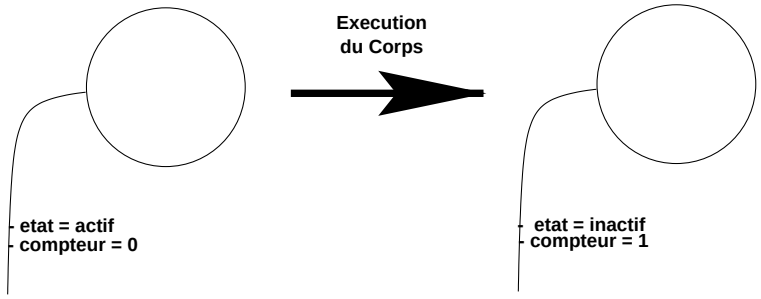

Figure 3.4: Un comportement non-déclenché

La transformation en notation pseudo-formelle nous donne :

$$\begin{array}{rc} < E : & \text{"comportement-non-declenché"}, \\ P : & nil, \\ G : & \text{A.etat} = \text{actif \&\& A.compteur} = 0, \\ C : & \text{A.etat} \leftarrow \text{inactif ; A.compteur} \leftarrow \text{A.compteur + 1} > \end{array}$$

Dans l'exemple précédent, on peut imaginer que c'est soit à la suite d'un changement d'état, soit à la suite de la remise à zéro du compteur ou bien soit tout simplement à la suite de la création de l'objet A (la **Garde** caractérisant ainsi l'état initial) que l'objet A se retrouve dans cet état. Pour éviter l'énumération de toutes ces **Portées**, ce qui alourdit la spécification, il nous a paru plus simple de définir la notion de comportement non-déclenché.

3.3.1.5 Composition de comportements:les comportements en cascade

Les comportements en cascade se produisent quand, dans la clause **Corps** d'un comportement, on exécute une action qui se révèle être l'événement déclencheur d'un autre comportement. Un exemple de ce type de comportement est donné par les comportements suivants :

Comportement 4 *"comportement-père" : Lorsque l'objet A reçoit le message O et si son état est actif, alors l'état de l'objet A devient inactif.*

Comportement 5 *"comportement-fils" : Lorsque l'objet A participant à la relation R dans le rôle r1 est inactif alors l'objet B avec qui il est en relation (via R) et qui a le rôle r2 devient lui aussi inactif.*

La Figure 3.5 représente schématiquement ce comportement.

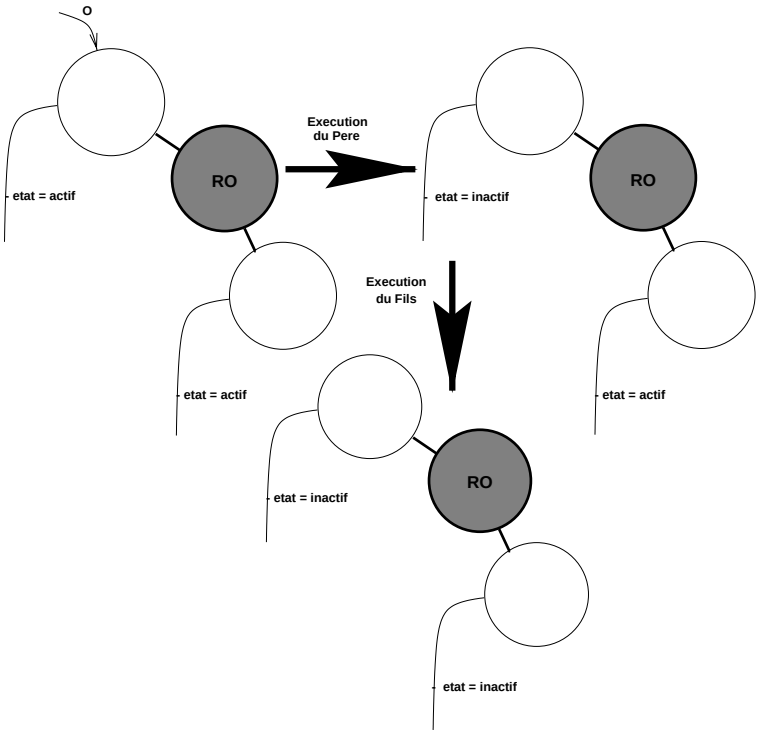

Figure 3.5: Cascade de comportements

La transformation en notation pseudo-formelle nous donne :

$< E$:	"comportement-père",	$< E$:	"comportement-fils",
P :	A $\Leftarrow$ O,	P :	*nil*,
G :	A.etat = actif,	G :	[R,r1].etat = inactif,
C :	A.etat $\leftarrow$ inactif $>$	C :	[R,r2].etat $\leftarrow$ inactif $>$

Les comportements en cascade montrent la puissance de notre formalisme lorsque l'on combine les possibilités suivantes :

- modélisation de comportement dynamique basé sur un modèle de relations :
  Ce que l'on veut modéliser dans cet exemple c'est la dépendance des états entre les objets A et B qui ont respectivement les rôles r1 et r2 de la relation R.

- utilisation des comportements non-déclenchés :
  Le comportement fils est un comportement non-déclenché. En effet dans ce cas, il n'est pas intéressant de savoir comment l'objet A devient inactif.

Ce type d'exécution plus compliquée laisse déjà entrevoir qu'il est nécessaire d'avoir une sémantique opérationnelle qui prend en compte le fait que l'exécution d'un comportement peut activer de nouveaux comportements dans un même cycle d'exécution.

3.3.2 Incorporation des assertions et traitement du nondéterminisme et de la sémantique opérationnelle

Maintenant que le lecteur est familier avec notre notion de comportement, il est temps d'équiper le formalisme BL des mécanismes nécessaires afin qu'il possède toutes les propriétés du formalisme idéal. Il s'agit des propriétés :

- P8 (formalisme doté d'assertions);

- P1 (formalisme permettant le nondéterminisme);

- P2 (formalisme doté d'une sémantique opérationnelle).

3.3.2.1 Traitement des assertions

L'idée des assertions est de donner à l'utilisateur la possibilité de spécifier des propriétés qui vont être vérifiées durant l'exécution des simulations. Dans le traitement des assertions, il faut donc dissocier deux étapes :

- la première étape est la spécification des assertions;

- la seconde étape est la prise en compte des assertions dans la sémantique opérationnelle.

a) Spécification Il est possible de spécifier deux types d'assertion :

- les préconditions sont des propriétés qui doivent être vérifiées avant l'exécution du **Corps** du comportement auquel elles sont associées;

- les postconditions sont des propriétés qui doivent être vérifiées après l'exécution du **corps** du comportement auquel elles sont associées.

La spécification des assertions est optionnelle car l'utilisateur peut ne pas avoir d'assertions à spécifier. Cependant l'utilisation des assertions, mêmes triviales, est fortement conseillée car elles se révèlent être un excellent moyen de tracer les exécutions compliquées où l'utilisateur peut avoir du mal à contrôler la dynamique du système (nondéterminisme, ajout incrémental de nouveaux comportements qui influent sur ceux déjà spécifiés de manière non prévue par l'utilisateur, ...).

Du point de vue de notre formalisme, il est donc nécessaire d'ajouter deux nouveaux champs à notre tuple de comportement : **Pre** et **Post** .

$$< E,\quad P,\quad G,\quad Pre,\quad C,\quad Post >$$

Les **Pré** et les **Post** sont des expressions booléennes.

b) Exécution On associe aux assertions (**Précondition et Postcondition**) une sémantique d'exécution très puissante car si elles devaient être violées, l'exécution serait stoppée. En cela, elles se différencient des **Gardes** qui autorisent ou pas l'exécution d'un comportement mais en aucun cas interrompent l'exécution en cours. La section qui présente la sémantique opérationnelle 3.4 montrera plus tard comment nous avons intégré les assertions.

Comportement 6 *"comportement-avec-assertions" : Si l'état de l'objet A est actif et si son compteur est égal à 0, lorsqu'il reçoit le message O alors vérifier que son état est actif et incrémenter son compteur. Vérifier que l'état de l'objet est toujours actif après l'incrémentation.*

La transformation en notation pseudo-formelle nous donne :

$$\begin{array}{rc} < E : & \text{"comportement-avec-assertions"}, \\ P : & \text{A} \Leftarrow \text{O}, \\ G : & \text{A.etat} = \text{actif \&\& A.compteur} = 0, \\ Pre : & \text{A.etat} = \text{actif}, \\ C : & \text{A.compteur} \leftarrow \text{A.compteur + 1}, \\ Post : & \text{A.etat} = \text{actif} > \end{array}$$

3.3.2.2 Traitement du nondéterminisme

De la même manière que pour le traitement des assertions, dans le traitement du nondéterminisme il faut dissocier deux étapes :

- la première étape est la spécification du nondéterminisme;
- la seconde étape est la prise en compte du nondéterminisme dans la sémantique opérationnelle.

a) Spécification Il n'y a pas d'opérateur explicite pour le parallélisme dans notre formalisme. Il est cependant possible de spécifier du nondéterminisme :

- spécification de comportements concurrents pour un objet :
  - cela se produit, par exemple, lorsque l'on a deux comportements ayant la même **Garde** et la même **Portée** par rapport à un message déclencheur donné.
  - cela se produit aussi lorsque l'objet recevant le message déclencheur a plusieurs rôles dans plusieurs relations différentes.
- envoi de messages :
  La primitive d'envoi de messages (cf section 3.3.4) permet l'envoi de plusieurs messages qui seront exécutés de manière concurrente.

b) Exécution Nous avons choisi de refléter le nondéterminisme dans les exécutions par l'entrelacement des actions concurrentes. La section qui présente la sémantique opérationnelle 3.4 montrera plus tard comment nous avons intégré cette politique d'entrelacement.

3.3.2.3 Les modes de connexion

L'exécution d'un comportement est simple lorsque celui-ci est isolé. Des problèmes se posent lorsqu'il y a plusieurs comportements à exécuter. Comme nous venons de le voir, cela se produit :

- lors des exécutions en cascade;
- lors des exécutions parallèles.

Une sémantique opérationnelle est nécessaire, mais elle doit être guidée par la personne qui spécifie les comportements par des attributs (que l'on qualifie "de synchronisation") qui doivent préciser :

- la phase de déclenchement :
  à quel moment la **Garde** est évaluée par rapport à l'occurrence de l'événement déclencheur;
- la phase d'exécution :
  à quel moment le **Corps** est exécuté par rapport à l'évaluation de la **Garde**.

Si l'on reprend l'exemple de la cascade de comportements de la section 3.3.1.5, la phase de déclenchement du comportement père doit être forcément différente de celle du comportement fils car c'est l'exécution du **Corps** du comportement père (le message qui change la valeur de l'attribut état) qui déclenche le comportement fils et non, la réception du message O comme pour le comportement père. Au moment de la réception de O, la **Garde** du comportement fils n'est pas vraie donc celui-ci n'est pas déclenchable. Sans un mécanisme en deux phases, la seconde **Garde** ne serait jamais évaluée au bon moment.

La phase d'exécution nous permet de spécifier l'exécution synchrone de comportement. Si l'on conserve toujours ce contexte de cascade défini dans la section 3.3.1.5, cela nous donne la possibilité de préciser si le comportement fils doit redonner la main au comportement père ou si celui ci peut s'exécuter de manière indépendante. Ces deux types d'exécutions sont modélisées dans la Figure 3.6.

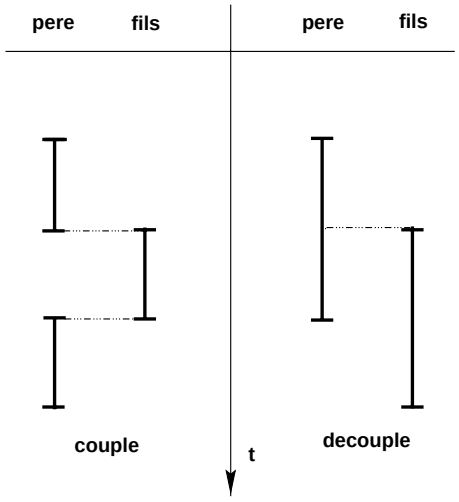

Figure 3.6: Les deux types d'exécutions

Pour la définition et l'implémentation de ces attributs de synchronisation (la phase de déclenchement et la phase d'exécution), nous nous sommes inspirés du concept de modes de connexion définis dans les règles actives ECA qui ont été définies pour les bases de données actives HIPAC [Ca89]. Ces modes de connexion permettent de spécifier :

- d'une part à quel moment la condition est évaluée par rapport à l'occurrence de l'événement (mode de connexion EC);
- et d'autre part à quel moment l'action est exécutée par rapport à l'évaluation de la condition (mode de connexion CA).

On a ainsi une équivalence d'une part du mode de connexion EC avec l'attribut phase de déclenchement et d'autre part du mode de connexion CA avec l'attribut phase d'exécution. Par analogie aux règles actives ECA, le couple d'attributs phase de déclenchement et phase d'exécution constitue les modes de connexion de BL.

Pour présenter les modes de connexion de BL, il est important d'introduire la notion de comportement "is-trigger" car c'est grâce à lui que l'on va synchroniser les exécutions de nos comportements dynamiques.

3.3.2.4 Les comportements "is-trigger" et les comportements "side-effect"

Les comportements "is-trigger" (que l'on peut traduire par "définit-le-déclencheur") sont ceux pour lesquels le comportement définit la sémantique du message qui le déclenche. C'est par exemple le cas des M-ACTION de CMIS ou des opérations CORBA pour lesquelles la sémantique de l'opération (au niveau du modèle d'information) n'est pas définie.

Ils se distinguent des comportements que l'on a présentés jusqu'à présent et que l'on peut définir de comportements "side-effect" (que l'on peut traduire par "effets de bords") qui sont des comportements qui agissent en addition du message déclencheur.

a) La phase de déclenchement Nous avons distingué deux phases :

1. la phase I :
   à chaque envoi de message, on évalue la **Garde** de tous les comportements de type phase I;

2. la phase II :
   la **Garde** des comportements de type phase II est évaluée une fois que tous les comportements "is-trigger" sont terminés (cad. ont envoyé leurs messages).

b) La phase d'exécution Nous avons distingué deux types d'exécution :

1. l'exécution de type couplé :
   Le comportement appelé est exécuté immédiatement après l'occurrence de l'événement appelant. Si l'événement provient d'un comportement (comportement père dans un contexte de cascade de comportements) l'exécution du comportement père est stoppée au profit de l'exécution du comportement fils;

2. l'exécution de type découplé :
   Le comportement est exécuté à un moment quelconque après l'occurrence de l'événement appelant. Si l'événement provient d'un comportement (comportement père dans un contexte de cascade de comportements) l'exécution du comportement père et celle du comportement fils s'effectuent en parallèle.

La Figure 3.6 illustre les deux types d'exécution : couplé et non-couplé lors de l'exécution en cascade de comportements.

c) Les notions de période et d'intervalle de validité Il est possible de spécifier des comportements "side-effect" durant trois périodes que l'on va calculer en fonction des comportements "is-trigger"

1. before-trigger :
   Lorsque les comportements "side-effect" sont potentiellement exécutables avant les comportements "is-trigger", ils sont qualifiés de "before-trigger"

2. during-trigger :
   Lorsque les comportements "side-effect" sont potentiellement exécutables pendant les comportements "is-trigger", ils sont qualifiés de "during-trigger"

3. after-trigger :
   Lorsque les comportements "side-effect" sont potentiellement exécutables après les comportements "is-trigger", ils sont qualifiés de "after-trigger"

La figure 3.7 illustre les différentes périodes de validité des comportements "side-effect" par rapport aux comportements "is-trigger".

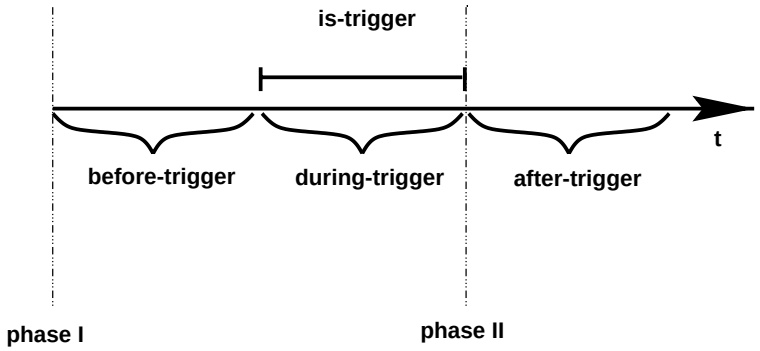

Figure 3.7: Les différentes périodes de validité

Dans le cas des exécutions de type couplé, nous avons étendu la notion de période de validité par la notion d'intervalle de validité. Un intervalle peut regrouper une, deux ou les trois périodes de validité. On peut ainsi spécifier la période de validité qui marque le début de l'intervalle de validité et la période de validité qui marque la fin de cet intervalle.

La notion d'intervalle n'a pas de sens dans le cas des exécutions découplées car il est impossible de déterminer la fin de l'intervalle. C'est pourquoi nous ne spécifions que l'une des trois périodes de validité qui marque en fait le début de l'intervalle de validité

d) Prise en compte dans le tuple de comportement BL On a donc pour les modes de connexion, la grammaire suivante :

$$\text{Mode-de-connexion} \rightarrow \text{phase-de-déclenchement phase-d-exécution}$$

Il est donc nécessaire d'ajouter un champ à notre tuple de comportement que nous nommons MC : Modes de Connexion .

$< E,\ MC,\ P,\ G,\ Pre,\ C,\ Post >$

Nous allons maintenant illustrer l'expression des modes de connexion sur l'exemple de la cascade de comportements (cf figure 3.5).

La transformation en notation pseudo-formelle nous donne :

$< E$:	"comportement-pere",	$< E$:	"comportement-fils",
MC :	phase I, coupled, after-trigger	MC :	phase II, uncoupled, during-trigger
P :	A $\Leftarrow$ O,	P :	*nil*,
G :	A.etat = actif,	G :	[R,r1].etat = inactif,
Pre :	nil,	*Pre* :	nil,
C :	A.etat $\leftarrow$ inactif,	C :	[R,r2].etat $\leftarrow$ inactif,
Post :	nil $>$	*Post* :	nil $>$

3.3.3 L'implémentation du formalisme BL

Le but de cette section est de donner quelques éléments de syntaxe sur le formalisme BL maintenant que le lecteur est familier avec notre notion de comportement. La syntaxe du formalisme se trouve en annexe A.

3.3.3.1 Scheme

Pour exprimer le **Corps** de nos comportements (qu'il faut voir en fait comme des bouts de code à exécuter), il faut se donner la puissance d'un langage de programmation classique. Il faut pouvoir déclarer des variables, utiliser des boucles, des expressions conditionnelles, les opérateurs sur les types de données classiques, etc ...

Plutôt que de définir notre propre langage, il a été décidé d'utiliser le langage Scheme [CR91]. En fait, nous avons étendu la syntaxe du langage Scheme par la syntaxe de BL. Ceci nous permet d'une part de réutiliser les environnements de programmation existants pour ce langage et d'autre part de nous reposer sur un langage standardisé. Scheme est un langage petit (la syntaxe repose sur 20 pages), simple et particulièrement propre. Par sa notation préfixe parenthésée, il ressemble fortement à LISP mais demeure nettement plus simple. Scheme est basé sur un modèle formel : le λ-calcul. De plus, il possède des fonctionnalités des langages fonctionnels modernes très attractives comme :

- le "garbage collecting";
- une syntaxe facilement extensible grâce à un mécanisme de macros de haut niveau;
- les "continuations".

Nous aurions pu choisir d'autres langages d'extension tels que TCL/TK, Perl, Python mais a contrario de Scheme aucun d'eux n'est standardisé. Du fait qu'il soit standardisé, il existe plusieurs interpréteurs Scheme (qui sont plus ou moins puissants), la plupart d'entre eux sont distribués publiquement dans des environnements de programmation. Tout comme le choix du langage, ils se sont révélés de très bons systèmes sur lesquels le simulateur TIMS a été bâti. Sans entrer dans les détails, ils possèdent des fonctionnalités très puissantes :

- la possibilité d'exécuter du code Scheme dans un environnement interprété ou bien compilé;
- celle d'interfacer du code C très facilement;

La description de l'implémentation du simulateur TIMS sera effectuée plus tard dans le chapitre 6.

3.3.3.2 L'analyse lexico-syntaxique du formalisme BL

Le fait d'étendre la syntaxe du langage Scheme par la syntaxe de BL apporte un bénéfice immédiat : l'interpréteur Scheme devient l'analyseur lexico-syntaxique de notre langage. Il ne reste ainsi plus qu'à gérer la sémantique dynamique.

3.3.4 Survol rapide de la syntaxe de BL

Ce paragraphe introduit les primitives de BL qui vont être utilisées durant la chapitre 7 de la partie III. Nous rappelons que la grammaire de BL est donnée dans l'annexe A.

3.3.4.1 Manipulation des messages

- déclaration d'un message *mess* qui comporte *n* attributs se fait par la primitive :

  (genrec:define *mess* $attr_1$ $attr_2$ ... $attr_n$**)**

- création du même message *mess* avec les valeurs des attributs correspondants :

  (`mess`**:make** val_1 val_2 ... val_n**)**

- envoi du message *mess* :

  (msgsnd *mess* **)**

- envoi en parallèle de plusieurs messages :

  (msgsnd $mess_1$ $mess_2$ ... $mess_n$**)**

- récupération de la valeur de l'attribut *attr* d'un message déclencheur dans un comportement :

  (msg-> *attr* **)**

Un message est soit associé à un objet (d'une classe donnée), soit à un objet (dans un rôle donné participant à une relation donnée) ou bien il est interne au système. On peut dans tous les cas associer un (ou plusieurs) comportement BL à un message. L'ensemble des messages se décompose en trois types :

- les messages IVP :
  Ils correspondent au tableau 3.2 de la section3.2.2;
- les messages CVP :
  Ils correspondent aux messages CMIS lors de la modélisation (comme c'est le cas dans le chapitre 7 d'applications qui utilisent ce service de communications;
- les messages internes au système (ils ne correspondent à aucun objet).

Pour la création et l'envoi de message IVP, nous avons augmenté BL de macros qui nous permettent de simplifier les spécifications. De manière générale on a pour le message IVP xxx, l'appel de ce message avec la première lettre du message en majuscule. Ce qui donne de manière générale
(msgsnd (ivpmsg-xyz:make ...)) = (Xyz ...). On obtient par exemple pour un create :
(msgsnd (ivpmsg-create:make ...)) $\rightarrow$ (Create ...).

3.3.4.2 Manipulation des relations

Comme nous l'avons vu dans le chapitre 3, dans BL,l'usage des relations est une fonctionnalité très importante. Nous avons muni le langage BL de trois fonctions qui nous permettent d'obtenir des informations sur les instances

- **(Card** *ri* $\Leftrightarrow$*inst role***)** nous permet d'obtenir la cardinalité du rôle *role* dans l'instance de relation *ri* $\Leftrightarrow$*inst* (telle qu'elle est défini dans le GRM dans le contexte du RGT et de la gestion OSI).
- **(Part** *ri* $\Leftrightarrow$*inst role***)** nous permet d'obtenir l'objet (ou la liste d'objet suivant la cardinalité) ayant le rôle *role* dans l'instance de relation *ri* $\Leftrightarrow$*inst*.
- **(Fetch-ris** *mo* $\Leftrightarrow$*inst role rel***)** nous permet d'obtenir la liste des instances de la relation *rel* dans laquelle l'objet *mo* $\Leftrightarrow$*inst* participe dans le rôle *role*.

3.3.4.3 Ossature d'un comportement BL

Un comportement BL a la syntaxe suivante :

```
BEHAVIOR -> (define-behavior behavior⇔label
            SCOPE
            (when ... )
            EXEC-RULES
            (pre ... )
```

(body . . . **)**
(post . . . **)**
)

SCOPE -> **(scope (msg** *msg*⇔*label***))**
| **(scope (ri** *rel*⇔*label***) (role** *role*⇔*label***) (msg** *msg*⇔*label***))**

EXEC-RULES -> **(exec-rules** FETCHING COUPLING**)**

FETCHING -> **(fetch-phase i)** | **(fetch-phase ii)**

COUPLING -> **(coupled** INTERVAL**)**
| **(uncoupled** PERIOD**)**

INTERVAL -> **before-trigger**
| **before-trigger during-trigger**
| **before-trigger after-trigger**
| **during-trigger**
| **during-trigger after-trigger**
| **after-trigger**
| **is-trigger**

PERIOD -> **before-trigger**
| **during-trigger**
| **after-trigger**

où

- la clause **define-behavior** est le constructeur de comportement
- *behavior*⇔*label* est l'**Etiquette**
- la clause **scope** est la **Portée**
- la clause **when** est la **Garde**
- la clause **exec-rules** définit les **Modes de Connexions**
- la clause **pre** est la **Précondition**
- la clause **body** est le **Corps**
- la clause **post** est la **Postcondition**

3.3.5 Résumé

Cette section a présenté le formalisme BL d'une manière didactique. Cette présentation s'est effectuée en deux étapes. La première étape a consisté à établir l'ossature pour la description d'un comportement BL. La seconde étape a consisté en l'ajout des mécanismes manquant à cette ossature. L'ensemble a débouché sur le tuple $< E, CM, P, G, Pre, C, Post >$ où

- E est l'**Etiquette** unique du comportement;
- la sélection des comportements potentiellement exécutables se fait suivant deux paramètres :
  1. P qui est la **Portée** et qui spécifie le message de déclenchement;
  2. G qui est la **Garde** et qui autorise l'exécution du comportement.
- l'exécution du comportement est régie suivant des **Modes de Connexions** spécifiées dans CM;
- la **Précondition** *Pre* et la **Postcondition** *Post* sont des assertions qui permettent de spécifier des propriétés qui doivent être vérifiées avant et après l'exécution de C le **Corps** du comportement qui est en fait un bout de code du langage de programmation Scheme étendu par la syntaxe de BL.

La dernière partie de cette section a quant à elle donné quelques éléments sur l'implémentation du langage BL. Des exemples sont donnés dans la partie III tandis que la grammaire du langage est donnée dans l'annexe A.

Le formalisme BL présente toutes les propriétés définies et reconnues nécessaires dans le formalisme "idéal" modulo le fait que nous puissions définir la sémantique opérationnelle de ce langage. Cette sémantique fait l'objet de la section suivante.

3.4 La sémantique opérationnelle de BL

La première partie de cette section décrit l'algorithme d'exécution des comportements : BPE (Behavior Propagation Engine). C'est ainsi que nous allons définir la sémantique opérationnelle de BL. Cette présentation fait référence à l'annexe B qui présente de manière plus formelle cette algorithme. Pour plus de détails, notamment l'algorithme lui même, le lecteur est convié à se référer à [Sid97].

La seconde partie de cette section montre quant à elle comment nous pouvons étendre les fonctions de base de BPE pour obtenir des exécutions de comportement que l'on appelle simulations comportementales. Ces simulations seront exploitées dans la partie II de ce mémoire (pour le test du comportement dynamique des systèmes à objets réparties).

3.4.1 Description de BPE et la création d'un BET

3.4.1.1 BPE

Le mécanisme de base de BPE est le chaînage avant. L'algorithme de chaînage avant correspond à la fonction WALK (définie par l'algorithme B.1.1). La fonction WALK prend en entrée une transition initiale. Il exécute cette transition (on dit aussi que l'on tire cette transition). L'exécution de cette transition génère à son tour des transitions (les transitions tirables). La fonction WALK s'arrête lorsqu'il n'y a plus de transitions à exécuter (cad lorsque la liste des transitions tirables est vide). On dit alors qu'il est à saturation.

Dans notre cas, chaque transition est un *BEN* (Behavior Execution Node) . Un *BEN* est une structure de données qui comprend principalement :

- *id* : l'identificateur de ce BEN;
- *state* : l'état de ce BEN;
- *beh* : un comportement (à exécuter);
- *bec* : un *BEC* (Behavior Execution Context) qui est un contexte de déclenchement (cad le message déclencheur qui a servi à déclencher *beh* et éventuellement le couple (relation, rôle) dans lequel l'objet affecté par *beh* participe).

Le cycle de vie d'un BEN (son exécution) est calculé par rapport à la description d'un comportement tel que nous venons de le présenter. De manière générale l'exécution de *beh* du BEN se résume en trois phases :

- l'évaluation de la précondition;
- l'évaluation du corps;
- l'évaluation de la postcondition.

Un pas du *Corps* d'un comportement *beh* d'un *BEN* représente une unité d'exécution. C'est en entrelaçant les différentes unités d'exécution des différents *BEN* potentiellement exécutables que l'on obtient une description nondéterministe du système.

3.4.1.2 création d'un BET

L'exécution de la fonction WALK nous permet de construire un arbre que l'on appelle le BET (Behavior Execution Tree : Arbre de comportement) où chacun des noeuds correspond à un BEN.

La figure 3.8 donne une vue schématique d'un arbre de comportement (BET) où chaque pas de comportement (BEN) est matérialisé.

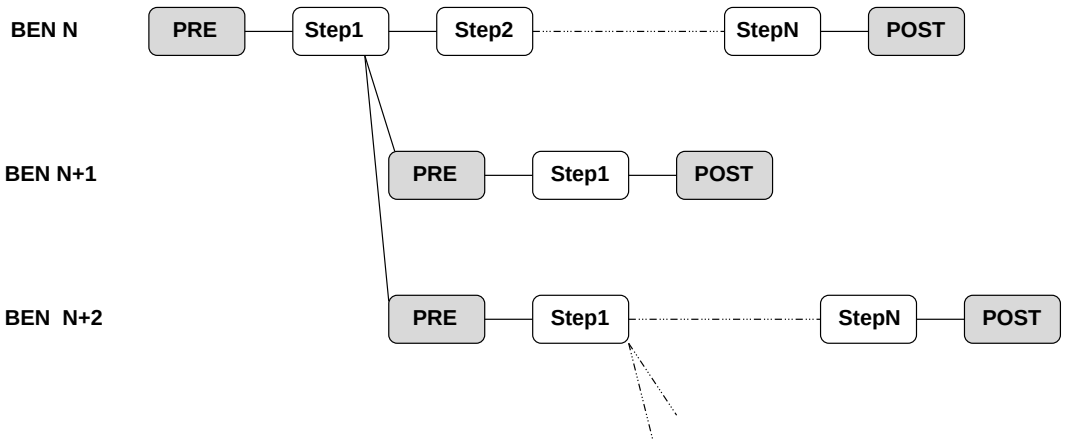

Figure 3.8: Le BET (Behavior Execution Tree)

3.4.2 La simulation dans TIMS

Une simulation exhaustive sur des modèles tels que les nôtres est impossible à cause de l'explosion combinatoire du nombre d'états possibles. Les systèmes que nous proposons de simuler peuvent comprendre plusieurs centaines de milliers d'instances pour lesquelles chacune peut avoir plusieurs attributs qui peuvent avoir potentiellement une infinité de valeurs.

Face à ce type de problème, il existe deux parades :

- la première parade consiste à simplifier le modèle. Il s'agit par exemple de décomposer le système en sous parties et/ou de contraindre l'espace des valeurs;
- la seconde parade consiste à n'explorer le système (que l'on conserve sans simplifications) qu'à travers des scénari de simulation.

C'est l'approche que nous suivons. Les scénari sont en fait des messages définis par l'utilisateur et que l'on applique à des configurations d'objets particulières.

Les simulations TIMS se déroulent de la manière suivante : on envoie un pas de scénario (un message qui constitue une transition initiale) à la fonction WALK. Lorsque celle-ci a terminé cette exécution (cad qu'il ne reste plus de transitions à tirer), on passe au pas suivant du scénario et on réitère le processus.

3.4.2.1 Les différents modes de simulation

Il existe plusieurs modes de simulations TIMS :

- le mode pas à pas : Le simulateur donne la main à l'utilisateur qui décide quel *BEN* parmi tous ceux qui sont potentiellement exécutables va être exécuté.
- le mode en profondeur d'abord : C'est une stratégie classique de parcours d'arbre. Ce mode rend la main à l'utilisateur lorsqu'il n'y a plus de pas à exécuter.

- le mode en largeur d'abord : C'est aussi une stratégie classique de parcours d'arbre. Là aussi, la main est rendue à l'utilisateur lorsqu'il n'y a plus de pas à exécuter.

- le mode aléatoire : Lorsqu'il y a plusieurs BEN à exécuter (dans une situation de nondéterminisme), le simulateur choisi aléatoirement lequel il exécute. Ce mode rend la main à l'utilisateur lorsqu'il n'y a plus de pas à exécuter.

Le mode pas à pas est en fait donné par l'algorithme B.1.1 WALK que l'on rend interactif. Les modes en profondeur d'abord, en largeur d'abord et aléatoire sont donnés par application de l'algorithme B.1.1 WALK; Seul change l'algorithme de la fonction TAKE-OUT-OF.

3.4.2.2 L'algorithme de simulation TIMS en mode exhaustif

L'algorithme B.2.1 de simulation TIMS en mode exhaustif nous permet de générer exhaustivement tous les comportements possibles du système en réponse à un stimuli (un pas de scénario) et ainsi d'avoir une vue partielle du comportement global de notre système. Sans rentrer dans les détails (notamment gestion du backtracking "UNDO"), la simulation exhaustive peut se résumer par l'algorithme B.2.1 DFS (et DFS-REC) qui est en fait une réécriture de la fonction WALK pour lequel grâce à la récursivité, on exécute toutes les combinaisons possibles. C'est une descente en profondeur d'abord.

A noter qu'il faut faire attention en l'utilisant car il est très coûteux en temps ...

3.4.3 Résumé

- La première partie de cette section a consisté en un exposé très simplifié de BPE (l'algorithme de traitement des comportements de BL). Cet algorithme donne la sémantique opérationnelle de BL. C'était la dernière propriété à établir pour faire de BL le formalisme "idéal".

- La seconde partie de cette section présente et explique le principe de simulations dans TIMS. Ces simulations sont basées sur des scénari ce qui nous évite de tomber dans le problème classique de l'explosion combinatoire du nombre d'états. Cette partie termine par la présentation de la simulation TIMS en mode exhaustif. Cet algorithme nous permet de générer exhaustivement tous les comportements possibles du système en réponse à un stimuli (un pas de scénario) et ainsi d'avoir une vue partielle du comportement global de notre système.

3.5 Conclusion

La première section de ce chapitre a présenté les principes de bases de BL (les règles actives ECA et la réutilisation de concepts de modélisation issus de RM-ODP) et a consisté à examiner l'adéquation entre ces principes de bases (de BL) et les propriétés du formalisme "idéal".

Cette section a présenté le formalisme BL d'une manière didactique. Cette présentation s'est effectuée en deux étapes. La première étape a consisté à établir l'ossature pour la description d'un comportement BL (à partir en fait de la traduction directe d'une règle active ECA). La seconde

étape a consisté en l'ajout des mécanismes manquants à cette ossature afin de remplir toutes les propriétés du formalisme "idéal".

La dernière partie de cette section a, quant à elle, donné quelques éléments sur l'implémentation du langage BL. Cette implémentation est largement basée sur l'utilisation du langage Scheme.

La troisième et dernière section de ce chapitre a consisté en un exposé très simplifié de BPE (l'algorithme qui donne la sémantique opérationnelle de BL). Cette présentation a ensuite enchaîné sur le principe de simulations dans TIMS et notamment sur l'algorithme de simulation TIMS en mode exhaustif. Cet algorithme qui conclut la partie I est fondamental pour la suite de ce mémoire car il est à la base de notre approche pour la génération de cas de tests du comportement dynamique des systèmes à objets répartis qui constitue le thème principal de la partie suivante : la partie II.

Partie II

Génération de tests du comportement dynamique des sytèmes à objets répartis

Chapitre 4

La génération de tests du comportement dynamique des systèmes à objets répartis

4.1 Introduction

Depuis plus de dix ans, la génération automatique de tests est perçue comme l'une des applications les plus rentables des techniques de spécifications formelles. Vu l'importance en terme de temps et de coût des phases de test, les chercheurs et praticiens du test ont cherché depuis plusieurs années à automatiser cette phase. Ainsi, de nombreuses méthodes de génération automatique de test de conformité ont vu le jour issues du test de circuit (voir par exemple les synthèses [LY96, Cav96, RC96]). Malheureusement, peu d'entre elles sont utilisées en pratique [LL95] et les tests sont encore écrits à la main à partir de spécifications informelles. Cette situation est particulièrement vraie dans le contexte des systèmes à objets répartis qui forment le cadre de travail de cette thèse.

Le but de cette partie (de ce chapitre et du chapitre suivant) va donc être de montrer que la génération de tests du comportement dynamique des systèmes à objets répartis est possible à partir de spécifications formelles adaptées et d'outils de génération de tests existants.

Ce chapitre se décompose en trois parties :

- la première partie de ce chapitre constitue une introduction à la problématique de la partie II de ce mémoire : la génération de tests du comportement dynamique des systèmes à objets répartis. Cette introduction s'effectue en deux temps :
  - dans un premier temps, nous présentons l'activité de test (et de manière plus précise la génération de tests) dans le contexte général des logiciels. Cette présentation vise simplement à donner une vue d'ensemble des principales techniques de génération de tests;
  - le problème de la répartition n'étant pas pris en compte dans la présentation précédente nous présentons, dans un second temps, l'activité de test dans le contexte des logiciels répartis. Cette activité est présentée par l'intermédiaire de la norme ISO/9646 qui constitue le standard de référence pour le test de conformité des logiciels répartis. A la fin de la présentation de ISO/9646, nous présenterons enfin le test de conformité dans le cadre de RM-ODP.

- la seconde partie de ce chapitre adopte en fait le même type de présentation que celui adopté dans la partie I. Elle débute par un rappel de nos objectifs et fixe des hypothèses de travail. La discussion de ces hypothèses nous permet de dégager un ensemble de propriétés qui caractérise l'approche à adopter pour la génération de tests du comportement dynamique des systèmes à objets répartis;

- enfin la troisième et dernière partie de ce chapitre explore les différentes techniques de génération de cas de tests et examine leur adéquation avec l'ensemble des propriétés identifiées dans la seconde partie de ce chapitre.

4.2 La génération de tests des logiciels

Le but de cette section d'introduction est d'une part de situer la génération de tests dans le contexte de l'activité de test et d'autre part d'examiner plus particulièrement la génération de tests des logiciels. Un effort particulier est fait lors de l'exposé des techniques de génération de tests fonctionnels car c'est le test le plus utilisé pour les logiciels répartis.

4.2.1 La génération de tests

D'une manière générale, tester un logiciel consiste à l'exécuter en ayant la totale maîtrise des données qui lui sont fournies en entrée tout en vérifiant que son comportement est celui attendu. Cette définition met en évidence quatre tâches distinctes nécessaires à l'activité de test :

- l'établissement des critères de test;

- l'identification des composants à tester;

- la constitution d'un ensemble de données qui seront fournies en entrée aux composants (préalablement identifiés) à tester (cad des jeux de tests) suivant les critères (préalablement identifiés);

- l'observation de l'exécution de ces composants, qui doit permettre l'identification des défaillances.

Il faut noter que la définition de l'activité de test que nous venons de donner n'inclut pas les activités nécessaires à l'élimination des défauts, à savoir leur localisation et leur correction. Ces deux activités appartiennent à la phase dîte de "debugging".

La génération de test peut se décomposer en deux phases : (i) l'établissement des critères de test et (ii) la construction proprement dite de la suite de tests.

4.2.1.1 La phase (i) : établissement des critères de sélection, d'adéquation et d'arrêt

Etant donné que la prise en compte de la totalité du domaine d'entrée du logiciel (test exhaustif) est presque toujours impossible, un ensemble de jeux de test doit être un sous-domaine de taille

réaliste. Idéalement, un critère doit définir un tel sous-domaine dont le pouvoir de révélation de défauts doit être quasi identique à celui du test exhaustif.

La construction d'un ensemble de jeux de tests conforme à un critère peut être effectuée suivant deux approches;

- la première consiste en la sélection à l'avance des éléments de cet ensemble suivant le critère qui est dans ce cas appelé critère de sélection;
- la seconde approche est de considérer un jeu de tests quelconque et d'évaluer à posteriori sa conformité au critère qui est dans ce cas appelé critère d'adéquation.

Concrètement, cette dernière approche revient à tester le logiciel avec des données quelconques en entrée jusqu'à ce que le critère soit satisfait (pour cette raison, on parle parfois de critère d'arrêt).

Le test aléatoire est souvent assimilé au critère le plus élémentaire puisque les jeux de tests sont sélectionnés au hasard. Dans les autres cas, un critère de sélection définit généralement une décomposition du domaine d'entrée du logiciel en un nombre fini de sous-domaines. Bien que ces sous-domaines ne soient pas toujours disjoints, on parle souvent de "partition" du domaine des entrées. Le pouvoir de détection de défauts des éléments d'un même sous-domaine est considéré équivalent, ce qui signifie qu'un seul élément du sous-domaine met en évidence tous les défauts que peut détecter le sous-domaine entier. Plusieurs critères de cette nature ont été proposés dans la littérature pendant ces vingt dernières années. Il faut cependant noter que l'efficacité des critères fondés sur une partition du domaine d'entrée est contestée car, dans de nombreux cas, le nombre de défauts détectés ne dépasse pas celui obtenu en testant le logiciel de manière purement aléatoire.

Evaluation théorique des critères de sélection Lors d'une des premières tentatives pour établir une théorie sur le test des logiciels, [GG75] se sont intéressés aux propriétés que doivent posséder les critères de sélection. Deux propriétés se sont dégagées :

- un critère est fiable si tous les jeux de tests conformes à ce critère détectent les mêmes défauts;
- un critère est valide si tout défaut du logiciel peut être détecté par au moins un jeu de test conforme au critère.

En pratique, nous n'avons aucun moyen de montrer qu'un critère de sélection est fiable ou valide ou de concevoir un critère possédant ces deux qualités. En pratique, un méthode intéressante pour l'évaluation de la fiabilité est la technique des mutants qui est une technique de test particulière qui sera présentée plus tard dans le paragraphe 4.2.2.2.

Il faut noter que des résultats théoriques fondamentaux sont issus de la recherche théorique :

- le problème de l'équivalence de deux programmes est non décidable. Une conséquence de ce résultat est que la sélection d'un jeu de tests montrant la correction d'un programme est également un problème non décidable;
- la sélection d'un jeu de tests activant une partie donnée du programme est un problème non décidable.

La conséquence de ces résultats est qu'il est donc nécessaire de simplifier la problématique. Par exemple dans plusieurs cas particuliers, ces problèmes deviennent décidables. En d'autres termes cela veut dire qu'autant le développement de techniques universelles est impossible autant plusieurs techniques adaptées à des cas particuliers (on parle souvent d'heuristiques) ont donné des résultats.

4.2.1.2 La phase (ii) : les techniques de génération de tests

Depuis ces vingt dernières années, le nombre des travaux sur la génération de tests n'a jamais cessé d'augmenter. Il est donc impossible de référencer la quantité de techniques de génération de tests mises au point. Il y a cependant deux paramètres qui nous permettent de les classifier : la granularité du test et le modèle du programme.

a) La granularité du test Le test peut être qualifié d'unitaire, d'intégration ou de système :

- le test unitaire concerne chacun des composants qui constituent le logiciel en le traitant de manière isolée. Le but de ce type de test est la recherche de défauts dans la réalisation de ce composant en ne considérant que la spécification de ce composant;
- le test d'intégration tente de mettre en évidence les anomalies dans les appels entre les composants ou dans la répartition des tâches entre composants pour traiter des fonctionnalités de plus haut niveau;
- le test système porte sur l'ensemble du logiciel et vise à identifier les cas où sa spécification initiale n'est pas respectée.

b) Le modèle du programme Il existe deux types de techniques de génération de tests :

- Les techniques structurelles (de boîtes blanches) :
  Elles peuvent être appliquées dès lors que la spécification et l'implémentation sont deux structures entièrement et librement explorables. Elles ont pour objectif de mettre en évidence des défauts d'implémentation.
- Les techniques fonctionnelles (de boîtes noires) :
  Elles se substituent aux techniques structurelles lorsque l'implémentation n'est accessible que via ses points d'interaction avec l'extérieur appelés ports de communication. Elles ont pour objectif de mettre en évidence l'existence de défauts dont l'origine est la mauvaise compréhension de la spécification du logiciel. La difficulté essentielle dans leur conception est la définition de critères permettant la sélection judicieuse de données d'entrée à partir de la spécification.

Il apparaît que ces deux méthodes sont complémentaires. C'est pourquoi on parle aussi de techniques hybrides (ou de boîtes grises). Les sections suivantes présentent tour à tour les techniques structurelles et les techniques fonctionnelles.

4.2.2 Les techniques de test structurel

Le test structurel est certainement l'approche la plus largement utilisée en milieu industriel. Il comprend des techniques définies sur la structure du contrôle du programme qui est le plus souvent représentée par un graphe de contrôle. Un second type de technique est la technique de mutation qui est une technique basée sur l'injection de fautes. Cette technique est, elle aussi, très souvent étudiée. Nous allons dans la suite de cette section présenter ces deux techniques.

4.2.2.1 Les techniques basées sur la construction du graphe de contrôle

a) Définition formelle des graphes de contrôle

Définition 3 *Un graphe de contrôle est un graphe orienté* $C = (N, A, e, s)$ *tel que :*

- *N est un ensemble de noeuds;*
- *A est une relation binaire sur N, appelé ensemble d'arcs;*
- $e \in N$ *et* $s \in N$ *sont les noeuds d'entrée et de sortie du graphe.*

Un noeud de N correspond à une instruction de programme autre qu'une instruction conditionnelle ou bien à l'évaluation de la condition d'une instruction conditionnelle ou d'une boucle.

Un arc $(n_i, n_j \in A)$ représente un transfert du contrôle de n_i à n_j. Un arc sera appelé branche si n_i est une condition. A chaque branche du graphe de contrôle, on peut associer un prédicat, appelé prédicat de branche, qui décrit les conditions requises pour que la branche soit traversée.

Un chemin du graphe de contrôle est une séquence $(n_1, \ldots, n_k)$ de noeuds de N tels que $n_1 = e, n_k = s$ et pour tout $i, 1 \leq i \leq k, (n_i, n_{i+1}) \in A$. Une variable d'entrée est une variable du programme apparaissant dans une instruction de lecture. Un chemin est exécutable s'il existe une valeur des variables d'entrée du programme pour lesquelles le chemin est traversé pendant l'exécution associée. S'il n'existe aucune valeur, le chemin est dit impossible.

b) Les critères de sélection Il existe différents critères de sélection :

- le critère le plus élémentaire est la couverture des instructions (ou couverture des noeuds) qui consiste à exécuter le logiciel de sorte que tous les noeuds de son graphe de contrôle soient exécutés au moins une fois;
- le critère le plus largement utilisé est la couverture des branches. Elle consiste à exécuter toutes les branches du graphe de contrôle ce qui revient à exécuter le logiciel de telle sorte que toutes les conditions figurant dans les instructions de branchement conditionnel soient évaluées au moins une fois à vrai et au moins une fois à faux;
- la couverture des chemins est satisfaite si chaque chemin du graphe de contrôle est exécuté au moins une fois.

En présence de boucles, le nombre de chemins est potentiellement infini. En conséquence la couverture des chemins est théoriquement impossible à atteindre. Pour cette raison, plusieurs autres critères ont été définis dans le but de combler le vide entre la couverture des branches et celle des chemins.

- La couverture des chemins intérieur-extérieur consiste à sélectionner pour chaque boucle un chemin tel que la condition d'entrée soit fausse (extérieur) et au moins un chemin tel que la condition d'entrée soit vraie (intérieur);
- La couverture des chemins de longueur *k* consiste à sélectionner tous les chemins provoquant au plus *k* itérations des boucles;
- La couverture des LCSAJ (Linear Code Sequence And Jump) est fondée sur la définition de portions de code comportant une séquence d'instructions consécutives dont la première est soit l'instruction d'entrée du programme, soit la destination d'une instruction de branchement et la dernière est soit une instruction de branchement, soit l'instruction de sortie du programme.

D'autres critères basés sur l'analyse du flux de données existent mais ne sont pas présentés dans cette introduction. Dans le cas ou le lecteur s'intéresse à ce type de critère, il peut se référer à [RW85, CPRZ89].

c) La génération de tests à partir d'un graphe de contrôle La génération de tests satisfaisant un critère peut se faire de deux manières différentes :

- la première consiste à construire à l'avance les jeux de tests en s'assurant qu'ils satisfont le critère de sélection retenu. On qualifie souvent cette méthode de "déterministe". On distingue deux étapes :
  - lors de la première étape, il faut sélectionner un ensemble (si possible minimal) de chemins satisfaisant le critère. La difficulté de cette tâche vient de l'existence potentielle de chemins impossibles parmi ceux sélectionnés de manière statique. Ce problème ne possède pas de solution car le calcul d'une entrée exécutant un chemin est un problème non décidable dans le cas général. Ainsi, on se contente d'utiliser des heuristiques qui tentent de minimiser le nombre de chemins impossibles [BM94].
  - la deuxième étape consiste à calculer pour chacun des chemins sélectionnés un jeu de tests l'exécutant. Il est en général non décidable de déterminer de manière statique les données d'entrée provoquant l'exécution d'un chemin donné. Même dans les cas particuliers où ce calcul devient décidable, il est nécessaire de procéder à une évaluation symbolique du programme dont le coût devient vite prohibitif au fur et à mesure que la complexité du graphe de contrôle augmente.
- la seconde consiste à exécuter le logiciel avec des jeux de tests aléatoires et à s'arrêter dès que le critère d'arrêt est satisfait. De cette manière, nous ne sommes plus confrontés aux problèmes de complexité de l'évaluation symbolique nécessaires au calcul du jeux de test.

Un cas intéressant de génération de ce type est le test statistique : la distribution des entrées est déterminée par une analyse de la structure du programme.

La seconde technique que nous allons présenter est la technique des mutants qui est une technique basée sur l'injection de fautes.

4.2.2.2 La technique de mutation

En ce qui concerne le test du logiciel, la seule technique d'injection de fautes qui existe aujourd'hui est l'injection de mutants [Dar96]. L'injection de mutants sert dans deux contextes :

- lorsque l'injection de mutants est utilisée, à priori, pour construire des jeux de tests, cette technique s'appelle le test de mutation;
- lorsque l'injection de mutants est utilisée, à posteriori, pour valider des jeux de tests, elle est dénommée analyse de mutation.

a) La génération des mutants Définie initialement par [DLS78], le test/l'analyse de mutation consiste dans un premier temps à créer, à partir d'un programme original P, un ensemble de programmes P_i, appelés mutants, qui diffèrent de P par une et une seule modification élémentaire syntaxiquement correcte introduite dans le code source de P. Cette modification est appelée mutation. On peut distinguer plusieurs catégories de mutation :

- remplacement d'un opérateur par un autre opérateur
- remplacement d'une constante par une autre constante
- remplacement d'un symbole (nom d'une variable ou d'un tableau) par un autre symbole, etc ...

Partant d'un programme P, des mutants associés et d'un jeu de tests T, l'analyse de mutation consiste dans un second temps à évaluer la proportion des mutations révélées par T. Une mutation m_i est révélée par T si le mutant correspondant P_i fournit au moins une valeur de sortie différente de celle produite par le programme original P en réponse au jeu T : le mutant P_i est alors dit tué par T. Dans le cas contraire, c'est à dire lorsqu'il fournit les mêmes sorties que le programme original P lors de l'exécution de T, P_i est dit vivant et deux cas sont alors possibles :

- soit le jeu de tests T n'est pas assez sensible pour révéler la mutation m_i
- soit le mutant P_i est équivalent au programme P, c'est à dire qu'il n'existe aucune entrée de test permettant de les distinguer.

b) Le score de mutation La mesure permettant de mesurer l'efficacité d'un jeu de tests à révéler des fautes de type de mutation est le score de mutation. Pour un programme P et un jeu de tests T, elle est notée $sm(P,T)$. Elle correspond à la proportion des mutants non équivalents à P qui sont tués par T. Un score de 100% indique que le jeu T tue tous les mutants non équivalents.

Afin de pallier le problème du nombre restreint de classes de fautes à injecter, l'analyse de mutation repose sur deux hypothèses :

- l'hypothèse du programmeur compétent :
  "Le programme testé a été écrit par un programmeur compétent. Par conséquent, si le programme est incorrect, il ne diffère du programme correct que par quelques fautes élémentaires."

- l'hypothèse dite d'effet de couplage :
  "les fautes complexes sont couplées à des fautes simples. En d'autres termes, des entrées de tests qui tuent tous les programmes qui diffèrent du programme original par seulement une faute simple, sont suffisamment sensibles pour révéler des fautes plus complexes."

Sous ces hypothèses, le score de mutation peut être considéré comme une mesure représentative de l'efficacité d'un jeu de test à révéler des fautes.

c) Les critiques L'analyse de mutation a suscité de nombreuses critiques en raison de son coût et du problème de la représentativité des fautes injectées par rapport aux fautes réelles (cad de la validité des deux hypothèses sous-jacentes citées précédemment). Cela a conduit à remettre en cause la validité du score de mutation en tant que mesure de l'efficacité d'un jeu de test. L'analyse de mutation n'a pourtant pas été abandonnée en tant qu'outil expérimental et a donné lieu à d'autres types d'utilisation que le lecteur peut trouver dans [Dar96].

4.2.3 Les techniques de test fonctionnel

Le test fonctionnel [Bei95] est généralement fondé sur l'analyse des spécifications, formelles ou informelles du logiciel. Le cas des spécifications informelles n'est pas très intéressant car il conduit à des méthodes qui ne sont pas (ou très peu) automatisables. Par contre l'existence de spécifications formelles permet de rendre partiellement ou même totalement (dans le meilleur des cas) automatique cette génération. Le test fonctionnel à partir de spécifications formelles est certainement l'approche la plus largement utilisée dans le milieu des systèmes répartis.

Comme nous l'avons déjà vu dans la section 2.4 du chapitre 2, deux types de spécifications formelles ont été largement exploités :

- les spécifications axiomatiques;

- les spécifications basées sur un modèle de transitions d'états. Ces spécifications représentent leur comportement dynamique par des automates d'états finis ou des systèmes de transitions étiquetés. C'est le cas classique de test des systèmes répartis (test des protocoles) et des spécifications comme SDL, Estelle, LOTOS.

Nous allons différer cette présentation des techniques de test fonctionnel à la section 4.5 car à la suite de l'énoncé des propriétés qui sera effectué dans la section 4.4.2, nous verrons qu'il est inutile de parcourir l'ensemble des techniques associées à ces deux techniques mais que l'examen des techniques associées à des méthodes formelles basées sur un modèle de transitions d'états est suffisant.

4.2.4 Résumé

La génération de tests s'effectue en deux étapes : l'établissement des critères de sélection et la construction des cas de test en fonction de ces critères. On peut classer les différents types de construction suivant deux paramètres :

- la granularité du test qui est soit unitaire, soit d'intégration ou soit système;
- le modèle de test qui est soit de boite blanche, soit de boite noire selon que l'on a ou pas accès au code source du programme.

Cependant la répartition n'est pas prise en compte dans cette introduction. Bien qu'elle n'influe pas vraiment sur les techniques de génération, c'est un facteur très important de l'architecture de test. La prochaine section va présenter la série de standard ISO/9646 qui traite du test de conformité des logiciels répartis.

4.3 La méthodologie OSI de test de conformité

Cette présentation de la méthodologie OSI de test de conformité est importante pour la suite de ce travail car elle définit le cadre architectural de notre travail et fixe la terminologie qui va être employée durant la suite de ce mémoire.

Pour définir une méthodologie générale et un cadre pour le test de conformité des produits, l'ISO et le CCITT ont défini dans la norme ISO/9646 [ISO92b] le contexte et les concepts importants du test en précisant comment les tests peuvent être développés pour un protocole donné et en donnant des directives pour l'exécution de ces tests.

4.3.1 La norme ISO/9646

La norme ISO/9646 a été définie pour des protocoles spécifiés en langage naturel. En fait au départ, elle a été développée pour les protocoles OSI, mais elle est utilisée pour tester tous les types de protocoles (ISDN, GSM). On la retrouve bien entendu dans le domaine des systèmes à objets répartis et plus particulièrement dans la gestion de réseaux OSI [Kab95, Bae93].

Cette norme comprend cinq parties :

- la partie 1 [IT92a] est une introduction et traite des concepts généraux;
- la partie 2 [IT92b] décrit le processus de spécification des suites de tests;
- la partie 3 [IT92c] définit la notation TTCN (Tree and Tabular Combined Notation);

- la partie 4 [IT92d] traite de la réalisation du test;
- la partie 5 [IT92e] définit le rôle du laboratoire de test et celui du client dans l'évaluation de la conformité.

4.3.1.1 Définition de la conformité

La norme [IT92a] définit la terminologie et donne une description complète des concepts de base. Une définition informelle de la conformité est :

Définition 4 *Un système est conforme s'il respecte toutes les contraintes de conformité de la norme correspondante*

Les contraintes de conformité sont explicitement indiquées dans la norme du protocole. Elles indiquent ce qu'une implantation conforme doit faire et ce qu'elle ne doit pas faire.

Le plus souvent, une norme de protocole ne spécifie pas un protocole unique, mais une classe de protocoles. La plupart des normes comportent un certain nombre d'options qui peuvent être ou non réalisées dans une implantation particulière du protocole. Toutes les options qui ont été réalisées dans une implantation donnée sont listées dans les (P)ICS ((Protocol) Implementation Conformance Statement). Des restrictions dans la sélection sont données par l'intermédiaire de deux types de contraintes : les contraintes statiques de conformité et les contraintes dynamiques de conformité.

- les contraintes statiques définissent les capacités minimales qu'une implémentation doit avoir (les combinaisons possibles des options);
- les contraintes dynamiques (la plus grande et la plus importante part de la norme) définissent le comportement observable de l'implémentation lors de la communication avec son environnement : l'ordre des événements, le codage des informations dans les PDU (Protocol Data Unit) et les relations entre les PDU.

En plus du (P)ICS, deux autres documents sont fournis par le programmeur : le SCS (System Conformance Statement) indique les options et les capacités du protocole réalisées dans l'implémentation considérée et le (P)IXIT ((Protocol) Implentation eXtra Information for Testing) donne des informations complémentaires (valeur de temporisateur, nombre maximum de connexions supportées, ...).

L'ensemble des tests nécessaires pour vérifier toutes ces contraintes est appelé **suite** ou **séquences de tests**. Un test de cette suite est appelé **cas de test**. Le cas de test est composé de plusieurs tests élémentaires appelés **pas de test**. Chaque cas de test a pour objectif de vérifier une contrainte particulière de conformité.

4.3.1.2 La spécification des suites de tests

La norme [IT92b] distingue plusieurs phases dans le processus de spécification des suites de tests :

1. la première phase consiste à spécifier une suite abstraite de tests pour le protocole considéré. Cette phase est connue sous le nom de génération de tests. Cette suite est abstraite dans le sens où elle a été développée indépendamment de toute implémentation. La génération de tests consiste à dériver systématiquement des cas de test à partir d'une spécification du protocole. Le but est de développer une suite de tests abstraite spécifiée dans une notation bien définie et qui teste tous les aspects de la spécification. Les cas de tests sont dérivés des contraintes dynamiques de conformité en plusieurs étapes :

   (a) premièrement, un ou plusieurs objectifs de test sont définis pour chaque contrainte. L'objectif de test donne une description précise de ce qui doit être testé pour décider de la satisfaction d'une contrainte particulière de conformité.

   (b) deuxièmement, un cas de test générique est généré pour chaque objectif de test. Le cas de test indique la succession d'actions à effectuer pour atteindre l'objectif de test, sans tenir compte de la méthode de test abstraite qui sera utilisée ni de l'environnement dans lequel le test sera réalisé.

   (c) la dernière étape consiste à dériver un cas de test abstrait pour chaque cas de test générique, en prenant en compte la méthode de test utilisée et les restrictions imposées par l'environnement de test.

2. la seconde phase consiste à déterminer une suite particulière de tests, exécutable pour une implantation donnée. Ceci s'appelle sélection ou implantation de tests. C'est à ce moment que les particularités de l'implémentation sont prises en compte.

3. la dernière phase est l'exécution du test. La suite de tests sélectionnée est exécutée sur l'implémentation par un testeur réel et les réponses correspondantes sont observées. Ceci conduit à un verdict sur la conformité de l'implémentation par rapport à la spécification. Les rapports du test sont consignés dans le rapport de test de conformité du protocole (PCTR Protocol Conformance test Report).

4.3.1.3 Les méthodes abstraites de test

La norme [IT92d] traite de la réalisation des méthodes abstraites de tests ou architectures de tests. Une norme de protocole spécifie le comportement d'une entité de protocole à ses points d'accès. L'idéal serait de tester l'entité à ces points. Malheureusement ces points d'accès ne sont pas toujours accessibles directement par le testeur. Les points où le testeur observe et contrôle l'IUT (Implementation Under Test, l'implémentation sous test) sont appelés point d'observation et de contrôle (PCO) .

Une méthode abstraite de test ou architecture de test définit un modèle d'accès à l'implémentation par les PCO. Nous pouvons distinguer les cas suivants :

- si un des SAP (Service Access Point) n'est pas accessible, il n'y a pas de PCO pour ce SAP;
- l'existence d'autres couches de protocole entre le PCO et le SAP, et le type de messages échangés entre eux;

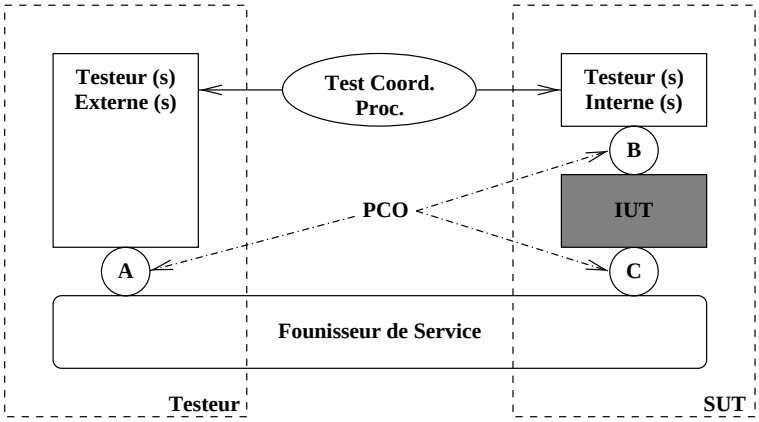

Figure 4.1: Architecture de test générique

- la localisation des PCO;
- le fonctionnement interne du testeur en terme de répartition des fonctions de test entre les différents éléments de test et leur règles de coordination.

En variant ces aspects, différentes méthodes de test peuvent être obtenues. Certaines ont été définies dans la norme pour être utilisées avec des suites abstraites de tests normalisées. Dans le contexte de la gestion OSI, ces méthodes abstraites ont été étudiées dans [Kab95, EWO95].

4.3.1.4 La notation de test TTCN

La norme [IT92c] définit le langage TTCN (Tree and Tabular Combined Notation) qui est une notation indépendante de toute implantation de description des suites de test. TTCN permet de décrire de manière précise le comportement attendu de l'implémentation et les séquences d'événements à envoyer à l'implémentation (cad les séquences d'entrées et de sorties qui surviennent au niveau des PCO). TTCN a deux formes : une forme graphique (TTCN-GR) facilement lisible par l'homme et une forme exécutable (TTCN-MP). Une suite de tests TTCN comprend quatre parties :

- une partie "Test Suite Overview" (survol de la suite de tests) qui contient les informations nécessaires à la présentation et à la compréhension de la suite de tests et une brève description de l'objectif global de la suite de tests.
- une partie "declarations" (déclarations) qui donne la liste et la définition de l'ensemble des composants utilisés dans toutes les parties de la description de la suite de tests: les PCO, les timers, les constantes, variables et types ASN.1.
- une partie "constraints" (contraintes) qui spécifie les valeurs des paramètres des PDU (Protocol Data Unit) et des ASP (Abstract Service Primitive) qui seront échangées entre le testeur et l'implémentation.

- une partie "behaviour" (comportement) composée d'un ensemble structuré de fiches. Chaque fiche décrit le déroulement d'un cas de test vu du testeur (envoi et réception de message). La description du comportement dynamique de la suite de tests inclut la description de PDU et des ASP utilisés dans les messages à envoyer ou à recevoir en se référant à la partie contrainte.

4.3.1.5 La partie comportement de TTCN

TTCN décrit la partie comportement sous forme d'arbre d'événements. Les événements sont définis par la notation donnée par le Tableau 4.1.

Event	⇔>	Send		
	|	Receive		
Send	⇔>	PCOIdentifier	!	ASPIdentifier
Recieve	⇔>	PCOIdentifier	?	ASPIdentifier

Tableau 4.1: Les événements en TTCN

Un envoi de message est précédé par un ! et une réception par un ?. Chaque fiche est structurée de la manière suivante :

- "Reference" définit le nom complet du cas de test et indique sa place dans la suite de tests;
- "Identifier" associe un indentificateur au cas du test;
- "Purpose" décrit l'objectif du cas de test;
- "Defaults Reference" indique la description d'un comportement par défaut s'il existe;
- la colonne "Behaviour Description" décrit le comportement de l'implémentation. Les événements sont décrits en ligne de deux manières : comme des événements en séquence ou comme des événements alternatifs (cf figure 4.2);
- la colonne "label" permet d'étiqueter chaque événement pour permettre des références ultérieures en particulier des instructions de branchement;
- la colonne "Constraints reference" indique des valeurs spécifiques pour certains PDU ou ASP;
- une séquence d'actions termine par la spécification du verdit correspondant à l'exécution de la séquence pour chaque alternative. La colonne "verdict" indique ce verdict. Le verdict est soit PASS (signifie le succès de la conformité), soit FAIL (signifie l'échec de la conformité) ou soit INCONCLUSIVE (cad ne se prononce pas - ni succès ni échec);
- la colonne "Comments" fournit des remarques pour aider à la compréhension du test;
- "Extended Comments" fournit des commentaires généraux.

événements en séquence	événements alternatifs
$PCO1!\ event_a$	$PCO1!event_a$
$PCO2?\ event_b$	$PCO2?event_b$
$PCO3!event_c$	$PCO3!event_c$

Figure 4.2: Les séquences d'événements en TTCN

4.3.2 Le test de conformité dans le contexte de RM-ODP

Au même titre qu'il n'est pas défini de notation pour les langages de points de vue, aucune méthode (technologie) n'est prescrite pour la génération de tests dans le contexte de RM-ODP.

Il existe cependant un cadre architectural pour la conformité des systèmes RM-ODP qui s'applique de la même manière que celui défini par ISO/9646. Ce cadre architectural est en fait une extension de ISO/9646. La majeure partie de cette extension est d'ailleurs déjà en cours d'élaboration dans la standardisation avec "Protocole Profile Testing Methodology" (modification des ICS et les IXIT (cf 4.3.1.1) à des profiles et des informations concernant aussi bien des objets que des protocoles) et "Multy-Party Testing Methodology" (résolution du problème des testeurs externes multiples et des testeurs internes multiples) comme des amendements à ISO/9646.

4.3.2.1 Les points de référence

Le modèle de référence ODP identifie certains points de référence dans l'architecture comme pouvant être retenus comme points de conformité dans les spécifications (cad des points où un essai peut être réalisé sur un objet pour vérifier s'il répond à un ensemble de critères de conformité). Il a été défini quatre points de référence :

- un point de référence de programmation : c'est un point de référence où une interface de programmation est mise en oeuvre pour permettre l'accès à une fonction;
- un point de référence physique : c'est un point de référence où il y a interaction entre le système et le monde extérieur, par exemple une interface homme-machine ou une interface de robot;
- un point de référence d'interfonctionnement : c'est un point de référence où une interface peut être établie pour permettre la communication entre deux ou plusieurs systèmes;
- un point de référence d'échange : c'est un point de référence où un support de stockage physique (par exemple une base de données), externe peut être introduit dans le système.

La figure 4.1 suggère que les PCO A et C soient en terme de RM-ODP à des points de référence d'interfonctionnement (ceci est sûrement dû au fait que le fournisseur de service suggère un fournisseur de service de communications OSI). Les points de références ne sont pas traités dans ISO/9646. Cependant le concept plus général de PCO peut être appliqué à chacun des points de référence.

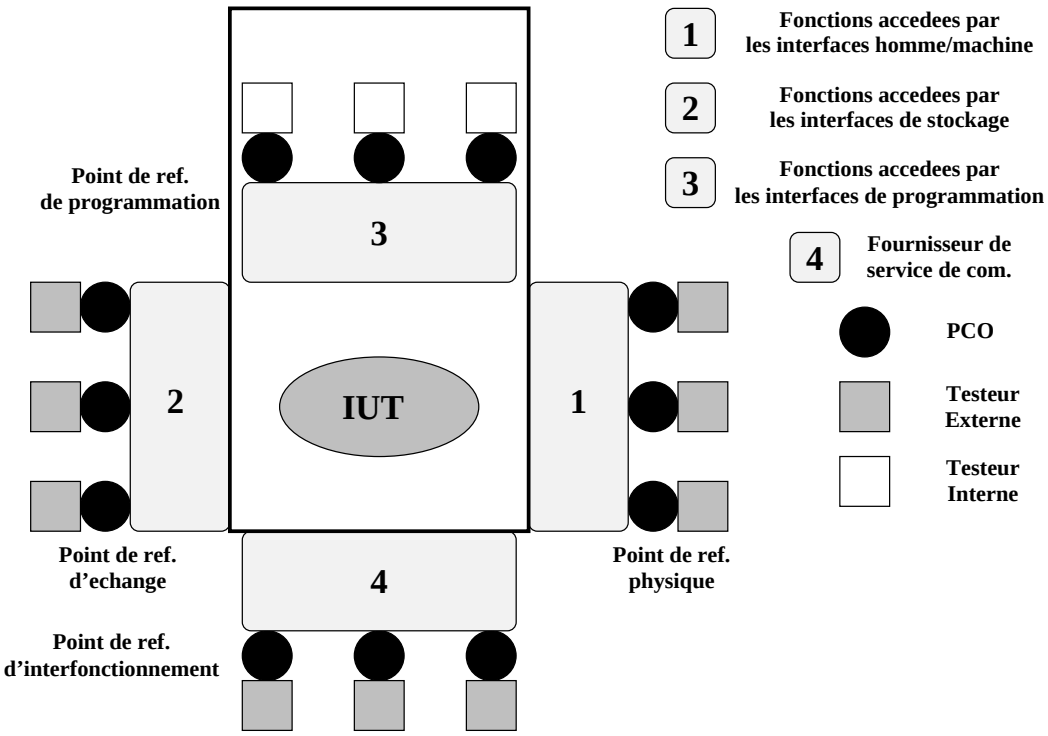

Figure 4.3: Architecture de test générique ODP

La figure 4.3 illustre le modèle générique d'architecture de test défini dans RM-ODP. Les points de référence sur lesquels les contraintes de conformité sont spécifiées peuvent être spécifiés dans tous les points de vue (au sens RM-ODP) excepté au point de vue technologie. Mais dans tous les cas, on doit spécifier au niveau du point de vue d'ingénierie comment les points de références sont utilisés pour le contrôle et l'observation. De fait tous les PCO font référence à leur "mapping" au niveau du point de vue d'ingénierie. C'est à la charge de l'IXIT de décrire le "mapping" des points de référence qui sont définis à un niveau de point de vue autre que celui d'ingénierie vers ce dernier. Bien sur l'IXIT aura pour charge aussi de décrire le "mapping" des points de référence du langage d'ingénierie sur les interfaces réelles du système sous test.

4.3.3 Résumé

La norme ISO/9646 présente le test de conformité des logiciels répartis. Les concepts essentiels sont :

- une définition informelle de la notion de test de conformité (qui est un test de spécification donc de boite noire);
- une méthodologie pour la dérivation (manuelle) des suites de tests;
- la présentation des architectures de test dont l'élément le plus important est la notion de points de contrôle et d'observation (PCO);
- la notation de test TTCN qui est le langage standardisé d'expression des suites de tests et qui est indépendant de toutes implémentation.

Cette section s'est terminée par la présentation du test de conformité dans le contexte de RM-ODP qui est en fait une extension de ISO/9646 en ce qui concerne les architectures de test. Un élément important de cette présentation est la notion de points de référence (extension des PCO) par rapport aux points de vue de RM-ODP.

4.4 Objectifs et hypothèses de travail

4.4.1 Rappel des objectifs

Le problème que l'on s'efforce de résoudre est le suivant : de nos jours, dans le contexte des systèmes à objets répartis, les tests sont encore écrits à la main à partir de spécifications informelles. Selon nous, cette écriture manuelle des suites de tests est le résultat d'un manque de description formelle adaptée à la spécification du comportement dynamique de ces systèmes (et d'outils de simulation comportementale) et non d'un manque d'outils de génération de tests car tous les outils existants imposent d'utiliser des spécifications formelles.

Si l'on se situe dans un contexte ISO/9646, les tests du comportement dynamique sont des tests qui expriment les contraintes dynamiques informelles et qui définissent en fait le comportement observable de l'implémentation lors de ses communications avec l'environnement. Notre

intuition est que les spécifications BL telles qu'elles ont été présentées dans la partie I sont une représentation formelle adaptée à la spécification de ces contraintes dynamiques informelles et que BL et le simulateur TIMS constituent un excellent point de départ pour la génération de tests du comportement dynamique.

L'objectif de ce travail est donc la proposition d'une méthode pour la génération de tests du comportement dynamique des systèmes à objets répartis à partir de BL.

4.4.2 Hypothèses de travail

De la même manière que dans la partie précédente, la formulation des hypothèses de travail nous permet d'une part de formaliser le cadre de notre travail et d'autre part de dégager l'ensemble des propriétés que devra posséder la méthode de génération de tests du comportement dynamique des systèmes à objets répartis. Par analogie à la partie I et pour la clarté de la suite du mémoire, nous nommons ces propriétés les propriétés de la méthode "idéale". Ces hypothèses de travail sont classées de la plus forte (type et modèle pour la génération de tests) à la plus faible (propriété sur les tests générés).

A noter que la réutilisation des spécifications BL (et du simulateur TIMS) constitue l'hypothèse de travail la plus forte. Elle n'entre pas en compte dans la suite de cet exposé car elle constitue une donnée du problème.

4.4.2.1 Hypothèse 1 : Types de tests

Il est important de préciser quels types de tests nous nous proposons de générer. On peut classifier les tests suivant leur appartenance aux types suivants :

- test de conformité;
- test de performance;
- test d'interopérabilité/test d'interfonctionnement;
- test de robustesse.

a) Le test de conformité vise à déterminer si l'implémentation est conforme à sa spécification. C'est le type de tests défini par exemple dans ISO/9646 (cf section 4.3). Il est clair que c'est le type de tests que nous allons chercher à générer et que nous allons nous placer pour ce travail dans le cadre défini par ISO/9646. Si l'on prend en compte le fait que la méthode de génération doit se baser sur les spécifications BL, les tests que nous allons obtenir visent à tester la conformité à la spécification BL.

Propriété 9 P_t0 *: Une méthode qui génère des tests de conformité aux spécifications BL*

b) Le test de performance Dès lors que des performances sont exigées par le cahier des charges des systèmes sous test, elles font de toute évidence partie des implémentations à tester. Le test de performance est une spécialisation du test de conformité qui vise en fait à tester les aspects temporels d'une spécification comportementale. Cependant si l'on se réfère à la section 2.3.2 du chapitre 2, l'aspect temporel n'est pas pris en compte dans BL et donc ne permet pas de spécifier des propriétés temporelles. La méthode, si elle se base sur les spécifications BL, ne pourra pas générer des tests de performances tout comme elle ne pourra pas tester les aspects temps réels de ces systèmes.

c) Le test d'interopérabilité/test d'interfonctionnement Ce type de tests est une spécialisation du test de conformité qui vise en fait à tester les aspects intéropérabilité ou interfonctionnement de ces systèmes. Comme nous l'avons vu dans la section 2.2.1 du chapitre 2, il faut dissocier ces deux types de spécifications :

- le test d'interopérabilité vise à vérifier si deux ou plusieurs composants d'un système réparti peuvent communiquer entre eux;

- le test d'interfonctionnement nécessite d'une part l'interopérabilité et d'autre part le test du comportement dynamique des systèmes répartis. Ces comportements sont testés par la propriété P_t0.

Si l'on fait la supposition que l'interopérabilité est assurée on peut alors dire que nous produisons des tests d'interfonctionnement. Lorsque l'on suppose que l'interopérabilité est assurée, on suppose que l'implémentation du modèle d'information statique et que l'implémentation des services de communications sont correctes. Dans le contexte de la gestion de réseaux OSI, le modèle d'information statique consistant en les spécifications GDMO et ASN.1 et les services de communications étant assuré par le protocole CMIP, ce type de tests peut être assuré en suivant l'approche préconisée par les travaux de [Kab95, Eri].

On généralise donc la propriété P_t0 qui devient P_t1 en disant que nous allons dériver des tests d'interfonctionnement sous l'hypothèse que l'interopérabilité a déjà été testée.

Propriété 10 P_t1 *: Une méthode qui génère des tests d'interfonctionnement*

d) Le test de robustesse La spécification des systèmes que l'on se propose de tester décrit surtout ce que l'on attend comme fonctionnement dans des conditions normales d'utilisation. Le test de robustesse consiste à vérifier la réaction du système face aux tentatives d'utilisation abusives. Cette notion, informelle comme on le voit, peut aussi bien inclure la réaction face aux pannes, que face aux fraudes ou aux simples erreurs de manipulation.

Ces réactions seront testables si elles sont spécifiées dans la spécification BL donc prises en compte dans le modèle que l'on veut simuler et tester.

4.4.2.2 Hypothèse 2 : Modèle de test

Selon la propriété P_t1, nous comptons réutiliser les spécifications BL et nous baser uniquement sur ces spécifications pour générer nos tests. Donc tout naturellement, le modèle de test que nous adoptons est un modèle de type boîte noire ou test fonctionnel (cf section 4.2.3).

Propriété 11 P_t2 *: Une méthode de type boîte noire*

4.4.2.3 Hypothèse 3 : Granularité des tests

La notion de granularité des tests définie dans la section 4.2.1.2 identifie trois types de tests : le test unitaire qui est le test du composant de base, le test d'intégration qui est le test de plusieurs composants et le test système qui est le test de l'ensemble des composants. Dans le test des programmes séquentiels, ce composant peut être une expression. Dans le test des logiciels objets, ce composant peut être un objet. La notion de granularité perd de sa signification dès lors que l'on effectue du test de systèmes répartis car la définition d'un composant de ces systèmes (l'implémentation sous test) fait appel à l'architecture de test. Or puisque l'on réutilise les spécifications BL (qui sont indépendantes de la répartition effective de l'implémentation et des technologies de répartition des objets), il faut que la méthode puisse être indépendante de l'architecture de test et de plus des technologies de répartition des objets.

Propriété 12 P_t3 *: Une méthode indépendante de l'architecture de test (et des technologies de répartition des objets)*

4.4.2.4 Hypothèse 4 : Exécutabilité des tests

Nous désirons que notre méthode produise des tests exécutables, ce qui est souvent référencé dans la littérature par des tests complets. Si l'on se réfère a ISO/9646, ce sont des tests décrits en TTCN.

Propriété 13 P_t4 *: Une méthode qui génère du TTCN*

L'hypothèse d'exécutabilité des tests est vérifiée à partir de la propriété P_t4 car il existe de nos jours des outils qui rendent les suites de TTCN exécutables.

4.4.3 Résumé

Les propriétés de la méthode "idéale" sont donc :

- P_t1 Une méthode qui génère des tests d'interfonctionnement
- P_t2 Une méthode de type boîte noire
- P_t3 Une méthode indépendante de l'architecture de test (et des technologies de répartition des objets)
- P_t4 Une méthode qui génère du TTCN

4.5 Les différentes approches pour la génération de tests du comportement dynamique des systèmes d'objets répartis

Maintenant que nous sommes munis de l'ensemble de propriétés que la méthode "idéale" doit posséder, nous allons examiner les différentes approches pour la génération de tests du comportement dynamique des systèmes d'objets répartis.

Compte-tenu des hypothèses très fortes que nous venons de faire comme la réutilisation de BL et en nous situant dans un cadre méthodologique comme ISO/9646, le nombre des approches est lui aussi très fortement réduit. C'est pourquoi nous n'examinerons que les approches de génération de tests issus du test des logiciels répartis (et notamment du test de protocoles). Cette restriction peut paraître discutable mais nous allons montrer que cet examen mène à une solution acceptable. De plus, ce choix nous garantit d'ores et déjà l'adéquation à P_t2 (méthode de type boîte noire) et P_t3 (méthode indépendante de l'architecture de test et des technologies de répartition des objets).

4.5.1 Discussion des techniques de tests en fonction des types de spécifications existantes

Avant d'examiner les différentes techniques de tests des logiciels répartis, nous proposons de discuter ces différentes techniques en fonction des types de spécifications existantes qui sont (comme nous l'avons vu dans la section 2.4 du chapitre 2) :

- les spécifications basées sur une approche axiomatique;
- les spécifications basées sur un modèle de transitions d'état.

Comme nous l'avons déjà vu, les spécifications basées sur une approche axiomatique sont plutôt dédiées à la spécification des propriétés tandis que les spécifications basées sur un modèle de transitions d'état sont plutôt dédiées à la construction du modèle de transitions d'état.

Si l'on considère la propriété P_t4 (méthode qui génère du TTCN) et la présentation du langage TTCN dans la section 4.3.1.4, la décomposition des tests sous la forme de séquences d'événements entre l'implémentation et le système de tests, fait immédiatement penser que c'est plutôt vers les techniques dédiées aux spécifications basées sur un modèle de transitions d'état qu'il faudra orienter notre examen.

Nous n'examinerons donc pas les techniques de tests basées sur les spécifications axiomatiques, nous ne considérerons que les techniques basées sur un modèle de transitions d'état.

4.5.2 Discussion entre les techniques de dérivation statiques et les techniques de dérivation dynamiques

L'ensemble des techniques de dérivation de test peut alors se décomposer en deux types de techniques :

- les techniques statiques de dérivation de tests;

- les techniques dynamiques de dérivation de tests.

Les techniques statiques de dérivation de tests appliquent des méthodes équivalentes à la méthode de la construction du graphe de contrôle du test structurel présentée dans la section 4.2.2 mais appliquées sur la spécification comportementale d'un système réparti. On peut citer notamment les graphes syntaxiques de SDL [TR96].

Les techniques dynamiques de dérivation de tests, quant à elles, effectuent la génération de tests en deux étapes :

- la première étape est une simulation exhaustive du comportement spécifié par la spécification formelle afin d'obtenir un modèle mathématique du comportement observable de l'implémentation. Ces modèles sont essentiellement les automates d'états finis (FSM) ou les systèmes de transitions étiquetés (LTS);

- la seconde étape consiste en l'application de techniques de génération de tests sur ces modèles mathématiques afin d'obtenir les cas de tests finaux.

Le problème avec les techniques statiques de dérivation de tests est qu'elles n'engendrent pas de tests complets [TR96]. C'est à la charge de l'utilisateur de compléter les cas de tests et d'écrire du TTCN. On ne remplit pas ainsi la propriété P_t4. C'est pourquoi nous allons nous concentrer sur les techniques de dérivation dynamiques basées sur des modèles mathématiques.

4.5.3 Les techniques dynamiques de dérivation de tests

4.5.3.1 Discussion préalable sur l'usage de modèles mathématiques

Une rapide exploration de la littérature montre clairement que les techniques de dérivation de tests à partir de FSM ont été beaucoup plus étudiées que les techniques basées sur les LTS. Ceci se justifie, selon nous, par le fait que les FSM étant un modèle plus puissant (en terme de propriétés mathématiques), des informations de tests comme : la couverture, la testabilité, ... peuvent être étudiées. Au contraire, les LTS sont un modèle bas niveau (très général) qui possède très peu de propriétés. On peut d'ailleurs traduire le comportement dynamique d'un système représenté par un FSM dans son équivalent représenté par un LTS de manière immédiate. Le contraire (cad passage de FSM vers LTS) est moins évident et représente peu d'intérêt.

Cependant, il est souvent noté dans la littérature que la construction automatique d'un FSM représentant le comportement dynamique à partir d'une spécification formelle n'est pas une chose aisée, tandis que la construction d'un LTS s'obtient immédiatement par la simulation exhaustive de cette spécification. Ces LTS particuliers sont appelés le graphe d'accessibilité de la spécification. Un grand nombre de méthodes de génération de tests se basent sur le graphe d'accessibilité pour ensuite générer des tests. Les approches citées dans [ACM+96] : Testgen et TVéda sont basées sur la construction de ce graphe d'accessibilité qui est ensuite transformé en un FSM pour appliquer des méthodes de génération de tests sur les FSM. D'autres approches basées sur la construction du graphe d'accessibilité sont la méthode Samstag [Nah95] et la méthode TGV [FJJV96b].

L'exposé que nous allons faire va donc présenter uniquement les techniques de dérivation de tests à partir de LTS.

4.5.3.2 Les systèmes de transitions étiquetés LTS

a) Définition formelle des systèmes de transitions étiquetés

Définition 5 *Un système de transition étiqueté est un quadruplet* $S = (Q, A, \rightarrow_s, q_0)$ *tel que :*

- *Q est un ensemble fini non vide d'états.* $q_0 \in Q$ *est l'état initial.*
- *A est un ensemble fini d'actions observables.* $\tau \notin A$ *représente les actions internes.*
- $\rightarrow_s \subseteq Q \times (A \cup \tau) \times Q$ *est la relation de transition.*
- on notera souvent $q \overset{a}{\Rightarrow_s} q'$ (ou simplement $q \overset{a}{\Rightarrow} q'$ quand il n'y a pas d'ambiguïté sur le LTS S) pour $(q, a, q') \in \rightarrow_s$
- $q \overset{a}{\Rightarrow_s}$ sera parfois utilisé à la place de $\exists q' : q \overset{a}{\Rightarrow_s} q'$
- l'ensemble des séquences d'actions de longueur finie est notée $A*$
- une trace est une séquence d'actions observables de A
- la séquence vide est notée ε
- la concaténation des traces est notée «.»
- comme d'habitude afin de ne pas parler que des actions visibles dans un système de transitions, nous utiliserons les notations suivantes où
  $q,\ q',\ q_1, \ldots,\ q_n \in Q,\ P \in 2^Q,\ a,\ a1, \ldots, a_n \in A,\ \sigma \in A^*$
- $q \overset{\varepsilon}{\Rightarrow} q'$ signifie $q \overset{\tau,\tau,\ldots,\tau}{\Rightarrow} q'$. On notera également $\tau^*(q) = \{q' | q \overset{\varepsilon}{\Rightarrow} q'\}$
- $q \overset{a}{\Rightarrow} q'$ pour $a \neq \varepsilon$ signifie $\exists q_1, q_2 : q \overset{\varepsilon}{\Rightarrow} q_1 \overset{a}{\Rightarrow} q_2 \overset{\varepsilon}{\Rightarrow} q'$
- $q \overset{a}{\Rightarrow}$ signifie $\exists q' : q \overset{a}{\Rightarrow} q'$
- $q \overset{a_1 \ldots a_n}{\Longrightarrow} q'$ signifie $\exists q_0, \ldots q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \ldots q_{n-1} \overset{a_n}{\Rightarrow} q_n = q'$
- $q \overset{a_1 \ldots a_n}{\Longrightarrow}$ signifie $\exists q' : q \overset{a_1 \ldots a_n}{\Longrightarrow} q'$
- $A(q) = \{a \in A | q \overset{a}{\Rightarrow}\} et A\{P\} = \bigcup_{q \in P} A(q)$
- q **after**$\sigma = \{\mathbf{q}' | \mathbf{q} \overset{\sigma}{\Rightarrow} \mathbf{q}'\}$ et P **after**$\sigma = \bigcup_{\mathbf{q} \in \mathbf{P}} \mathbf{q}$ **after**σ
- $traces(q) = \{\sigma | q \overset{\sigma}{\Rightarrow}\}$ et $traces(S) = traces(q_0)$
- q **can**$\sigma \Leftrightarrow \mathbf{q} \overset{\sigma}{\Rightarrow}$
- q **cannot**$\sigma \Leftrightarrow \mathbf{q} \overset{\sigma}{\not\Rightarrow}$
- q **after** σ **refuses A** $=_{\mathbf{def}} \exists \mathbf{q}' \in \mathbf{Q}$**after** $\sigma, \forall \alpha \in \mathbf{A}, \mathbf{q}'$ **cannot** σ

b) Relations d'implantation pour la conformité L'utilisation des spécifications formelles pour spécifier les systèmes concurrents a conduit à l'introduction des équivalences de tests et à une définition formelle de la notion de conformité sous formes de relations d'implantation. Une relation d'implantation est une relation qui doit lier toute implantation conforme à sa spécification formelle (cad elle exprime formellement les aspects de la spécification que toute implantation conforme doit respecter). Etant donné une relation d'implantation $\leqslant_R$, une spécification S et une implémentation I, on a $I \leqslant_R S$ si et seulement si I est une implantation correcte de la spécification S. La question qui se pose ensuite, est de savoir quelles sont les relations d'implantations les plus appropriées. La réponse à cette question peut être obtenue en essayant de définir la classe des implantations valides pour une spécification formelle donnée. En variant cette classe, plusieurs types de relations d'implantation peuvent être obtenus :

- relations d'équivalence :
  Plusieurs LTS peuvent décrire intuitivement le même comportement observable. Cela peut être dû par exemple aux actions τ qui sont considérées comme non observables. Ainsi quelques relations d'équivalence ont été définies basées sur la notion de comportement observable : l'équivalence observationelle [Mil80], la bisimulation forte et la bisimulation faible [Mil89], l'équivalence d'échec [Hoa85] ...
  Ces relations sont basées sur certains critères d'observation (cad deux systèmes sont dits équivalents si et seulement si ils sont indistinguables par certains types d'observations extérieures). Ces équivalences permettent de faire abstraction de certains détails internes.
  Par exemple, pour deux LTS P_1 et P_2 :

  - $P_1 \leqslant_{tr} P_2 =_{def} traces(P_1) \subseteq traces(P_2)$
  - $P_1 \geqslant_{tr} P_2 =_{def} traces(P_1) \supseteq traces(P_2)$
  - $P_1 \approx_{tr} P_2 =_{def} traces(P_1) = traces(P_2)$

  $\leqslant_{tr}$ et $\geqslant_{tr}$ sont des préordres. $\approx_{tr}$ est une relation d'équivalence appelée équivalence de traces [1].

- relation d'implantation :
  Pour les systèmes de transitions étiquetés, la conformité peut être caractérisée par une relation de test appelée relation d'implantation. Les relations d'implantation sont également obtenues par comparaison du comportement observable de l'implantation avec celui de la spécification. Cependant pour une relation d'implantation, il est seulement demandé que le comportement de l'implantation soit en relation avec celui de la spécification (cad le comportement de l'implantation peut être expliqué à partir du comportement de la spécification). En général, c'est une relation asymétrique et moins restrictive qu'une relation d'équivalence. Par exemple, on peut admettre qu'une implémentation puisse être plus déterministe qu'une spécification, ou que sa fonctionnalité puisse être réduite ou étendue. Une relation d'implantation serait donc plutôt considérée comme un préordre (bien choisi). Le couple *spécification + relation d'implantation* détermine l'ensemble des implantations

[1] Un préordre est une relation réflexive et transitive. Une relation d'équivalence est un préordre symétrique.

conformes à la spécification. Cet ensemble est infini en général. Quelques relations d'implantation basées sur cet idée ont été définies dans [Bri87b, Bri87a, Led91].

- **Définition 6** *(Relation de réduction)*
  I **red** $S \Leftrightarrow \forall \sigma \in A*, \forall P \subseteq Q:$ $\quad traces(I) \subseteq traces(S)$ *et*
  I **after** σ **refuses P** $\Rightarrow$ **S after** σ **refuses P**

  La relation de réduction exprime le critère de conformité basé sur la réduction du non-déterminisme. Une implantation I est correcte par rapport à une spécification S si les traces observées en testant l'implantation I avec un test t doivent également être observées en testant S avec t; et les blocages observés dans I doivent l'être également dans S. La relation d'implantation obtenue notée **red**, est aussi appelée préordre de test. Elle est notée $\leqslant_{te}$.

- **Définition 7** *(Relation d'extension)*
  I **ext** $S \Leftrightarrow \forall \sigma \in A*, \forall P \subseteq Q:$ $\quad traces(I) \supseteq traces(S)$ *et*
  I **after** σ **refuses P** $\Rightarrow$ **S after** σ **refuses P**

  La relation d'extension **ext** exprime la notion d'extension du comportement de l'implantation par rapport à celui de la spécification. Une implémentation I est conforme à une spécification S suivant cette relation, si les traces observées en testant S avec un test t doivent également être observées en testant l'implantation I avec le test t; et les blocages observés dans I doivent l' être également dans S.

  Remarque 1 *La relation d'équivalence induite par* **red** *et* **ext** *est l'équivalence de test* $\approx_{te}$.

  La relation **red** implique $\leqslant_{tr}$. Tester **red** pose donc le problème de vérifier que toutes les traces qui ne sont pas dans la spécification ne le sont pas dans l'implémentation, conformément à la caractérisation de $\leqslant_{tr}$. Ceci pose un problème car cet ensemble de traces est en général infini. Dans [Bri88], une autre relation d'implantation **conf** a été introduite pour résoudre ce problème. Cette relation utilise uniquement les traces qui existent dans la spécification et donc ne teste pas $\leqslant_{tr}$.

- **Définition 8** *(Relation conf)*
  I **conf** $S =_{def} \forall \sigma \in traces(S), \forall P \subseteq A:$
  I **after** σ **refuses P** $\Rightarrow$ **S after** σ **refuses P**

  Tester si I **conf** S revient à vérifier si, placé dans un environnement dont les traces sont réduites à celles de S, I n'a pas de blocages qui ne figurent pas dans S. Autrement dit, I bloque moins souvent que S. Ainsi **conf** ne permet pas de détecter des traces de I qui ne figurent pas dans S. **conf** est une relation réflexive mais pas transitive et on a **red** $= (\leqslant_{tr} \bigcap$ **conf**$)$. On peut donc tester la correction d'une implémentation par rapport à **red** en la testant par rapport à $\leqslant_{tr}$ et à **conf** séparément.

c) La génération de tests à partir des systèmes de transitions étiquetés Les deux méthodes les plus référencées sont la méthode des testeurs canoniques et la méthode CO-OP :

- La méthode des testeurs canoniques :
  Pendant l'exécution d'un test t sous l'implantation I, l'observation consiste à considérer les apparitions ou les non apparitions d'actions observables. Un modèle de test pour un ensemble de LTS Q peut être défini par :

  – un ensemble T de testeurs (ou tests);

  – une application $runs : Q \times T \rightarrow A*$

  Dans [Bri87b, Tre92], une théorie a été proposée pour définir des testeurs appelés testeurs canoniques, pour des LTS finis. Ces testeurs permettent de décider si une implantation satisfait la relation d'implantation **conf**. Dans cette théorie, les systèmes à tester et les testeurs sont tous des LTS. L'application d'un test t à un LTS p est conduit par l'opération de composition parallèle $\| : runs(p,t) = \{\sigma \in A* \,|\, p \| t \stackrel{\sigma}{\Rightarrow}\}$
  Appliquer une fois le testeur canonique au processus p revient à tester une seule trace de la spécification et conduit toujours à un succès si p est une implantation correcte : c'est un LTS ayant les mêmes traces que la spécification et qui exécuté de manière synchrone avec le LTS p, n'aboutit jamais à un blocage. En appliquant le testeur canonique jusqu'à ce que toutes les traces de la spécification soient testées, nous pouvons décider si Q **conf** S. Le testeur canonique pour la spécification S est défini de la façon suivante :

  Définition 9 *(Testeur canonique)*
  Soit S un LTS. Un testeur canonique pour S est le LTS $T(S)$ satisfaisant les conditions suivantes :

  – $traces(T(S)) = traces(S)$

  – $\forall I, I\ bf conf S \Leftrightarrow \forall \sigma \in Q*, \exists P \subseteq A$
  I **after** σ **refuses P** $\Rightarrow$ **S after** σ **refuses P**

  Le testeur canonique se comporte comme un observateur averti, qui exige que l'implémentation exécute toujours une action choisie parmi un ensemble d'actions possibles, sans que ce choix n'entraîne un blocage. L'arrêt de l'exécution est accepté si c'est le testeur qui l'a décidé.

- La méthode CO-OP :
  A partir de la définition des testeurs canoniques, [Wez90] a proposé une méthode, appelée méthode CO-OP pour dériver des testeurs canoniques à partir d'une spécification donnée. Dans cette méthode, un cas de test est défini comme une réduction de $T(S)$ (au sens de la définition **red**) qui ne peut plus être réduite. Une suite de tests est l'ensemble de tous les cas de test (cad toutes les réductions possibles de $T(S)$).

  Définition 10 *(CO-OP)*
  Soit L un LTS et q un état de Q

  1. $out(L) = \{\sigma \in L | L \stackrel{\sigma}{\Rightarrow}\}$ *(toutes les actions exécutables à partir de L)*
  2. *L est dit stable si* $L \stackrel{\tau}{\not\Rightarrow}$ *(le système ne peut pas faire de transition interne à partir de L)*

La méthode CO-OP est basée sur l'idée suivante : si un LTS L peut évoluer de manière interne jusqu'à un état stable Q, alors tout cas de test pour L doit pouvoir exécuter au moins une des actions x telle que $L \overset{x}{\Rightarrow}$. Les cas de test qui ne contiennent aucune de ces actions peuvent bloquer en communiquant avec une implantation correcte de L. Pour couvrir toutes les traces possibles de L dans un testeur canonique, pour tout $a \in out(L)$, il doit exister au moins un cas de test qui peut exécuter a.

d) Choix de l'approche La méthode des testeurs canoniques et la méthode CO-OP ne sont pas très utilisables sur des problèmes de taille réelle car elles conduisent à un très grand nombre de tests (le nombre de tests est proportionnel à la taille du graphe d'accessibilité). Nous préférons adopter une approche basée sur la construction du graphe d'accessibilité et ensuite appliquer des algorithmes de génération de tests sur ce graphe d'accessibilité comme c'est le cas avec : TVeda, Testgen, Samstag et TGV.

Ce choix est motivé par deux raisons :

- nous avons l'intuition que nous allons pouvoir générer le graphe d'accessibilité à partir de spécifications BL et du simulateur TIMS. Il faudra ensuite adapter ce graphe pour qu'il serve de point d'entrée d'un outil spécifique de génération de tests.

- ce type d'approche permet de générer des tests sur des problèmes de taille réelle (de complexité réelle). Ceci est bien sûr très important dans notre contexte. On jugera d'ailleurs de cette faculté de notre approche à appréhender des complexités importantes lors de l'application de ce travail de génération de tests sur les interfaces Xcoop dans la partie III.

Parmi l'ensemble des outils disponibles, nous choisissons d'utiliser TGV (qui sera présenté dans le chapitre suivant : le chapitre 5) comme outil de génération de tests. Tous les autres outils aurait pu faire l'affaire (Samstag, TVeda ou Testgen) car la méthode que nous comptons utiliser est complètement générique. Notre choix s'est porté sur TGV car :

- il génère du TTCN (et donc de fait il remplit la propriété P_t4, une méthode qui génère du TTCN);

- il permet une sélection des tests générés en fonction d'objectifs de test (tel que c'est préconisé dans ISO/9646 et tel que c'est effectué dans la pratique);

- le formattage du graphe d'accessibilité entre TIMS et TGV n'est pas du tout difficile à réaliser comme il sera montré dans le chapitre suivant.

4.5.3.3 Examen des propriétés de la méthode idéale

a) P_t1 **(méthode qui génère des tests d'interfonctionnement)** Il faut rappeler que cette propriété signifie d'une part que nous allons générer des tests de conformité par rapport aux spécifications BL et que d'autre part l'interopérabilité du système est assurée. La démonstration de cette propriété est immédiate dès lors que nous faisons l'hypothèse que nous sommes capables d'obtenir un graphe d'accessibilité à partir de spécifications BL et du simulateur TIMS.

b) P_t2 **(méthode de type boîte noire)** Le choix des méthodes de dérivations dynamiques qui génèrent un modèle mathématique du comportement observable de la spécification vérifie immédiatement P_t2.

c) P_t3 **(méthode indépendante de l'architecture de test et des technologies de répartition des objets)** La conclusion pour P_t3 est la même que pour P_t2.

d) P_t4 **(méthode qui génère du TTCN)** Le choix de TGV est justifié par le fait que cet outil, à partir d'un graphe d'accessibilité, génère des tests dans le format TTCN. La propriété P_t4 est donc de fait vérifiée.

4.5.4 Résumé

Lors de cette section, nous avons examiné les différentes alternatives pour la génération de tests du comportement dynamique des systèmes à objets répartis. L'examen s'est en fait résumé en l'étude des différentes techniques de dérivation de tests à partir de spécifications formelles des systèmes répartis. Nous avons restreint cette étude aux techniques dédiées aux spécifications qui ont comme modèle de base le modèle de transition d'états.

Nous nous sommes ensuite intéressés aux techniques dynamiques de dérivation de tests à partir de ce type de spécifications qui

- dans un premier temps simulent exhaustivement la spécification afin d'en obtenir une représentation sous la forme d'un modèle mathématique (comme les FSM et les LTS);
- et qui dans un second temps génèrent des tests à partir d'outils de génération de tests.

Cette présentation a ensuite débouché sur une ébauche de solution à notre problème. Cette solution se base sur l'hypothèse que l'on est capable d'obtenir un graphe d'accessibilité à partir de simulations exhaustives de spécifications BL, et que l'on applique ce graphe d'accessibilité dans un outil de génération de tests approprié comme TGV pour obtenir les tests sous la forme TTCN. On obtient alors la méthode de génération de tests du comportement dynamique des systèmes à objets répartis, qui constitue l'objectif de ce travail.

Enfin, cette section a terminé par un examen sommaire qui montre que l'approche que nous nous proposons d'adopter est en parfaite adéquation par rapport aux propriétés de la méthode "idéale".

4.6 Conclusion

La première partie de ce chapitre était une introduction à la problématique : la génération de cas de tests du comportement dynamique des systèmes à objets répartis. Nous avons débuté par une présentation dans le contexte général des logiciels de cette activité. Puis nous avons raffiné notre étude aux systèmes répartis par la présentation de la norme ISO/9646.

A partir d'un rappel de nos objectifs initiaux et du modèle ODP, la seconde partie de ce chapitre a consisté en l'énoncé d'un ensemble de propriétés que la méthode idéale doit posséder pour permettre la génération de cas de tests du comportement dynamique des systèmes à objets répartis.

Enfin la troisième partie de ce chapitre a consisté en l'étude des différentes techniques existantes. Cette étude peut se résumer en l'étude des techniques de dérivation dynamiques de tests à partir des spécifications formelles des systèmes répartis.

La méthode idéale modulo l'hypothèse que nous sommes capables d'obtenir le graphe d'accessibilité à partir de spécifications BL et du simulateur TIMS, consiste alors en l'intégration d'un outil de génération de tests TGV qui à partir de ce graphe d'accessibilité va générer des cas de tests TTCN.

Le prochain chapitre qui présente notre approche pour la génération de tests du comportement dynamique des systèmes à objets répartis va donc montrer comment on peut intégrer le simulateur TIMS qui va servir à la génération du graphe d'accessibilité et l'outil TGV.

Chapitre 5

La génération de tests du comportement dynamique des systèmes répartis avec TIMS

5.1 Introduction

L'objectif global de ce travail (de cette partie) est la génération de tests du comportement dynamique des systèmes répartis. Le chapitre précédent après un examen des propriétés que doit posséder la méthode "idéale" de génération de tests, a proposé d'adopter une approche basée sur la construction du graphe d'accessibilité qui est un LTS (système de transitions étiqueté) particulier. Ce chapitre va montrer comment nous avons mis en oeuvre cette approche (notamment l'intégration des deux outils TIMS et TGV).

Ce chapitre se compose de trois parties :

- la première partie présente les principes de base de notre approche pour la génération de tests du comportement dynamique des systèmes à objets répartis. Cette approche se base sur l'intégration du simulateur TIMS de spécifications BL et de l'outil TGV de génération de tests.
  TIMS sert en fait de pré-processeur en générant un graphe d'accessibilité et des objectifs de test. TGV sert de post-processeur en dérivant un cas de test TTCN à partir du graphe d'accessibilité pour chaque objectif de test.

- la seconde partie présente les modifications que nous avons apportées au simulateur TIMS pour que d'une part il génère le graphe d'accessibilité et que d'autre part il génère les objectifs de tests. La génération du graphe d'accessibilité se fait en deux étapes: construction d'un arbre de comportement par simulation exhaustive et génération du graphe d'accessibilité. La construction de l'arbre de comportement utilise l'algorithme B.2.1 présenté dans la partie précédente (la partie I). Nous savons d'ores et déjà que ces performances sont limitées dès lors que la complexité est moyenne. La génération des objectifs de tests se fait par application d'une technique particulière des ordres partiels (les Sleep Set [God95]) sur l'arbre de comportement.

- la troisième et dernière partie débute par un compte rendu d'expérimentations qui confirme

que dès lors que les simulations comportementales sont un peu compliquées, la génération de l'arbre de comportement (et donc du graphe d'accessibilité) devient très longue. Cette partie présente donc la solution que nous avons apportée à ce problème et démontre sur quelques expérimentations son efficacité.

5.2 Les principes de base de notre approche

Notre approche pour la génération de tests du comportement dynamique des systèmes à objets répartis s'appuie donc sur une technique particulière de dérivation dynamique de tests : la construction d'un graphe d'accessibilité qui est en fait un LTS particulier. Ce type de techniques génère des tests en deux étapes :

1. étape 1 : Construction du graphe d'accessibilité par simulation exhaustive du comportement observable. Cette construction est immédiate à partir de spécifications basées sur des modèles de transitions d'états.

2. étape 2 : Génération des tests à partir du graphe d'accessibilité à l'aide d'un outil de génération de tests.

Pour effectuer l'étape 1, nous proposons d'utiliser les spécifications BL et le simulateur TIMS présentés dans le chapitre 3. Pour effectuer l'étape 2, nous proposons d'utiliser l'outil TGV qui va être présenté dans la suite de cette section.

La figure 5.1 résume l'approche que nous comptons adopter pour la génération de tests.
L'étude de cette figure nous montre que :

- l'outil TIMS peut se résumer à l'équation suivante : $Arbre_{exhaustif} = f_{TIMS}(Spec_{BL}, Scen)$ où $Arbre_{exhaustif}$ est l'arbre de comportement résultat de la fonction de simulation exhaustive f_{TIMS} qui prend en entrée la spécification BL $Spec_{BL}$ auquel on applique le scénario *Scen*.

- l'outil TGV peut se résumer à l'équation suivante : $TC_{TTCN} = f_{TGV}(G, TP)$ où TC_{TTCN} (le cas de test en TTCN) est le résultat de la fonction de génération de tests f_{TGV} qui prend en entrée G le graphe d'accessibilité et TP l'objectif de test tous deux décrits sous la forme d'un IOLTS (Input Output Labeled Transition System). Un IOLTS est un système de transitions étiqueté dans lequel on distingue deux types d'actions, les entrées et les sorties. Une définition formelle est donnée plus tard dans la section 5.2.2.

5.2.1 TIMS

Le problème qui se pose donc est comment à partir des spécifications BL et du simulateur TIMS, nous allons pouvoir générer d'une part le graphe d'accessibilité G et d'autre part les objectifs de tests TP.

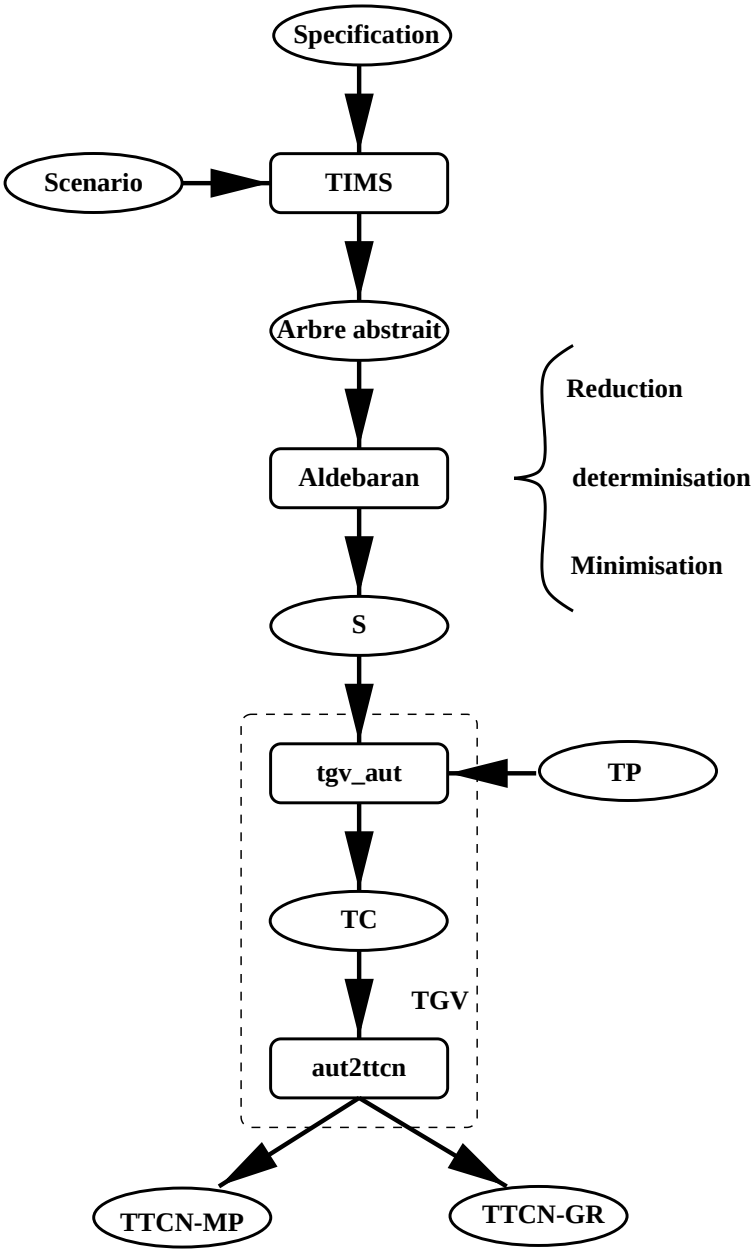

Figure 5.1: Génération de tests à partir des spécifications BL et de TIMS

5.2.1.1 La génération du graphe d'accessibilité

L'idée de base pour la génération du graphe d'accessibilité est de transformer l'arbre de comportement : $Arbre_{exhaustif}$ issu d'une simulation TIMS en mode exhaustif (présenté par l'algorithme B.2.1) sous la forme attendue par TGV (cad un IOLTS). Ce que l'on peut formaliser par l'équation : $G = f_{TIMS2}(Arbre_{exhaustif})$

Cette transformation d'un arbre de comportement TIMS $Arbre_{exhaustif}$ à celle d'un LTS nécessite que l'on précise les points suivants :

- Comment passer d'un arbre de comportement TIMS $Arbre_{exhaustif}$ où la notion d'état est implicite à un LTS où la notion d'état devient explicite.

- Comment passer de la notion d'arbre de comportement TIMS $Arbre_{exhaustif}$ où la transition de base correspond à l'exécution d'un *BEN* (Behavior Execution Node) à celle d'un IOLTS où la transition de base est étiquetée. Cette transition de base étiquetée prend une sémantique soit de messages observables (entrée du testeur) ou contrôlables (sortie du testeur) soit de messages internes (invisibles par le testeur) nécessaire pour l'activité de test.

a) Rendre les états du système explicites Pour transformer un arbre de comportement en un LTS, il faut rendre explicite la notion d'état. La notion d'état dans notre modélisation des systèmes à objets répartis (tels que nous l'avons décrite dans le chapitre 3) est implicite. Elle est donnée par le couple (IR, BET) où IR est le conteneur d'instances (l'ensemble des objets du système avec leurs attributs et les valeurs associées) et où BET est l'arbre de comportement (représentation de l'exécution d'une simulation). L'IR représente la vue du flux des données de l'état du système tandis que le BET représente la vue du contrôle de l'état du système.

Rendre les états explicites ne signifie pas que nous sommes capables de les qualifier (cad de le reconnaître des autres états). Cela est malheureusement impossible dans le simulateur TIMS compte-tenu de certains choix d'implémentations(cf [Sid97]) (c'est notamment la linéarisation du BET qui est impossible car linéariser l'IR ne pose pas vraiment de problèmes). Rendre explicite l'état signifie qu'il faut identifier le message (que l'on appelle transition t_{ij}) qui nous fait passer d'un état global $E_i = (IR_i, BET_i)$ à un état global $E_j = (IR_j, BET_j)$ tel que : $E_i \overset{t_{ij}}{\Rightarrow} E_j$.

Si l'on résume la section 3.4.1, l'arbre de comportement exhaustif $Arbre_{exhaustif}$ est un arbre de *BEN* (Behavior Execution Node). Chaque BEN représente une unité d'exécution d'une simulation TIMS. Un *BEN* correspond à un contexte de déclenchement le *BEC* et à un comportement à exécuter. Le *BEC* représente quant à lui le message déclencheur *msg* et éventuellement la relation et le rôle auxquels participe l'objet. C'est ce message déclencheur qui fait passer le système d'un état E_i à E_j et c'est donc lui qui fera office de transition dans le graphe d'accessibilité.

b) La sémantique des messages Dans un IOLTS (formalisme utilisé par TGV), les transitions sont étiquetées. Elles indiquent si le message est observable (entrée) ou contrôlable (sortie) ou bien elles indiquent si c'est un message interne. Ce choix d'étiqueter les transitions nous oblige à parler d'architecture de tests et de positionnement des PCO (Points of Control and Observation). C'est une étape qui dans notre modèle de spécification comportementale n'est pas prise en compte

(compte-tenu du choix délibéré de nous situer au niveau du point de vue de traitement et du point de vue d'information) et pour laquelle il faudra rajouter des informations. Ces informations appartiennent au point de vue d'ingénierie si l'on se réfère à ce qui est préconisé dans le test de conformité des systèmes RM-ODP (cf section 4.3.2).

Un exemple d'architecture de test peut être le suivant :

On considère le modèle générique de systèmes à objets répartis défini dans la section 3.2.2. Les objets CVP sont effectivement répartis et représentent des composants que l'on veut tester, les IUT (Implementation Under Test) alors

- les PCO seront situés aux interfaces de ces objets;
- les messages observables (et contrôlables) seront les messages CVP;
- les messages internes seront alors les messages IVP car les objets d'information ne seront pas visibles (conformément à la propriété $P_t 2$ méthode de type boite noire).

Notre approche conserve tout de même sa généricité car c'est au dernier moment que l'on fixe la localisation des PCO et donc les messages du système (observables, contrôlables ou internes). De plus ces informations d'architecture de test sont données en paramètre à la méthode de test. En cela la propriété $P_t 3$ reste toujours valide.

5.2.1.2 La génération des objectifs de test

La spécification des objectifs de test (TP Test Purpose) est une tâche difficile qui nécessite une expertise d'une part de la spécification à tester (pour déterminer ce qu'il est pertinent de tester) et d'autre part de la modélisation de cette même spécification pour l'activité de test. C'est une étape cruciale qui va permettre de sélectionner les tests que l'on va appliquer sur l'implémentation. Cette sélection est importante car il faut obtenir le plus petit nombre de cas de tests ayant un pouvoir de détection de fautes le plus important possible. Il faut rappeler que, notamment dans le contexte de TGV, pour chaque objectif de test, on a génération d'un cas de test.

La génération "automatique" des objectifs de test est une solution intéressante au problème de la spécification manuelle de ces objectifs. La génération "automatique" des objectifs de test à partir de spécifications BL ou du simulateur TIMS (du résultat de simulation) constitue donc l'objectif de ce travail. Cependant, elle nécessite une discussion préliminaire car le choix de la technique pour la dérivation des objectifs de test n'est pas aussi immédiat. Ce choix doit surtout prendre en compte le fait que le but de la génération d'objectifs de test n'est pas d'obtenir le plus grand nombre d'objectifs de tests mais au contraire d'obtenir le plus petit nombre d'objectifs de tests qui prend en compte le plus grand nombre de fautes détectables. Nous avons donc étudié les différentes approches citées dans la littérature concernant l'expression et la dérivation automatique des objectifs de test.

Les hypothèses que nous proposons sont les suivantes :

- dérivation finale d'un objectif de test sous la forme d'un IOLTS conforme à TGV;
- du fait de l'hypothèse précédente) en terme d'expressivité la restriction est liée aux langages spécifiables par les LTS.

a) Définition informelle d'un objectif de test La norme ISO/9646 définit la notion d'objectif de test de manière complètement informelle. C'est l'unité de couverture des contraintes (statiques ou dynamiques de conformité) (cf 4.3.1.1) qui doivent conduire à un cas de test. A titre d'indication, [Che96] recense les besoins suivants pour l'expression d'objectifs de test :

- tester une transition particulière est de loin le besoin le plus souvent utilisé. Un tel test consiste à exécuter un préambule qui a pour but de positionner l'implantation dans un certain état d'origine de la transition à tester, exécuter la dite transition, et éventuellement exécuter un postambule (souvent un reset). Ce type de test est appelé méthode de test de "transition unique" et est à la base de nombreux autres besoins de test;
- tester une transition avec tous les préambules possibles;
- tester une séquence donnée de transitions;
- effectuer un test *n* fois consécutivement.

Le besoin d'un formalisme pour l'expression des TP, a été reconnu par la communauté du test assez rapidement. Malheureusement, il n'existe pas de formalismes standardisés pour leur expression. Nous allons présenter ces différents formalismes (existants) en les classifiant suivant deux approches :

- Les approches dites "génériques", qui sont basées sur l'extraction de l'objectif de test à partir de la spécification (formelle) ou de la modélisation du comportement dynamique de cette spécification.
- Les approches dites "empiriques", qui sont extraites d'informations que le testeur doit ajouter (manuellement) à la spécification.

b) Les approches génériques De la même manière que pour la dérivation des tests, on distingue deux types de méthodes pour l'expression des objectifs de test : les méthodes dynamiques basées sur le modèle mathématique représentant le comportement dynamique et les méthodes statiques basées sur la spécification formelle.

- les méthodes dynamiques :
  Elles sont issues d'un traitement sur le modèle mathématique représentant le comportement de la spécification :
  - le test de "transition unique" comme nous venons de le voir;
  - dans [BTV91], une notation (**after** $\Leftrightarrow$**must**) basée sur les LTS est proposée pour l'expression des objectifs de test. Cette notation est utilisée suivant la forme (**after** σ **must** A) : où σ dénote une séquence d'événements (cad une trace), menant jusqu'à un certain état et $A \subseteq L$ un sous-ensemble d'un ensemble L de LTS. A peut être soit un événement, soit une séquence. Cette méthode revient donc à extraire une branche (ou une partie d'une branche) du graphe d'accessibilité;

  - les automates (en particulier les IOLTS de TGV). Cette méthode revient donc à extraire un (sous-arbre) du graphe d'accessibilité.

- les méthodes statiques :
  Elles sont issues d'un traitement sur la spécification formelle :

  - dans [CGPT95], un but de test correspond à une transition du graphe syntaxique de la spécification formelle (SDL en l'occurrence). Une définition des graphes syntaxiques a été donnée dans la section 4.5.2;
  - dans [TR96], la méthode précédente est étendue par extraction de chemins depuis le graphe syntaxique. Dans cette approche, l'extraction d'un chemin est justifiée par le fait que de tels chemins "permettent de décrire des comportements de bout en bout ce qui est particulièrement adapté aux protocoles de haut-niveau".

Le problème avec les méthodes statiques issues d'un traitement sur la spécification formelle est que les objectifs de test ne sont pas complets. Le problème est qu'il faut instancier manuellement les paramètres des messages de tests avant d'obtenir des test exécutables [TR96]. A noter qu'à cette occasion, on retrouve le même type d'objection à utiliser cette méthode que pour la dérivation de tests (cf section 4.5.2). [TR96] propose de se servir des objectifs de tests incomplets pour tout de même piloter la génération du graphe d'accessibilité.

c) Les approches empiriques

- dans [Che96], un formalisme basé sur les expressions rationnelles de messages et d'états est proposé. Ce formalisme est doté des opérateurs classiques des expressions rationnelles et de quelques opérateurs spécifiques : LENGTH, MIN, LOOP qui permettent de spécifier respectivement la longueur, le minimum ou un préambule sans boucle. Ce formalisme est donc puissant en terme d'expressivité et se situe au niveau du modèle mathématique et aurait pu être classé dans la catégorie précédente.

- dans [GK96], le système expert SELEXPERT utilise une représentation privée (basée sur les objets) pour décrire de manière semi-formelle, les objectifs de test. Ce formalisme permet de spécifier des TP formalisant aussi bien des propriétés statiques que des propriétés dynamiques qui se subdivisent elles mêmes en propriétés comportementales et en propriétés temporelles.

- dans [Nah95], les objectifs de tests sont exprimés sous forme d'une séquence de messages en MSC [IT92g] (Message Sequence Charts). Un MSC décrit l'échange de messages entre processus sous la forme d'un ordre partiel. La méthode consiste ensuite à composer parallèlement l'arbre représentant le comportement avec le MSC par simulation. Dans [NGH93], les MSC sont utilisés pour spécifier des propriétés en logique temporelle (guarantee properties).

De manière générale, ces approches ne sont pas automatisables à partir d'une spécification BL et donc ne sont pas utilisables dans notre contexte. Il faut rappeler que l'objectif de ce travail

est de dériver des objectifs de tests automatiquement à partir de spécification BL ou de TIMS (de simulations TIMS plus précisément).

d) Les différentes approches pour la dérivation automatique des TP à l'aide de TIMS Cette section présente différentes suggestions pour la dérivation des TP automatiquement à partir de simulations TIMS. Il s'agit en fait d'examiner les différentes approches qui viennent d'être citées et de les appliquer dans notre contexte de travail. Nous nous sommes concentrés sur les approches génériques dynamiques car ce sont les seules qui nous permettent de générer des objectifs de tests complets de manière automatique ce qui constitue l'objectif initial de cette section.

- Le test de "transition unique" est de loin le plus simple à réaliser. Il s'agit de proposer comme objectif de test, chacun des messages observables (les messages déclencheurs qui traversent un PCO) lors de la construction du graphe d'accessibilité. Cette méthode génère cependant beaucoup d'objectifs de tests (et donc de cas de tests), c'est pourquoi nous ne la retenons pas.

- L'extraction d'une séquence de transitions depuis l'arbre exhaustif $Arbre_{exhaustif}$ est aussi très facile à réaliser. Si cette séquence commence par l'état initial et se termine par un état final, on parle alors de branche et donc d'extraction de branche de l'arbre exhaustif $Arbre_{exhaustif}$. C'est l'approche que nous comptons adopter.
  Le problème à résoudre devient celui de la sélection de cette branche car le nombre de branches peut être très important. Nous verrons dans le section qui présente la mise en oeuvre quelle approche nous avons adopter et comment nous proposons de faire une sélection pour l'extraction de branches pertinentes. On obtient alors l'équation suivante : $TP = f_{TIMS3}(Arbre_{exhaustif})$

5.2.2 Présentation de TGV

Le prototype TGV[1](pour Test Generation by Verification Technology) [FJJV96b, FJJV96a] permet de produire automatiquement des tests de conformité pour un système réparti, à partir d'un graphe d'accessibilité représentant les comportements possibles d'une spécification formelle du système à tester.

Un choix important de TGV est d'utiliser des algorithmes efficaces issus du domaine de la vérification. Cette approche rejoint en cela les approches décrites dans [ACM+96]. Un autre choix important de TGV est de modéliser formellement les différents objets et relations intervenant dans le test :

- En particulier comme les praticiens du test et contrairement à beaucoup de méthodes automatiques, il est utilisé des objectifs de test afin de sélectionner les cas de tests;

[1]Ce prototype a été réalisé grâce à une collaboration étroite entre les équipes de Vérimag Grenoble et Pampa de l'Irisa Rennes. Une partie du développement de TGV a été réalisée en utilisant des bibliothèques Open Caesar fournies par l'environnement CADP (Caesar Aldebaran Distribution Package) [FGK+96].

- Le modèle des systèmes de transitions étiquetés est utilisé pour modéliser la spécification, l'implantation, les cas de test, et les automates pour modéliser les objectifs de tests. La conformité est formalisée par une relation entre le modèle de l'implantation et le modèle de la spécification en s'inspirant d'une relation de conformité déjà définie dans [Tre92, Pha94].

Enfin, TGV produit des arbres de tests (pas seulement des séquences) dans le format TTCN.

5.2.2.1 Modélisation de la conformité

a) Les systèmes de transitions à entrées sorties Dans le cadre du test, il est utilisé un modèle dérivé des LTS, les IOLTS (pour Input/Output Labelled Transition Systems) dans lequel on distingue deux types d'actions, les entrées et les sorties. La raison fondamentale est que du point de vue du test, il faut distinguer les actions contrôlables (sorties) et les actions observables (entrées). En particulier le testeur contrôle ses émissions donc peut faire un choix sur celles qu'il envoie à l'implantation sous test. Mais il ne contrôle pas ses réceptions qui lui sont envoyées par l'implantation. Il doit donc considérer toutes les possibilités de réception afin de fournir un verdict correct. Etant donné que pour le test de conformité on ne s'intéresse qu'aux interactions entre testeur et testé, les actions internes seront indifférenciées. Elles seront toutes représentées par l'action τ qui n'est ni une entrée, ni une sortie.

Définition 11 *Un IOLTS est un LTS* $S = (Q, A, \rightarrow_s, q_0)$ *tel que A soit partitionné en deux sous-ensembles distincts* $A = A_I \bigcup A_O$ *avec* $A_I \bigcap A_O = \emptyset$ *où* A_I *est l'alphabet d'entrée et* A_O *est l'alphabet de sortie*

b) La spécification A partir de la sémantique formelle de la spécification, on peut décrire l'ensemble des comportements possibles de *Spec* par un LTS S_{init}[2].

Le LTS S_{init} comporte à la fois des actions internes et des actions visibles. Mais comme on ne s'intéresse qu'aux interactions avec l'environnement, on supposera que les actions internes sont indifférenciées et représentées par l'action τ . Si on partitionne l'ensemble des actions en actions de sorties et d'entrées, ce LTS peut alors être vu comme un IOLTS que nous noterons $S_{init} = (Q, A, \rightarrow_{S_{init}}, q_0)$ avec $A = A_I \bigcup A_O$.

En général les méthodes de génération automatique partent d'une spécification sous la forme de LTS (ou autre) qui ne comporte pas d'action interne. Il est supposé que cette spécification du comportement visible a été obtenue à partir de S_{init} par :

1. abstraction des τ
2. déterminisation au sens des traces
3. et éventuellement minimisation du LTS

afin d'obtenir le LTS S_{vismin} qui servira à la génération de tests.

Le LTS S_{vismin} peut par exemple être obtenu à partir de S_{init} grâce à l'outil Aldébaran [FGK+96], en trois phases (abstraction puis déterminisation puis minimisation). La version de TGV que nous

[2] comme cela est le cas avec les spécifications BL

allons utiliser, suppose que le graphe d'accessibilité qu'elle considère en entrée est un graphe S_{vis} qui ne comprend pas de τ et qui est déterministe. La minimisation n'est pas nécessaire mais est recommandée.

c) L'implantation C'est une boîte noire. Ce n'est pas un objet mathématique mais pour pouvoir définir formellement la conformité, il faut supposer que son comportement peut être modélisé par un IOLTS noté $I = (Q_I, A', \rightarrow_I, q_{0_I})$ avec $A' = A'_I \bigcup A'_O$ et $A_I \subseteq A'_I$ et $A_O \subseteq A'_O$.

d) L'objectif de test Tel qu'il est défini dans la norme ISO/9646, il décrit de manière informelle quel est le but particulier d'un cas de test. Son intention est donc de permettre de sélectionner un cas de test afin de tester un aspect particulier du comportement du système sous test. Il décrit généralement un enchaînement d'actions non nécessairement consécutives que l'on veut tester. Dans notre approche, un objectif de test sera un IOLTS noté $TP = (Q_{TP}, A, \rightarrow_{TP}, q_{0_{TP}})$ muni d'un ensemble d'états accepteurs $Accept \subseteq Q_{TP}$ lui donnant ainsi la structure d'automate d'état fini. Il permettra de reconnaître dans la spécification des séquences d'actions afin de sélectionner celles qui permettront de construire le cas de test.

e) Le cas de test Un cas de test est modélisé par un IOLTS noté $TC = (Q_{TC}, \overline{A}, \rightarrow_{TC}, q_{0_{TC}})$ avec $\overline{A} = \overline{A_I} \bigcup \overline{A_O}$ et $\overline{A_O} = A_I$ et $\overline{A_I} = A_O$. Il sera acyclique et sans τ . En plus de cette structure d'IOLTS, un cas de test sera muni d'une fonction qui à certains états attribue des verdicts. On distingue quatre types de verdicts.

- FAIL signifie que le test a détecté une erreur dans l'implantation.
- (PASS) est un verdict provisoire signifiant que l'objectif a été satisfait par la séquence d'actions courante.
- PASS est un verdict définitif pour le cas de test courant. Il est attribué après un verdict (PASS) quand on est revenu dans l'état initial. Une séquence entre des verdicts (PASS) et PASS est appelée postambule. Elle sert à enchaîner les tests en revenant à chaque fois dans l'état initial.
- INCONCLUSIVE lorsque l'on a ni PASS, ni FAIL.

f) La relation de conformité La relation de conformité est une relation permettant de dire si un IOLTS modélisant une implantation est correct vis à vis de la spécification. Plusieurs choix sont possibles et on pourrait être tenté de prendre une relation très discriminante telle que la bisimulation. Mais il faut aussi tenir compte des différences entre l'implantation et la spécification qu'un test peut réellement découvrir ainsi que des différences que l'on souhaite admettre. Cette relation doit aussi prendre en compte la différence entre actions contrôlables et actions observables. Il a été choisi une relation **ioconf** issue des travaux de [Pha94, Tre92]. Une implantation sera conforme à une spécification pour la relation **ioconf** si, après toute trace de la spécification, l'implantation ne peut produire que des sorties existant dans la spécification. Ceci s'exprime formellement par : I **ioconf** S ssi $\forall\, \sigma\ Traces(S), O(I \textbf{ after } \sigma) \subseteq O(S \textbf{ after } \sigma)$ où $O(P) = A_O \bigcap A(P)$

g) Principes de l'algorithme de génération de tests Les principes de l'algorithme de génération ont été publié dans [FJJV96b, FJJV96a]. Nous résumons ici les principes de la construction qui se compose de trois phases :

1. TGV utilise un objectif de test comme un automate d'état fini reconnaisseur, afin de sélectionner des séquences d'interactions de S_{vis} acceptées par l'objectif de test. TGV effectue l'image miroir de ces séquences d'interactions (entrées $\leftrightarrow$ sorties) pour les considérer comme des interactions du testeur. L'ensemble des séquences d'interactions est produit sous la forme d'un graphe dirigé acyclique (DAG) qui constitue le squelette du cas de test et qui comporte les verdicts associés à certaines actions.

2. Le DAG est ensuite décoré par la gestion des temporisateurs (timers) par modification des étiquettes de transitions.

3. Enfin le DAG résultant peut être traduit en TTCN.

5.2.3 Résumé

Cette section présente les principes de base de notre approche. Notre approche est en fait basée sur l'intégration de deux outils : le simulateur TIMS et l'outil de génération de tests TGV. Le simulateur TIMS doit générer d'une part un graphe d'accessibilité et d'autre part des objectifs de tests. L'outil de génération TGV génère à partir du graphe d'accessibilité pour chaque objectif de test un cas de test TTCN.

La construction de l'arbre d'accessibilité à partir de TIMS est basée sur la détection des messages déclencheurs (qui constitueront les transitions du graphe) et sur la décoration de ces transitions en fonction d'informations liées à l'architecture de test (la localisation des PCO et l'énumération des messages observables/contrôlables et des messages internes).

En ce qui concerne la génération des objectifs de test à partir de TIMS, c'est une solution qui vise à l'extraction de branches pertinentes depuis l'arbre exhaustif représentant le comportement dynamique qui sera adoptée.

5.3 Mise en oeuvre de notre approche

La mise en oeuvre de notre approche consiste donc d'une part en la génération du graphe d'accessibilité et d'autre part en la génération des objectifs de test.

5.3.1 Génération du graphe d'accessibilité

La génération du graphe d'accessibilité se décompose en deux phases :

- la phase I consiste en la modification du simulateur (cf 5.2) pour qu'il génère des données intermédiaires;

- la phase II consiste en la construction du graphe à partir de ces données intermédiaires.

Cet enchaînement est représenté par la Figure 5.2.

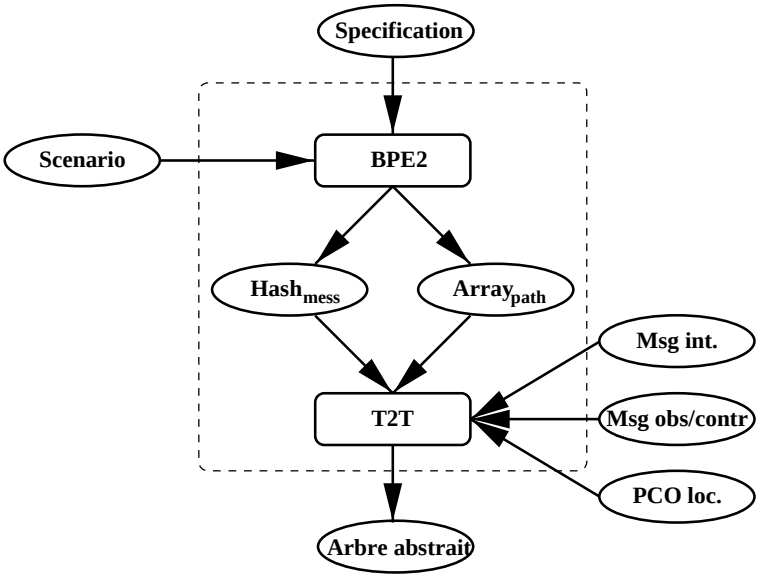

Figure 5.2: TIMS adapté à l'activité de test

5.3.1.1 Phase I : BPE2 (Récupération des simulations de TIMS)

On peut décomposer les modifications de BPE vers BPE2 en deux points :

- la détection des messages déclencheurs (donc des futures transitions du graphe d'accessibilité) et
- la construction de $Array_{path}$ la liste des messages déclencheurs qui ont été exécutés depuis l'état initial jusqu'à un état final (cad lorsque l'exécution est terminée, ce qui constitue un chemin du graphe d'accessibilité).

a) Détection des messages Elle se fait par modification de l'algorithme B.1.6 d'exécution d'un pas unitaire du simulateur TIMS et elle est donnée par l'algorithme C.1.1. On peut résumer cette modification par :

- l'ajout de chaque message *msg* exécuté dans la liste des messages $Stack_{mess}$ qui ont été détectés durant le parcours du chemin depuis l'état initial et ce jusqu'à la terminaison de l'exécution;
- l'ajout des nouveaux messages *msg* dans une table de hashing $Hash_{mess}$. On associe une clé (un entier unique) qui représente symboliquement chaque message et nous permet par la suite de ne traiter que des entiers.

b) Construction de $Array_{path}$ Elle se fait par modification de l'algorithme B.2.2 de simulation exhaustive et elle est donnée par l'algorithme C.1.2. On peut résumer cette modification par :

- initialisation des structures de données spécifiques à l'activité de test :$Stack_{mess}$, $Hash_{mess}$, et $Array_{path}$.

- à chaque terminaison d'exécution (lorsqu'il n'y a plus de BEN à exécuter), on ajoute $Stack_{mess}$ au tableau des chemins $Array_{path}$. L'ajout dans $Array_{path}$ n'est effectué que si $Stack_{mess}$ est un nouveau chemin.

5.3.1.2 Phase II : T2T (Génération du graphe d'accessibilité)

On peut résumer l'algorithme de T2T en trois étapes :

1. la construction du graphe d'accessibilité
2. la décoration des transitions
3. le formattage de sortie

a) Construction du graphe d'accessibilité Comme le montre la Figure 5.3, il s'agit de passer de la représentation de $Array_{path}$ à la construction du graphe d'accessibilité G_{init}.

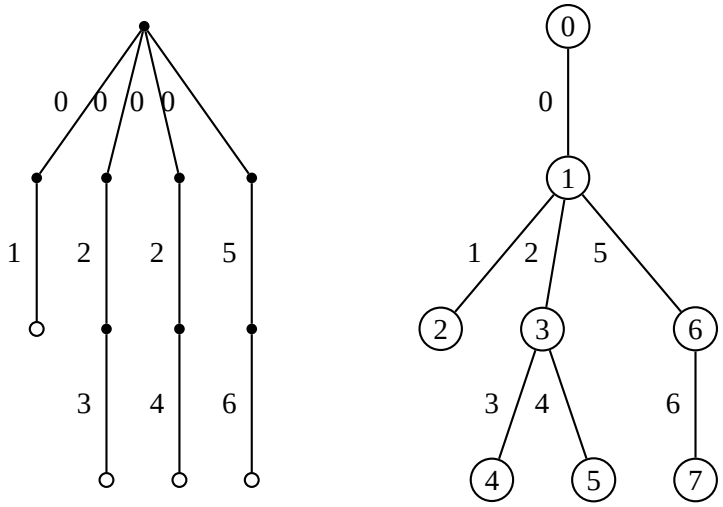

Figure 5.3: Construction du graphe d'accessibilité G_{init} depuis $Array_{path}$

Cette transformation est donnée par l'algorithme C.2.2 dans l'annexe C. En résumé, il s'agit d'un parcours en parallèle d'une branche de $Array_{path}$ et du graphe G_{init}. L'idée est de factoriser les même séquences de messages dans la même branche. La Figure 5.3 le montre pour les deux branches (0,2,3) et (0,2,4) qui conduisent à la construction du sous-grahe (0, 2, (3 | 4)).

C'est notamment à cette occasion que l'on attribue un entier à chaque état et qui donc rend explicite la notion d'état (du LTS). Toujours en suivant l'exemple de la Figure 5.3, on se retrouve dans l'état final 5 après la succession de messages (0,2,4).

b) Décoration des transitions La décoration des transitions (donnée par l'algorithme C.2.4) se sert d'informations spécifiques à l'activité de test comme la localisation des PCO. A partir de cette localisation, on en déduit les différentes listes :

- la liste des messages observables;
- la liste des messages contrôlables;
- la liste des messages internes.

A noter que ces informations sont dépendantes de l'architecture de test. Elles doivent être fournies par le testeur une fois pour toute au début de l'activité de test.

La décoration d'une transition est donnée par la règle suivante : On marque d'un ! les messages

transition	$\rightarrow$	PCO_name	! ou?	transition_name
	|		i	

observables, d'un? les messages contrôlables et d'un i les messages internes. On retrouve ici le format d'une transition telle qu'elle va être spécifiée dans le TTCN (cf le tableau 4.1 de la section 4.3.1.5).

c) Formattage de sortie Le formattage de sortie est donné par l'algorithme C.2.5. On parcourt simplement le graphe G_{init} interne résultant de l'algorithme C.2.1 pour générer l'IOLTS représentant la spécification dans le format aldébaran qui est donné ci dessous.

$$\begin{array}{ccccc} \textbf{des(} & q_0\textbf{,} & nb\_tr\textbf{,} & nb\_st\textbf{)} \\ & & \ldots & \\ \textbf{(} & q_i\textbf{,} & tr_{ij}\textbf{,} & q_j\textbf{)} \\ & & \ldots & \\ \textbf{(} & q_m\textbf{,} & tr_{mn}\textbf{,} & q_n\textbf{)} \end{array}$$

Figure 5.4: Format Aldébaran

- q_0 est l'état initial de l'IOLTS;
- nb_tr donne le nombre de transitions;
- nb_st donne le nombre d'états;
- $q_i, q_j, q_m et q_n$ sont des états;
- $tr_{ij} et tr_{mn}$ sont des transitions.

5.3.2 Génération des objectifs de test

Cette section présente la solution que nous avons adoptée pour la génération des objectifs de test. Comme nous l'avons vu dans la section précédente, elle consiste en l'extraction de branches de l'arbre de comportement exhaustif $Arbre_{exhaustif}$.

La stratégie qui consiste à dériver comme objectif de test chacune des branches faisant apparaître un comportement différent (en terme de fonctionnalité) semble être la plus utilisée. Cependant, plus le graphe d'accessibilité est grand, plus le nombre de branches est important. Il est de plus complètement inutile de générer des objectifs de test pour chacune des branches car beaucoup de ces branches sont équivalentes (du point de vue observationnel) du fait de l'entrelacement des actions. On appelle extraction de branches pertinentes la solution qui va nous permettre d'extraire les branches de l'arbre de comportement exhaustif qui sont fonctionnellement différentes.

La technique la plus connue qui s'attaque à l'élimination des branches entrelacées est appelée la méthode d'ordre partiel [WG93].

5.3.2.1 Une courte introduction aux méthodes d'ordre partiel

Les techniques de réduction par les méthodes d'ordre partiel s'attaquent à l'une des causes de l'explosion combinatoire : la représentation du parallélisme par entrelacement des actions. En effet, lorsque deux composants du système peuvent exécuter deux actions en parallèle, on reporte l'entrelacement de ces deux actions dans le graphe de comportement. Ces deux actions si elles sont indépendantes mènent au même état global(cf Figure 5.5).

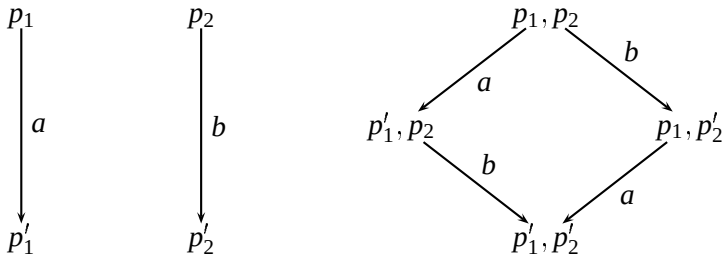

Figure 5.5: Entrelacement d'actions

Définition 12 *Formellement, deux actions a et b sont dites indépendantes ssi :* $\forall p, p_a, p_b$ *états :* $p \overset{a}{\Rightarrow} p_a \wedge p \overset{b}{\Rightarrow} p_b \Rightarrow \exists p'$ *état :* $p_a \overset{b}{\Rightarrow} p' \wedge p_b \overset{a}{\Rightarrow} p'$. *Deux actions non indépendantes sont dites en conflit.*

Grâce au travail de [Maz86], il est possible d'utiliser cette relation d'indépendance pour définir une relation d'équivalence sur les séquences d'actions du graphe de comportement : deux séquences d'actions sont équivalentes ssi elles peuvent être obtenues l'une de l'autre par permutations d'actions adjacentes et indépendantes.

La classe d'équivalence d'une séquence correspond à un ordre partiel sur les occurrences d'actions; toute extension linéaire de cet ordre représentant une trace dans la classe d'équivalence.

Les méthodes d'ordre partiel sont issues de l'idée de réduire le graphe de comportement grâce à l'équivalence de trace. Elles visent principalement à éliminer l'entrelacement.

On cherche pour cela à obtenir un sous graphe du graphe de comportement comportant le moins de traces équivalentes possibles [WG93]. Il existe deux types d'algorithmes :

- les "Persitent Set" [Val90];
- les "Sleep Set" [God95].

Ces algorithmes sont définis sous la forme d'ensembles d'actions qui expriment des relations de réduction précisées au moyen de critères locaux (cad attachés à chaque état). Ces deux types d'ensemble sont calculables en chaque état du système et permettent de limiter le nombre d'actions à explorer à partir de cet état. Ainsi, la construction du graphe réduit directement à partir de la description est possible. Ces deux approches sont compatibles et peuvent donc être combinées [GW91]. Alors que les "persitent sets" anticipent les entrelacements multiples, les "sleep sets" se servent de l'exploration déjà effectuée pour les éliminer.

5.3.2.2 Mise en oeuvre de la génération des objectifs de tests

En utilisant les ordres partiels, nous comptons extraire les branches sur des arbres réduits en réutilisant un algorithme implémenté dans le simulateur TIMS et présenté dans le détail dans [Sid97] : la technique des "Sleep Set".

- L'objectif de D. Sidou était de réduire l'arbre de comportement pour détecter dans un temps raisonnable les propriétés de terminaison ou de détecter la violation d'assertions. Cette technique préserve l'ensemble des états finaux, mais regroupe les branches identiques (en terme d'ordre partiels sur les événements indépendants) en une seule.
- Notre objectif (en réutilisant la technique des Sleep Set) est de réduire le nombre des branches indépendantes en une seule en opérant l'extraction à partir de l'arbre réduit obtenu par application de l'algorithme des Sleep Set.

Cette technique n'est pas parfaite car elle ne préserve pas l'équivalence observationnelle. Elle réduit (élimine) donc des branches qui auraient constitué des alternatives correctes (comme comportement du système). Il s'avère que dans la pratique seule une des alternatives constitue un objectif de test. Rappelons que le but de l'activité de test est de découvrir des erreurs, ce n'est pas de garantir la correction. Cette hypothèse est donc acceptable dans un premier temps. Nous verrons à la fin de ce chapitre comment améliorer cet état de fait notamment en préservant l'équivalence observationnelle dans la prochaine section (cf section 5.4).

On obtient ainsi un objectif de test pour chaque branche fonctionnellement différente. Il faut ensuite, pour chaque branche, enlever les actions internes, ce qui dans le contexte d'une branche est immédiat. A noter que l'on réutilise les fonctions de génération de l'arbre d'accessibilité pour la décoration des transitions et le formattage de sortie.

La méthode des "Sleep Set" et son implémentation dans TIMS Intuitivement, les "Sleep Set" vont garder la trace des exécutions déjà explorées dans le graphe réduit afin d'empêcher la construction d'exécutions équivalentes. Les "Sleep Set" sont donc construits au cours de l'exploration et contiennent les actions qui ne doivent pas être exécutées à partir de l'état atteint.

Définition 13 *(Sleep Set)*
Formellement, lorsqu'un état p est atteint par un état p' via une action a, l'état p hérite du Sleep set associé à p' de la manière suivante :
$Sleep \Leftrightarrow Set(p) =_{def} (Sleep \Leftrightarrow Set(p') \bigcup \{c \ / \ c$ *déjà exécuté de* $p\}) \setminus \{b \ / \ a$ *et* b *en conflit* $\}$.
La valeur initiale du "Sleep Set" à l'état initial est l'ensemble vide.

Sans entrer dans les détails, la simulation exhaustive avec les "Sleep Set" (inspirée de [God97]) est présentée par l'algorithme B.3.1. C'est en fait l'algorithme B.2.1 auquel on ajoute la gestion des "Sleep Set" et pour lequel on définit la relation de dépendance *Dop* entre deux transitions qui se base sur les opérations (modifications) bas niveau de l'IR (le conteneur d'instances) qui sont :

- (*create entry*)
- (*delete entry*)
- (*get entry property*)
- (*set entry property value*)

5.3.3 Résumé

Cette section vient de présenter les modifications qui ont été apportées au simulateur TIMS pour qu'il génère un graphe d'accessibilité et des objectifs de test conformes au format attendu par TGV. Ceci aurait pu constituer la conclusion de notre travail. Mais lors des expérimentations que nous avons menées pour valider notre approche (cf partie III), nos soupçons quant aux limites du simulateur TIMS à appréhender des simulations exhaustives compliquées se sont retrouvés immédiatement vérifiés. La section suivante énonce le problème, examine les alternatives pour la solution de ce problème et décrit l'approche que nous avons adoptée.

5.4 Le problème de la génération du graphe d'accessibilité

Dans un premier temps, cette section énonce le problème que nous avons rencontré lors de la génération du graphe d'accessibilité à partir d'exécutions compliquées. Elle présente dans un second temps l'approche que nous avons adoptée et termine par un ensemble d'alternatives auxquelles nous avons pensé mais qui n'ont pu être mises en oeuvre dans le contexte de cette thèse et qui constitue le sujet de travaux futurs.

5.4.1 Enoncé du problème

Une fois que les algorithmes ont été écrits, nous les avons exécutés sur le cas d'étude qui sera présenté dans la partie III. Dès lors que nous nous sommes retrouvés face à des exécutions compliquées (faisant intervenir un grand nombre de comportements concurrents), nous avons obtenu des temps de génération du graphe d'accessibilité désastreux. Nous nous attendions à ce type de problème car la génération du graphe d'accessibilité est un problème classique [TR96, JM97]. Nous pensions échapper à ce problème car nous n'effectuons pas des simulations complètement exhaustives (comme c'est classiquement le cas) mais des simulations basées sur des scénari.

D'ailleurs, comme le montre le Tableau 5.1, dans les cas simples, nous obtenons quand même des résultats. Dès lors que l'exécution est complexe, le temps est trop long (suivant le même Tableau 5.1, la simulation exhaustive qui mène à la génération du graphe d'accessibilité lors d'une exécution complexe a été arrêtée après 50.000 secondes).

	Exécution simple	Exécution moyenne	Exécution compliquée
DFS	90 s	2.245 s	??
DFS-SS	1,4 s	2,8 s	55 s

Tableau 5.1: Temps d'exploration du graphe d'accessibilité en fonction de la complexité de l'exécution et de l'algorithme DFS (simulation exhaustive) et DFS-SS (simulation exhaustive avec la méthode des "Sleep Set")

Une solution triviale qui consiste à augmenter la puissance de calcul de nos machines n'est pas envisageable car la complexité de notre problème est exponentielle. Il est donc préférable de proposer un meilleur algorithme.

Notre algorithme de simulation exhaustive est très inefficace car il parcourt autant de fois le même état qu'il y a de l'entrelacement entre des actions nondéterministes et donc il développe souvent des sous-arbres identiques. Une solution immédiate serait de stocker les états déjà rencontrés (afin d'éviter de re-parcourir le sous-arbre correspondant). Mais nous avons déjà vu (cf section 5.2.1.1) que c'est impossible dans notre cas car nous n'avons pas la possibilité de linéariser un BET, qui est une des deux composantes qui forment l'état du système (l'autre composante étant l'IR: le conteneur d'instances qu'il est par contre facile à linéariser).

L'approche intuitive

Si l'on se place dans le contexte de l'exemple donné dans la section 5.2.1.1 et que l'on observe un graphe d'accessibilité "caractéristique" donné par la figure 5.6, on s'aperçoit qu'il comprend beaucoup d'entrelacements de messages internes. Ceci est dû en fait à notre modélisation qui exprime la sémantique d'un message CVP par des messages IVP.

Cette observation est corroborée par les chiffres du tableau 5.1 qui montrent que la simulation exhaustive sur les méthodes d'ordres partiels ("Sleep Set") réussit à appréhender les simulations de complexité moyenne ainsi que l'exécution compliquée dans des temps raisonnables. Le problème est que nous ne pouvons pas appliquer cet algorithme car il ne conserve pas l'équivalence observationnelle. En d'autres termes, il supprime des comportements corrects, ce qui du point de vue du test n'est pas envisageable.

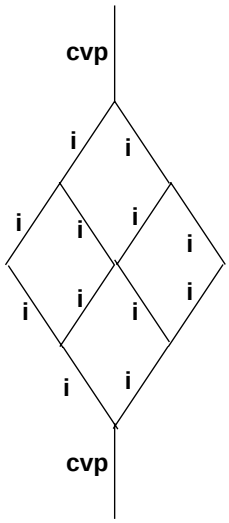

Figure 5.6: Graphe d'accessibilité caractéristique

La solution serait donc d'éliminer cet entrelacement local de transitions (messages) internes, ce qui ne change rien du point de vue de l'observabilité. Cette approche intuitive formalise l'idée présentée dans [VAM96] avec les "Graphes des Pas Couvrants". L'approche des "Pas Couvrants" consiste à regrouper, sous certaines conditions, certains événements "indépendants" en un pas de transitions et à considérer la réalisation de ce pas de transitions comme un événement atomique. On obtient ainsi une structure de Pas Couvrants. Le bénéfice potentiel réalisé en substituant un pas de calcul au sous-graphe engendré par l'entrelacement d'événements indépendants est localement exponentiel. L'intérêt de cette méthode est qu'en fonction d'une relation d'indépendance bien choisie, on peut obtenir des graphes de pas couvrants préservant l'équivalence observationnelle au système de transition initial.

5.4.2 L'approche choisie et sa mise en oeuvre

L'approche choisie constitue une alternative à l'approche intuitive qui vient d'être présentée dans la section suivante (graphe de pas couvrants préservant l'équivalence observationnelle).

Il s'agit d'appliquer l'algorithme des "Sleep Set" tout en maintenant l'équivalence observationnelle au système de transition initial. On conserve cette équivalence en augmentant la relation de dépendance Dop donnée dans la section B.3. On rend simplement dépendants les messages observables et contrôlables entre eux et on laisse indépendants les messages internes. Ainsi on introduit le fait que deux messages observables (ou contrôlables) dès lors qu'ils se produisent deviennent dépendants. On réduit ainsi un grand nombre de branches mais on conserve les branches qui sont observationnellement dépendantes.

5.4.2.1 Mise en oeuvre

La mise en oeuvre consiste juste en la spécification d'une nouvelle relation de dépendance. L'algorithme de construction de l'arbre réduit suivant les "Sleep Set" est le même qu'à la section 5.3.2. La seule modification (qui n'est pas détaillée) consiste à instrumenter les opérations sur les messages (observables et contrôlables) pour le calcul de la relation de dépendance.

Nouvelle relation de dépendance La nouvelle relation de dépendance consiste en l'ajout de la dépendance entre les messages observables et contrôlables.

Définition 14 *(relation de dépendance Dop+obs)*
La relation de dépendance Dop+obs est donnée par les trois règles qu'il faut appliquer en séquence :

1. *deux transitions sont dépendantes dès lors qu'elles appartiennent à l'ensemble des messages observables et contrôlables;*

2. *l'opération create et l'opération delete sont dépendantes avec toutes autres opérations sur la même entrée entry;*

3. *l'opération set est dépendante avec les opérations set et get sur la même propriété property d'une même entrée entry.*

5.4.2.2 Nouvelles Expérimentations

Nous avons dès lors mené une nouvelle série d'expérimentations (cf le Tableau 5.2) qui montrent que cette solution simple et pragmatique nous permet de générer un graphe d'accessibilité même pour des exécutions (simulations exhaustives) compliquées.

	Exécution simple	Exécution moyenne	Exécution compliquée
DFS	90 s	2.245 s	??
DFS-SS	1,4 s	2,8 s	55 s
DFS-SS+obs	3 s	11 s	5.730 s

Tableau 5.2: Temps d'exploration du graphe d'accessibilité en fonction de la complexité de l'exécution et de l'algorithme DFS (simulation exhaustive) et DFS-SS (simulation exhaustive avec la méthode des "Sleep Set") et DFS-SS+obs (simulation exhaustive avec la méthode des "Sleep Set" et nouvelle relation d'indépendance)

5.4.2.3 Utilisation de la nouvelle relation de dépendance pour la génération des objectifs de test

En utilisant cette nouvelle relation de dépendance (Dop+obs), ceci nous permet de construire un graphe réduit qui conserve l'équivalence observationelle. Extraire les branches de cet arbre nous permet de dériver tous les objectifs de test observationnellement différents, alors que la première

relation d'indépendance (Dop) ne conservait pas l'équivalence observationnelle et donc réduisait (éliminait) des objectifs de test qui pouvaient être pertinents.

Le point crucial est de déterminer si obtenir un plus grand nombre d'objectifs de test est une bonne chose car cela ne correspond pas à l'activité de test observée dans la pratique. Lorsqu'on génère tous les TPs possibles, on génère un grand nombre de cas de tests à exécuter. Seule la confrontation de nos résultats avec des équipes de testeurs va nous permettre de déterminer quelle stratégie il faut adopter.

5.4.3 Une alternative à notre solution à implémenter dans le futur

Une autre alternative qui vise à traiter le problème avant qu'il ne survienne serait intéressante à implémenter. Cette alternative se base sur l'intuition qu'ont eue les concepteurs de TGV que le problème de la génération du graphe d'accessibilité se pose systématiquement dès lors que l'on s'attaque à des cas d'études de taille réelle. L'idée présentée dans [JM97] est de générer les tests à la volée lors de la simulation exhaustive en évitant ainsi la construction totale du graphe d'accessibilité. Il s'agit par exemple de ne pas explorer les sous-arbres d'une transition qui, de toutes façons, conduit à un verdict "FALSE".

Il s'offre alors deux alternatives pour réaliser ce couplage :

- pilotage de la génération de tests par la simulation exhaustive :
  L'algorithme ("maître") de la simulation exhaustive pilote l'algorithme ("esclave") de génération de tests par une API qui serait offerte par le générateur de tests;
- pilotage de la simulation exhaustive par la génération de tests :
  L'algorithme ("maître") de la génération de tests pilote l'algorithme ("esclave") de simulation exhaustive par une API qui serait offerte par le simulateur. Cette approche est plus facile à mettre en oeuvre entre TGV et TIMS, car il a été défini dans [JM97], par les concepteurs de TGV, une API que peut offrir TIMS, nécessitant les fonctions suivantes :
  - $q_0 \leftarrow Init()$: qui rend l'état initial du LTS;
  - $\{t\} \leftarrow Fireable(q)$: qui rend la liste des transitions tirables $\{t\}$ pour un état q;
  - $q' \leftarrow Succ(q,t)$: qui rend l'état successeur q' obtenu par le tir de la transition t sur l'état q.
  - $(true|false) \leftarrow Comp(e,e')$ qui rend le booléen *true* si deux états e et e' sont égaux et $false$ sinon.

Cette approche n'a pu être mise en oeuvre car cette version de TGV n'est pas encore disponible. Elle constitue cependant une approche très élégante qui pourra être mise en oeuvre dans un futur proche.

5.4.4 Résumé

Cette section débute par poser le problème de la génération d'un graphe d'accessibilité lors de simulations exhaustives compliquées. Ce problème a été résolu en utilisant la technique des ordres

partiels des "Sleep Set" avec une nouvelle relation de dépendance qui nous permet de conserver l'équivalence observationnelle. Cette section montre ensuite comme cette solution est simple à implémenter (dès lors que l'on dispose de l'implémentation de l'algorithme des "Sleep Set" dans TIMS) et elle montre aussi que la solution est efficace lors de la génération de graphe d'accessibilité correspondant à des exécutions compliquées.

5.5 Conclusion

Ce chapitre a proposé dans sa première partie notre approche pour la génération de tests du comportement dynamique des systèmes à objets répartis. L'approche est basée sur l'intégration de deux outils : le simulateur TIMS et l'outil de génération de tests TGV. Le simulateur TIMS doit générer d'une part un graphe d'accessibilité et d'autre part des objectifs de tests. L'outil de génération TGV génère à partir du graphe d'accessibilité pour chaque objectif de test un cas de test TTCN.

Dans la seconde partie, ce chapitre présente les modifications qui ont été apportées au simulateur TIMS pour qu'il génère un graphe d'accessibilité et des objectifs de test conformes au format attendu par TGV. Cette première version pour la génération de tests nous permet de traiter des cas simples (des simulations exhaustives simples). Mais comme nous nous y attendions, lors des exécutions de complexité moyenne, elle montre ses faiblesses (le temps de génération du graphe d'accessibilité est très grand.

La troisième et dernière partie de ce chapitre présente notre solution pour résoudre le problème de la génération du graphe d'accessibilité. L'idée de base est d'utiliser la technique des ordres partiels des "Sleep Set" avec une nouvelle relation de dépendance qui nous permet de conserver l'équivalence observationnelle. L'avantage de cette solution est qu'elle est simple à implémenter et qu'elle se révèle très efficace sur des expérimentations que nous avons menées lors de la génération de graphe d'accessibilité correspondant à des exécutions compliquées.

Partie III

Application à la gestion des interfaces Xcoop

Chapitre 6

La simulation de modèles d'informations du RGT

6.1 Introduction

Jusqu'à présent le modèle de comportement a été exposé dans un contexte générique de systèmes d'objets répartis. En développant un modèle générique, ceci nous permet de spécifier le comportement dynamique pour des systèmes tels que : CORBA, JAVA, TINA, RGT ...

Le but de la partie III est d'appliquer les résultats de cette thèse dans un contexte réel issu du RGT : la gestion des interfaces Xcoop (Xcoop est un acronyme pour coopération inter-opérateurs à travers les interfaces X).

- La première partie de ce chapitre présente donc le RGT (Réseaux de Gestion des Télécommunications [TMN92b]) qui est le cadre de travail des interfaces Xcoop. Les communications de ces systèmes reposent sur les protocoles de la gestion de réseaux OSI, c'est pourquoi nous introduisons ensuite la gestion de réseaux OSI et particulièrement ses services de communications. L'objectif et l'ambition de ces deux présentations sont de donner les éléments suffisants pour la compréhension du cas d'étude pour les non initiés au RGT et à la gestion de réseaux OSI.

- La deuxième partie de ce chapitre présente l'environnement de simulation TIMS qui est un simulateur de comportement dynamique de spécifications du RGT. Cette présentation montre essentiellement les interfaces fonctionnelles du simulateur : l'interface de communication, l'interface de modèle d'information, l'interface des scénari et l'interface visuelle.
  Une attention particulière sera donnée à l'intégration du modèle d'information statique : GDMO, GRM et ASN.1, qui avec les spécifications BL qui spécifient le modèle d'information dynamique (présentées dans la partie I) doivent constituer le modèle d'information du RGT complet.

6.2 Une introduction au RGT

L'évolution structurelle des réseaux de télécommunications va actuellement dans le sens d'une claire séparation fonctionnelle des services de communication et de gestion. Il s'agit de faire interfonctionner ces deux domaines de service au travers d'interfaces de communication normalisées, et d'intégrer harmonieusement les divers systèmes de gestion par l'emploi d'outils et de méthode normalisés. Le CCITT a élaboré le concept de Réseau de Gestion des Télécommunications (RGT ou TMN pour Telecommunications management Network) pour définir une architecture fonctionnelle d'un système de gestion de réseaux souple, complet et évolutif [TMN92b].

Le RGT offre un cadre modulaire de développement de la gestion (grâce à la normalisation d'interfaces) dans lequel les opérateurs, les applications et les équipements de télécommunications communiquent de façon normalisée. Un des points importants de ce cadre architectural est la définition claire des responsabilités de chaque acteur. Les apports escomptés d'un RGT sont la suppression de la redondance inutile des informations de gestion, la facilité de dialogue entre les applications, l'accroissement des capacités d'évolution (ajout, intégration d'applications et d'équipements nouveaux), l'amélioration de la qualité de service ainsi que des gains de productivité. Le RGT permet d'automatiser toute la chaîne de gestion des réseaux et de favoriser une plus grande souplesse dans la gestion des opérateurs. La notion de RGT est purement fonctionnelle. Elle ne préjuge en rien de la taille et des particularités des implantations physiques la réalisant. Elle s'applique aussi bien aux réseaux de télécommunications publics (téléphone, téléphone de voiture, RNIS, réseaux intelligents), qu'aux réseaux de transmission (multiplexeurs, nouvelle hiérarchie synchrone) et aux réseaux privés (PABX, réseaux locaux d'entreprise, réseaux grande distance).

Le RGT est fonctionnellement distinct du réseau de télécommunications qu'il gère, interroge et commande, même s'il peut utiliser dans la pratique les ressources de ce dernier. Il est logiquement séparé de ce réseau qui peut être dédié à l'acheminement de la voix, des données ou de l'image.

6.2.1 Fonctions offertes par le RGT

[TMN92a] propose une liste non exhaustive des fonctions de gestion qu'un RGT doit supporter. Il s'agit des fonctions de gestion des anomalies (par le traitement des alarmes, la localisation des pannes et la réalisation d'essais), de gestion des informations comptables (par la collecte des relevés de compte et la gestion des paramètres de facturation), de gestion de la configuration (par la gestion des paramètres de configuration, d'installation, de mise en service et de la gestion des états et des commandes), de gestion de la charge et des performances du réseau (par observation du trafic et de la qualité de service, par la collecte de données générées par le comportement du réseau), et de gestion de la sécurité (par la protection du système contre les disfonctionnements et contre les accès non autorisés aux fonctions et aux données). Le Tableau 6.1 résume ces principales fonctions dénommées également aires fonctionnelles de gestion.

Fonctions	Rôles
Gestion des anomalies	surveillance par alarmes localisation des pannes essais et mesures (test)

Gestion des informations comptables	collecte des relevés de compte gestion des paramètres de facturation
Gestion de la configuration	gestion des paramètres de configuration gestion de la mise en service gestion des états gestion des commandes
Gestion des performances	collecte des données gestion du trafic gestion de la qualité de service
Gestion de sécurité	protection authentification habilitaton

Tableau 6.1: Aires fonctionnelles de gestion du RGT

6.2.2 Architecture fonctionnelle

Comme nous venons de le voir, l'architecture du RGT est fonctionnelle. Il s'agit d'une architecture modulaire constituée de groupements fonctionnels dédiés à la réalisation des tâches particulières relatives au transport et au traitement des informations de gestion.

6.2.2.1 Groupements fonctionnels

Le groupement fonctionnel OSF (Operations Systems Function pour fonction de système de gestion) traite l'information de gestion pour surveiller, ou pour effectuer en partie ou en totalité, une fonction de gestion.

Les groupements fonctionnels MF (Meditation Function pour fonction de méditation) et QAF (Q Adaptor Function pour fonction d'adaptation d'interface Q) participent au transfert d'informations entre l'OSF et le NEF (Network Element Function pour fonction d'éléments de réseau).

La fonction de médiation comprend d'une part une fonction de conversion d'informations qui traduit un modèle de données dans un autre et modifie ainsi un contenu des messages d'informations de gestion (fonction minimale et caractéristique d'un groupement fonctionnel de médiation), une fonction de conversion de protocoles d'autre part, et éventuellement des fonctions complémentaires (journalisation, filtrage, concentration d'informations).

Si un équipement présente une interface de gestion ne répondant pas aux normes de gestion OSI, il est nécessaire d'utiliser la fonction d'adaptation d'interface Q (QAF) pour effectuer la traduction entre le "langage OSI" et celui spécifique à l'équipement.

Le groupement fonctionnel d'éléments de réseau (NEF) communique avec le RGT pour être dirigé et surveillé. Il faut noter que le RGT interagit avec la NEF et la QAF, mais n'inclut pas ces fonctions en tant que composantes puisqu'il ne s'agit que d'un aspect des équipements (leur gestion).

Le groupement fonctionnel WSF (WorkStation Function pour fonction de poste de travail) permet à l'opérateur de communiquer avec la fonction de médiation (MF) et la fonction de système de gestion (OSF). Il est caractérisé par la fonction de présentation qui traduit l'information en

provenance des machines dans une forme compréhensible par un opérateur humain et vice versa. Le RGT interagit avec la fonction de poste de travail (WSF), mais ne l'inclut pas en tant que composante à proprement parler.

Toutes ces fonctions font appel à la fonction de communication (DCF Data Communication Function) qui correspond aux services offerts par les couches 1 à 3 du modèle OSI.

6.2.2.2 Points de référence

Les points de référence (cf Figure 6.1) sont des points de passage d'informations entre des groupements fonctionnels. Un point de référence devient une interface lorsque les groupements fonctionnels connectés sont réalisés et supportés par des équipements différents.

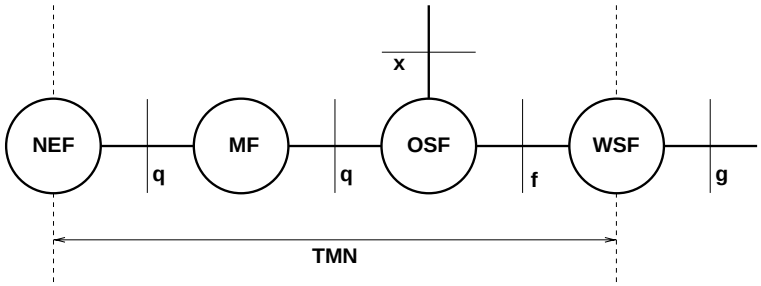

Figure 6.1: Les points de référence du RGT

En ce qui concerne le choix des noms des points de référence, la lettre f a été choisie car elle précède la lettre g (g comme graphique), retenue pour l'interface homme-machine. La lettre x a été retenue car au-delà du point de référence x s'étend l'inconnu, souvent désigné par x en mathématiques. Quant à la lettre q, il semblerait qu'elle ait été choisie un peu par hasard : à l'époque du choix, elle était disponible.

6.2.3 Architecture physique et interfaces de communication

A l'architecture fonctionnelle préalablement déclinée correspond une architecture physique. Un système physique peut réaliser plusieurs des groupements fonctionnels OSF, QAF, MF, NEF, ou WSF. Par ordre de priorité, si un système physique contient

- la NEF, il est qualifié d'élément de réseau (NE Network Element);
- la MF, il est qualifié d'entité de médiation (MD Meditation Device);
- l'OSF, il est qualifié de système de gestion (OS Operations System);
- la WSF, il est qualifié de poste de travail (WS Work Station).

L'interface correspondant à un point de référence est désignée par la même lettre, mais en majuscule et non plus en minuscule (interfaces Q, F, X et G). L'interface de type Q regroupe

les interfaces Q3 et Q_x. L'interface Q_x est un sous-ensemble de l'interface Q3. L'adaptateur d'interface Q est un dispositif devant mettre à niveau, à la fois en termes de protocole et de modèle de données, un équipement n'ayant pas d'interface de type Q en mode natif; par définition, il ne réalise que la QAF.

Selon les recommandations Q.811 et Q.812, qui définissent les protocoles pour les interfaces Q3, deux types de besoins ont été identifiés :

- l'un correspondant à la manipulation de petits volumes de données; c'est l'élément de service CMISE qui a été choisi. CMISE va être présenté dans la section suivante 6.3;
- l'autre au transfert des données en masse; c'est le protocole normalisé de transfert de fichiers FTAM (File Transfert, Access and Management).

6.2.4 Application de gestion

6.2.4.1 Notion de domaine

Pour des besoins de gestion, il peut être nécessaire de partionner l'ensemble des objets gérés selon divers critères (organisationnels, fonctionnels, géographiques, technologiques). On peut, par exemple, avoir des centres d'exploitation pour l'ensemble du réseau spécialisés dans un certain type d'équipements. Un tel choix peut ne pas être stable dans le temps, mais est dépendant des évolutions de la configuration du réseau et des entités organisationnelles (sociales et humaines). Le regroupement des objets pour réaliser une telle partition définit le domaine de gestion. Les domaines peuvent être disjoints, inclus, intersécants ou interagissants. Cette notion de domaine est particulièrement intéressante dans le cas des grands réseaux, où une gestion à étages (régionale, nationale, internationale) est courante.

6.2.4.2 Structure hiérarchique en couches

Le modèle en couches vise à définir une allocation des fonctions de gestion de réseaux structurée, modulaire et efficace. En général une couche contient des objets plus synthétiques que ceux de la couche inférieure. La répartition des objets selon les différentes couches définit la structure en couches, et à l'intérieur d'une couche, on regroupe les objets gérés par la même OSF dans un domaine d'OSF. Ce domaine peut lui aussi contenir d'autres domaines. On obtient ainsi une structure récursive.

Une mise en service fréquente de ce type de structure consiste à définir quatre couches principales :

- gestion des éléments de réseau (Network Element Management Layer);
- gestion du réseau (Network Management Layer);
- gestion des services (Service Management Layer);
- gestion commerciale (Business Management Layer).

La couche de gestion des éléments de réseau regroupe des entités responsables d'un ou de plusieurs éléments de réseau et de leur environnement (énergie, détection d'incendie, ... etc). Ces entités gèrent la charge des éléments en planifiant les actions (optimisation du séquencement des ordres, gestion des demandes contradictoire, ... etc). Cette couche met à disposition de la couche supérieure les compte-rendus d'exécution. Elle peut de plus effectuer des pré-traitements pour agréger les données et présenter une vue plus synthétique à la couche supérieure.

La couche de gestion du réseau est concernée par les aspects "réseau" de la gestion (acheminements, adressage, ... etc). Elle offre une vision unifiée (indépendante des types d'équipements) et permet l'installation, la suppression et la modification des capacités du réseau pour supporter les services offerts aux clients.

Le rôle principal de la couche de gestion des services est de servir d'interface technique envers les clients des services. Cela recouvre en particulier des aspects contractuels lors de la programmation et de l'attribution des services. Cette couche peut se subdiviser en deux sous-couches : l'une de gestion des services support, l'autre de gestion des services à valeur ajoutée.

Quant à la couche commerciale, elle a pour rôle d'utiliser la couche de gestion des services au mieux des intérêts économiques de l'organisation. Elle prend en compte en particulier les aspects marketing, commerciaux et législatifs pour la gestion du réseau.

6.2.5 L'interface X

L'interface X permet de dialoguer avec le RGT depuis l'extérieur; Il offre ainsi une certaine visibilité accompagnée de possibilités d'actions (limitées) sur le réseau. La sécurité est un point crucial dans cette ouverture qu'il faut savoir maîtriser.

Les interfaces X entre opérateurs recouvrent l'échange des informations techniques entre opérateurs. Cependant les données de nature commerciale ne sont pas exclues de ce transfert (guichet unique par exemple). Le cas d'étude qui sera présenté dans le prochain chapitre porte sur ce type d'interfaces.

Les interfaces avec les clients permettent une ouverture du RGT aux entreprises. Ceci apporte la sûreté des services (qui est accrue si l'entreprise peut réagir elle même à une panne) et la réduction des coûts (une attitude de "self-service" est moins coûteuse). Par conséquent, l'ouverture vers le client nécessite l'échange d'informations de service entre opérateur et client. De manière plus spécifique, on distingue :

- la remontée d'alarmes réseau;
- l'émission par le client de rapports d'anomalie et de requêtes de correction (la demande la plus pressante des clients semble porter sur la gestion des fautes; En effet, les clients ont souvent une vision plus fine de leurs problèmes que l'opérateur qui agit généralement de façon plus globale sur un environnement plus complexe);
- la remontée des informations de taxation (en temps réel et en temps différé);
- la visualisation du trafic, en cas de réseau privé virtuel;
- les essais et mesures;

- les reconfigurations en cas de réseau privé virtuel.

6.3 La gestion de réseaux OSI

Notre présentation de la gestion de réseaux OSI va se dérouler de la manière suivante. Nous présentons dans un premier temps le cadre architectural de gestion de réseaux OSI et nous présentons dans un second temps les communications (le protocole et les services) dans la gestion de réseaux OSI.

6.3.1 Le cadre architectural

Le cadre architectural est défini par [FRM]. La gestion OSI est une application. Puisque l'environnement à gérer est distribué, les composants individuels des activités de gestion sont eux aussi répartis. Les processus de gestion exécutent donc leurs activités de façon répartie et coopérante. Ils communiquent entre eux en utilisant des services de divers éléments de service d'application ou (ASE Application Service Element).

On distingue deux types de processus de gestion.

1. Les processus gestionnaires ont la responsabilité d'une ou de plusieurs activités de gestion. Ils émettent des opérations et reçoivent des notifications (résultant éventuellement d'opérations).

2. Les processus agents exécutent les opérations de gestion sur les objets gérés (situés dans la base d'information de gestion). Ils peuvent émettre des notifications d'événements.

Un même système ouvert peut supporter des processus gestionnaires et/ou des agents. Le regroupement des processus gestionnaires et agents en domaine fonctionnel de gestion ou (MD Management Domain) correspond à des choix organisationnels de gestion de réseau. Un MD est géré par une autorité privée ou publique de gestion. Les domaines fonctionnels de gestion peuvent être disjoints ou non. Seules les interactions, entre et à l'intérieur des domaines fonctionnels de gestion sont normalisées.

L'ensemble conceptuel des informations de gestion d'un système ouvert est la base d'information de gestion ou MIB (Management Information Base). La base d'information de gestion est définie comme un ensemble d'objets gérés accessibles par l'interface de gestion.

Un système ouvert doit être capable d'identifier correctement les constituants de sa base, de définir une structure logique pour les objets qu'il manipule, de comprendre les actions à exécuter sur les objets et les événements qu'elles peuvent générer.

6.3.1.1 GDMO : Guidelines for the Definition of Managed Objects

Nous ne présentons qu'un rapide survol du langage GDMO, pour plus de détails, le lecteur peut se référer à la norme [GDM]

Afin d'uniformiser la modélisation des ressources, l'ISO a retenu une approche orientée-objets pour la modélisation des ressources de gestion. Cette approche s'appuie sur un ensemble de

concepts de modélisation (héritage, encapsulation, ... etc) et comprend une notation normalisée appelée GDMO (Guidelines for the Definition of Managed Object pour "directives pour la description des objets gérés"). GDMO est utilisé comme technique de description (pseudo)-formelle pour la spécification des interfaces de gestion des ressources gérées.

Le langage GDMO fournit un ensemble de neuf formulaires permettant une description modulaire et normalisée des objets gérés et de leur organisation au sein d'une base d'information de gestion :

- le formulaire de classe d'objets gérés :
  Il permet de spécifier des classes d'objets gérés en définissant les paquetages qui les composent et en les liant aux super-classes dont elles héritent.

- le formulaire de paquetage (package) :
  Il est utilisé pour définir les paquetages en spécifiant pour chacun d'eux les attributs, actions, notifications et le comportement qui les composent.

- le formulaire d'attribut :
  Il permet de définir des attributs types.

- le formulaire de groupe d'attributs :
  Il permet de regrouper sous une étiquette un certain nombre d'attributs. Un groupe peut par la suite être référencé dans un paquetage.

- le formulaire d'action :
  Il permet de définir des actions autorisées sur des objets gérés.

- le formulaire de notification :
  Il permet de spécifier des notifications pouvant être émises par des objets gérés.

- le formulaire de paramètre :
  Il permet de définir des paramètres d'attributs, d'actions ou de notifications.

- le formulaire de comportement :
  Il permet de définir et d'enregistrer sous une étiquette, la définition d'un comportement.

- le formulaire de corrélation de noms (name binding) :
  Il est utilisé pour spécifier les liens de contenance entre les différents objets gérés au sein d'une base d'information de gestion.

Le lien entre les différents formulaires se fait toujours au travers de références. Par exemple les attributs utilisés par un formulaire de paquetages font référence par le champ attribut de ce formulaire à des formulaires d'attributs.

On définit GDMO comme une technique de description (pseudo)-formelle car la description des comportements des objets gérés (via leur formulaire) se fait en prose (par une description en langue naturelle).

6.3.1.2 ASN.1

ASN.1 (Abstract Syntax Notation One) [ISO87a] est le langage formel de spécification des unités de protocole de la couche application retenu par l'ISO. Ce langage possède des compilateurs capables de générer des structures de données spécifiques au système qui les supporte. Ainsi tous les protocoles - incluant la description d'objets gérés, partie intégrante du protocole - utilisent ASN.1. L'ensemble des unités de protocole d'un (ou de plusieurs) éléments de service de la couche application (CMISE, ROSE, ACSE [ACS]) constitue une syntaxe abstraite enregistrée comme OBJECT-IDENTIFIER (type ASN.1). Ainsi sur une Association, les entités homologues connaissent les syntaxes abstraites qui seront utilisées.

Puisque la représentation interne des données dépend du système qui les supporte il est nécessaire de convenir d'un mode de représentation des données lors du transfert (représentation commune concrète). Le codage retenu est le BER (Basic Encoding Rules) [ISO87b]. C'est un codage de type TLV (Type, Length, Value pour Type, Longueur, Valeur). Une syntaxe abstraite doit toujours être accompagnée d'une syntaxe de transfert. Le couple (syntaxe abstraite, syntaxe de transfert) constitue un contexte de Présentation.

Ainsi pour une APDU (Application Protocol Data Unit) transmise par pointeur à l'entité de Présentation, cette dernière reconnaissant la structure logique (abstraite) de l'APDU, est capable de la transmettre selon la syntaxe de Transfert reconnue. L'entité de Présentation réceptrice reconnait l'APDU et la délivrera par pointeur, sous la structure spécifique du système, à l'entité d'Application concernée.

Tous les types ASN.1 sont reconnaissables par une valeur d'étiquette. On peut citer :

- les types primitifs {integer, boolean, bit-string, object-identifier, ... etc}
- les types construits {sequence (record de Pacal), sequence-of (array de Pascal), ... etc}
- les types prédéfinis {numeric-string, time, ... etc}

Ces valeurs d'étiquettes sont véhiculées par la syntaxe de transfert comme spécification de type de valeur transmise.

6.3.2 Le protocole et ses services

Un élément de service d'application permettant d'effectuer des échanges d'information de gestion sous la forme de demande et/ou de demande/réponse a été normalisé. Il s'agit de CMISE (Common Management Information Service Element). Le protocole est défini par [CMIa] tandis que les services sont définis par [CMIb]. C'est un véhiculaire de base qui offre aux applications particulières de gestion les moyens de réaliser, sur des objets gérés, des opérations de gestion et de rapporter des notifications.

Les six services normalisés autorisant la manipulation d'informations de gestion sont listés dans le Tableau 6.2 :

Services	Type	Fonctions
M-EVENT-REPORT	C/NC	fait part d'un événement survenu pour un objet géré

M-GET	C	Demande d'information de gestion
M-SET	C/NC	Modification d'information de gestion
M-ACTION	C/NC	Réalisation d'une action particulière sur un objet géré
M-CREATE	C	Création d'un objet géré
M-DELETE	C	Suppression d'un objet géré

Tableau 6.2: Les services CMIS

Comme nous le verrons plus tard dans la section 6.3.1.1, les informations de gestion sont vues comme un ensemble d'objets gérés, possédant chacun des attributs sur lesquels les événements et des actions définis peuvent survenir. Les noms de ces objets gérés sont hiérarchiquement organisés dans un arbre d'information de gestion (ou MIT pour Management Information Tree). La manipulation d'un objet géré particulier passe par sa sélection, sa localisation et son accès dans l'arbre d'information de gestion. Pour cela les mécanismes de profondeur (scoping) et de filtrage (filtering) sont définis.

- le scoping permet de localiser le (les) niveau(x) hiérarchique(s) de l'arbre d'information de gestion où se trouve(nt) l'(les) objet(s) sujet(s) à manipulation.
- le filtering autorise le test de la présence (ou de l'absence) de certaines valeurs d'attributs d'un objet géré préalablement scoppé afin de pouvoir le sélectionner.

Ces mécanismes peuvent être complétés par l'utilisation d'une procédure de contrôle d'accès pour chaque objet. Elle peut être mise en oeuvre pour donner la permission de rechercher des valeurs d'attributs, de les modifier, d'effectuer des actions ainsi que pour créer ou supprimer un objet.

Si plusieurs objets ont été sélectionnés par les processus précédemment décrits, l'utilisateur de service CMISE doit pouvoir spécifier la façon dont les opérations doivent être synchronisées. Il existe deux types de synchronisation :

- la synchronisation atomique qui assure que l'exécution des opérations n'est appliquée sur l'ensemble des objets que si toutes les opérations peuvent être exécutées.
- La synchronisation dite au mieux, par opposition, permet d'exécuter au moins les opérations réalisables même si toutes ne le sont pas.

Les additifs 1 et 2 aux documents ISO/9595 et ISO/9596 étendent l'ensemble de base des services communs. Les extensions à CMIS/CMIP sont :

- Le service d'annulation (M-CANCEL-GET) est une extension du service de consultation (M-GET) qui autorise un émetteur à demander, de la part du fournisseur de service CMISE, l'annulation de l'envoi de nouvelles réponses à une demande de service de consultation M-GET préalablement effectuée. Elle représente la volonté de l'émetteur de ne plus recevoir de résultats (capacité de stockage saturée, les réponses déjà obtenues le satisfont, etc ...).
- L'extension du service M-SET autorise l'ajout/suppression de valeurs à l'ensemble des valeurs qui représentent un attribut. Cela accorde une meilleure description du comportement des attributs et une plus grande souplesse d'utilisation du service de mise à jour.

6.4 Le simulateur de modèles d'information du RGT : TIMS

TIMS est un simulateur de modèles d'information du RGT. Il se compose d'un noyau générique pour la simulation dont le composant essentiel est le BPE (qui a été présenté dans la section 3.4) et de quatre interfaces dédiées au RGT :

- l'interface de modèle d'information;
- l'interface de communication;
- l'interface des scénari;
- et l'interface de visualisation graphique.

La figure 6.2 donne une vue fonctionnelle de l'environnement de simulation TIMS.

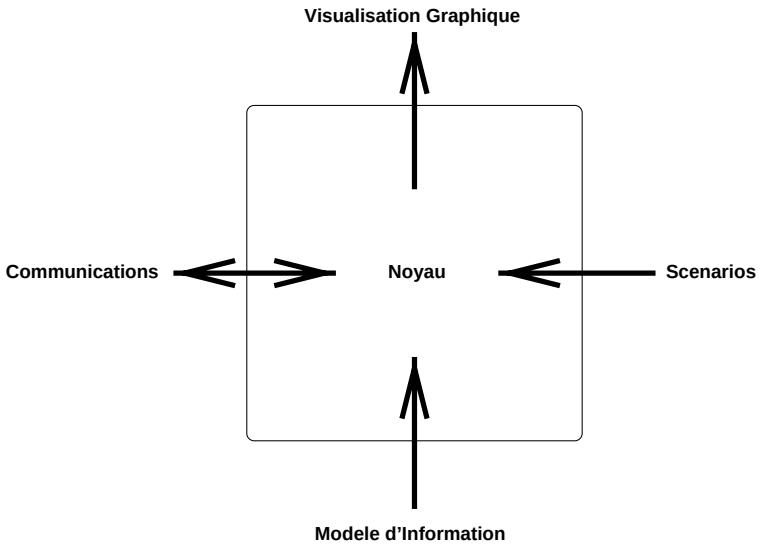

Figure 6.2: Vue fonctionnelle du simulateur

Cette section présente tour à tour chacune de ces interfaces.

6.4.1 L'interface de modèle d'information

Le modèle d'information se décompose en deux parties : (i) un modèle d'informations dynamique et (II) un modèle d'informations statique.

- le modèle d'informations dynamique se compose des spécifications BL (présentées dans la partie I).
- le modèle d'informations statique, dans le contexte du RGT (mais aussi plus généralement dans la gestion OSI), se compose de trois spécifications : les spécifications GDMO, ASN.1 et GRM.

6.4.1.1 GRM

GRM [GRM](Generiq Relationship Model) est un modèle pour l'expression des relations entre objets. Des exemples de relations spécifiées avec le GRM seront donnés dans le chapitre suivant. Le GRM permet de décrire des relations génériques basées sur le concept de rôle. Chaque entité représente un ou plusieurs objets. La syntaxe de GRM s'apparente à celle de GDMO. Une spécification GRM est composée de deux gabarits : le gabarit de classe de relation et le gabarit de "mapping" de relation.

a) Le gabarit de classe de relation se compose de plusieurs champs :

- un champ qui exprime l'héritage éventuel avec d'autres classes de relation;
- un champ pour l'expression du comportement (à la GDMO cad en prose);
- un champ qui énumère les opérations supportées par la relation (cf tableau 6.3);
- un champ de description des rôles. Pour chaque rôle, on peut définir la classe GDMO dont il dépend, les différentes contraintes de cardinalité et le comportement dynamique quant à la relation (cad sa possibilité à se joindre ou à se retirer de la relation);
- un dernier champ contraint le nombre d'instances qu'une classe de relation peut supporter.

Services	Fonctions
ESTABLISH	établissement de la relation
TERMINATE	terminaison de la relation
QUERY	intèrrogation de la relation
NOTIFY	notification de la relation
USER-DEFINED	action spécifique définie par l'utilisateur

Tableau 6.3: Les services du GRM

b) Le gabarit de "mapping" de relation Ce gabarit décrit deux types d'information :

- Il décrit d'une part comment "mapper" les services du GRM par rapport aux services CMIS car les services du GRM n'ont pas été prévus dans CMIP. Si on veut les intégrer à moindre coup dans un contexte de gestion OSI, ce mapping est nécessaire.
- Il décrit d'autre part comment le rôle est implanté par rapport à un contexte de gestion OSI. On exprime comment la relation explicite du GRM est implicitement définie dans un contexte de gestion OSI. Quatre types de relation sont dénotés :
  - la relation de contenance basée sur les Name Binding;
  - la relation entre objets qui font références à d'autres objets via des attributs qui jouent le rôle de pointeurs;

– la relation implicite qui s'instancie par des actions CMIS munies d'une sémantique additionnelle;

– les objets gérés GDMO de relation (qui ne modélisent pas une ressource par exemple).

6.4.1.2 Intégration du modèle d'informations statique dans TIMS

Les spécifications sont celles issues de la standardisation. Pour les intégrer, nous avons choisi de les transformer sous la forme de repository (conteneur) implémenté en Scheme. Comme le montre la figure 6.3, ces repository sont générés automatiquement par les analyseurs lexico-syntaxiques (parser).

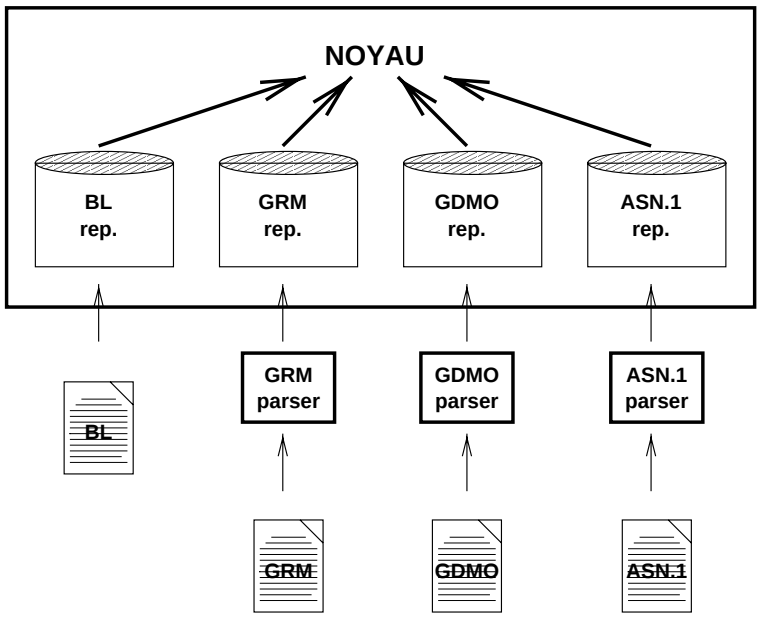

Figure 6.3: L'interface de modèle d'information

Nous avons développé le parser de GRM avec les outils BISON et FLEX. Le parser de GDMO a, quant à lui, été récupéré de l'outil OSIMIS [PMB+95]. Le cas d'ASN.1 a été beaucoup plus délicat. Son traitement s'est effectué en trois étapes :

1. le parser ne génère pas un repository, mais un snapshot. Celui-ci est en fait un fichier binaire de codage et de décodage BER. La génération de ce fichier est faite par la première passe d'un interpréteur ASN.1 : l'outil ASNFVT. Cette première passe est appelée dans la figure 6.4 COD/DECOD. Cette étape est exécutée une fois lors de l'initialisation du système.

2. la deuxième passe concerne la manipulation des valeurs ASN.1. Principalement cette étape transforme des valeurs ASN.1 standardisées sous forme de chaîne en leur équivalent (que nous avons inventé) sous forme Scheme et vice-versa (fichier VAL.SCM dans la Fig 6.4). La couche Scheme a été ajoutée car il n'est pas très facile de manipuler les valeurs ASN.1 sous forme de chaîne.

Cependant la manipulation des valeurs ASN.1 sous forme Scheme s'est elle aussi révélée délicate, c'est pourquoi nous avons ajouté des opérateurs (comme Make-Sequence, Make-Choice, GetFied, ...) à notre version Scheme du langage ASN.1 (fichier OPE.SCM dans la Fig 6.4). Ces opérateurs permettent à l'utilisateur de ne pas manipuler des listes, ce qui facilite grandement l'utilisation.

3. la troisième et dernière passe consiste en la transformation de l'ASN.1, sous sa forme standardisée (cad chaîne) en sa valeur de transfert BER par l'interpreteur ASNFVT [ASN] (cf Fig 6.4). Cette transformation se sert bien entendu du snapshot généré lors de la première passe. A noter que la transformation finale est une valeur PE qui est le format utilisé par l'interface de communications (OSIMIS/ISODE) qui sera décrite dans la section 6.4.2.

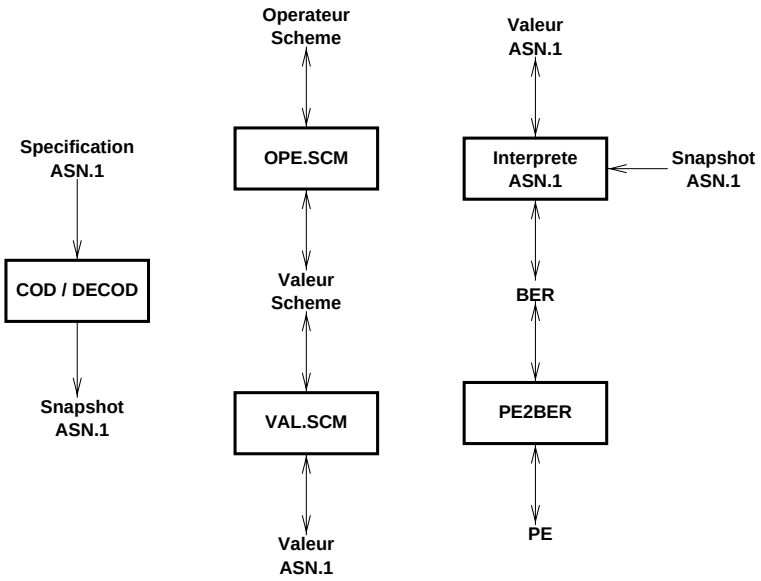

Figure 6.4: Manipulation de l'ASN.1 dans le simulateur

6.4.2 L'interface de communications

L'interface de communications permet au simulateur (lorsqu'il émule un ou plusieurs objets de traitement) de dialoguer avec d'autres objets de traitement. Dans le contexte du RGT, c'est donc une interface Q3 (CMIP) dont est équipé le simulateur. Comme le montre la figure 6.5, l'interface de communications dialogue avec le noyau via une API CMIS écrite en Scheme. Cette API invoque des services CMIS provenant de la librairie C "msap" issue d'Osimis [PMB+95]. Cette librairie est elle-même liée à Isode [ROR91] qui fournit la pile OSI. Nous utilisons en fait l'implémentation du RFC1006 qui se sert de TCP/IP comme protocole de transport.

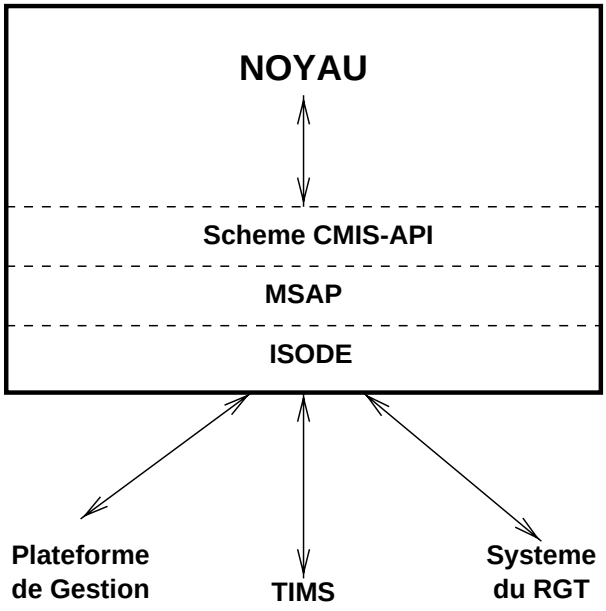

Figure 6.5: L'interface de communications

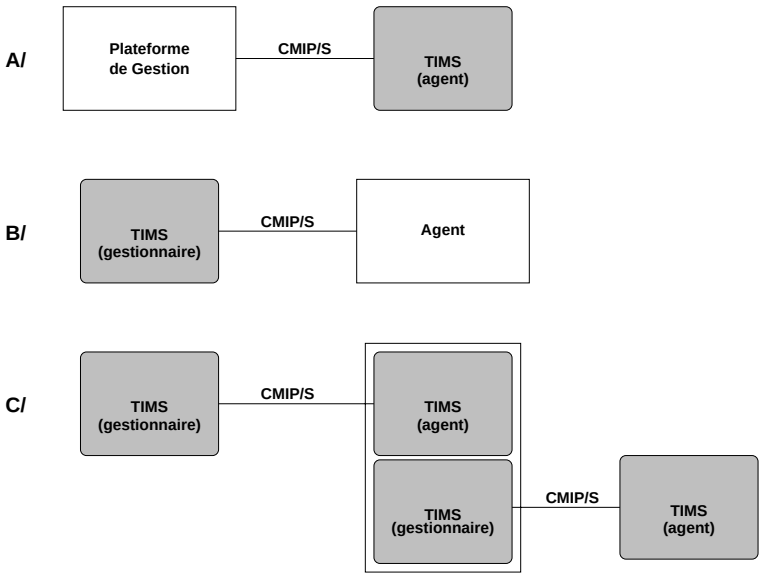

Figure 6.6: Les différentes configurations de simulation

6.4.2.1 Les différentes configurations de simulation

Cette section présente rapidement les différentes configuration possibles du simulateur. Comme le montre la figure 6.6 l'ajout de l'interface de communication nous permet d'utiliser TIMS suivant différentes configurations de simulation :

- émulation d'un agent OSI (cas A de la fig 6.6) :
  On décrit le comportement de l'agent en réaction aux messages envoyés par le gestionnaire.

- émulation d'un gestionnaire OSI (cas B de la fig 6.6) :
  On décrit le comportement du gestionnaire en réaction aux messages envoyés par l'agent.

- émulation d'une architecture complète (cas C de la fig 6.6) :
  Différentes instances du simulateur participent à ce type de simulation. On remarquera un troisième type de participant les entités de médiation (qui jouent le rôle de gestionnaire et d'agent). C'est ce type de boite qui va être utilisé pour représenter un composant dans l'application qui va être présentée dans le cas d'étude présenté dans le chapitre suivant.

A noter que TIMS sans interface de communications est tout de même utilisable. Le seul moyen d'injecter des messages est simplement l'interface des scénari qui va être présentée dans la section suivante.

6.4.3 L'interface des scénari

6.4.3.1 Les scénari

Les scénari constituent un acteur très important de la simulation (cf section 3.4.2). Ils permettent d'envoyer des messages au simulateur en émulant soit un message de traitement[1], soit un message d'information. Un scénario peut être un message ou bien un ensemble de messages. Un message peut déclencher un comportement. A son tour, ce comportement peut lors de son exécution déclencher un ou des messages.

Un scénario particulier est nommé le "loader" (chargeur de configuration). C'est celui qui est responsable de la mise en place de la configuration de référence. On verra dans la partie III que le loader se résume en un seul message qui déclenche en cascade l'exécution de plusieurs messages.

6.4.3.2 Les ressources réelles

Avec la possibilité d'injecter des messages IVP (via l'interface des scénari) est née l'idée de modélisation des ressources réelles dans le contexte des objets gérés de la gestion OSI.

Cette interface que nous appelons "RRIface" est modélisée par un objet particulier qui n'appartient pas au modèle d'information initial. L'idée est de communiquer (via les interfaces standardisées) avec cet objet pour lequel on a défini des comportements (au sens spécifications BL) qui agissent directement sur le conteneur d'instances (IR).

[1] dans le contexte de la gestion OSI, les messages de traitement sont des messages CMIS

Pour communiquer avec cet objet, nous avons défini une classe GDMO qui comprend des actions GDMO. C'est ensuite à travers l'interface de communication (ou l'interface des scenari) que nous communiquons avec l'objet "RRIface".

6.4.4 L'interface de visualisation graphique

La visualisation graphique est une fonction essentielle pour la compréhension de ce qui se passe lors d'une simulation. Elle rend utilisable l'ensemble des informations en faisant abstraction des détails et en présentant de manière conviviale les informations. Nous avons retenu deux types d'informations :

- le graphe des objets;
- l'arbre de comportements.

6.4.4.1 Le graphe des objets

Il donne à tout moment l'état du point de vue des données du système (cad sa configuration en terme d'objets d'information et d'objets de relation). L'utilisateur en cliquant sur un des cercles qui représente une instance voit s'afficher l'état de l'objet d'information (les attributs et leur valeurs) ou bien l'état de l'objet de relation (les rôles et les instances d'objet de relation qui les remplissent).

- Ce graphe pouvant être très grand, il est possible de le couper (ciseaux dans la figure 6.7) et ainsi de ne garder que les sous-graphes qui nous intéressent.
- De manière à aider durant la phase de spécification/simulation, un objet lorsqu'il change d'état (modification de la valeur d'un attribut) change de couleur. Ceci nous permet de voir immédiatement, l'impact d'un pas de scénario (et des comportements BL).

6.4.4.2 L'arbre de comportements

C'est tout simplement une représentation graphique du BET tel qu'il a été introduit dans la section 3.4.1. Autant nous pouvons dire que le graphe des objets représente l'état des données, l'arbre de comportements représente l'état du point de vue contrôle.

L'arbre de comportements est intéressant quand il est utilisé comme trace de l'exécution et montre comment s'enchaînent les différents comportements. Cependant, il se révèle encore plus puissant quand il est utilisé lors des simulations interactives compliquées (simulation en mode pas à pas). Par exemple lors d'un contexte de nondéterminisme, il permet de comprendre la combinatoire mise en jeu en montrant l'ensemble des BEN potentiellement exécutables.

- Cet arbre pouvant être très grand, il est possible de le couper (ciseaux dans la figure 6.7) et ainsi de ne garder que les sous-arbres qui nous intéressent.
- L'utilisateur en cliquant sur un des rectangles qui représente un BEN voit s'afficher les informations de celui-ci. De manière à donner du relief à cet arbre, la couleur de chaque noeud informe l'utilisateur de l'état du BEN 3.4.

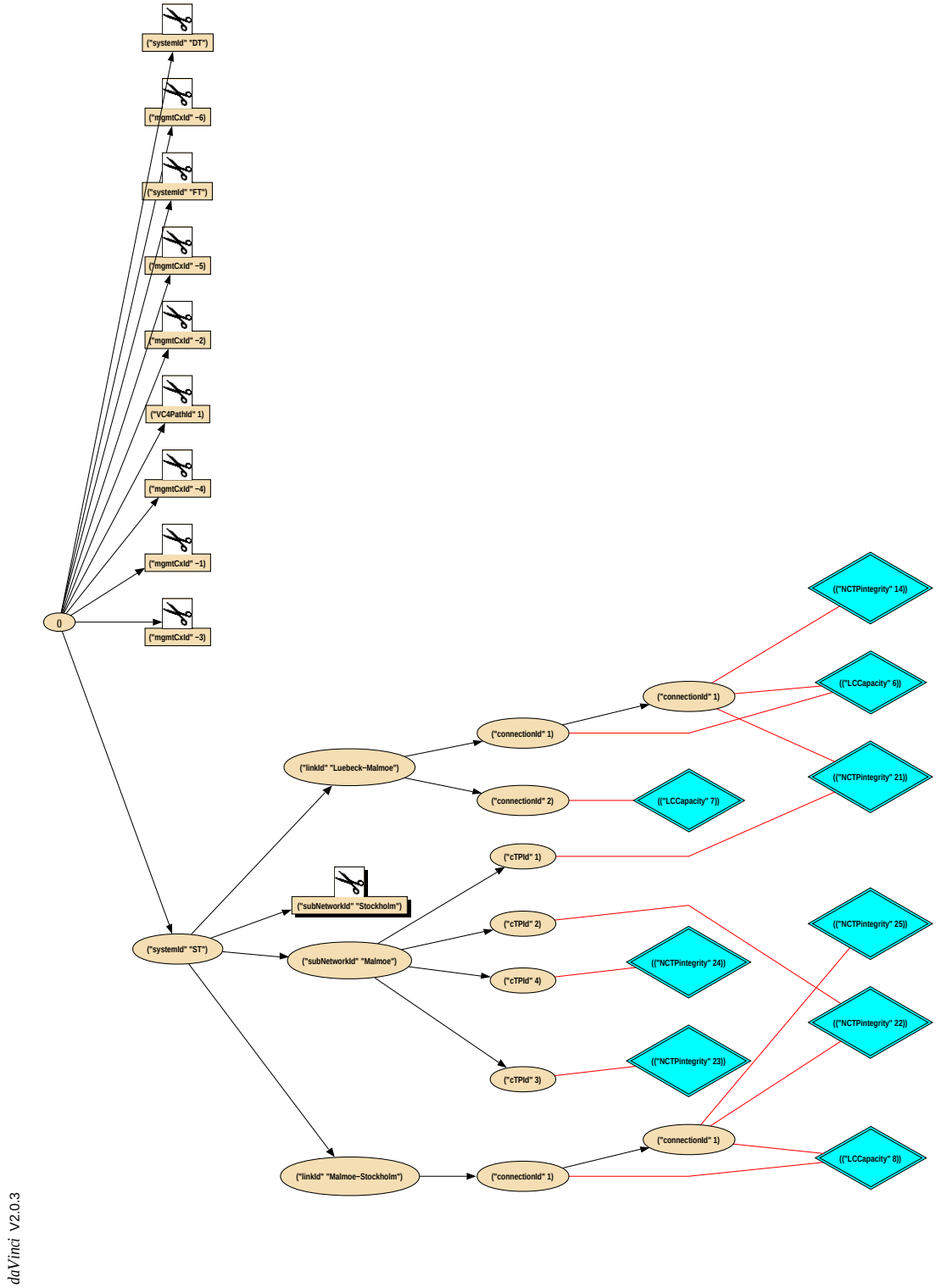

Figure 6.7: Un exemple de graphe d'objets

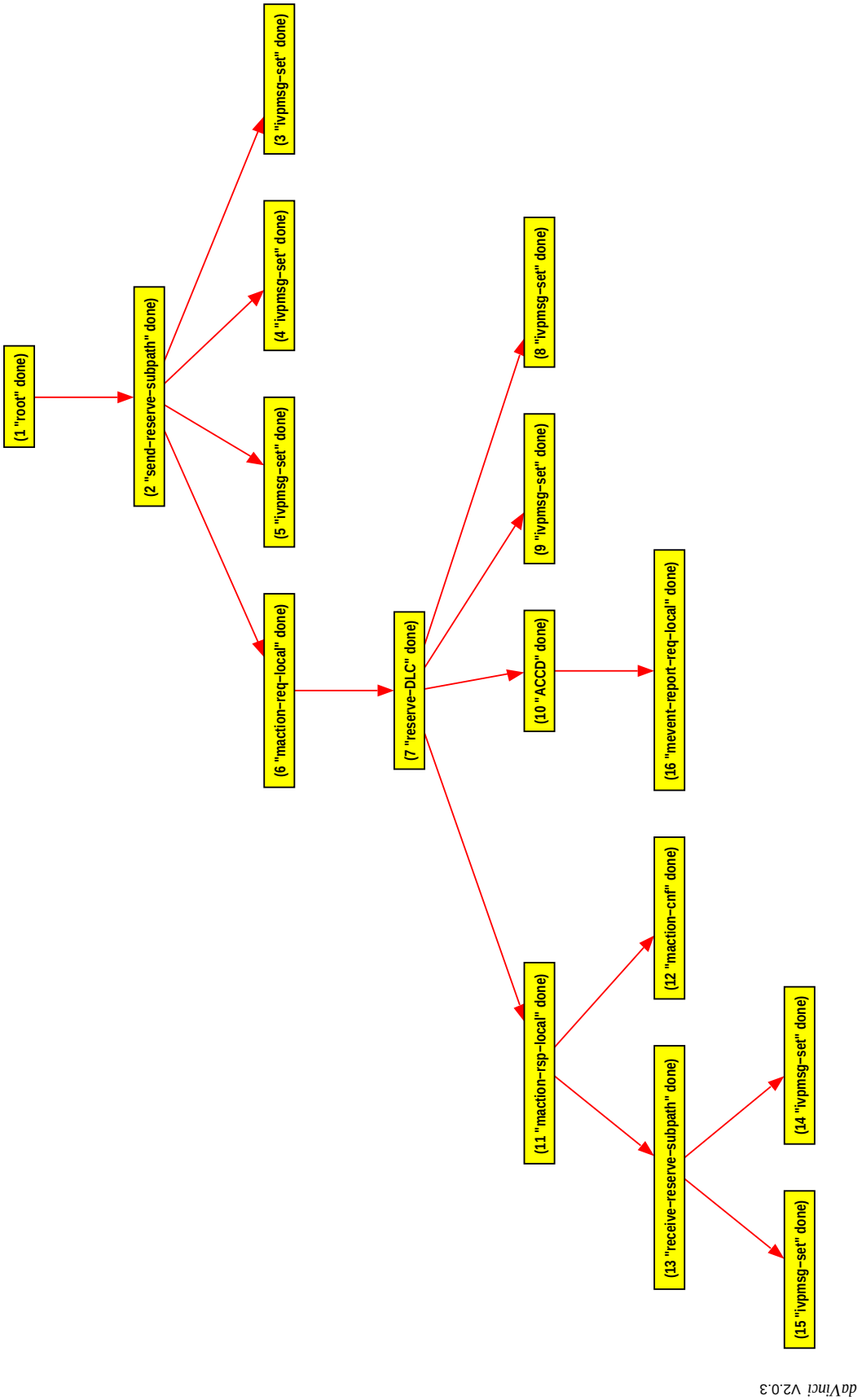

Figure 6.8: Un exemple d'arbre de comportements

6.5 Conclusion

La première partie de ce chapitre a présenté le RGT et la gestion de réseau OSI. Le but de cette présentation est de présenter le cadre de travail sur lequel nous allons expérimenter nos recherches présentées dans la partie I et la partie II (spécification, simulation et génération de tests) dans le contexte d'un cas d'étude réel : Xcoop.

La seconde partie de ce chapitre a présenté TIMS, qui est un simulateur dédié à la simulation de modèles d'information du RGT. On a notamment présenté les différentes interfaces de ce simulateur : l'interface de modèle d'information, l'interface de communications, l'interface des scenari et l'interface visuelle. C'est grâce à ce simulateur que nous avons modélisé, simulé et généré des tests sur le cas d'étude Xcoop. C'est l'ensemble de ces expérimentations que nous allons décrire dans le chapitre suivant.

Chapitre 7

Spécification du comportement dynamique et génération de tests pour les interfaces Xcoop à l'aide de TIMS

7.1 Introduction

Le but de ce chapitre est d'appliquer les résultats de cette thèse (un formalisme pour la description du comportement dynamique et une méthode de génération de tests) dans un contexte réel issu du RGT : la gestion des interfaces Xcoop (Xcoop coopération inter-opérateurs à travers les interfaces X du RGT) pour les réseaux de transport SDH.

Ce chapitre se décompose en trois parties :

- Dans la première partie de ce chapitre, nous présentons les interfaces Xcoop. Au cours de cette présentation, nous allons rapidement aborder la gestion de la SDH. Nous présentons ensuite plus précisément la gestion de ces interfaces et le test de ces interfaces.
- La seconde partie de ce chapitre présente une partie de la spécification du comportement dynamique de ces interfaces. Le comportement est d'abord donné en prose ou sous la forme de schéma (tel qu'il est formalisé dans les documents originaux). Il est ensuite donné en BL. Nous allons présenté à cette occasion la plupart des caractéristiques de BL dans un contexte réel d'utilisation.
- La troisième partie de ce chapitre présente des exécutions de TIMS d'une part en tant que simulateur des spécifications et d'autre part en tant que générateur de tests. A cette occasion, nous faisons une comparaison entre les tests écrits manuellement par les concepteurs de ces interfaces et ceux générés automatiquement grâce à TIMS.

Les données qui ont servi à la spécification du comportement dynamique et à la génération de tests qui vont suivre proviennent du serveur d'EURESCOM et sont consultables publiquement [P4096]. La spécification est composée de cinq volumes :

- volume 1 : Partie Principale

- volume 2 : Annexe A l'ensemble concernant l'allocation de chemins
- volume 3 : Annexe B l'ensemble concernant la gestion de fautes
- volume 4 : Annexe C le modèle d'information
- volume 5 : Annexe D Suite abstraite de test

On trouve aussi sur ce serveur des tests sous forme TTCN-MP qui serviront à notre comparaison dans la section 7.4.

7.2 Introduction générale au cas d'étude

Ces dernières années, EURESCOM, dans plusieurs projets liés au RGT, a proposé des spécifications de gestion de réseaux à travers les interfaces X entre opérateurs, mais une méthodologie (des directives) pour l'implémentation de ces interfaces n'a jamais été étudiée.

Pour combler ce manque, EURESCOM a créé le projet P408 et a lancé le concept de Pan-European TMN Laboratory (PET-Lab)[1]. Deux technologies de réseaux de transport ont été étudiées : la SDH et l'ATM. Nous ne nous intéresserons par la suite uniquement à la technologie SDH. Le réseau de transport SDH qui va être géré se nomme METRAN (Managed European TRAnsmission Network) .

- la spécification de l'interface Xcoop (coopération inter-opérateurs à travers les interfaces X) fut la première activité du projet P408. L'approche choisie pour documenter la spécification des interfaces Xcoop est celle des "Ensembles" du NMF (Network Manangement Forum). La spécification des interfaces Xcoop pour les réseaux de transport SDH se compose de deux ensembles :
  - un ensemble concernant l'allocation de chemins (path provisioning);
  - un ensemble concernant la gestion de fautes.

  Nous ne nous intéresserons par la suite qu'à l'ensemble concernant l'allocation de chemins.
- la seconde activité du projet P408 fût de spécifier des tests pour ces interfaces. L'approche choisie pour la spécification des tests de ces interfaces a été de suivre celle préconisée par ISO/9646.
- l'ultime activité du projet P408 fut d'exécuter ces tests sur les implémentations de ces interfaces.

7.2.1 Spécification des interfaces Xcoop

Avant de présenter l'ensemble concernant l'allocation de chemins, nous allons donner quelques informations sur la gestion des interfaces Xcoop et sur la méthodologie de description adoptée.

[1] que l'on peut traduire par laboratoire européen d'expérimentations sur le RGT

7.2.1.1 Introduction aux interfaces Xcoop

a) Gestion de la SDH Les réseaux SDH peuvent être modélisés par différentes couches réseaux (cf Figure 7.1). Chacune d'entre elles a une relation de client-serveur avec ses couches contiguës (une couche serveur offre un service de transport à la couche supérieure cliente et est la couche cliente d'une couche inférieure dont elle utilise les services de transport). Ce modèle en couches est présenté de haut en bas :

- la couche circuit qui supporte la transmission de bout en bout des services de télécommunications sur le réseau SDH.
- la couche chemin qui supporte le transport de conteneurs virtuels (VC Virtual Container). Des couches supportant des chemins de bas et de haut niveaux sont définies conformément à la capacité des VC.
- la couche section qui garantit la transmission des trames SDH entre les noeuds du réseau.

Chaque couche est connectée aux couches contiguës à travers des points d'accès qui s'occupent des fonctions d'adaptation nécessaires pour convertir le format des signaux des différentes couches. Chaque réseau peut être partitionné en des sous-réseaux différents SN (SubNetwork) qui sont connectés par des LC (Link Connection). La motivation pour l'adoption d'une telle approche est que chaque couche peut être reconfigurée et gérée indépendamment des autres et le réseau peut être réarrangé en changeant l'association des points de connexion.

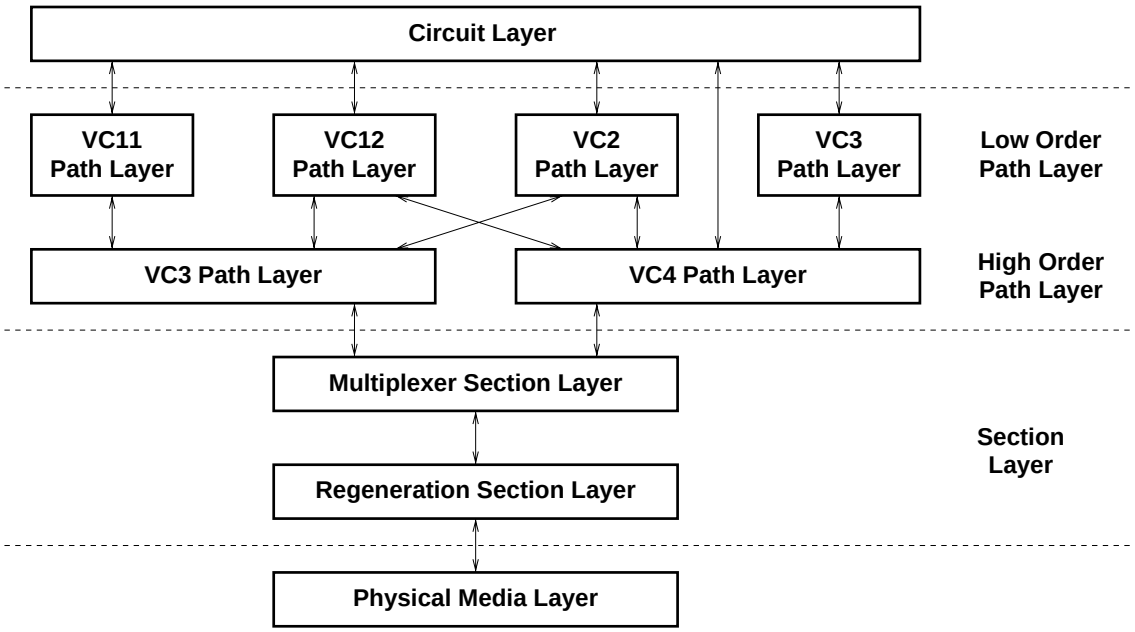

Figure 7.1: Le modèle en couches de la SDH

b) Gestion de METRAN La gestion de METRAN se situe au niveau couche chemin :

- la couche serveur de METRAN est uniquement constituée de VC-4 : les LC (Link Connection) .
- la couche client de METRAN est constituée de VC-low (VC-12, VC-2 et VC-3). Pour des raisons de simplifications, METRAN ne considère que des VC-12 : les DLC (Deliverable Link Connection) .

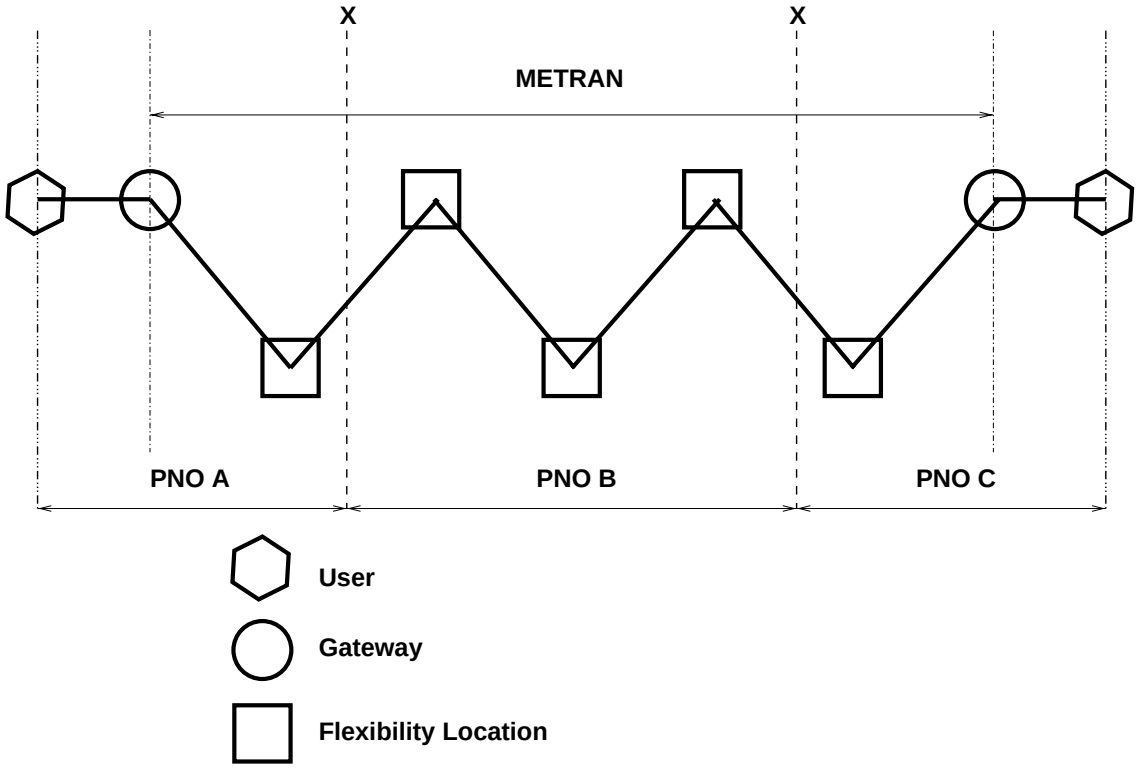

Figure 7.2: La gestion de METRAN

L'objectif principal de gestion de METRAN, pour l'Ensemble de l'allocation de chemins est d'automatiser la gestion des chemins au niveau VC-12 (DLC) à travers le réseau européen formé par des VC-4 (LC). Les ressources qui constituent le réseau METRAN à travers l'Europe sont des LC et des SN.

Les LC qui traversent une frontière entre deux opérateurs (PNO Public Network Operator) sont gérés par seulement un des deux opérateurs. Chaque PNO doit déclarer toutes ses ressources qui vont être dédiées à METRAN à tous les autres PNO.

Les chemins de DLC doivent démarrer et terminer dans des SN particuliers qui sont appelés les "gateway". Les autres SN sont appelés des "flexibility location" (point de cross-connection).

Par exemple (en suivant la Figure 7.2), un PNO (A) peut demander de la bande passante sur un LC de METRAN qui est possédé par un autre PNO (B). La demande de A est toujours dédiée à

un LC particulier (l'hypothèse est faite que tous les PNO connaissent la bande passante disponible sur un LC). Si B a de la bande passante disponible, il informe A que sa requête est acceptée et il lui donne l'identité du DLC alloué. Lorsque A a alloué tous les DLC sur un chemin de LC (appartenant éventuellement à un ou plusieurs autres PNO) , il peut demander l'activation du chemin de DLC et l'activation des cross-connection au niveau des différents SN empruntés.

Il n'y a pas de base de données centralisée contenant la topologie du réseau METRAN donc chaque PNO maintient sa propre vue du réseau. Un PNO est de fait responsable de son propre sous-réseau. Il est notamment responsable de la dissémination de l'information contenant chaque modification d'une partie de son sous-réseau à tous les autres PNO. De tels changements sont par exemple : un changement dans la bande passante disponible d'un LC.

Vu la taille importante du réseau METRAN, le nombre d'alarmes (de notifications) qui peuvent circuler peut être très important. C'est pourquoi, il est stipulé que seulement les LC et les DLC sont autorisés à émettre des alarmes. De plus, il est recommandé que seulement les PNO utilisateurs de la ressource émétrice de l'alarme soient informés.

7.2.1.2 Méthodologie

Pour décrire les services de gestion (MS Management Services), le concept d'"Ensemble" issu du NMF a été utilisé et adapté pour l'usage particulier de P408 (cad la coexistence du rôle de gestionnaire et d'agent dans le même système). Un Ensemble est un document contenant la spécification complète d'un MS à travers une interface de gestion donnée. On peut voir un Ensemble comme une collection de fonctions que l'on appelle MFS (Management Function Set) . Un MFS utilise une ou plusieurs fonctions de base : les MF (Management Function) . Un MF décrit les messages échangés à travers l'interface de gestion. La description des MF utilise le formalisme le plus proche possible du protocole de communication (CMIP). La modularité est assurée au niveau MF puisque chaque MF peut être réutilisé par plusieurs MFS.

7.2.1.3 L'Ensemble concernant l'allocation de chemins

Cet ensemble s'intérresse aux messages à travers l'interface X (interface entre différents RGT). La spécification des messages traversant les interfaces HMI (interfaces hommes-machines), les interfaces Q (à l'intérieur d'un RGT) et les messages internes aux applications de gestion peuvent être décrits de manière séparée et ne sont pas traités dans cet ensemble.

Dans notre cas, l'information contenue dans l'Ensemble est structurée de la manière suivante :

- description des ressources et du modèle d'information;
- description des fonctions (MFS) et description des scénari (MF).

a) Les ressources Cette section définit toutes les ressources et tous les composants de ces ressources qui vont être utilisés dans cet ensemble. Les ressources sont définies par des descriptions textuelles ou par des références à d'autres documents contenant la description des ressources concernées. Essentiellement, la description des ressources gérées est basée sur G.803. METRAN

interprète ces définitions pour les adapter aux ressources qui sont gérées dans le contexte de cet ensemble.

Pour décrire l'activité dynamique des ressources, le cycle de vie de ces ressources se décompose en deux processus :

- le processus de commissionnement (comissioning process) : il concerne l'installation et la configuration des ressources et existe avant le processus d'approvisionnement.
- le processus d'approvisionnement (provisioning process) : il concerne la réservation et l'activation des ressources.

Cet ensemble ne traite que du processus d'approvisionnement. La liste des ressources gérées dans cet ensemble est la suivante :

- les composants de la topologie : aucun des composants de la topologie ne peut être créé ou modifié (en termes de caractéristiques propres) à travers les interfaces X.
  - la couche réseau : Le réseau de transport est construit sous formes de couches réseau avec des relations client/serveur mutuelles. L'information transférée à travers le réseau est caractéristique pour la couche (en terme de débit et de format). Une couche réseau est composée de liens et de sous-réseaux.
    Dans METRAN, il n'est considéré que la couche "chemin" du réseau global. METRAN est constituée d'une couche cliente VC-12 et d'une couche serveur VC-4. METRAN est constitué de sous-réseaux ("flexibility location" et "gateway") et de liens entre ces sous-réseaux.
  - le lien : Le lien décrit la relation fixe entre deux sous-réseaux dans la même couche réseau. Un lien offre une capacité (débit pour le transport) qui peut être utilisée par des connections de lien.
    Dans METRAN (*mLink*), les liens peuvent être internes à un opérateur (PNO) ou bien traverser la frontière entre deux opérateurs. Seul un des opérateurs est responsable de la gestion de ce lien.
  - le sous-réseau : Un sous-réseau décrit la possibilité d'interconnexion des liens par l'intermédiaire des connexions de sous-réseau (subNetworkConnection SNC). Les sous-réseaux peuvent se subdiviser en des sous-réseaux de haut niveau et des sous-réseaux de bas niveau. Les sous-réseaux de haut niveau représentent des partitions. Les sous-réseaux de bas niveau représentent la matrice de "cross-connection" dans l'élément de réseaux.
    Dans METRAN (*mSubNetwork*) , un sous-réseau correspond toujours à une matrice de "cross-connection" dans un élément de réseau (sous-réseau de bas niveau et pas de partitionnement). METRAN fait l'hypothèse d'équipements non bloquants, ce qui veut dire que les SNC n'ont pas besoin d'être réservés avant leur activation. Enfin les sous-réseaux se subdivisent en "flexibility location" et "gateway". Les "gateway" sont des "flexibility location" qui ont la particularité de commencer et de terminer un chemin au niveau VC-12.

- les entités de transport :
  - la connexion de lien : c'est une relation fixe entre deux points de connexion qui utilisent une partie de la capacité de transport d'un lien.
    Dans METRAN (*mLinkConnection*) , le terme LC dénote une connexion de lien de type VC-4. Ces connexions sont mises en place durant le processus de commissionnement. Un LC est toujours dédié uniquement à METRAN.
    Dans METRAN (*mDelivLinkConnection*), le terme DLC dénote une connexion de lien de type VC-12.Ces connexions sont mises en place durant le processus de commissionnement. Les DLC sont supportés par des LC.
  - la connexion de sous-réseau : c'est une relation fixe entre deux points de connexion dans un sous-réseau.
    Dans METRAN (*mSubNetworkConnection*), c'est une connexion entre deux points de connexions dans un "flexibility location" ou dans un "gateway". Les SNC sont établies de manière dynamique durant le processus d'approvisionnement (cad qu'un objet géré correspondant au SNC est créé). Dans METRAN, seules les connexions points à points sont supportées.
- les points de connexions : Les points de connexions sont créés durant le processus de commissionnement. Ils correspondent aux points d'accès ou de terminaison d'un réseau en couche.
  - Dans METRAN (*mNetworkCTP*), les points de connexions se situent de chaque coté des connexions de lien et des connexions de sous-réseau.

Un chemin METRAN VC-12 est composé d'une ensemble de DLC et de SNC entre deux "gateway".

b) Le modèle d'information Le modèle d'information statique original est constitué d'un fichier GDMO et d'un fichier ASN.1. Nous avons écrit un troisième fichier contenant la spécification GRM. Ces trois fichiers sont donnés dans l'annexe E.

Une partie du modèle d'information dynamique (les spécifications BL) sera donnée dans l'annexe D. Pour la clarté des explications qui vont suivre, nous donnons l'arbre d'héritage (cf Figure 7.3) et l'arbre de contenance (cf Figure 7.4) de METRAN.

c) les scénari (MF) et les fonctions (MFS) On conserve le nom original des scenari et fonctions du document d'origine.

- les scénari (MF) :
  - MF DLC Reservation : réservation d'un DLC;
  - MF ACCD (Available Connections Change Dissemination) : émission d'une notification indiquant un changement de capacité d'un LC (nombre de DLC disponibles);
  - MF DLC Unreservation et MF DLC Release : déréservation et désactivation d'un DLC;

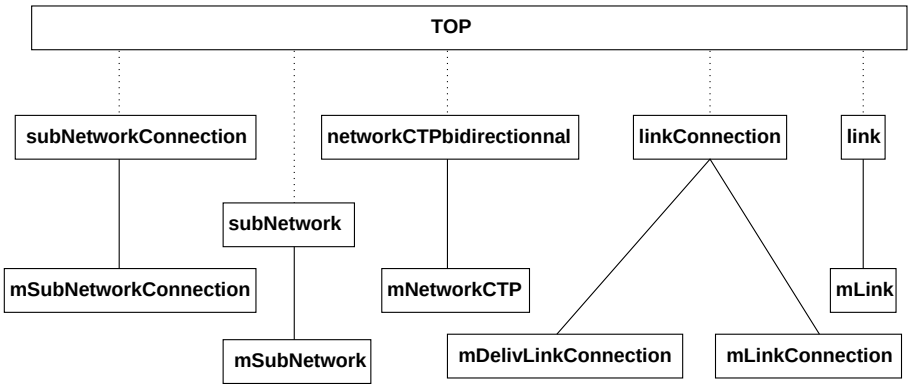

Figure 7.3: arbre d'héritage de METRAN

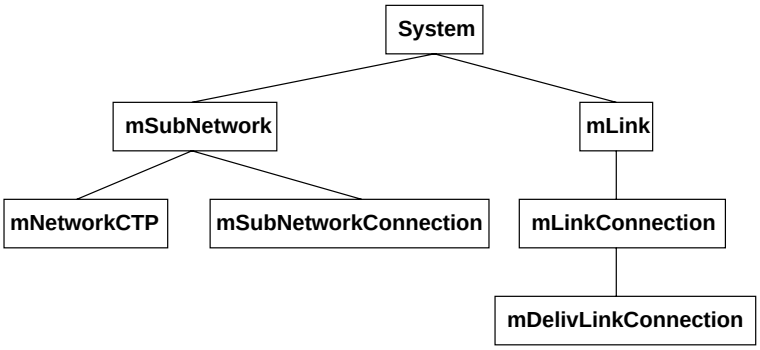

Figure 7.4: arbre de contenance de METRAN

  - MF DLC Activation : activation d'un DLC;
  - MF SNC Set-up : mise en place d'un SNC;
  - MF SNC Release : terminaison d'un SNC;
  - MF ACR (Available Connections Read) : lecture de l'attribut "AvailableConnection" qui indique la capacité d'un LC (nombre de DLC disponibles);
  - MF AtCD (Ability to Connect Dissemination) : émission d'une notification indiquant un changement de capacité d'un SN (nombre de SNC disponibles);
  - MF AtCR (Ability to Connect Read) : lecture de l'attribut "abilityToConnect" qui indique la capacité d'un SN (nombre de SNC disponibles).

- les fonctions (MFS) :
  - MFS Reserve DLCs : réservation d'un ensemble de DLC dans un chemin;
  - MFS Unreserve DLCs : déréservation d'un ensemble de DLC dans un chemin;
  - MFS Reservation time-out :déréservation d'un DLC à cause de sa non activation après un certain délai (expiration d'un timer);
  - MFS Activate Path : activation d'un ensemble de DLC et de SNC dans un chemin (l'activation doit avoir lieu immédiatement après la réservation);
  - MFS Terminate Path : désactivation et déréservation d'un ensemble de DLC et de SNC dans un chemin;
  - MFS Available Connections Update : notification indiquant un changement du nombre de DLC disponible dans un LC;
  - MFS Ability to Connect Update : notification indiquant un changement du nombre de SNC disponible dans un SN;

Les scénari (MF) et les fonctions (MFS) qui sont soulignés seront décrits en détail lors de leur modélisation dans TIMS dans la section 7.3. Ces MF et des MFS sont donnés dans l'annexe E.

7.2.2 Test des interfaces SDH Xcoop

L'objectif de cette spécification de tests est de fournir des scénari pour tester l'interfonctionnement entre les applications de gestion à travers les interfaces Xcoop. Il ne s'agit pas de fournir des scénari pour tester complètement la conformité de ces applications. Néanmoins, l'interfonctionnement n'est possible que si un certain degré de conformité est assuré localement (sous la responsabilité de chaque PNO de chaque coté de l'interface X). C'est par exemple le cas pour le test des piles OSI et des objets gérés qui doit être effectué systématiquement mais qui est hors du propos de cette spécification. De la même manière, il est hors du propos de cette spécification de donner des directives sur le développement et l'intégration des applications réelles. La spécification des tests se présente de la manière suivante :

- description de la portée des tests;

- description de la structure de la suite de test;
- description de l'architecture de test.

7.2.2.1 La portée des tests

Le projet P201 d'EURESCOM, qui avait pour mission de donner des directives pour le test des applications du RGT, a défini quatre types de tests qui doivent être exécutés dans l'ordre de 1 à 4 (cad que l'on passe à l'étape suivante que si l'étape courante a été validée)[2].

1. test unitaire : qui consiste au test de la pile de protocole, des objets gérés et du test des MF;
2. test d'intégration : qui consiste au test du modèle d'information et des MFS;
3. test système : qui consiste au test d'un Ensemble (de plusieurs MFS);
4. test inter-Ensembles : qui consiste au test de l'application de gestion avec tous les Ensembles qui la constitue.

La spécification de tests du projet P408 ne considère que le test des MF pour test unitaire et le test des MFS pour test d'intégration et comprend des tests système et des tests inter-Ensembles.

Le projet P201 d'EURESCOM a aussi défini trois aires de validation pour le test des applications du RGT.

1. La première aire de validation est appelée "aire de validation interne de l'opérateur réseau" et couvre le test d'un domaine de gestion d'une application de gestion international (INMS International Network Management System). Ceci inclut le test d'une interface X avec un autre INMS mais exclut tous les autres INMS[3].
2. La seconde aire de validation est appelée "aire de validation pour l'accès au service par l'utilisateur". Elle couvre le test de l'INMS et d'un utilisateur qui appartient à un domaine de gestion différent (client d'un autre PNO). A contrario de la gestion ATM où il peut y avoir des interfaces (Xuser) de ce type entre un INMS et un client, il n'y a pas ce type d'interfaces dans la gestion SDH.
3. La troisième et dernière aire de validation est appelée "aire de validation du service bout à bout" et couvre le test de deux ou plusieurs domaines de gestion. Dans le cas de la SDH, ceci signifie l'union des domaines de gestion de plusieurs INMS et donc des interfaces Xcoop.

Compte-tenu de la non-applicabilité de la seconde aire de validation, le test de l'"aire de validation interne de l'opérateur réseau" et le test de l'"aire de validation du service bout à bout" sont très voisins. Pour le test interne à un PNO, on utilise le plus souvent un simulateur pour simuler l'INMS de l'autre coté de l'interface X. Cette spécification ne traite que des tests de l'"aire de validation du service bout à bout".

[2] C'est en fait la notion de granularité des tests (présentée dans la section 4.2.1.2 du chapitre 4) qui est appliquée à la méthodologie des Ensembles du NMF.

[3] C'est un test de boite grise (test fonctionnel mais avec accès au code) interne à un PNO (cf section 4.2.1.2 du chapitre 4).

7.2.2.2 La structure de la suite de tests

Cette section décrit la structure de la suite de tests qui a été développé dans le contexte du projet P408. Celle ci peut se schématiser par un arbre où les feuilles sont les cas de tests (cf Figure 7.5).

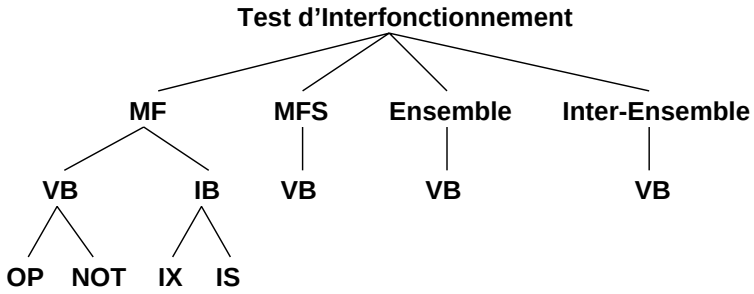

Figure 7.5: Structure de la suite de test

Le Tableau 7.1 définit brièvement quelques groupes de tests génériques définis dans ISO/9646 et raffinés par P201.

CA	Test de capacité (paramètres ont la valeur par défaut)
VB	test de comportement valide (paramètres varient à l'intérieur des limites autorisées)
IB	test de comportement non valide (paramètres varient en dehors des limites autorisées)
OP	test d'opération (test qui vérifie une requête et une réponse)
NOT	test de notification (test qui vérifie l'émission d'une notification)
IX	test de syntaxe non valide (test simulant une erreur au niveau syntaxique par exemple adressage d'une entité qui n'existe pas)
IS	test de sémantique non valide (test simulant une erreur au niveau sémantique par exemple déréservation d'une identité qui n'est pas réservée)

Tableau 7.1: Groupes de tests génériques utilisés dans METRAN

- les tests CA constituent le cas initial des tests VB;
- les tests VB sont divisés en deux ensembles : les tests OP et les tests NOT;
- les tests IB sont divisés en deux ensembles : les tests IX et les tests IS;

- pour les branches MFS, Ensembles et inter-Ensembles, il n'y a pas de distinctions entre les tests IB et les tests VB. De plus il n'y a pas de distinctions entre OP et NOT, seul est important la présence et l'ordre des opérations et des notifications.

Une convention de nommage est mise en place pour étiqueter les objectifs de test et les cas de test afin de clarifier la spécification. Etant donné que nous ne présenterons pas toute la spécification (cad tous les objectifs de test) nous ne donnons pas cette convention de nommage.

La spécification donne un ensemble d'objectifs de tests dont quelques exemples seront traités dans le détail lors de la génération de tests avec TIMS dans la section 7.4.2.

7.2.2.3 L'architecture de test

Avant de produire des tests, il faut d'une part définir une structure de suite de tests et d'autre part définir des objectifs de test. Il faut ensuite écrire un cas de test pour chaque objectif de test. Lors de cette étape, il faut tenir compte de l'architecture de test. Celle ci définit le positionnement des PCO et la syntaxe et la sémantique des messages qui traversent ces PCO (dans un sens ou dans l'autre).

Il est important de noter que la définition de l'architecture de test va permettre l'exécution des tests dans le milieu opérationnel. C'est donc l'élément qui dicte le choix d'exécutabilité des tests (il se peut notamment qu'à cause de l'architecture de test, des objectifs de test ne puissent pas être transformés en cas de test).

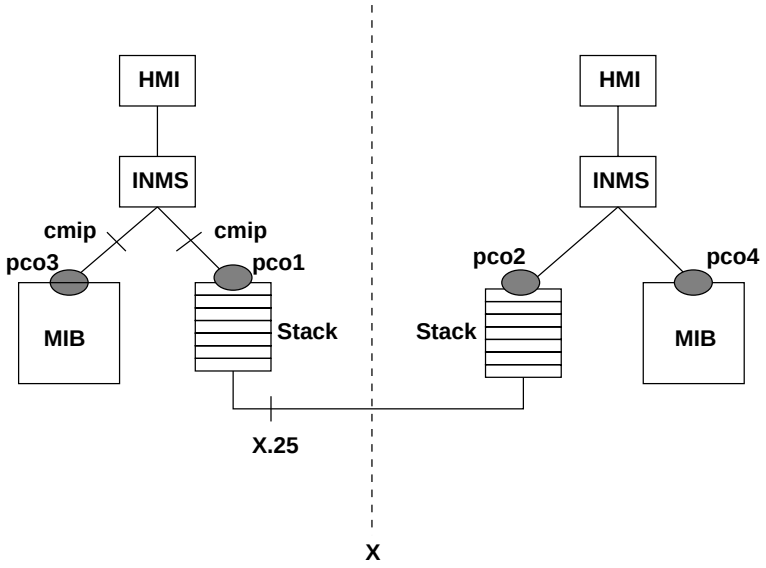

Figure 7.6: L'architecture de test

La figure 7.6 donne une vue simplifiée de l'architecture adoptée dans P408 durant l'activité de test.

- HMI : est une interface homme machine qui contient les fonctions de transformation des informations contenues dans la MIB dans un format intelligible;
- INMS : est le système de gestion international (INMS). Il est composé des fonctions qui interagissent avec l'interface homme-machine, des fonctions d'émission et de réception des primitives CMIS et des fonctions du contrôle de la MIB;
- MIB : est la base de gestion contenant les objets gérés;
- Pile de protocole OSI : qui fournit le service de transport utilisé par les messages X.

Pour chaque PNO, on a deux PCO : un sur la pile et un autre sur la MIB. Ce choix se justifie par :

- l'architecture de test doit être la plus simple possible. Une complexité non nécessaire rend difficile la production de la suite de tests;
- la spécification des MF et des MFS consiste en la spécification de service CMIS, ce qui oblige à positionner un PCO à ce niveau;
- les PCO sur la MIB sont là pour le test d'Ensemble (à ce niveau il est parfois désiré d'exercer des manipulations (et/ou des vérifications) sur des attributs d'objets gérés). Dans le but d'équiper l'implémentation par rapport à ces besoins et pour pouvoir tracer les sources d'erreurs, il est nécessaire qu'un PCO se situe au niveau de la MIB.

Enfin avant de tester effectivement il faut mettre en place une configuration de référence. Cette configuration de référence est aussi nommée topologie de référence. Il est donné une liste d'objets avec un nommage respectant M1400. Quelques restrictions pour la simplicité du test ont été effectuées (par exemple un LC contient 20 DLC au lieu de 63). Il est enfin donné la valeur initiale de certains attributs.

7.3 Spécification des interfaces SDH Xcoop avec TIMS

Cette section présente la spécification des comportements en BL des objets des interfaces SDH Xcoop. Afin d'éviter un niveau de détails inutile, certains comportements sont simplifiés (on ne donne pas la liste des attributs et des valeurs ASN.1). On trouvera les spécifications complètes dans l'annexe D.

7.3.1 La topologie de référence

Comme nous l'avons vu dans l'introduction précédente, le processus de commissionnement n'est pas spécifié (cad que la façon dont on installe et configure les ressources n'est pas spécifiée). On dispose en tout et pour tout de deux types d'informations :

- le document qui décrit la topologie de référence nous donne la structure de la MIB (l'arbre de contenance);

- la valeur initiale des attributs des objets est donnée de manière complètement non-structurée dans trois documents : le document qui décrit le modèle d'information, le document qui décrit l'Ensemble et dans celui qui décrit la topologie de référence.

7.3.1.1 Création de la topologie de référence

La création de la topologie initiale peut se faire par une simple énumération de création d'objet. Ceci peut se révéler très fastidieux dans le cas d'une MIB gigantesque. C'est notamment le cas de METRAN; c'est pourquoi nous modélisons la construction par l'intermédiaire d'envoi de messages et de comportements BL.

Cette création nous montre un exemple de cascade de comportements (vu dans la section 3.3.1.5). Le message "*metran⇔topology*" déclenche le comportement "*metran⇔topology⇔setup*" qui a dans son corps la création d'un objet CVP de la classe "system" (un objet de la MIB) et deux envois de message : "*SN⇔creation*" et "*LINK⇔creation*". Le message "*SN⇔creation*" déclenche à son tour un comportement "*subsetwork⇔creation*" qui envoie à son tour un message pour la création des "mNetworkCTP" ...

Cette cascade de messages et de comportements est montrée dans la Figure 7.7. Cette figure montre le développement en parallèle de l'exécution du scénario (messages et comportements) et la construction de l'arbre de nommage.

a) Scénario La création de la topologie de référence est initialisée par l'envoi du message interne "metran-topology".

```
(msgsnd (metran-topology:make))
```

b) Comportement en BL Le comportement "*metran⇔topology⇔setup*" se déclenche lors de la réception du message "*metran⇔topology*".

```
(define-behavior "metran-topology-setup"
  (scope (msg "metran-topology"))
  (when)
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
    (Create "system" ... )                  ;1
    (msgsnd (SN-creation:make operator      ;2
                              subnetlist))  ;3
    (msgsnd (LINK-creation:make operator    ;4
                                linklist))  ;5
  )
  (post)
)
```

- Ligne 1 du body : création d'un objet de la classe "system"; Ceci correspond (dans notre cas) à la création d'un domaine de gestion d'un opérateur.

- Ligne 2 du body : création et envoi d'un message interne "*SN ⇔creation*" pour la création de tous les sous-réseaux de la liste *subnetlist* pour l'opérateur *operator*;
- Ligne 3 du body : création et envoi d'un message interne "*LINK ⇔creation*" pour la création de tous les liens de la liste *linklist* pour l'opérateur *operator*.

On notera que tous ces comportements sont des "is-trigger" (cad qu'ils définissent la sémantique du message qui les déclenche cf section 3.3.2.3)).

c) Comportement en BL Le comportement "*subsetwork ⇔creation*" se déclenche lors de la réception du message "*SN ⇔creation*".

```
(define-behavior "subnetwork-creation"
  (scope (msg "SN-creation"))
  (when)
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
    (for-each (lambda (snid)                          ;1
       (Create "mSubNetwork" ... )                  ;2
       (msgsnd (NCTP-creation:make (msg-> operator)  ;3
                                   sn-name           ;4
                                   nb-nctp)))        ;5
    (msg-> subnetlist))                              ;6
  )
  (post)
)
```

- Ligne 1 du body : $(for \Leftrightarrow each\ (lambda\ (X)\ body)\ list \Leftrightarrow elt)$ est une primitive Scheme qui applique la lambda *body* à tous les éléments X de la liste $list \Leftrightarrow elt$; Dans notre cas, on exécute l'envoi des deux messages pour tous les éléments de la liste *subnetlist* la liste des sous-réseaux d'un opérateur donné.
- Ligne 2 du body : création d'un objet de la classe "mSubNetwork";
- Ligne 3 du body : création et envoi d'un message interne "*NCTP ⇔creation*" pour la création d'un nombre *nb ⇔nctp* de networkCTP pour le sous-réseau *sn ⇔name* de l'opérateur *operator*.

d) Comportement en BL Le comportement "*nctp ⇔creation*" se déclenche lors de la réception du message "*NCTP ⇔creation*".

```
(define-behavior "nctp-creation"
  (scope (msg "NCTP-creation"))
  (when)
```

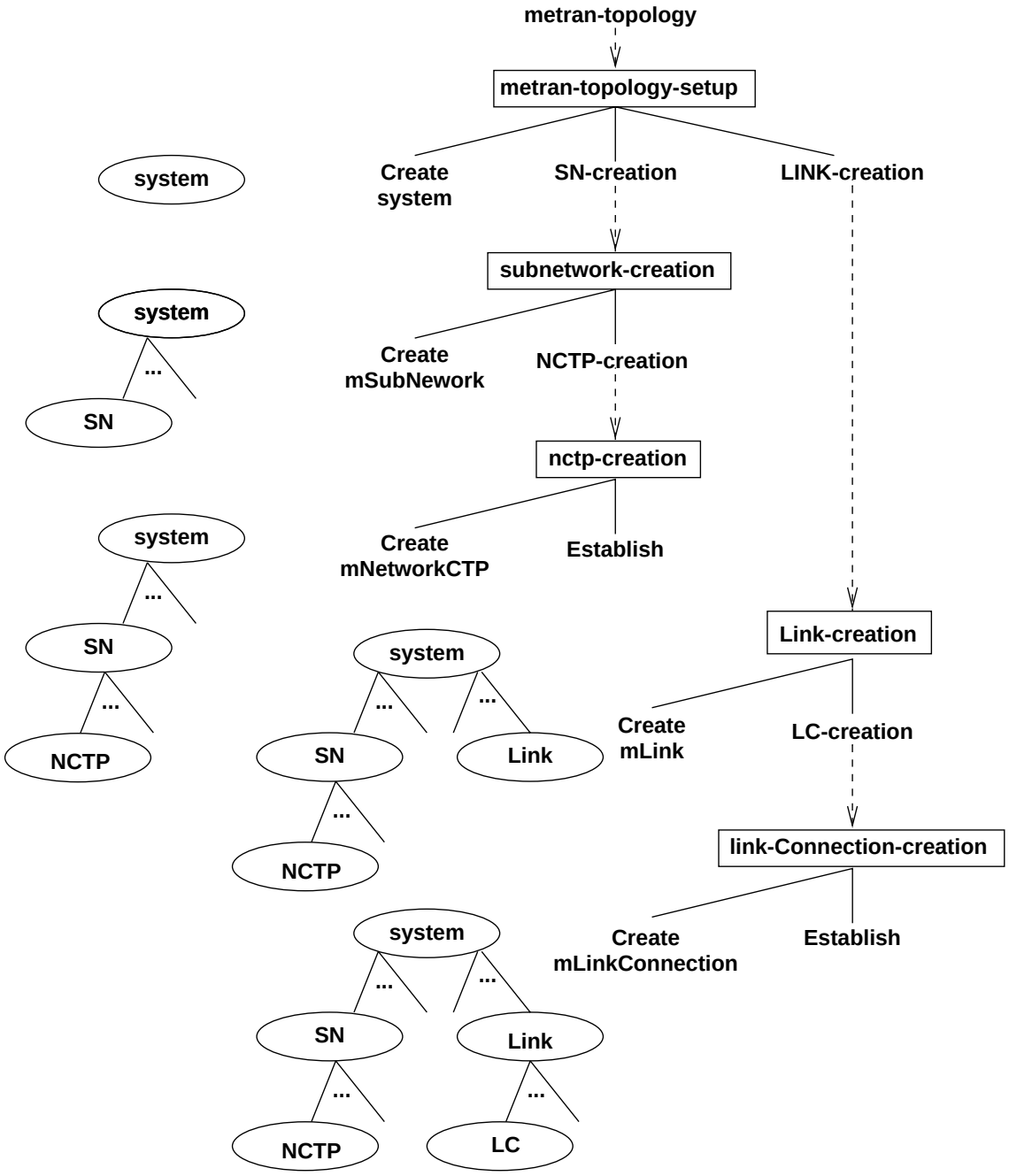

Figure 7.7: Création automatique de la topologie de METRAN

```
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
    (do ((nctpid 0 (+ 1 nctpid)))                    ;1
        ((= nctpid (msg-> nb-nctp)) #t)              ;2
       (Create:make "mNetworkCTP" ... )           ;3
       (Establish "NCTPintegrity" ri                 ;4
                  '(("nctp" nctpid)                  ;5
                    ("snc" *unspecified*)            ;6
                    ("connectivity" *unspecified*))) ;7
    )
  )
  (post)
)
```

- Ligne 1 du body : $(do\ ((val\ init\ p1))\ (p2\ res)\ body)$ est une primitive Scheme qui applique la boucle *body* tant que $p2$ est faux et renvoie *res*. A chaque itération applique le prédicat $p1$ sur la variable d'itération *val* dont *init* est la valeur initiale. Dans notre cas, on obtient une boucle qui s'exécute $nb \Leftrightarrow nct\,p$ fois.

- Ligne 2 du body : création d'un objet de la classe "mNetworkCTP"

- Ligne 3 du body : création et envoi d'un message IVP Establish qui crée une instance de la relation "NCTPintegrity". Cette relation contient 3 rôles : le rôle "nctp", le rôle "snc" et le rôle "connectivity". Lors de l'établissement de la relation, seul le rôle "nctp" est connu. Les autres rôles seront joints à la relation de manière dynamique (par l'intermédiaire d'une opération BIND). Cette relation exprime le fait qu'un NCTP est un point de terminaison d'une part d'une SNC (rôle SNC) et d'autre part d'un DLC (rôle connectivity).
  Une alternative aurait été d'écrire un comportement associé à la création du NCTP et qui aurait établi la relation.

e) Comportement en BL Le comportement "$link \Leftrightarrow creation$" se déclenche lors de la réception du message "$LINK \Leftrightarrow creation$".

```
(define-behavior "link-creation"
  (scope (msg "LINK-creation"))
  (when)
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
    (for-each (lambda (linkid)                ;1
       (Create:make "mLink" ... )          ;2
       (msgsnd (LC-creation:make operator  ;3
                                 aend      ;4
                                 zend      ;5
```

```
                                    nb-lc)))  ;6
    (msg-> linklist))                         ;7
  )
  (post)
)
```

- Ligne 2 du body : création d'un objet de la classe "mLink"
- Ligne 3 du body : création et envoi d'un message interne "*LC⇔creation*" pour la création d'un nombre *nb⇔lc* de LC entre les sous-réseau *aend* et *zend*. Le lien appartient à l'opérateur *operator*.

f) Comportement en BL Le comportement "*link⇔connection⇔creation*" se déclenche lors de la réception du message "*LC⇔creation*".

```
(define-behavior "link-connection-creation"
  (scope (msg "LC-creation"))
  (when)
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
    (do ((lcid 0 (+ 1 lcid)))              ;1
        ((= lcid (msg-> nb-lc)) #t)        ;2
       (Create "mLinkConnection" ... )  ;3
       (Establish "LCCapacity" ... )    ;4
    )
  )
  (post)
)
```

- Ligne 2 du body : création d'un objet de la classe "mLinkConnection"
- Ligne 3 du body : création et envoi d'un message IVP Establish qui crée une instance de la relation "LCCapacity". Cette relation contient 2 rôles : le rôle "lc" et le rôle "dlc". Lors de l'établissement de la relation, seul le rôle "lc" est connu. Cette relation exprime la relation de contenance entre un LC et un ensemble de DLC.

7.3.1.2 Création d'un DLC

La topologie est complète lors de la création des DLC. Cet exemple de comportement est intéressant car c'est le premier comportement "side-effect" (comportement de type effet de bords 3.3.2.3). Dans le cas présent, après la création d'un DLC, il s'agit de faire les "Bind" dynamiquement dans les trois instances de relation dans lesquelles un DLC joue un rôle. Un DLC a le rôle de "lc" dans une relation "LCCapacity". Il a le rôle de "connectivity" dans la relation "NCTPIntegrity".

a) Scénario

```
(Create "mDelivLinkConnection" ... ))
```

b) Comportement BL

```
(define-behavior "DLC-setup-after-creation"
  (scope (msg "ivpmsg-create"))
  (when (equal? objectClass "mDelivLinkConnection"))
  (exec-rules (fetch-phase i) (coupled after-trigger))
  (pre)
  (body
     (let ((ri-aend (FetchRis (msg-> "aEndNWTPList")   ;1
                              "nctp"                   ;2
                              "NCTPIntegrity"))        ;3
           (ri-zend (FetchRis (msg-> "zEndNWTPList")   ;4
                              "nctp"                   ;5
                              "NCTPIntegrity"))        ;6
           (ri-lccap (FetchRis (superior (msg-> inst)) ;7
                               "lc"                    ;8
                               "LCCapacity"))          ;9
           (oldval (Get (Part (ri) "lc")               ;10
                        "availableLinkConnections")))  ;11
       (Bind ri-lccap  (msg-> inst) "dlc")             ;12
       (Bind ri-aend (msg-> inst) "connectivity")      ;13
       (Bind ri-zend (msg-> inst) "connectivity")      ;14
       (Set (Part ri-lccap "lc")                           ;15
            "availableLinkConnections"                 ;16
            (+ 1 oldval))                              ;17
     )
  )
  (post)
)
```

L'exécution de ce comportement se déroule en deux temps. Dans un premier temps il s'agit de récupérer les instances de relation par l'opérateur FetchRis et les attributs pointeurs du DLC qui pointent sur des objets avec lesquels il va être en relation. Ensuite dans un second temps, il faut faire le BIND du DLC dans les différentes instances que l'on vient de déterminer et mettre à jour l'attribut "availableLinkConnections" du LC supérieur qui contient le nombre de DLC disponibles (dont l'état "assignmentState" est "free"). Il est important de noter que lors du FetchRis, on se sert d'informations liées à l'implémentation des relations (relation de contenance entre un LC et un DLC) et attributs pointeurs réciproques. A partir du moment où l'objet est joint à la relation (par un BIND), on peut utiliser la relation et ses mécanismes (opérateurs comme Part, Card ...). C'est notamment le cas dans le dernier Set (Ligne 15 du Body), où on atteint le LC supérieur par l'intermédiaire de la relation et du rôle ("lc" en l'occurrence).

let est une primitive Scheme qui permet de déclarer des variables locales (au comportement). Avec les primitives for-each et do que nous venons de voir, le lecteur commence à entrevoir la justification du choix d'un véritable langage de Programmation (Scheme dans notre cas) était une fonctionnalité importante. Les opérateurs arithmétiques, les structures de données, tout ce dont un programmeur a besoin fait partie de BL. Nous allons voir que dans le cas des MFS, qui sont des algorithmes plus complexes, ce choix se justifie totalement.

L'exemple du comportement suivant est une illustration sur l'usage d'un scope avec une relation, un rôle et un message (déclencheur). Ce mécanisme très puissant de BL, nous permet de spécifier le comportement d'un objet de manière complètement générique (sans le connaître précisément tout au moins). Le message déclencheur est le Bind précédent. Une alternative aurait été de spécifier ce Set à la suite du Bind, mais nous préconisons une spécification plutôt modulaire.

Le Corps de ce comportement est trivial. C'est un Set sur l'attribut "connectivityPointer" d'un NetworkCTP sur l'instance (de mNetworkCTP) que l'on obtient par son rôle "nctp" dans la relation "NCTPIntegrity" (via l'opérateur "Part"). Le lecteur peut remarquer que les modes de connexion de ce types de comportements sont :

- fetch-phase ii : La garde doit être évaluée durant l'exécution du comportement père.
- uncoupled : Le moment où le comportement va s'exécuter n'est pas important.
- after-trigger : comportement de type "side-effect) en réaction après l'occurrence du message.

```
(define-behavior "set-nctp-connectivitypointer"
  (scope (ri "NCTPIntegrity") (role "connectivity") (msg "ivpmsg-bind"))
  (when)
  (exec-rules (fetch-phase ii) (uncoupled after-trigger))
  (pre)
  (body
     (Set (Part (ri) "nctp")
          "connectivityPointer"
          (Part (ri) "connectivity"))
  )
  (post)
)
```

Ce comportement termine la présentation de l'algorithme qui génère automatiquement la référence de configuration.

7.3.2 Le niveau MF

La présentation des deux MF que nous allons donner, respecte scrupuleusement les documents d'origine. La traduction est fidèle (cad qu'il n'y a pas de changements dans la présentation ou dans l'emploi du vocabulaire). La présentation en prose d'un MF est le plus souvent structurée de la manière suivante :

- scénario;

- résultat;
- précondition;
- postcondition.

Pour les scénari, il y a correspondance un à un entre les scénari dans METRAN et les scénari dans TIMS. Il s'agit de stimuli du système d'informations. Le résultat décrit la réponse à cette requête et ne pose donc pas de problèmes pour son intégration dans TIMS. Le problème réside en une différence de sémantique entre précondition (et postcondition) dans TIMS et dans METRAN. Dans le cas de TIMS, on associe aux assertions une sémantique bloquante de l'exécution (en d'autres termes en cas de violation des assertions, on stoppe l'exécution). Les assertions de METRAN ont une tout autre sémantique : la précondition est un ensemble de propriétés à vérifier pour lancer l'exécution tandis que la postcondition est une séquence d'action à exécuter en cas de succès ...

La notion de succès (et/ou d'échec) est très informelle. En terme de METRAN, un succès signifie que la requête s'exécute ("maction-req" par exemple) et qu'il y a émission d'une réponse positive ("maction-resp" avec le résultat de la requête en paramètre). L'échec est alors une requête qui ne s'exécute pas et qui renvoie un réponse d'erreur ("maction-resp" champs Failed en paramètre de la réponse avec le motif). Ce succès et cet échec sont au sens de TIMS deux comportements valides. Il faut en aucun cas poser une postcondition (au sens TIMS), qui arrête la simulation en cas d'échec (au sens METRAN).

Les préconditions de METRAN sont plutôt des gardes de TIMS car les préconditions de METRAN ne sont ni plus ni moins que des propriétés qui permettent l'exécution du comportement ce qui constitue la définition des gardes de TIMS.

Les postconditions de METRAN forment l'algorithme de la branche succès d'un comportement.

La spécification des assertions (au sens de TIMS) se fait de manière complètement indépendante à la spécification de METRAN.

7.3.2.1 MF 'DLC Reservation'

Comportement 7 *MFDLCRes : Un PNO (dans le rôle gestionnaire) peut demander la réservation d'un DLC en envoyant un M-ACTION avec l'opération "reserveConnection" à une instance de la classe "mLinkConnection" (un LC). L'identité du PNO demandeur est passée par paramètre dans l'action.*

Le résultat en cas de succès, contient l'identité du DLC réservé et les points de connexion (de la classe "mNetworkCTP") associés. Le résultat en cas d'échec contient la raison de l'échec :

- *plus de DLC libre $\rightarrow$ problemCause = resourceFull*
- *LC hors service $\rightarrow$ problemCause = resourceDisabled*

Précondition : la valeur de l'attribut "operationalState" du LC doit être "enabled" et la valeur de l'attribut "availableConnections" du LC doit être supérieure à 0.

Postcondition : En cas de succès, l'action change l'attribut "assignmentState" du DLC affecté à la valeur "reserved", la valeur de l'attribut "availableConnection" du LC est décrémentée de 1, l'attribut "currentOriginPNO du DLC affecté est positionné à l'identité du PNO demandeur et enfin une notification "attributeValueChange" est envoyée conformément à la MF ACCD.

a) Scénario

```
(msgsnd (moreq-action:make "reserveConnection" ... ))
```

b) Comportement BL

```
(define-behavior "reserve-DLC"
  (scope (ri "LCCapacity") (role "lc") (msg "moind-action"))
  (when  (and (equal? (msg-> actype) "reserveConnection")
              (asn=? (Get (msg-> moi) "operationalState")
                     'enabled)
              (> (Get (msg-> moi) "availableLinkConnections")
                      0)))
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
    (let ((get-dlc #f)                                          ;1
          (dlc-res '()))                                        ;2
      (for-each                                                 ;3
        (lambda (one-dlc)                                       ;4
            (if (not get-dlc)                                   ;5
               (if (asn=? (Get one-dlc "assignmentState")       ;6
                          'free)                                ;7
                  (begin (set! get-dlc #t)                      ;8
                         (set! dlc-res one-dlc)))))             ;9
        (Part (ri) "dlc"))                                      ;10
      (if get-dlc                                               ;11
         (begin (Set dlc-res                                    ;12
                     "assignmentState"                          ;13
                     'reserved)                                 ;14
                (Set (Part (ri) "lc")                           ;15
                     "availableLinkConnections"                 ;16
                     (- (Get (Part (ri) "lc")                   ;17
                             "availableLinkConnections")        ;18
                        1))                                     ;19
                (msgsnd (morsp-action:make ... ))             ;20
         )                                                      ;21
         (msgsnd (morsp-action:make "reserveConnection" ... )) ;22
      )))                                                       ;23
  (post)
)
```

- Ligne 1 du body : get-dlc est un booléen initialisé à faux.
- Ligne 8 du body : ce booléen lorsqu'il est à vrai indique nous avons récupéré un DLC.
- Ligne 2 du body : dlc-res est un indentificateur de DLC initialisé à vide.
- Ligne 9 du body : Si on récupère un DLC libre, on met son identificateur dans dlc-res.
- Dans ce comportement, on notera l'utilisation de la puissance de la modélisation par les relations. L'action est envoyée à un LC (qui a le rôle "lc" dans "LCCapicity") et des effets de bords comme le changement de l'attribut "assignmentState" du DLC (qui a le rôle "dlc" dans "LCCapicity"). D'autres effets de bords sont cablés comme le changement de la valeur du nombre des DLC disponibles, représenté par l'attribut "availableLinkConnections" du LC.
- Si à la fin de la boucle for-each (Ligne 3 -> 10), si get-dlc est toujours faux, nous sommes dans le cas d'échec. On retourne la réponse de l'action en remplissant le champs "Failed" avec la valeur "resourceFull".

Les deux comportements suivants examinent les différentes alternatives pour la non-validation de la garde (précondition au sens METRAN). On examine dans un premier temps le cas où l'attribut "availableLinkConnections" du LC indique qu'il n'y a plus de ressources disponibles et dans un second temps, on examine le cas où l'attribut "operationalState" du LC indique que celui est hors service. Dans le premier cas, on envoie comme réponse l'action en remplissant le champs "Failed" avec la valeur "resourceFull". Dans le second cas, on envoie comme reponse l'action en remplissant le champs "Failed" avec la valeur "resourceDisabled".

```
(define-behavior "can-not-reserve-DLC-resource-full"
  (scope  (ri "LCCapacity") (role "lc") (msg "moind-action"))
  (when  (and (equal? (msg-> actype) "reserveConnection")
              (asn=? (Get (msg-> moi) "operationalState")
                     'enabled)
              (not (> (Get (msg-> moi) "availableLinkConnections")
                      0)))
         )
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
   (msgsnd (morsp-action:make ... ))
   )
  (post)
)

(define-behavior "can-not-reserve-DLC-resource-disabled"
  (scope (msg "moind-action"))
  (when  (and (equal? (msg-> actype) "reserveConnection")
              (asn=? (Get (msg-> moi) "operationalState")
```

```
                    'disabled)
               (> (Get (msg-> moi) "availableLinkConnections")
                  0))
         )
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre )
  (body
   (msgsnd (morsp-action:make ... ))
   )
  (post)
  )
```

Le comportement d'un MF DLC Reservation est pratiquement totalement modélisé. La seule opération qui n'est pas modélisée est l'affection du PNO demandeur dans l'attribut "currentOriginPNO" du DLC réservé. Ceci n'est pas fait pour une raison de limitation de l'implémentation de CMISE dans notre simulateur. La gestion des ACCESS CONTROL n'est pas supportée dans la version actuelle du simulateur. Ceci n'est pas très grave et sera modélisé dans un futur proche.

La dernière référence de comportement est l'envoi d'une notification via la MF ACCD pour informer tous les PNO du changement de capacité du LC. Ce comportement est traité dans la section suivante.

7.3.2.2 MF 'ACCD (Available Connections Change Dissemination)'

Comportement 8 *MFACCD : Un PNO (dans le rôle agent) reporte que le nombre de DLC disponibles dans un LC a changé en envoyant un M-EVENT-REPORT du type "attributeValueChange". Ceci est causé par une notification créée par une instance de la classe "mLinkConnection" (un LC) qui est émise lors d'un changement sur l'attribut "availableConnection". Le M-EVENT-REPORT doit être envoyé à tous les PNO.*

a) Comportement BL La modélisation avec BL est immédiate. Nous avons spécifié ce MF comme un comportement "side-effect" qui survient après la modification de l'attribut "availableLinkConnections" du LC. On notera que l'on atteint l'objet dans un contexte de déclenchement constitué d'une relation, d'un rôle et d'un message.

```
(define-behavior "ACCD"
  (scope  (ri "LCCapacity") (role "lc") (msg "ivpmsg-set"))
  (when (equal? (msg-> attr) "availableLinkConnections"))
  (exec-rules (fetch-phase ii) (uncoupled after-trigger))
  (pre (asn=? (Get (Part (ri) "lc") "operationalState")
              'enabled))
  (body
   (msgsnd (moreq-event-report:make "attributeValueChange" ... ))
  )
  (post (asn=? (Get (Part (ri) "lc") "operationalState")
              'enabled))
```

)

Ce comportement est intéressant car il nous permet de montrer d'une part la spécification des assertions et d'autre part leur utilité. Il faut noter que dans ce cas particulier, la précondition et la postcondition sont les mêmes. L'émission de la notification "attributeValueChange", ne peut avoir lieu que si l'action "reserve-DLC" a réussi. Les conditions pour sa réussite sont : le LC est en service et le LC a suffisamment de ressources disponibles. Une précondition est levée si la ressource (le LC en l'occurrence) est hors-service lors de l'exécution du comportement. Ceci peut se produire par exemple si le LC tombe en panne, durant l'accomplissement de l'action précédente (cf MF 'DLC reservation'). La simulation est donc stopée car on se retrouve avec un système incohérent.

7.3.3 Le niveau MFS

Pour chacun des MFS, nous donnons le texte en prose issu du document d'origine. La traduction est fidèle (cad qu'il n'y a pas de changements dans la présentation ou l'emploi du vocabulaire). Les Figures 7.8, 7.9 et 7.10 sont elles aussi issues du document original. La prose et les figures constituent l'unique source de spécification. On s'aperçoit assez rapidement que le modèle d'information est déficient. L'exemple le plus frappant concerne l'objet Path (chemin (de VC-4)) qui ne fait partie du modèle d'informations alors que les comportements au niveau MFS les manipulent très fréquemment. Cette déficience du modèle d'information s'explique par le fait que l'on modélise des comportements d'objets de gestionnaires (INMS) qui sont propres à chaque opérateur. Ces objets ne sont pas visibles de l'autre côté de l'interface X (ne sont ni consultés et ni manipulés).

Nous donnons dans un premier temps les MFS tels qu'ils sont décrits dans le document et nous présentons dans un second temps notre modélisation.

7.3.3.1 MFS 'Reserve DLCs' et MFS 'UnReserve DLCs'

Comportement 9 *MFSResDLCs : La MFS 'Reserve DLCs' commence quand un utilisateur appartenant au domaine d'un PNO (demandeur) veut établir un chemin VC-12 vers une destination appartenant à un domaine d'un autre PNO. Avant la réservation effective, une recherche de la meilleure route (du point de départ à celui de destination) est calculé dans l'INMS du PNO demandeur. Le résultat de cette recherche est une liste de VC-4 (LC). Ces LC appartiennent au PNO demandeur, au PNO destination et à un ensemble de PNO intermédiaire. L'hypothèse est faite que les DLC sont réservés dans l'ordre. Le résultat de chaque demande de réservation est un DLC et l'identité des 2 NCTP qui vont être utilisés dans le chemin. La frontière effective entre 2 PNO est entre 1 DLC et 1 SN. Ceci impose que le PNO possédant le SN d'arrivée doit déclarer ces NCTP au PNO propriétaire du Lien. Cette déclaration a lieu pendant le processus de commissionnement.*

Le MFS 'Reserve DLCs' contient tous les messages qui traversent l'interface X pour la réservation des DLC le long d'une route de LC.

Le rôle du PNO répondeur peut être joué soit par le PNO de destination, soit par un PNO de transit (intermédiaire). Le rôle de PNO "autres" représente tous les PNO autres que le PNO répondeur qui doivent recevoir l'information que le nombre de DLC disponibles a changé (MF ACCD).

Les raisons d'échec de cette MFS sont : manque de capacité, ressource inconnue, ou LC hors service.

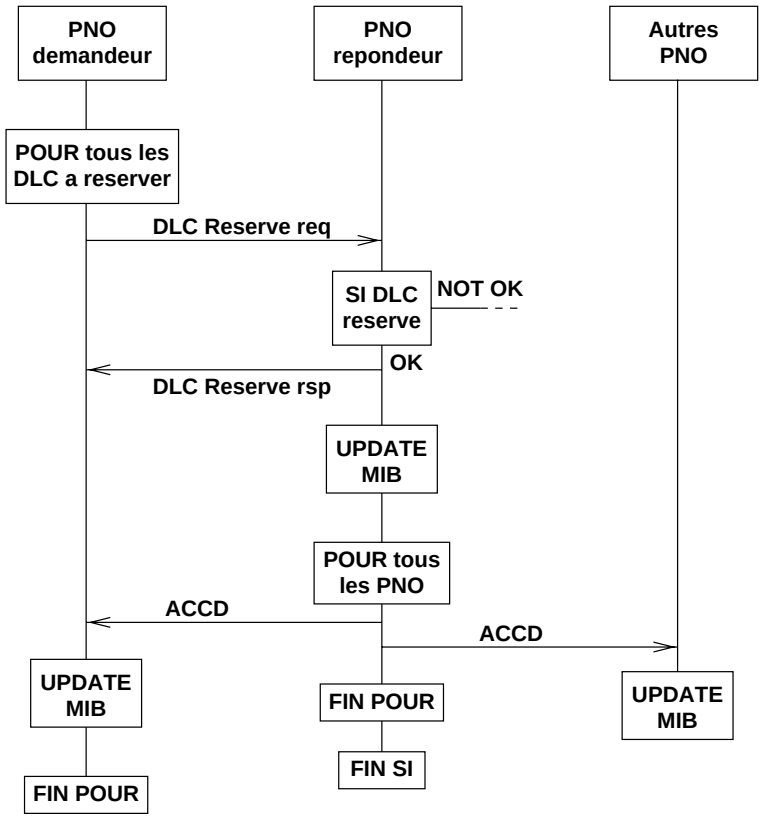

Figure 7.8: Réservation d'un ensemble de DLC

Comportement 10 *MFSUnResDLCs : La MFS 'Unreserve DLCs' est initiée lorsqu'un PNO demandeur a besoin de changer la valeur de l'attribut "assignmentState" d'un DLC ou d'un ensemble de DLC à "free". Cela peut se produire lorsqu'un LC dans le chemin refuse de donner un DLC. Dans ce cas l'ensemble des DLC étant déjà réservé doit être déréservé et une nouvelle route calculée.*

Si la déréservation d'un DLC échoue, c'est à la charge du PNO demandeur de refaire une requête de déréservation. Le PNO demandeur peut aussi choisir de ne pas faire de nouvelle requête, dans ce cas, le DLC va être déréservé par le PNO répondeur après un certain délai (MFS 'DLC time-out').

Le MFS 'Unreserve DLCs' contient tous les messages qui traversent l'interface X pour la déréservation des DLC le long d'une route de LC.

Le rôle du PNO répondeur peut être joué soit par le PNO de destination, soit par un PNO de transit (intermédiaire). Le rôle de PNO "autres" represente tous les PNO autres que le PNO répondeur qui doivent recevoir l'information que le nombre de DLC disponibles a changé (MF ACCD).

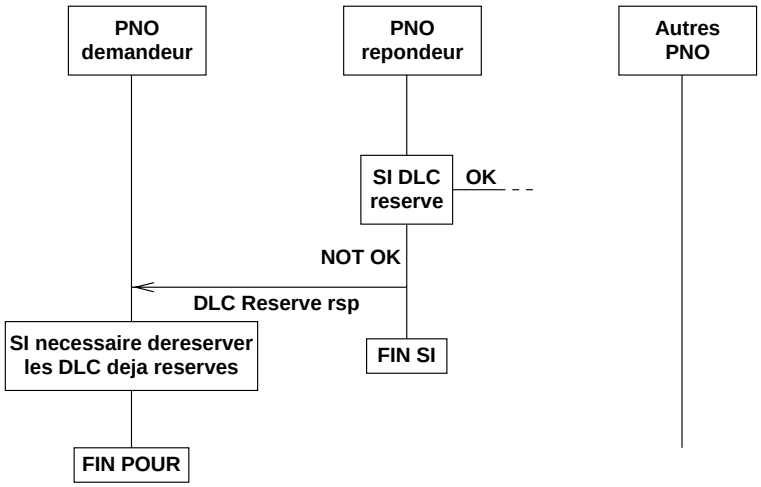

Figure 7.9: Echec dans la réservation d'un ensemble de DLC

Les raisons d'échec de cette MFS sont : déréservation d'un DLC réservé/activé par un autre PNO, tentative de déréservation d'un DLC non réservé

7.3.3.2 Modélisation en BL

Le code qui correspond à ces deux comportements se trouve en annexe D. Il est particulièrement difficile à lire puisqu'il s'agit à ce niveau de mettre en place des algorithmes de haut-niveau. En effet, nous voyons un MFS comme un méta-scenari qui doit appeler les scénari MF que l'on vient de présenter dans la section précédente.

La première partie de ce travail a consisté à spécifier un objet de la classe "mPath" (un chemin) avec ses attributs. Nous avons rapidement convergé vers un identificateur de ce chemin et un attribut qui contient le chemin de VC-4 effectif. On suppose que l'on dispose d'une fonction "best-route" qui nous rend ce chemin. L'objet s'est très vite trouvé inefficace dès lors qu'il fallait prendre en compte la gestion automatique des réservations et des déréservations en cas d'incident. C'est pourquoi cette classe a deux attributs de plus ("to-do" et "done") qui sont des attributs qui vont nous servir à effectuer les automatismes. Enfin il fallait instaurer un certain contrôle dans l'enchaînement des comportements pour qu'un méta-scénario (ou le comportement associé) puisse envoyer des messages qui déclenchent des comportements au niveau MF. Il faut ensuite récupérer les réponses de ces comportements au niveau MF pour les analyser et décider si il faut continuer la réservation ou lancer la déréservation de manière automatique.

En suivant la Figure 7.11 on peut avoir une idée de la modélisation de ces MFS. Cette modélisation commence par l'envoi d'un message "reserve-path-msg" qui déclenche le comportement "send-reserve-subpath". Par effet de cascade, des messages s'exécutent dont un moreq-action "reserveConnection" qui déclenche les scénari MF (MF Reservation et MF DLC). Ces scénari se terminent par l'envoi d'un message mocnf-action "reserveConnection". Celui-ci déclenche un comportement "receive-reserve-subpath" qui teste si la réservation s'est bien passée. Si c'est le cas

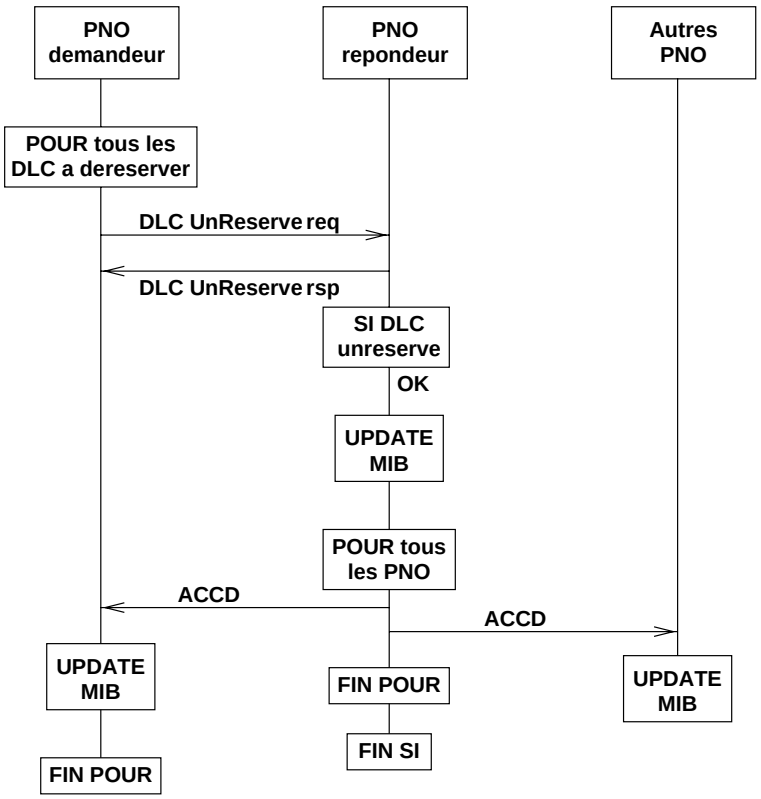

Figure 7.10: Déréservation automatique d'un ensemble de DLC

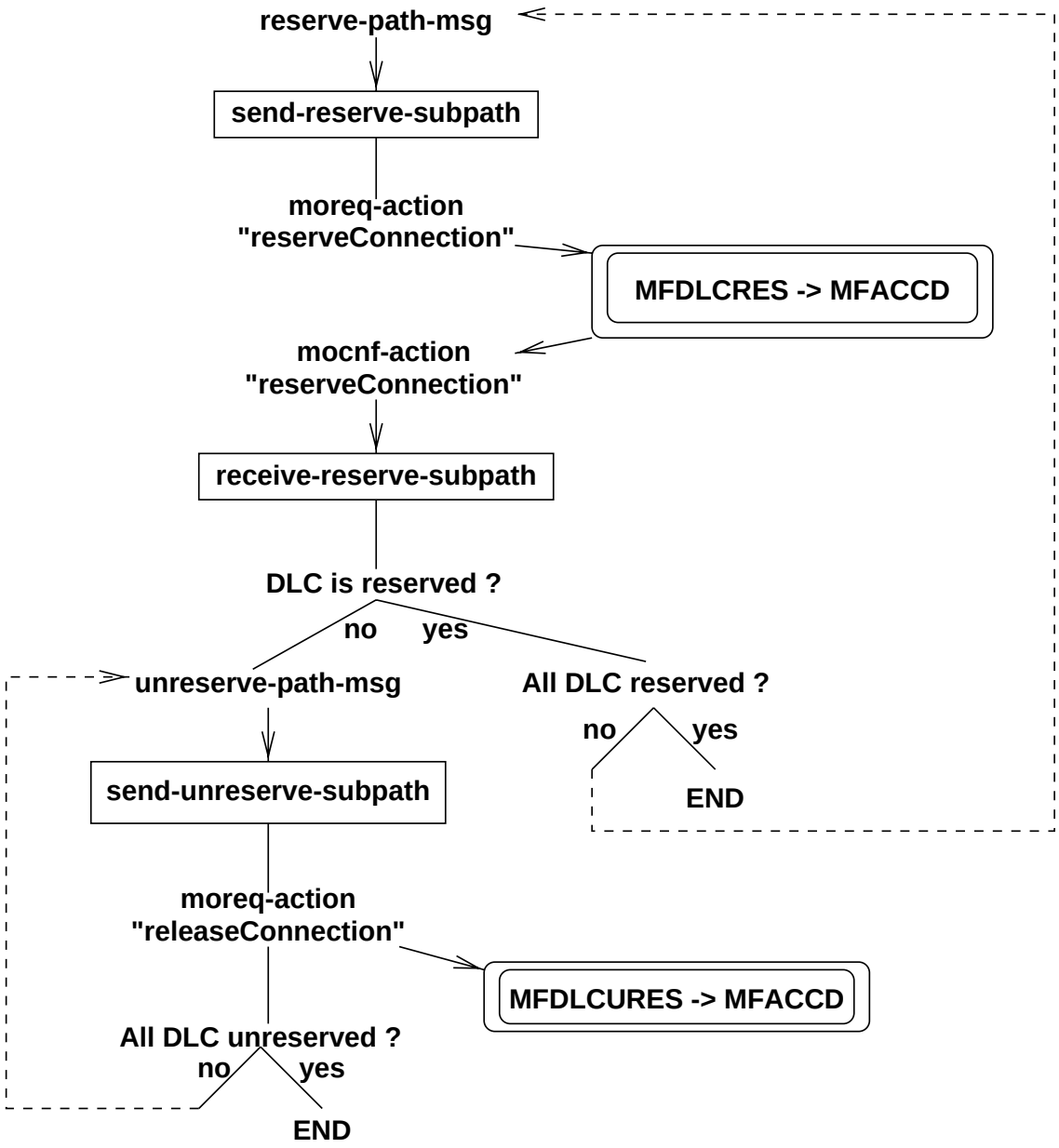

Figure 7.11: Modélisation du MFS "reserve DLCs" (et du MFS "unreserve DLCs") en BL. L'appel à des MF (DLCRES, ACCD et DLCURES) est représenté par un double rectangle. Les rectangles sont des comportements en BL.

et s'il reste des DLC à réserver, on recommence en envoyant un message "reserve-path-msg" sinon on termine. Dans le cas où la réservation s'est mal passée, on lance la déréservation automatique par l'envoi du message "unreserve-path-msg". On boucle sur ce message tant qu'il y a des DLC déjà réservés à déréserver.

A ce niveau de complexité en terme d'algorithme des comportements, un langage de comportement doit posséder toutes les fonctionnalités d'un langage de programmation. Notre choix d'intégrer BL dans le langage Scheme s'est révélé très judicieux.

7.3.4 Analyse de la spécification en BL

La couverture en terme de spécification avec BL sont :

- formalisation de la création de la topologie de référence;
- formalisation de tous les MF et de trois MFS.

BL se révèle tout autant puissant pour la spécification des comportements simples (niveau MF et création de la topologie) que pour la spécification des comportements décrits par des algorithmes assez compliqués (niveau MFS). Le résultat le plus intéressant à notre sens est que le seul problème est la compréhension de la complexité des algorithmes de METRAN eux mêmes, une fois cette complexité maitrisée, le passage à BL est facile.

En tant que concepteur du langage, il est difficile de donner des critères objectifs quant à la simplicité du langage et son utilisabilité. Des expériences ont été conduites à Swiss Telecom pour l'utilisation de BL (et du simulateur) par des ingénieurs qui n'ont pas participé au projet TIMS. Ceux-ci ont spécifié des comportements en BL sur un cas d'étude analogue au nôtre : les interfaces V5. En dehors du fait qu'il a fallu un temps normal pour que ceux-ci soient à l'aise avec l'environnement de spécification et de simulation, on peut dire que ce fut un succès en terme d'utilisabilité du système par des non-experts des méthodes formelles.

Avec ces deux remarques, on peut conclure que BL est un succès.

7.4 Exploitation de TIMS

Cette section examine TIMS dans un premier temps en tant que simulateur et dans un second temps en tant que générateur de tests. L'objectif de cette section est de montrer qu'est ce qu'il est possible d'obtenir avec TIMS, une fois que les spécifications BL et les scénari sont écrits.

7.4.1 TIMS en tant que simulateur

Pour obtenir des simulations avec TIMS :

- il faut dans un premier temps intégrer le modèle d'information (comme nous l'avons vu dans le chapitre 6 par l'interface de modèles d'information). Le modèle d'information statique (GDMO, ASN.1 et GRM) correspondant à ce cas d'étude est donné en annexe E.

- il faut dans un second temps écrire un scénario que l'on exécute à travers l'interface des scenari (comme nous l'avons vu dans le chapitre 6).
- on peut ensuite lancer une simulation. Il suffit de choisir le mode de simulation.

Le graphe d'objets et l'arbre de comportements de la simulation d'un MF sont donnés respectivement dans les Figures 6.7 et 6.8. On peut inspecter au shell la valeur d'un objet (cf Figure 7.12)

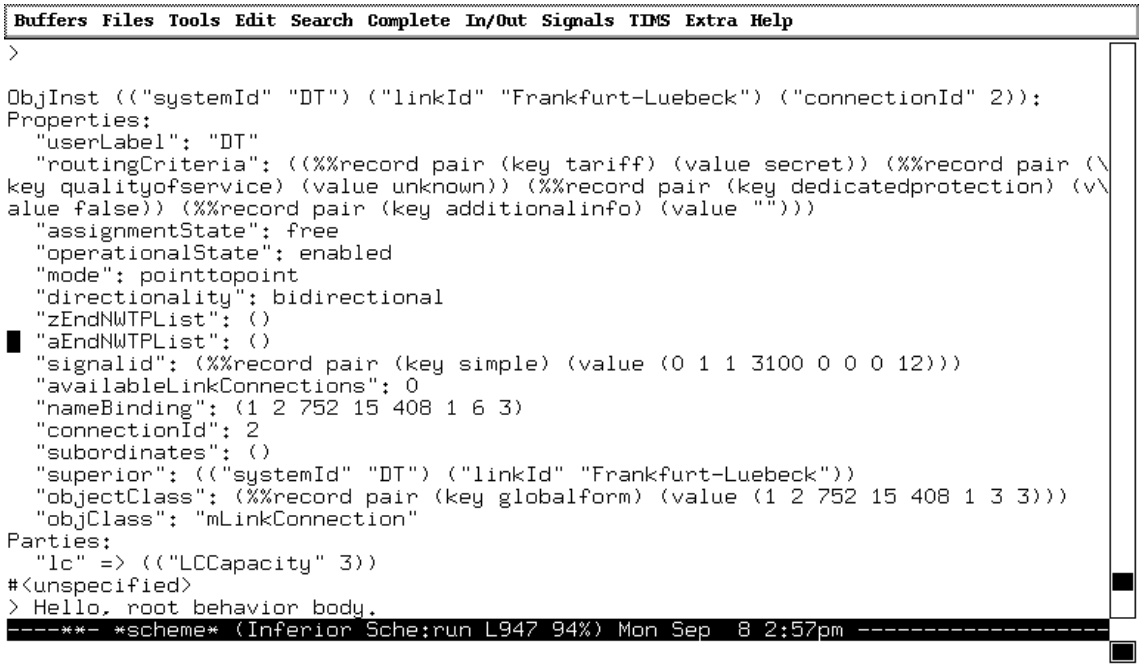

Figure 7.12: Un exemple d'objet (un LC)

ou bien celle d'un BEN (cf Figure 7.13).

7.4.2 TIMS en tant que générateur de tests

L'objectif de ce travail n'a pas été de couvrir tous les tests du projet P408, mais plutôt d'évaluer la capacité de notre méthode à générer des tests dans un contexte réel et de les comparer aux tests existants qui ont été écrits à la main dans le contexte du projet P408. Pour information, dans le contexte du projet P408, 110 tests ont été écrits. 85 sont des tests de niveau MF, 22 pour des tests de comportements valides (VB) et 63 pour des tests de comportements non valides. Les 25 tests restants sont des tests de niveau MFS. Les tests de niveau Ensembles et Inter-Ensembles n'ont pas été écrits par le projet P408.

7.4.2.1 Génération des tests MF

Ces tests sont très faciles à générer avec TIMS car l'essentiel du travail (une fois que les spécifications BL sont écrites) consiste à écrire des scénari (au sens du simulateur TIMS). En fait les

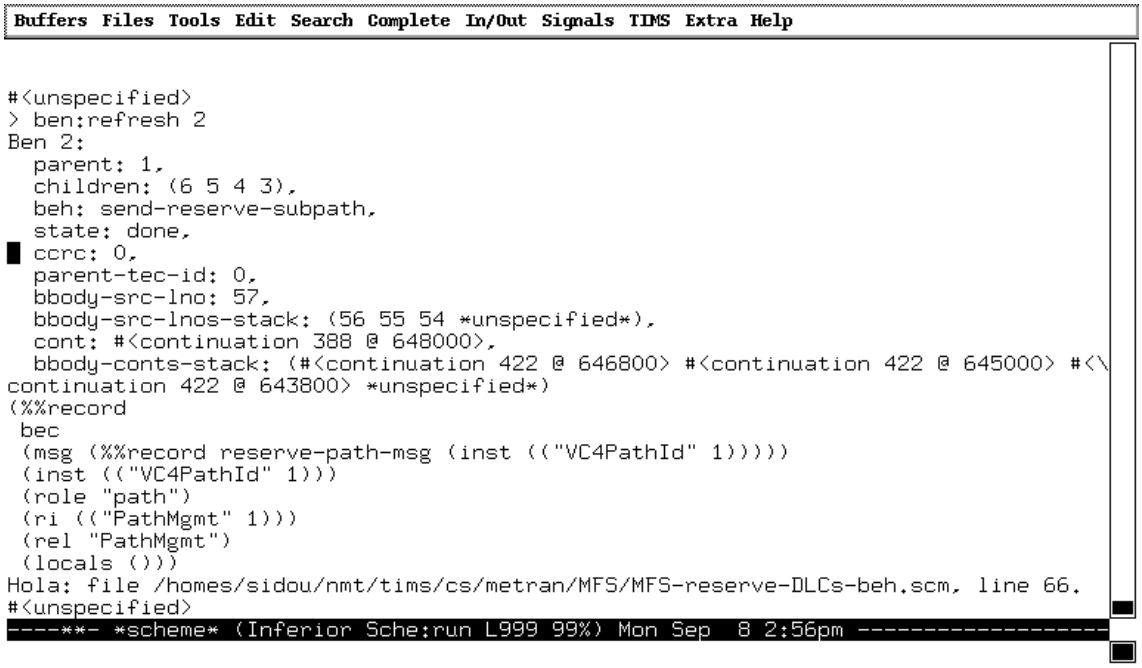

Figure 7.13: Un exemple de BEN

tests MF sont tellement basiques que nous avons pu les générer avec une simulation exhaustive classique de TIMS (cf la section 5.3 de la partie II).

7.4.2.2 Génération des tests MFS

Les problèmes en terme de puissance de TIMS ont commencé dès lors que nous sommes passés au niveau MFS où la complexité comportementale commence à être importante. A noter que cette complexité se retrouve dans le fait que très peu de tests ont été écrits à la main à ce niveau et pas du tout à des niveaux supérieurs (Ensembles et Inter-Ensembles).

C'est donc à ce moment qu'il a fallu utiliser la version de TIMS qui génère un graphe d'accessibilité réduit grâce à la méthode des ordres partiels "Sleep Set" muni de la relation de dépendance qui conserve l'équivalence observationnelle (cf la section 5.4 de la partie II).

a) Comparaison entre un test écrit à la main et un test généré automatiquement Pour effectuer la comparaison entre le test écrit à la main dans le contexte du projet P408 et le même mais généré automatiquement avec TIMS, nous nous plaçons dans le cas où il n'y a qu'une interface X à traverser (pas de PNO intermédiaires).

Si on ne considère que le corps du cas de test écrits à la main (partie body), on peut dire que le test généré automatiquement avec TIMS est quasi-identique.

- la ligne 2 de la Figure 7.15 est équivalente à la ligne 1 de la Figure 7.14;
- la ligne 4 de la Figure 7.15 est équivalente à la ligne 2 de la Figure 7.14;

```
+-------------------------------------------------------------------------+
|     Test Case Dynamic Behaviour                                         |
+-------------------------------------------------------------------------+
| Test Case Name      : beh2_res                                          |
| Group               :                                                   |
| Purpose             :                                                   |
| Default             :                                                   |
| Comments            :                                                   |
+---+---+--------------------------------------+-----------------+---+---+
| Nr| L | Behaviour Description                | Constraints Ref | V | C |
+---+---+--------------------------------------+-----------------+---+---+
| 1 |   | pco1! moreq_action, St tmocnf_action | moreq_action0   |   |   |
| 2 |   |   pco1? mocnf_action,                | mocnf_action1   |   |   |
|   |   |         Cl tmocnf_action,            |                 |   |   |
|   |   |         St tmoind_evrep              |                 |   |   |
| 3 |   |     pco1? moind_evrep, Cl tmoind_evrep|                | P |   |
| 4 |   |     ? tmoind_evrep                   |                 | F |   |
| 5 |   |   ? tmocnf_action                    |                 | F |   |
+---+---+--------------------------------------+-----------------+---+---+
```

Figure 7.14: Le test MFS ’reserve DLCs’ avec TIMS

SDHXint Aug 21, 1997 ITEX 3.01

Test Case Dynamic Behaviour

Test Case Name : MRESDLCV1
Group : MFS/MRESDLC/VB/
Purpose : Ensure that the IUT can handle all actions concerning the MFS DLC Reservation.
Configuration :
Default :
Comments :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+getAvailableConnections(LCId_1)			1.
2	body	PCO1 ! M_ACTION_req START max_rsp_timer	CMIS_dlcres_req_valid(LCId_1)		2.
3		?TIMEOUT max_rsp_timer		I	3.
4		PCO1 ? M_ACTION_cnf START max_ACCD_timer	CMIS_dlcres_cnf_valid(LCId_1)		4.
5		?TIMEOUT max_ACCD_timer		F	5.
6		PCO1 ? M_EVENT_REPORT_ind	CMIS_eventreport_ind_valid(LCId_1)	(P)	6.
7		+checkAvailConn(LCId_1, globalvarAvailConn−1)			7.

Figure 7.15: Le même test généré à la main

- la ligne 6 de la Figure 7.15 est équivalente à la ligne 3 de la Figure 7.14.

La différence réside uniquement en la gestion des timers. A cette occasion, il ne faut pas oublier que dans notre méthode, les timers sont générés automatiquement par TGV.

La différence essentielle entre les deux tests est que dans le cas des tests générés à la main, la personne qui spécifie a le loisir de spécifier des actions de haut niveau qui n'appartiennent pas au comportement observable. Elles ne sont pas présentes dans une exécution TIMS et donc on ne peut pas les générer automatiquement.

Deux requêtes de ce type sont visibles sur la Figure 7.15 :

- ligne 1 : la requête "+getAvailableConnections(LCid_1) qui stocke le nombre de DLC disponibles dans la variable globalvarAvailCon;
- ligne 7 : la requête "+checkAvailConn(LCid_1,globalvarAvailConn-1)" qui vérifie que le nombre de DLC disponibles est bien égal à la valeur de globalvarAvailConn - 1.

Avec ces requêtes, la personne qui écrit les tests, spécifie des actions de plus haut niveau sur les données. La Figure 7.16 montre la requête getAvailableConnections.

SDHXint Aug 21, 1997 ITEX 3.01

Test Step Dynamic Behaviour

Test Step Name : getAvailableConnections(lcid: lcId)
Group : Preambles/
Objective :
Default :
Comments :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		PCO3!M_GET_req	CMIS_getreq_LC_AvailConn(lcid)		1.
2		PCO3?M_GET_cnf (globalvarAvailConn := M_GET_cnf.AttributeList.mgetConfirmPDU.value)	CMIS_getcnf_LC_AvailConnAny(lcid)		

Detailed Comments : 1. Do a GET on the attribute availableConnections of managed object lcid.
2. Receive the confirmation of the get. Store the availableConnections attribute in global variable globalvarAvailConn.

Figure 7.16: Préambule généré à la main

Avec la génération du corps du test, le travail le plus difficile est fait. La génération du préambule (getAvailableConnections par exemple) peut être partiellement accomplie automatiquement. Il est facile de générer la partie séquencement d'événements observables : ligne 1 et 2 de la Figure 7.16). Ainsi, il ne reste plus au testeur qu'à écrire à la main, les appels de fonction dans le test principal (ligne 1 et 7 de la figure 7.15) et la manipulation de la variable globale dans le préambule (fin de la ligne 2 de la Figure 7.16).

Les tests dans le cas où on franchit plusieurs interfaces Xcoop n'ont pas été écrits à la main alors qu'avec TIMS, cela n'a posé aucun problème.

b) Un exemple de test au niveau Ensemble Cet exemple correspond au test du comportement spécifié dans la section précédente où on tente de réserver un chemin (un ensemble de DLC) et à cause d'un LC hors-service, il faut déréserver les DLC déjà réservé.

Là encore la génération automatique de ce test n'a pu avoir lieu que grâce à la simulation exhaustive qui construit l'arbre réduit (avec la méthode des Slep Set muni la relation de dépendance qui conserve l'équivalence observationnelle).

7.4.2.3 Analyse sur le test avec TIMS

a) Ecriture à la main La meilleure conclusion concernant le travail sur le test peut avoir pour point de départ une constatation issue du rapport final du projet P408 : "l'investissement (en terme de temps) pour apprendre à spécifier des tests et les spécifier en TTCN a été beaucoup plus important que le temps passé à spécifier les interfaces elles mêmes (Ensembles, GDMO, ASN.1)".

De plus les tests écrits à la main ne couvrent que la partie du comportement simple de ces interfaces (le niveau MF et une partie du niveau MFS).

L'avantage est que les tests écrits à la main sont mieux finis dans le sens où le testeur peut spécifier des vérifications de plus haut niveau (comme vérifier le changement de valeur d'un attribut).

b) Génération automatique Avec notre approche, la génération est entièrement automatique. Ceci évite à l'ingénieur RGT qui n'est pas spécialiste du test d'apprendre à spécifier et de spécifier des tests dans un langage inconnu (TTCN). Un avantage très important est que la génération automatique ne génère pas de tests faux (à partir du moment où la spécification des comportements est validée).

De plus nous avons montré que concernant la complexité, nous étions capables de générer des tests jusqu'au niveau Ensemble (plusieurs MFS). Le niveau inter-Ensemble ne pose théoriquement pas de problèmes particuliers [4].

Etant donné que le corps d'un test généré automatiquement est quasi identique à celui d'un test écrit à la main, on peut dire que notre méthode est un excellent générateur du squelette du test et que le travail d'augmenter le test avec des vérifications de haut niveau n'est pas très difficile (même pour un non spécialiste). En effet avec la génération du corps le travail le plus difficile est fait. La génération du préambule (donnée à la figure 7.16) peut elle aussi être accomplie automatiquement (pour la partie séquencement d'événements observables). Ainsi, il ne reste plus au testeur qu'à écrire à la main uniquement les appels de fonction dans le test principal et la manipulation de la variable globale dans le préambule.

7.5 Conclusion

Ce chapitre a débuté par une partie introductive sur le cas d'étude que constitue les interfaces Xcoop. La présentation de ces interfaces s'est déroulée en trois temps. Nous avons dans un premier

[4] nous ne l'avons pas fait dans notre cas car nous n'avons pas spécifié le deuxième Ensemble qui concerne la gestion de fautes

temps abordé la gestion de la SDH. Nous avons ensuite présenté la gestion de ces interfaces et nous avons enfin terminé par évoquer le test de ces interfaces.

La deuxième partie du chapitre a présenté dans un premier temps la modélisation et la formalisation de la création de la topologie de référence avec le langage BL. Nous avons ensuite donné des comportements BL sur quelques exemples simples (niveau MF) et sur deux exemples compliqués (niveau MFS). Outre le fait que ce dernier algorithme est difficile à comprendre, nous avons montré comment aborder ce type de problème plutôt délicat. Le résultat en terme de modélisation est très intéressant car BL se révèle très puissant pour la spécification des comportements simples aussi bien que celle des comportements compliqués.

La troisième et dernière partie de ce chapitre donne un aperçu de l'exécution de TIMS en tant que simulateur dans un premier temps et en tant que générateur de tests dans un second temps. A cette occasion, nous avons comparé les tests écrits à la main (au sein du projet P408) et ceux générés automatiquement à l'aide de TIMS. Là encore les résultats sont très encourageants car à un coût tout à fait négligeable, nous avons obtenu des tests quasi-équivalents à ceux écrits à la main et ce sans être contraint par la complexité du comportement à tester.

Chapitre 8

Conclusion Générale

8.1 Résumé du travail et des contributions

Les objectifs initiaux de cette thèse visaient en l'amélioration de la mise en service des applications du RGT. Ce problème s'est assez rapidement généralisé en la proposition de formalismes et d'outils adaptés à la spécification du comportement dynamique des systèmes à objets répartis.

- Dans le chapitre 3, nous avons proposé un formalisme : BL pour la description du comportement dynamique des systèmes à objets répartis.
- Dans le chapitre 5, nous avons proposé une méthode pour générer des tests du comportement dynamique des systèmes à objets répartis à partir de TIMS.
- Nous avons présenté le simulateur TIMS qui se compose d'un noyau générique de simulation (présenté dans le chapitre 3) et d'interfaces dédiées au RGT (présentées dans le chapitre 6).
- Nous avons ensuite appliqué, dans le chapitre 7, l'ensemble des résultats précédents à un cas d'étude réel standardisé issu du RGT que sont les interfaces Xcoop.

Outre le fait que nous proposons un formalisme et une méthode de génération des tests, notre travail de recherche a aussi consisté en la formulation des propriétés du "formalisme idéal" pour la spécification du comportement dynamique des systèmes à objets répartis (cf chapitre 3) et la formulation des propriétés de la "méthode idéale" pour la génération de tests du comportement dynamique des systèmes à objets répartis (cf chapitre 5). Ces deux ensembles de propriétés peuvent servir de point de référence pour tout autre projet qui viserait à résoudre les mêmes problèmes que les nôtres.

Il est important de préciser que le travail de spécification et de simulation a été utilisé par des ingénieurs à Swiss Telecom sur un cas d'étude du même type que le nôtre : les interfaces V5. Cette expérience nous a permis de confronter des utilisateurs qui n'ont pas participé au projet TIMS face au simulateur et au formalisme BL. En dehors du fait qu'il a fallu un temps normal pour que ceux-ci soient à l'aise avec l'environnement de spécification et de simulation, on peut dire que ce fut un succès en terme d'utilisabilité du système par des non-experts des méthodes formelles. De plus

pour ces adeptes de TIMS, maintenant que le plus dur est fait, le seul problème qu'ils auront, sera la compréhension du cas d'étude à modéliser qui est un temps nettement plus important que celui de l'apprentissage de TIMS et de BL.

Le travail sur la génération de tests a prouvé sur le papier tout au moins qu'il est utilisable dans un contexte réel. Ce travail nécessite maitenant une confrontation avec des équipes de testeurs pour évaluer son exploitation dans un contexte opérationnel.

8.2 Perspectives pour de nouveaux travaux

Les travaux futurs peuvent se subdiviser en deux catégories : les travaux à court terme et les travaux à long terme.

- les travaux à court terme concernent essentiellement des travaux d'implémentation qui seront conduits dans un futur très proche :
  - ce sont les travaux d'implémentation évoqués lors de la section 5.4 qui consistent à améliorer le temps de génération du graphe d'accessibilité. Ces travaux consistent en la génération à la volée des tests lors de la simulation exhaustive (couplage TIMS/TGV). Cette technique devrait nous permettre de garantir une génération de tests à partir de graphe d'accessibilité très grand (voire même infini).
  - ce sont les travaux d'implémentation concernant l'environnement de simulation des modèles d'informations de CORBA. Ce travail doit nous permettre de valider l'indépendance de BL et de notre méthode de génération de tests à une technologie de distribution des objets particulière. Ce travail est déjà en cours d'implémentation avec le développement uniquement d'interfaces dédiées à CORBA (ce sont en fait des interfaces avec les fonctionnalités équivalentes des interfaces présentées dans le chapitre 6). Il faudra ensuite trouver un cas d'étude (identique à Xcoop) sur lequel on pourra valider l'aspect générique du langage BL et de la génération de tests avec TIMS.
- les travaux à long terme concernent essentiellement des travaux qui peuvent constituer éventuellement des nouveaux sujets de thèses :
  - le premier travail à long terme vise à éliminer la seule limitation conséquente de notre travail qui consiste en l'hypothèse de ne pas prendre en compte le temps dans BL. Ce type de limitation nous empêche malheureusement de spécifier, simuler et tester un ensemble de propriétés liées aux performances et/ou aux aspects temps-réel des systèmes de télécommunications. Une piste possible à suivre serait d'adapter le travail effectué sur les extensions temporelles de LOTOS à BL car ces deux langages sont sémantiquement voisins.
  - le second travail à long terme concerne le problème épineux de la couverture des tests (et de leur sélection). Ce problème se pose d'autant plus vite dès lors que l'on génère des tests automatiquement (comme nous le proposons dans cette thèse). Le problème de la génération automatique est que l'on obtient un grand nombre de tests qu'il est

souvent d'un point de vue économique impossible de pratiquer dans la réalité. Une piste possible à suivre serait d'intégrer les techniques de mutation évoquées lors de la section 4.2.2.2 dans TIMS.

Annexe A

BL BNF

define⇔behavior ⇒ (**define-behavior** *label⇔spec*
scope
when
exec⇔rules
pre
body
post)

define⇔behavior⇔old ⇒ (**define-behavior-old** *label⇔spec*
scope
when
exec⇔trigger
pre
body
post)

scope ⇒ (**scope** *scope⇔spec*)

when ⇒ (**when**) |
(**when** *scheming*)

exec⇔rules ⇒ (**exec-rules** *fetch⇔phase*
coupled) |
(**exec-rules** *fetch⇔phase*
uncoupled)

fetch⇔phase ⇒ (**fetch-phase i**) |
(**fetch-phase ii**)

coupled ⇒ (**coupled before-trigger**) |
(**coupled before-trigger**
during-trigger) |
(**coupled before-trigger**
after-trigger) |
(**coupled during-trigger**) |
(**coupled is-trigger**) |
(**coupled during-trigger**
after-trigger) |
(**coupled after-trigger**)

uncoupled ⇒ (**uncoupled before-trigger**) |
(**uncoupled during-trigger**) |
(**uncoupled after-trigger**)

exec⇔trigger ⇒ (**exec-trigger before-when**) |
(**exec-trigger before-body**) |
(**exec-trigger after-body**) |
(**exec-trigger is-body**)

pre ⇒ (**pre**) |
(**pre** *scheming*)

body ⇒ (**body**) |
(**body** *scheming*)

post ⇒ (**post**) |
(**post** *scheming*)

bec ⇒ (**bec**)

inst ⇒ (**inst**)

role ⇒ (**role**)

ri ⇒ (**ri**)

rel ⇒ (**rel**)

msg ⇒ (**msg**)

msg⇔> ⇒ (**msg->** *field*) |
(**msg->** *field*1
...)

msg? ⇒ (**msg?** *rec*⇔*name*)

local⇔*get* ⇒ (**local-get** *var*)

local⇔*set*! ⇒ (**local-set!** *var*
val)

msgsnd ⇒ (**msgsnd** *msg*1
*msg*2
...)

msgsndl ⇒ (**msgsndl** *msgs*)

atomic ⇒ (**atomic** *se*1
...)

msgbufsnd ⇒ (**msgbufsnd** *se*1
...)

Annexe B

Les algorithmes de la sémantique opérationnelle de TIMS

Dans cette description, nous avons volontairement fait un grand nombre de simplifications.

- la gestion des attributs qui permettent la construction de l'arbre de comportement BET : relation entre les BEN "père" et les BEN "fils".
- la gestion des informations qui permet l'exécution en fonction du type de l'exécution (couplé ou découplé).
- la gestion des informations *undo-informations* qui permet le backtracking car le simulateur est équipé d'une fonction UNDO, (cad qu'il est possible de revenir en arrière) et qui en fait exécute le contraire de la fonction BEN_EXEC.

B.1 Chaînage avant

Le mécanisme de base de BPE est le chaînage avant. Celui-ci est donné par l'algorithme B.1.1.

Algorithme B.1.1 *Chaînage avant :*
WALK(ben_0)
1 $Stack_{enabled} \leftarrow \emptyset$
2 PUSH($Stack_{enabled}, ben_0$)
3 **while** $Stack_{enabled} \neq \emptyset$
4 **do** $ben \leftarrow$ TAKE_OUT_OF($Stack_{enabled}$)
5 $< \{ben'\} > \leftarrow$ BEN_EXEC(ben)
6 PUSH($Stack_{enabled}, \{ben'\}$)

La fonction WALK prend en entrée une transition initiale que l'on nomme ben_0 et ne rend pas de résultat. Elle utilise une structure de données : le tas $Stack_{enabled}$ des transitions tirables et les fonctions usuelles sur ce tas (PUSH qui met un élément dans le tas et TAKE_OUT_OF qui retire

un élément du tas). L'algorithme WALK s'arrête lorsqu'il n'y a plus de pas à exécuter (cad lorsque $Stack_{enabled}$ est vide. On dit alors qu'il est à saturation.

- Ligne 1 : $Stack_{enabled}$ est initialisé à vide.
- Ligne 2 : On place la transition initiale ben_0 dans le tas $Stack_{enabled}$.
- Ligne 3 : Tant que le tas $Stack_{enabled}$ n'est pas vide, on effectue les opérations suivantes:
- Ligne 4 : retirer la transition *ben* du tas $Stack_{enabled}$.
- Ligne 5 : exécuter la transition *ben* par l'appel de fonction BEN_EXEC donné par l'algorithme B.1.3. Cette exécution génère un ensemble de transitions que l'on note $\{ben'\}$.
- Ligne 6 : placer cet ensemble de transitions $\{ben'\}$ dans le tas $Stack_{enabled}$.

Dans notre cas, chaque transition est un *BEN* (Behavior Execution Node). Un *BEN* est une structure de donnée qui comprend principalement :

- *id* : l'identificateur de ce BEN;
- *state* : l'état de ce BEN;
- *beh* : un comportement (à exécuter);
- *bec* : un *BEC* (Behavior Execution Context) qui est un contexte de déclenchement cad le message déclencheur qui a servi à déclencher *beh* et éventuellement le couple (relation, rôle) dans lequel l'objet affecté par *beh* participe.

Cette structure de données représente une unité d'exécution.

B.1.1 Automate d'état simplifié d'un BEN

Le cycle de vie d'un BEN (et donc la gestion de l'état *state*) est donné par l'algorithme B.1.2.

Algorithme B.1.2 *BEN FSM :*
NEXT_STATE_BEN(*ben*)
1 **if** *ben.state* = *"wait"*
2 **then** *ben.state* ← *"ready⇔pre"*
3 **if** *ben.state* = *"ready⇔pre"*
4 **then** *ben.state* ← *"ready"*
5 **if** *ben.state* = *"ready"*
6 **then** *ben.state* ← *"ready⇔post"*
7 **if** *ben.state* = *"ready⇔post"*
8 **then** *ben.state* ← *"done"*

- Ligne 1 : L'état initial d'un BEN est obtenu à la création du BEN, celui-ci est alors dans l'état "wait";
- Ligne 2 : Lorsque l'on vérifie la précondition du comportement *beh* affecté à un BEN, celui-ci passe de l'état "wait" à "ready-pre";
- Ligne 4 : Lorsque l'on commence l'exécution du comportement *beh* affecté à un BEN, (donc après la vérification de la précondition) , celui-ci passe de l'état "ready-pre" à "ready";
- Ligne 6 : Lorsque l'on vérifie la **Postcondition** du comportement *beh* affecté à un BEN, celui-ci passe de l'état "ready" à "ready-post";
- Ligne 8 : L'état final d'un BEN est obtenu après la vérification de la postcondition, on passe alors de l'état "ready-post" à "done";

Ce cycle de vie est calculé par rapport à la description d'un comportement tel que nous venons de le présenter. De manière générale l'exécution de *beh* du BEN se résume en trois phases : l'évaluation de la précondition, l'évaluation du corps et enfin l'évaluation de la postcondition. On obtient ainsi principalement l'algorithme de l'exécution d'un BEN donné dans la section suivante.

B.1.2 Exécution d'un BEN

Comme nous venons de le voir dans le cycle de vie d'un BEN (dans la section précédente). L'exécution d'un BEN est régie par les trois phases du comportement *beh* associé à un BEN. L'exécution d'un BEN est donnée par l'algorithme B.1.3.

Algorithme B.1.3 *Exécution d'un BEN :*
BEN_EXEC(*ben*)

```
1  switch
2     case ben.state = "ready⇔pre" :
3          VERIF_PRE(ben)
4     case ben.state = "ready" :
5          return EXEC_STEP_BEN(ben)
6     case ben.state = "ready⇔post" :
7          return VERIF_POST(ben)
```

- Ligne 1 : Si le BEN *ben* est dans l'état *state* "ready-pre"
- Ligne 2 : alors vérifier la précondition du *beh* affecté à ce BEN par l'appel de la fonction VERIF_PRE donnée par l'algorithme B.1.4.
- Ligne 3 : Si le BEN *ben* est dans l'état *state* "ready"
- Ligne 4 : alors exécuter un pas unitaire de comportement *beh* affecté à ce BEN par l'appel de la fonction EXEC_STEP_BEN donné par l'algorithme B.1.6. Retourner la liste des BEN qui ont été éventuellement créés lors de l'exécution de cette fonction (dû au fetch phase II).

- Ligne 5 : Si le BEN *ben* est dans l'état *state* "ready-post"
- Ligne 6 : alors vérifier la postcondition du *beh* affecté à ce BEN par l'appel de la fonction VERIF_POST donnée par l'algorithme B.1.5. Retourner la liste des BEN qui ont été éventuellement créés lors de l'exécution de cette fonction (dû au fetch phase I).

B.1.3 Vérification de la Précondition

Cette fonction évalue tout simplement la précondition.

Algorithme B.1.4 *Vérification de la Précondition :*
VERIF_PRE(*ben*)
1 **if** *ben.beh.pre* = *True*
2 **then** NEXT_STATE_BEN(*ben*)
3 **else** **error** *"Precondition violated"*

- Ligne 1 : Si la précondition *Pre* du comportement *beh* affecté au BEN *ben* est vraie
- Ligne 2 : alors changer d'état par l'appel de la fonction NEXT_STATE_BEN décrite par l'algorithme B.1.2.
- Ligne 3 : sinon lever l'exception "Precondition violated".

B.1.4 Vérification de la Postcondition

Cette fonction évalue la postcondition et met le BEN dans l'état *state* "done". Le comportement *beh* a été entièrement exécuté. Si le comportement est de type "is-trigger" (cf section 3.3.2.3), il faut calculer les comportements à exécuter de types phase II.

Algorithme B.1.5 *Vérification de la Postcondition :*
VERIF-POST(*BEN*)
1 **if** *ben.beh.post* = *True*
2 **then** NEXT_STATE_BEN(*ben*)
3 **if** *ben.beh.type* = *"is⇔trigger"*
4 **then** $\{ben'\} \leftarrow$ FETCH(*ben.bec*, *phase⇔II*)
5 **return** $\{ben'\}$
6 **else** **error** *"Postcondition violated"*

- Ligne 1 : Si la postcondition *Post* du comportement *beh* affecté au BEN *ben* est vraie
- Ligne 2 : alors changer d'état par l'appel de la fonction NEXT_STATE_BEN décrite par l'algorithme B.1.2.

- Ligne 3 : Maintenant que le comportement *beh* est entièrement exécuté, si *beh* est de type "is-trigger"
- Ligne 4 : alors exécuter l'appel de la fonction FETCH qui est décrite par l'algorithme B.1.7 en passant par paramètre le BEC et la phase (II en l'occurrence). Le résultat est l'ensemble des nouveaux BEN à exécuter { ben' }.
- Ligne 5 : Retourner la liste des nouveaux BEN à exécuter { ben' } à la fonction appelante BEN_EXEC (cf algo algo:ben-exec).
- Ligne 6 : sinon lever l'exception "Postcondition violated".

B.1.5 Exécution d'un pas de BEN

Cette fonction exécute un pas de BEN quand il est dans l'état *state* "ready". On exécute alors un pas du *Corps* d'un comportement. Si le comportement est entièrement exécuté, on passe dans l'état "ready-post".

Algorithme B.1.6 *Exécution d'un pas de BEN :*

```
EXEC_STEP_BEN(ben)
1  msg⇔list ← NEXT_STEP_BODY(ben.beh.corps)
2  if ben.beh.corps = NIL
3    then NEXT_STATE_BEN(ben)
4  repeat
5         msg ← POP(msg⇔list)
6         {ben'} ← FETCH(bec ← msg, phase⇔I)
7         APPEND(ben⇔list, {ben'})
8    until msg⇔list = NIL
9  return ben⇔list
```

- Ligne 1 : L'unité d'exécution des comportements se calcule sur la base des envois de messages qui sont spécifiés dans le **Corps** *corps* du comportement *beh* affecté à un BEN *ben*. Le calcul de ce pas unitaire se fait par l'appel de fonction NEXT_STEP_BODY (qui n'est pas décrite). Cette fonction retourne la liste des messages susceptibles de déclencher des comportements $msg \Leftrightarrow list$.
- Ligne 2 : S'il ne reste plus de pas de comportements à exécuter dans le *corps*.
- Ligne 3 : alors changer d'état par l'appel de la fonction NEXT_STATE_BEN décrite par l'algorithme B.1.2.
- Ligne 4 : tant que la liste des messages susceptibles de déclencher des comportements $msg \Leftrightarrow list$ n'est pas vide
- Ligne 5 : retirer un message *msg* de la liste $msg \Leftrightarrow list$

- Ligne 6 : exécuter l'appel de la fonction FETCH qui est décrite par l'algorithme B.1.7 en passant par paramètre le BEC (comprenant msg) et la phase (I en l'occurrence). Le résultat est l'ensemble des nouveaux BEN à exécuter { ben' }.

- Ligne 7 : Ajouter la liste des nouveaux BEN à exécuter { ben' } à la liste des nouveaux BEN à exécuter déjà calculée $ben \Leftrightarrow list$.

- Ligne 9 : Retourner la liste des nouveaux BEN à exécuter $ben \Leftrightarrow list$ à la fonction appelante BEN_EXEC (cf algo algo:ben-exec).

B.1.6 Sélection de comportements à exécuter

La recherche des nouveaux comportements à exécuter se fait par un algorithme (Fetch) qui comprend deux étapes: Le Scopping et le Filtering qui ne sont pas décrits.

Algorithme B.1.7 *Sélection de comportements :*
FETCH$(bec, phase)$
1 $bec \Leftrightarrow list \leftarrow$ SCOPPING$(bec, phase)$
2 $ben \Leftrightarrow list \leftarrow$ FILTERING$(beh \Leftrightarrow list)$
3 **return** $ben \Leftrightarrow list$

- Ligne 1 (Scopping) : En fonction de leur clause **Portée** et de la clause **Mode de Connexions**, on sélectionne les comportements potentiellement exécutables en parcourant le fichier de spécification de comportements (en fait un conteneur de comportements). On regroupe chaque comportement potentiellement exécutable dans une liste de BEC $bec \Leftrightarrow list$.

- Ligne 2 (Filtering) : En fonction de la **Garde** et de la liste des becs $bec \Leftrightarrow list$ qui correspond aux comportements potentiellement exécutables (issue du scopping), on sélectionne les comportements à exécuter. Ces comportements à exécuter forment un ensemble de nouveaux BEN $ben \Leftrightarrow list$.

- Ligne 3 : Retourne la liste des nouveaux BEN à exécuter $ben \Leftrightarrow list$ à la fonction appelante VERIF_POST ou EXEC_STEP_BEN suivant la phase.

B.2 L'algorithme de simulation TIMS en mode exhaustif

Sans entrer dans les détails (notamment gestion du backtracking fonction UNDO), la simulation exhaustive peut se résumer par l'algorithme B.2.1 DFS (et DFS-REC) qui est en fait une réécriture de la fonction WALK pour lequel grâce à la récursivité, on exécute toutes les combinaisons possibles. C'est une descente en profondeur d'abord.

Algorithme B.2.1 *DFS :*
DFS$()$

```
1  Stack_enabled ← ∅
2  PUSH(Stack_enabled, ben_0)
3  Stack_exec ← ∅
4  DFS-REC()
```

La fonction DFS prend en entrée une transition initiale que l'on nomme ben_0 et ne rend pas de résultat. Elle utilise deux structures de données : le tas $Stack_{enabled}$ des transitions tirables et le tas $Stack_{exec}$ des transitions exécutées. Elle utilise, de plus, les fonctions usuelles de manipulation de tas (PUSH qui met un élément dans le tas et POP qui retire un élément dans le tas).

- Ligne 1 : le tas $Stack_{enabled}$ des transitions à tirer est initialisé à vide.
- Ligne 2 : On place la transition initiale ben_0 dans le tas $Stack_{enabled}$ des transitions à tirer.
- Ligne 3 : le tas $Stack_{exec}$ des transitions exécutées est initialisé à vide.
- Ligne 4 : Appel de la fonction récursive DFS-REC.

Algorithme B.2.2 *DFS-REC :*

```
DFS-REC()
1  repeat
2        ben ← TAKE_OUT_OF(Stack_enabled)
3        PUSH(Stack_exec, ben)
4        {ben'} ← BEN_EXEC(ben)
5        PUSH(Stack_enabled, {ben'})
6        DFS-REC()
7        ben ← POP(Stack_exec)
8        UNDO(ben)
9    until Stack_enabled = ∅
```

- Ligne 1 : Tant que le tas $Stack_{enabled}$ n'est pas vide, on effectue les opérations suivantes:
- Ligne 2 : On retire la transition *ben* du tas des transitions à tirer $Stack_{enabled}$.
- Ligne 3 : On place cette transition *ben* dans le tas des transitions exécutées $Stack_{exec}$.
- Ligne 4 : On exécute la transition *ben* par l'appel de la fonction BEN_EXEC décrite par l'algorithme B.1.3. Cette exécution génère un ensemble de transitions que l'on note $\{ben'\}$.
- Ligne 5 : On place cet ensemble de transitions $\{ben'\}$ dans le tas $Stack_{enabled}$ des transitions à tirer.
- Ligne 6 : On effectue un appel récursif sur la fonction DFS-REC.
- Ligne 7 : On retire la transition *ben* du tas des transitions exécutées $Stack_{exec}$.
- Ligne 8 : On exécute l'inverse de la transition *ben* par l'appel de fonction UNDO.

B.3 Algorithme des "Sleep-set"

C'est en fait l'algorithme B.2.1 auquel on ajoute la gestion des "Sleep Set" et pour lequel on définit la relation de dépendance Dop entre deux transitions qui se base sur les opérations (modifications) bas niveau de l'IR (le conteneur d'instances). Ces opérations sur l'IR (qui est en fait une table de hashing) sont les suivantes :

- (*create entry*)
- (*delete entry*)
- (*get entry property*)
- (*set entry property value*)

Définition 15 *(relation de dépendance Dop)*
La relation de dépendance Dop entre des opérations basiques est donnée par les deux règles qu'il faut appliquer en séquence :

1. *l'opération create et l'opération delete sont dépendantes avec toutes autres opérations sur la même entrée entry;*
2. *l'opération set est dépendante avec les opérations set et get sur la même propriété property d'une même entrée entry.*

Algorithme B.3.1 *DFS-SLEEP-SET :*
DFS()
1 $Stack_{enabled} \leftarrow \emptyset$
2 PUSH($Stack_{enabled}, ben_0$)
3 $Stack_{exec} \leftarrow \emptyset$
4 $Sleep \leftarrow \emptyset$
5 DFS-REC($Sleep$)

- Ligne 4 On initialise le tas des Sleep Set *Sleep*.
- Ligne 5 On appelle la fonction récursive DFS-REC en passant par paramètre le tas des Sleep Set *Sleep*.

Algorithme B.3.2 *DFS-SLEEP-SET-REC :*
DFS-REC($Sleep$)
1 **repeat**
2 $ben \leftarrow$ TAKE_OUT_OF($Stack_{enabled}$)
3 PUSH($Stack_{exec}, ben$)
4 $\{ben'\} \leftarrow$ BEN_EXEC(ben)
5 PUSH($Stack_{enabled}, \{ben'\}$)

```
6        DFS-REC(Sleep \ {ben' ∈ Sleep : ben' et bensont indpendants})
7        ben ← POP(Stack_exec)
8        UNDO(ben)
9        Sleep ← Sleep ⋃ ben
10     until Stack_enabled = ∅
```

- Ligne 6 On appelle récursivement en ajoutant la transition ben' au tas des Sleep Set $Sleep$ si ben' est indépendant de ben.
- Ligne 9 On ajoute la transition ben au tas des Sleep Set $Sleep$.

Annexe C

Les algorithmes pour le test

C.1 Les modifications de la sémantique opérationnelle de TIMS

C.1.1 Détection des messages

Elle se fait par modification de l'algorithme B.1.6 d'exécution d'un pas unitaire du simulateur TIMS par

- d'une part ajout de chaque message dans la liste des messages $List_{mess}$ qui ont été détectés durant le parcours du chemin depuis l'état initial et ce jusqu'à la terminaison de l'exécution (ligne 6 de l'algorithme C.1.1).

- d'autre part ajout de chaque message dans une table de messages $Hash_{mess}$. On associe une clé (un entier unique) qui représente symboliquement chaque message et nous permet par la suite de ne traiter que des entiers (ligne 7 de l'algorithme C.1.1).

Algorithme C.1.1 *Exécution d'un pas de BEN :*

EXEC_STEP_BEN(ben)

```
 1  msg⇔list ← NEXT_STEP_BODY(ben.beh.corps)
 2  if ben.beh.corps = NIL
 3     then NEXT_STATE_BEN(ben)
 4  repeat
 5          msg ← POP(msg⇔list)
 6          PUSH(List_mess, msg)
 7          PUSH(Hash_mess, msg)
 8          {ben'} ← FETCH(bec ← msg, phase⇔I)
 9          APPEND(ben⇔list, {ben'})
10     until msg⇔list = NIL
11  return ben⇔list
```

C.1.2 Construction de $Array_{path}$

Elle se fait par modification de l'algorithme B.2.2 de simulation exhaustive en deux étapes:

- initialisation des structures de données $List_{mess}$, $Hash_{mess}$, et $Array_{path}$ (ligne 3, 4 et 5 de l'algorithme C.1.2)
- A chaque terminaison d'exécution (lorsqu'il n'y a plus de BEN à exécuter), on ajoute $List_{mess}$ au tableau des chemins $Array_{path}$. L'ajout n'est effectué que si $List_{mess}$ est un nouveau chemin. (ligne 10 de l'algorithme C.1.3)

Algorithme C.1.2 *DFS :*

```
DFS()
1  Stack_enabled ← ∅
2  PUSH(Stack_enabled, ben_0)
3  List_mess ← ∅
4  Hash_mess ← ∅
5  Array_path ← ∅
6  Stack_exec ← ∅
7  DFS-REC()
```

Algorithme C.1.3 *DFS-REC :*

```
DFS-REC()
 1  repeat
 2        ben ← TAKE_OUT_OF(Stack_enabled)
 3        PUSH(Stack_exec, ben)
 4        {ben'} ← BEN_EXEC(ben)
 5        PUSH(Stack_enabled, {ben'})
 6        DFS-REC()
 7        ben ← POP(Stack_exec)
 8        UNDO(ben)
 9    until Stack_enabled = ∅
10  ADD_IF_NEW(Array_path, List_mess)
```

C.2 Les algorithmes propres au test

C.2.1 Construction du graphe d'accessibilité

L'algorithme MAKE-TREE appelle l'algorithme TREE-WALK (cf Algo C.2.2 pour chacune des listes de messages $List_{Mess}$ (qui represésente un chemin) de $Array_{path}$.

Algorithme C.2.1 *Make-tree :*

```
MAKE-TREE()
1  repeat
2        List_Mess ← POP(Array_path)
3        TREE-WALK(List_Mess, 0)
4   until Array_path = ∅
```

- Ligne 1 : retire la liste des messages $List_{Mess}$ de $Array_{path}$;
- Ligne 2 : appel de l'algorithme TREE-WALK en passant par paramètre la liste des messages et la racine du graphe d'accessibilité.

L'algorithme TREE-WALK parcourt en parallèle la liste *List* et la table de hashing *Hash* qui représente le graphe d'accéssiblité. Pour chaque noeud à partir de la racine, on regarde s'il y a un noeud fils qui peut être atteint par le message en tête de liste. L'examen qui nous permet de savoir si un noeud fils existe est donné par l'algorithme INSPECT-SONS (cf Algo C.2.3).

Algorithme C.2.2 *Tree-Walk :*

```
TREE-WALK(List, current ⇔node)
1  if List ≠ NIL
2    then Node ← HASH(current ⇔node)
3         res ← INSPECT-SONS(HEAD(List), Node.sons)
4         if FIRST(res) = true
5           then return TREE-WALK(Tail(List), SECOND(res))
6           else  return TREE-WALK(Tail(List), PUSH(Hash, HEAD(List), current ⇔node))
```

- Ligne 1 : si la liste de messages n'est pas vide;
- Ligne 2 : alors récupère le noeud du graphe dont l'identificateur est passé en paramètre à la fonction;
- Ligne 3 : appel de l'algorithme INSPECT-SONS en passant par paramètre le message en tête de liste des messages (de la liste $List_{Mess}$)
- Ligne 4 : si INSPECT-SONS retourne un couple dont le premier élément est vrai
- Ligne 5 : appel récursivement la fonction Tree-Walk en passant le reste de la liste des messages (de la liste $List_{Mess}$) et l'identificateur du noeud courant (donné par le second élément du couple résultant de INSPECT-SONS)
- Ligne 6 : (INSPECT-SONS retourne un couple dont le premier élément est faux)
  appel récursivement la fonction Tree-Walk en passant le reste de la liste des messages (de la liste $List_{Mess}$) et en créeant un nouveau noeud.

L'algorithme INSPECT-SONS regarde pour chaque fils s'il a été atteint par le message *mess*. Si c'est le cas, il retourne un couple avec le booléen vrai et l'indentificateur du noeud fils.

Algorithme C.2.3 *Inspect-Sons :*

INSPECT-SONS($mess, list \Leftrightarrow of \Leftrightarrow sons$)

```
1  res ← < false, 0 >
2  repeat
3        current ⇔son ← POP(list ⇔of ⇔sons)
4        Node ← HASH(current ⇔son)
5        if FIRST(res) = false
6          then if Node.msg = mess
7                  then res ← < true, Node.id >
8    until list ⇔of ⇔sons = ∅
9  return res
```

- Ligne 1 : initialise le résultat de cette fonction au couple (booléen faux et identificateur 0);
- Ligne 2 : tant que la liste des fils $list \Leftrightarrow of \Leftrightarrow sons$ n'est pas vide;
- Ligne 3 : retire l'identificateur du fils courant $current \Leftrightarrow son$ de la liste $list \Leftrightarrow of \Leftrightarrow sons$;
- Ligne 4 : récupère le noeud du graphe depuis l'identificateur du fils courant $current \Leftrightarrow son$;
- Ligne 5 : si le premier élément du couple résultat est faux;
- Ligne 6 : si le message passé par paramètre est égal au message qui a servi à atteindre le noeud du fils courant;
- Ligne 7 : alors change le couple résultat par le booléen vrai pour premier élément et l'identificateur du fils courant comme second élément.

C.2.2 Décoration des transitions

Cet algorithme se sert alors d'informations spécifiques à l'activité de test comme :

- la liste des messages observables $*incoming \Leftrightarrow list*$;
- la liste des messages contrôlables $*outgoing \Leftrightarrow list*$;
- la liste des messages internes $*internal \Leftrightarrow list*$;
- la localisation des PCO.

On marque d'un ! les messages observables, d'un ? les messages contrôlables et d'un "i" les messages internes.

Algorithme C.2.4 *Treat-msg :*

```
TREAT-MSG(mess)
1  if messin ∗ internal ⇔list∗
2    then res ← "i"
3    else switch
4           case messin ∗ incoming ⇔list∗ :
5                res ← STRING-APPEND(GET-PCO(mess), "!", mess.name)
6           case messin ∗ outgoing ⇔list∗ :
7                res ← STRING-APPEND(GET-PCO(mess), "?", mess.name)
8         return res
```

- Ligne 1 : si le message *mess* passé par paramètre est un message interne (appartient à la liste $*internal \Leftrightarrow list*$
- Ligne 2 : alors étiquette la transition avec un i
- Ligne 3 : sinon suivant le cas où
- Ligne 4 : le message *mess* passé par paramètre est un message observable (appartient à la liste $*incoming \Leftrightarrow list*$)
- Ligne 5 : étiquette la transition avec le nom du PCO, un ! et le nom du message
- Ligne 6 : le message *mess* passé par paramètre est un message contrôlable (appartient à la liste $*outgoing \Leftrightarrow list*$)
- Ligne 7 : étiquette la transition avec le nom du PCO, un? et le nom du message

L'algorithme de la fonction GET-PCO est dépendant de la technologie de distribution des objets et de son implémentation.

C.2.3 Formattage de sortie

On parcourt simplement *Hash* qui est la représentation en mémoire du graphe d'accessibilité pour générer l'IOLTS représentant la spécification dans le format aldébaran qui est donné par la figure 5.4.

Algorithme C.2.5 *Tree2Aldebaran :*

```
TREE2ALDEBARAN()
1  WRITE("(des(0, ∗nb_tr∗, ∗nb_st∗)"))
2  while (Node ← POP(Hash)) ≠ NIL
3     do NEWLINE()
4        WRITE("(")
5        WRITE(Node.father)
```

```
6        WRITE(",")
7        WRITE(TREAT-MSG(Node.msg))
8        WRITE(",")
9        WRITE(Node.id)
10       WRITE(")"))
11
```

- Ligne 1 : 0 est l'état initial du LTS, $*nb_{tr}*$ donne le nombre de transitions du LTS et $*nb_{st}*$ donne le nombre d'états.
- Ligne 2 : Pour tous les noeuds *Node* du graphe d'accessibilité représenté par *Hash*
- Ligne 5 : écrire l'identificateur du noeud père
- Ligne 7 : écrire le résultat de l'appel de la fonction TREAT-MSG donnée par l'algorithme C.2.4.
- Ligne 9 : écrire l'identificateur du noeud

Annexe D

Modélisation de Xcoop en BL

Pour chacun des MF et des MFS, le texte en prose issu du document d'origine (et les figures dans le cas des MFS) se trouve dans le chapitre 7.

D.1 Le niveau MF

D.1.1 MF DLC Reservation

```
(define-behavior "reserve-DLC"
  (scope (ri "LCCapacity") (role "lc") (msg "moind-action"))
  (when  (and (equal? (msg-> actype) "reserveConnection")
              (asn=? (Get (msg-> moi) "operationalState") 'enabled)
              (> (Get (msg-> moi) "availableLinkConnections") 0)))
  (exec-rules (fetch-phase i) (coupled is-trigger))
  (pre)
  (body
   (let ((get-dlc #f)
         (dlc-res '()))

   ;;; gets a Free DLC -> DLCres that belongs to the LC

     (for-each (lambda (one-dlc)
                 (cond (get-dlc)
                       ((asn=? (Get one-dlc "assignmentState") 'free)
                        (set! get-dlc #t)
                        (set! dlc-res one-dlc))))
               (Part (ri) "dlc"))

     (cond (get-dlc

   ;;; If SUCCESS sets DLCres.assignmentState to reserved
            (Set dlc-res "assignmentState" 'reserved)

   ;;; If SUCCESS sets DLCres.currentOriginPNO to the requesting PNOId
          ;;; we should implement access control field in CMISE...

   ;;; If SUCCESS decreases LC.availableLinkConnection by one
            (Set (Part (ri) "lc") "availableLinkConnections"
                 (- (Get (Part (ri) "lc") "availableLinkConnections")
                    1))

   ;;; If SUCCESS returns reserveConnectionReply: the passed field
            (msgsnd (morsp-action:make2
```

```
                    '(msd ,(msg-> msd))
                    '(invoke ,(msg-> invoke))
                    '(moi ,(msg-> moi))
                    '(actype "reserveConnection")
                    '(acreply
                      ,(asn:choice-value
                        'passed
                        (asn:seq
                         (asn:field-value 'connectionId (moi:scm->asn dlc-res))
                         (asn:field-value 'aNetworkCTPId
                                          (list-ref (Get dlc-res "aEndNWTPList") 0)) ; ds: fixme
                         (asn:field-value 'zNetworkCTPId
                                          (list-ref (Get dlc-res "zEndNWTPList") 0)) ; ds: fixme
                         )))))

           );;; get-dlc is true
          (else
          ;;; If FAILED returns reserveConnectionReply: the failed field
           (msgsnd (morsp-action:make2
                    '(msd ,(msg-> msd))
                    '(invoke ,(msg-> invoke))
                    '(moi ,(msg-> moi))
                    '(actype "reserveConnection")
                    '(acreply
                      ,(asn:choice-value
                        'failed
                        (asn:choice-value
                         'logicalProblem
                         (asn:seq (asn:field-value 'problemCause
                                                   (asn:choice-value 'integerValue 22))))))))
           );;; get-dlc is false
          );;;cond
      );;;let
     );;;body
    (post)
    )

  (define-behavior "can-not-reserve-DLC-resource-disabled"
    (scope (msg "moind-action"))
    (when  (and (equal? (msg-> actype) "reserveConnection")
                ;(asn=? (Get (msg-> moi) "operationalState") 'enabled)
                (asn=? (Get (msg-> moi) "operationalState") 'disabled)
                (> (Get (msg-> moi) "availableLinkConnections") 0))
           )
    (exec-rules (fetch-phase i) (coupled is-trigger))
    (pre )
    (body
     (msgsnd (morsp-action:make2
                    '(msd ,(msg-> msd))
                    '(invoke ,(msg-> invoke))
                    '(moi ,(msg-> moi))
                    '(actype "reserveConnection")
                    '(acreply
                      ,(asn:choice-value
                        'failed
                        (asn:choice-value
                         'logicalProblem
                         (asn:seq (asn:field-value 'problemCause
                                                   (asn:choice-value 'integerValue 23))))))
     )
    (post)
    )

  (define-behavior "can-not-reserve-DLC-resource-full"
    (scope (msg "moind-action"))
```

```
      (when  (and (equal? (msg-> actype) "reserveConnection")
                  (asn=? (Get (msg-> moi) "operationalState") 'enabled)
                  (not (> (Get (msg-> moi) "availableLinkConnections") 0)))
             )
105   (exec-rules (fetch-phase i) (coupled is-trigger))
      (pre )
      (body
       (msgsnd (morsp-action:make2
                     `(msd ,(msg-> msd))
110                  `(invoke ,(msg-> invoke))
                     `(moi ,(msg-> moi))
                     `(actype "reserveConnection")
                     `(acreply
                       ,(asn:choice-value
115                      'failed
                         (asn:choice-value
                          'logicalProblem
                          (asn:seq (asn:field-value 'problemCause
                                                    (asn:choice-value 'integerValue 22))))))))
120    )
      (post)
      )
```

D.1.2 MF ACCD (Available Connections Change Dissemination)

```
1   (define-behavior "ACCD"
      (scope (ri "LCCapacity") (role "lc") (msg "ivpmsg-set"))
      (when (equal? (msg-> attr) "availableLinkConnections"))
      (exec-rules (fetch-phase ii) (uncoupled after-trigger))
5     (pre (asn=? (Get (Part (ri) "lc") "operationalState")
                  'enabled))
      (body
       ;;; send attributeValueChangeNotification ACCD to all the PNOs
       (msgsnd (moreq-event-report:make2
10               `(msd ,*notif-msd*)
                 `(invoke ,(msif:new-invoke))
                 `(moc ,(Get (msg-> inst) "objClass"))
                 `(moi ,(msg-> inst))
                 '(evtype "attributeValueChange")
15               `(evinfo
                   ,(asn:seq
                     (asn:field-value
                      'attributeValueChangeDefinition
                      (asn:set-of
20                     (asn:seq
                        (asn:field-value
                         'attributeId
                         (asn:choice-value 'globalForm
                                           (oid:str->oid "availableLinkConnections")))
25                      (asn:field-value
                         'newAttributeValue
                         (asn:ber->any
                          (asn:scm->ber
                           (Get (msg-> inst) "availableLinkConnections")
30                         (mor:attribute-syntax "availableLinkConnections")))))))))
                ))
       )
      (post (asn=? (Get (Part (ri) "lc") "operationalState")
                   'enabled)))
35
```

D.2 Le niveau MFS

D.2.1 MFS Reserve DLCs

```
1
    (genrec:define boot-path-reservation path-inst path mgr-list)

    (genrec:define reserve-path-msg inst)
5
    (define-behavior "boot-reserve-path"
      (scope (msg "boot-path-reservation"))
      (when )
      (exec-rules (fetch-phase i) (coupled is-trigger))
10    (pre )
      (body
       (newline)
       (display "-->")
       (newline)
15     (display "Je boote reserve path")
       (newline)

       (Create (mir:root)
               "VC4Path"
20             (msg-> path-inst)
               '(("path" ,(msg-> path))
                 ("status" idle)
                 ("to-do" ,(msg-> path))
                 ("done" ,(list))))
25
       (Establish "PathMgmt" '(("PathMgmt"  ,(ir:genid "PathMgmt")))
                  '(("cnx" ,(msg-> mgr-list))
                    ("path" ,(list (msg-> path-inst)))))
       )
30    (post))




35
    (define-behavior "send-reserve-subpath"
      (scope  (ri "PathMgmt") (role "path") (msg "reserve-path-msg"))
      (when (and (equal? (Get (Part (ri) "path") "status")
                         'idle)
40               (not (equal? (Get (Part (ri) "path") "to-do")
                              '()))
                 ))
      (exec-rules (fetch-phase i) (coupled is-trigger))
      (pre)
45    (body
       (let* ((path-inst (Part (ri) "path"))
              (subpath (car (Get path-inst "to-do")))
              (mgmt-msd (cadr (Get path-inst "to-do"))))
         (newline)
50       (display "-->")
         (newline)
         (display subpath)
         (newline)
         (Set path-inst "status" 'busy)
55       (Set path-inst "to-do" (cddr (Get path-inst "to-do")))
         (Set path-inst "done" (cons subpath (cons mgmt-msd (Get path-inst "done"))))
         (msgsnd (moreq-action:make2
                  '(msd ,mgmt-msd)
                  '(invoke ,(msif:new-invoke))
60                '(moc "mLinkConnection")
```

```
                ‘(moi ,subpath)
                ‘(actype "reserveConnection")
                ‘(acinfo ,*unspecified*)))
        ))
65    (post))


    (define-behavior "receive-reserve-subpath"
70    (scope  (ri "PathMgmt") (role "cnx") (msg "mocnf-action"))
      (when (and (equal? (Get (Part (ri) "path") "status")
                         'busy)
                 (equal? (msg-> actype) "reserveConnection")
                 ))
75    (exec-rules (fetch-phase i) (coupled is-trigger))
      (pre)
      (body
       (let* ((path-inst (Part (ri) "path"))
              (result (asn:choice (msg-> acreply))))
80       (newline)
         (display "-->")
         (newline)
         (display (msg-> acreply))

85
         (cond ((equal? result 'passed)
                (Set path-inst "done" (cons (moi:asn->scm (val:get (msg) '(acreply value 0 value)))
                                            (Get path-inst "done")))
                (cond ((null? (Get path-inst "to-do"))
90                     (Set path-inst "status" 'done)
                       ;;; faire le terminate
                       ;;; (Terminate (ri))
                       )
                      (else (Set path-inst "status" 'idle)
95                          (msgsnd (reserve-path-msg:make (Part (ri) "path")))
                            ))
                )
               ((equal? result 'failed)
                (Set path-inst "status" 'busy2)
100             (Set path-inst "done" (cddr (Get path-inst "done") ))
                (newline)
                (display "-->")
                (newline)
                (display "unreserve")
105             (msgsnd (unreserve-path-msg:make (Part (ri) "path")))
                )
               (else (display "FATAL ERROR")))))
      (post))
```

D.2.2 MFS Unreserve DLCs

```
1   (genrec:define unreserve-path-msg inst)


    (define-behavior "send-unreserve-subpath"
5     (scope  (ri "PathMgmt") (role "path") (msg "unreserve-path-msg"))
      (when)
      (exec-rules (fetch-phase i) (coupled is-trigger))
      (pre)
      (body
10     (let* ((path-inst (Part (ri) "path"))
              (dlcid (car (Get path-inst "done")))
              (subpath (cadr (Get path-inst "done")))
```

```
            (msd-mgr (caddr (Get path-inst "done")))
            )

      (msgsnd (moreq-action:make2
               '(msd ,msd-mgr)
               '(invoke ,(msif:new-invoke))
               '(moc "mLinkConnection")
               '(moi ,subpath)
               '(actype "releaseConnection")
               '(acinfo ,(asn:seq (asn:field-value 'connectionId (moi:scm->asn dlcid))))))

      (Set path-inst "done" (cdddr (Get path-inst "done")))

      (cond ((null? (Get path-inst "done"))
             (Delete path-inst)
             (Terminate (ri)))

            (else (msgsnd (unreserve-path-msg:make (Part (ri) "path")))))
      )
     )
    (post)
  )
```

Annexe E

Modéle d’information statique Xcoop

E.1 Le fichier GDMO

```
-- 4.1 mDelivLinkConnection

mDelivLinkConnection MANAGED OBJECT CLASS
DERIVED FROM "I-ETS GOM VH.3 ": linkConnection;
CHARACTERIZED BY

"Recommendation M.3100:1992" : createDeleteNotificationsPackage,
"I-ETS GOM VH.3" : zEndNWTPListPackage,
mDelivLinkConnectionPackage PACKAGE

BEHAVIOUR mDelivLinkConnectionBehaviour BEHAVIOUR DEFINED AS "An
instance of this class represents a capacity between 2
mNetworkCTPs. This capacity, which can be equivalent to a VC-12, a
VC-2, a VC-3 or a VC-4, is a section of a METRAN deliverable. All the
instances of this class are created during the commissioning process
with an assignmentState attribute set to free. The states of the DLC
are mapped to the combination of two attributes of managed object
class mDeliverableLinkConnection, assignmentState and
operationalState. The assignment state is changed by operations over
the X interface. The operationalState changes are caused by
PNO-internal actions or problems. The operationalState of a DLC is
dictated by the operationalState of the LC that contains the DLC. The
following table shows the states and the corresponding attribute
values.  DLC state assignmentState operationalState unreserved free
enabled reserved reserved enabled activated assigned enabled out of
service free/reserved/assigned disabled ";;

ATTRIBUTES
"Recommendation M.3100:1992": userLabel GET,
"Recommendation X.721:1991" : operationalState GET,
currentOriginPNO GET,
"I-ETS GOM VH.3" : assignmentState GET;;;

REGISTERED AS {MetranDefinedTypesModule.metranObjectClass 1};

-- 4.2 mLink

mLink MANAGED OBJECT CLASS
DERIVED FROM "I-ETS GOM VH.3" : link;
CHARACTERIZED BY

"I-ETS GOM VH.3" : externalLinkPackage,
mLinkPackage PACKAGE
```

```
BEHAVIOUR mLinkBehaviour BEHAVIOUR DEFINED AS "An instance of this
class represents a topological description of capacity sharing the
same routing criteria between two mSubNetworks. A mLinkConnection
inside a mLink is assigned (released) using the reserveConnection and
assignConnection (using the releaseConnection) actions, which as a
result, change the mLinkConnection's assignmentState attribute. The a
and zEndPoints attributes point to the 2 mSubNetworks terminating the
mLink. In case of a cross-border link these 2 mSubNetworks belong to
separate PNOs, but the mLink belongs to only one of them. The
attributeValueChange notification is sent to all PNOs only when
attribute noOfConnections or availableConnections
changes. AvailableConnections changes when capacity is used but also
when underlying resources are out of order, or added / removed from
METRAN. NoOfConnections changes when underlying resources are added /
removed from METRAN.";;

ATTRIBUTES
routingCriteria GET,
"Recommendation M.3100:1992": userLabel GET;

ACTIONS
reserveConnection,
assignConnection,
releaseConnection;;;

REGISTERED AS {MetranDefinedTypesModule.metranObjectClass 2};

-- 4.3 mLinkConnection

mLinkConnection MANAGED OBJECT CLASS
DERIVED FROM "I-ETS GOM VH.3" : linkConnection;
CHARACTERIZED BY

"Recommendation M.3100:1992" : createDeleteNotificationsPackage,
"Recommendation M.3100:1992" : attributeValueChangeNotificationPackage,
"I-ETS GOM VH.3": zEndNWTPListPackage,
mLinkConnectionPackage PACKAGE

BEHAVIOUR mLinkConnectionBehaviour BEHAVIOUR DEFINED AS "An instance
of this class represents a capacity equivalent to a VC-4 between 2
mNetworkCTPs and is used for managing the Setup and release of
mDelivLinkConnections. All the contained mDelivLinkConnections have
the same routing criteria as the mlinkConnection. The
availableConnections attribute gives the number of
mDelivLinkConnections which are still possible to Setup inside the
mLinkConnection. All changes of its value should be reflected by an
attributeValueChange notification to all the PNOs. All the instances
of this class are created during the commissioning process with an
assignmentState attribute set to free. The assignmentState changes to
assigned when all the possible mDelivLinkConnections are set up and
the availableConnections reaches the null value. There is no
attributeValueChange notification for the assignmentStates change. The
operationalState is either enabled or disabled, depending on the
presence/absence of a defect on the underlying resource. There is no
attributeValueChange notification for this change, but a
metranCommunicationAlarm is sent (once) to the currentOriginPNO of all
the contained mDelivLinkConnections. A mDelivLinkConnection can be set
up (can be released) inside a mLinkConnection at a PNO's request using
the reserveConnection and assignConnection actions (using the
releaseConnection action). These actions change the value of
availableConnections attribute and an attributeValueChange
notification is sent to all PNOs.";;

ATTRIBUTES
```

```
    "Recommendation X.721:1991" : operationalState GET,
"I-ETS GOM VH.3" : availableConnections GET,
    "Recommendation M.3100:1992": userLabel GET,
    routingCriteria GET,
    "I-ETS GOM VH.3" : assignmentState GET;

ACTIONS
    reserveConnection,
    assignConnection,
    releaseConnection;

NOTIFICATIONS
    metranCommunicationsAlarm;;;

    REGISTERED AS {MetranDefinedTypesModule.metranObjectClass 3};

-- 4.4 mNetworkCTP

    mNetworkCTP MANAGED OBJECT CLASS
    DERIVED FROM  "I-ETS GOM VH.3" :networkCTPBidirectional;
    CHARACTERIZED BY
"I-ETS GOM VH.3" : connectivityPointerPackage,
    "I-ETS GOM VH.3" : sncPointerPackage,
    "Recommendation M.3100:1992" : ctpInstancePackage,
    mNetworkCTPPackage PACKAGE

BEHAVIOUR mNetworkCTPBehaviour BEHAVIOUR DEFINED AS "An instance of
    this class represents the point where the input-output of a
    mDelivLinkConnection or of a mLinkConnection is bound to the
    input-output of a mSubNetworkConnection, or vice-versa.";;

ATTRIBUTES
    "Recommendation M.3100:1992": userLabel GET;;;

    REGISTERED AS {MetranDefinedTypesModule.metranObjectClass 4};

-- 4.5 mSubNetwork

    mSubNetwork MANAGED OBJECT CLASS
    DERIVED FROM  "I-ETS GOM VH.3" : subNetwork;
    CHARACTERIZED BY
"I-ETS GOM VH.3": subNetworkIdPackage,
    "I-ETS GOM VH.3": signalidPackage,
    "I-ETS GOM VH.3": basicConnectionPerformerPackage,
    "Recommendation M.3100:1992" : attributeValueChangeNotificationPackage,
    mSubNetworkPackage PACKAGE

    BEHAVIOUR mSubNetworkBehaviour BEHAVIOUR DEFINED AS "An instance of
    this class represents a logical collection of mNetworkCTPs and is used
    for managing the Setup and release of
    mSubNetworkConnections. According to G.803 definition of sub-Network,
all termination points can be connected together, in addition a METRAN
    assumption says that there is no blocking situation. Consequently it
    is not possible / required to reserve in advance a
    mSubNetworkConnection and the only actions supported by the
    mSubNetwork are for immediate activation / release of
mSubNetworkConnections. An attributeValueChangeNotification is sent
    only when the value of abilitytoConnect attribute becomes null, ceases
    to be null or crosses a threshold.";;

    ATTRIBUTES
abilityToConnect GET;;;

    REGISTERED AS {MetranDefinedTypesModule.metranObjectClass 5};
```

```
    -- 4.6 mSubNetworkConnection

    mSubNetworkConnection MANAGED OBJECT CLASS
    DERIVED FROM  "I-ETS GOM VH.3" : subNetworkConnection;
    CHARACTERIZED BY
    "I-ETS GOM VH.3" : zEndNWTPListPackage,
    mSubNetworkConnectionPackage PACKAGE

    BEHAVIOUR mSubNetworkConnectionBehaviour BEHAVIOUR DEFINED AS "An
    instance of this class represents a connection across a mSubNetwork
    between 2 networkCTPs. An instance is created as the result of a
    setupSubNetworkConnection action sent to the containing
    mSubNetwork. The assignmentState attribute value is set to assigned
    during the entire lifetime of an instance.
    The operationalState is either enabled or disabled, depending on the
    presence/absence of a defect on the underlying resource. There is no
    attributeValueChange notification for this change, but a
    metranCommunicationAlarm is sent to the currentOriginPNO. An instance
    is deleted as the result of a releaseSubNetworkConnection action sent
    to the containing mSubNetwork.";;

    ATTRIBUTES
    "Recommendation M.3100:1992": userLabel GET,
    "Recommendation X.721:1991" : operationalState GET,
    currentOriginPNO GET,
    "I-ETS GOM VH.3" : assignmentState GET;

    NOTIFICATIONS metranCommunicationsAlarm;;;

    REGISTERED AS {MetranDefinedTypesModule.metranObjectClass 6};

    -- 5 Standard Object Classes

    -- 5.1 alarmRecord This Object Class is defined in [X.721].
    -- 5.2 log This Object Class is defined in [X.721].
    -- 5.3 system This Object Class is defined in [X.721].

    -- 6 Packages There is no METRAN-specific Packages to be implemented in
    -- P408/T8

    -- 7 Attributes

    -- 7.1 abilityToConnect

    abilityToConnect ATTRIBUTE
    WITH ATTRIBUTE SYNTAX MetranDefinedTypesModule.AbilityToConnect;
    MATCHES FOR EQUALITY;
    BEHAVIOUR abilityToConnectBehaviour BEHAVIOUR DEFINED AS "This
    attribute refers to the ability of the mSubNetwork to operate new
    mSubNetworkConnections changes. Possible values are : cannot satisfy
    any new Setup requests (normal parameter is false), cannot satisfy
    requests for more than N set-ups (nbOfPossibleNewSetups gives the
    value of N), or normal capability (normal parameter is true). A change
    of the value of this attribute generates an attributeValueChange
    notification.";;
    REGISTERED AS {MetranDefinedTypesModule.metranAttribute 1};

    -- 7.2 currentOriginPNO

    currentOriginPNO ATTRIBUTE
    WITH ATTRIBUTE SYNTAX MetranDefinedTypesModule.OriginPNO;
    MATCHES FOR EQUALITY;
    BEHAVIOUR currentOriginPNOBehaviour BEHAVIOUR DEFINED AS "This
    attribute identified the PNO who requested the associated
    mDelivLinkConnection or mSubNetworkConnection. A change of the value
```

```
    of this attribute does not generate an attributeValueChange
240 notification.";;
    REGISTERED AS {MetranDefinedTypesModule.metranAttribute 2};

    -- 7.3 routingCriteria

245 routingCriteria ATTRIBUTE
    WITH ATTRIBUTE SYNTAX MetranDefinedTypesModule.RoutingCriteria;
    BEHAVIOUR routingCriteriaBehaviour BEHAVIOUR DEFINED AS "This
    attribute is a place holder for giving the routing criteria shared by
    all mLinkConnections belonging to the referred mLink (or all the
250 mDelivLinkConnections contained in a mLinkConnection). The routing
    criteria could be expressed in terms of performance characteristics,
    cost, length, availability, etc. The exhausted list of criteria and
    the way to model them are for further study as they will depend
    largely from the Operating Agreement to be signed between METRAN
255 parties. For P408/T8 purposes, the syntax is reduced to the following
    : tariff, which is expressed in terms of normal, discount, high or
    secret, qualityOfService which indicates a contractual level of
    performance expected from the contained mLinkConnections (or
    mDelivLinkConnections), it is expressed in terms of low, medium, high
260 or unknown, dedicatedProtection, a boolean which indicates if the
    mLink is protected or not, additionalInfo, which is a free text. It is
    not expected that any of these criteria changes during the mLink
    lifetime.";;
    REGISTERED AS {MetranDefinedTypesModule.metranAttribute 3};
265
    -- 8 Actions

    -- 8.1 assignConnection

270 assignConnection ACTION
    BEHAVIOUR assignConnectionBehaviour BEHAVIOUR DEFINED AS "This action
    activates an already reserved mDelivLinkConnection. As a result of it,
    the assignmentState of the mDLC changes to assigned. No other
    attribute is affected, no attributeValueChange notification is
275 generated. If this action fails, the action reply indicates one of the
    following reasons: DLC unknown to responding PNO: noSuchDLCInstance
    DLC not reserved by requesting PNO: noReservation DLC already assigned
    to other PNO: useConflict DLC already assigned to requesting PNO:
    noEffect DLC disabled: resourceDisabled. ";;
280 MODE  CONFIRMED;
    WITH INFORMATION SYNTAX MetranDefinedTypesModule.AssignConnectionInfo ;
    WITH REPLY SYNTAX MetranDefinedTypesModule.AssignConnectionReply;
    REGISTERED AS {MetranDefinedTypesModule.metranAction 1};

285 -- 8.2 releaseConnection

    releaseConnection ACTION
    BEHAVIOUR releaseConnectionBehaviour BEHAVIOUR DEFINED AS "This action
    discontinues an active mDelivLinkConnection. As a result of it, the
290 assignmentState of the mDLC changes to free. The value of the
    availableConnections attribute changes and an attributeValueChange
    notification is generated.. If this action fails, the action reply
    indicates one of the following reasons : - DLC unknown to responding
    PNO : noSuchDLCInstance - DLC not assigned to requesting PNO :
295 useConflict - DLC not reserved by requesting PNO : useConflict - DLC
    already free : noEffect ";;
    MODE CONFIRMED;
    WITH INFORMATION SYNTAX MetranDefinedTypesModule.ReleaseConnectionInfo;
    WITH REPLY SYNTAX  MetranDefinedTypesModule.ReleaseConnectionReply;
300 REGISTERED AS {MetranDefinedTypesModule.metranAction 2};

    -- 8.3 reserveConnection
```

```
    reserveConnection ACTION
305 BEHAVIOUR reserveConnectionBehaviour BEHAVIOUR DEFINED AS "This action
    reserves a mDelivLinkConnection contained in a given
    mLinkConnection. As a response to this action, the identity of the
    reserved mDelivLinkConnection is received with the corresponding a and
    z networkCTP identities. As a result of it, the assignmentState of the
310 mDLC changes to reserved. The value of the availableConnections
    attribute changes and an attributeValueChange notification is
    generated. If this action fails, the action reply indicates one of the
    following reasons : no free DLC in LC : resourceFull LC disabled :
    resourceDisabled ";;
315
    MODE CONFIRMED;
    WITH REPLY SYNTAX MetranDefinedTypesModule.ReserveConnectionReply;
    REGISTERED AS {MetranDefinedTypesModule.metranAction 3};

320 -- 9 Notifications

    -- 9.1 metranCommunicationsAlarm

    metranCommunicationsAlarm NOTIFICATION
325 BEHAVIOUR metranCommunicationsAlarmBehaviour BEHAVIOUR DEFINED AS
    "This notification type is used to report when an alarm concerning the
    referred object occurs. The list of probable cause and their
    associated perceived severity will be fixed for P408 experiment. The
    alarm can be either a communication, an equipment, or a software
330 alarm. An estimated time of repair can be indicated in additionalText
    parameter.";;
    WITH INFORMATION SYNTAX MetranDefinedTypesModule.AlarmInfo;
    REGISTERED AS {MetranDefinedTypesModule.metranNotification 1};

335 -- 10 Name bindings

    -- 10.1 log-system
    -- smi2NameBinding2
    -- 10.2 alarmRecord-log
340
    alarmRecord-log NAME BINDING
    SUBORDINATE OBJECT CLASS alarmRecord AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS log AND SUBCLASSES;
    WITH ATTRIBUTE "Recommendation X.721 : 1991": logRecordId;
345 REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 7};

    -- 10.3 mSubNetwork-system

    mSubNetwork-system NAME BINDING
350 SUBORDINATE OBJECT CLASS mSubNetwork AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS "Recommendation X.721: 1992":system AND SUBCLASSES;
    WITH ATTRIBUTE "I-ETS GOM VH.3" : subNetworkId;
    REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 5};

355 -- 10.4 mSubNetworkConnection-mSubNetwork

    mSubNetworkConnection-mSubNetwork NAME BINDING
    SUBORDINATE OBJECT CLASS mlink AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS mSubNetwork AND SUBCLASSES;
360 WITH ATTRIBUTE "Recommendation M.3100: 1992": connectionId;
    REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 6};

    -- 10.5 mNetworkCTP-mSubNetwork

365 mNetworkCTP-mSubNetwork NAME BINDING
    SUBORDINATE OBJECT CLASS mNetworkCTP AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS mSubNetwork AND SUBCLASSES;
    WITH ATTRIBUTE "Recommendation M.3100 : 1992" : cTPId;
```

```
    REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 4};

    -- 10.6 mLink-system

    mLink-system NAME BINDING
    SUBORDINATE OBJECT CLASS mLink AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS "Recommendation X.721: 1992":system AND SUBCLASSES;
    WITH ATTRIBUTE "I-ETS GOM VH.3" : linkId;
    REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 2};

    -- 10.7 mLinkConnection-mLink

    mLinkConnection-mLink NAME BINDING
    SUBORDINATE OBJECT CLASS mLinkConnection AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS mLink AND SUBCLASSES;
    WITH ATTRIBUTE "Recommendation M.3100 : 1992" : connectionId;
    REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 3};

    -- 10.8 mDelivLinkConnection-mLinkConnection

    mDelivLinkConnection-mLinkConnection NAME BINDING
    SUBORDINATE OBJECT CLASS mDelivLinkConnection AND SUBCLASSES;
    NAMED BY SUPERIOR OBJECT CLASS mLinkConnection AND SUBCLASSES;
    WITH ATTRIBUTE "Recommendation M.3100 : 1992" : connectionId;
    REGISTERED AS {MetranDefinedTypesModule.metranNameBinding 1};
```

E.2 Le fichier ASN.1

```
1
    -- 11 Supporting ASN.1 Productions

    -- 11.1 METRAN-defined types module
5
    MetranDefinedTypesModule { iso member(2) sweden(752) telia(15) p408(408) t8(1) asn1Module(2) }
    DEFINITIONS IMPLICIT TAGS ::= BEGIN
    IMPORTS
    AlarmInfo FROM Notification-ASN1Module { joint-iso-ccitt ms (9) smi (3) part2 (2) asn1Module (2) 2}
10  ObjectInstance FROM CMIP-1 { joint-iso-ccitt ms (9) cmip (1) version1 (1) protocol (3) }
    Failed, ProblemCause FROM PrETSx {ccitt(0) identified-organization(4) etsi(0) ets(653) informationModel(0) as

    metran OBJECT IDENTIFIER ::= {iso member(2) sweden(752) telia(15) p408(408) t8(1)}
    metranObjectClass OBJECT IDENTIFIER ::= {metran managedObjectClass(3)}
15  metranAttribute OBJECT IDENTIFIER ::= {metran attribute(7)}
    metranNameBinding OBJECT IDENTIFIER ::= {MetranDefinedTypesModule.metran nameBinding(6)}
    metranAction OBJECT IDENTIFIER ::= {metran action(9)}
    metranNotification OBJECT IDENTIFIER ::= {metran notification (10)}

20  -- The following value assignment are specific extension for ProblemCause -- within the P408 context.
    noSuchDLCInstance ProblemCause ::= integerValue : 20
    useConflict ProblemCause ::= integerValue : 21
    resourceFull ProblemCause ::= integerValue : 22
    resourceDisabled ProblemCause ::= integerValue : 23
25  noReservation ProblemCause ::= integerValue : 24
    noEffect ProblemCause ::= integerValue : 25

    -- The following value assignments are mentioned as comments in GOM and
    -- shall be used within the P408 context.
30
    noSuchTPInstance ProblemCause ::= integerValue : 0
    noSuchSNCInstance ProblemCause ::= integerValue : 3
    -- Integer values 26 to 39 are reserved for unforeseen ProblemCauses
    -- that will be encountered during the course of the P408 SDH experiments.
35
```

```
AbilityToConnect ::= CHOICE {
normal  BOOLEAN ,
nbOfPossibleNewSetups  INTEGER
}

AssignConnectionInfo ::= ObjectInstance

AssignConnectionReply ::= CHOICE {
passed [0] NULL, failed [1] Failed }

OriginPNO ::= GraphicString

QualityOfService ::= ENUMERATED {
low (0), medium (1), high (2), unknown(3) }

ReleaseConnectionInfo ::= SEQUENCE {
connectionId  ObjectInstance }

ReleaseConnectionReply ::= CHOICE {
passed [0]  NULL, failed [1] Failed }

ReserveConnectionReply ::= CHOICE {
passed [0] SEQUENCE {
        connectionId  ObjectInstance ,
        aNetworkCTPId  ObjectInstance ,
        zNetworkCTPId  ObjectInstance } ,
failed [1] Failed }

RoutingCriteria ::= SEQUENCE {
tariff Tariff,
qualityOfService QualityOfService,
dedicatedProtection BOOLEAN,
additionalInfo GraphicString }

Tariff ::= ENUMERATED {
discount (0), normal (1), high (2), secret (3) }

END
```

E.3 Le fichier GRM

```
LCCapacity RELATIONSHIP CLASS
  BEHAVIOUR LCCapacityBehavior;
  SUPPORTS ESTABLISH, TERMINATE;

  ROLE lc
    COMPATIBLE-WITH mLinkConnection
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
  REGISTERED AS { metranRole 1 }

  ROLE dlc
    COMPATIBLE-WITH mDelivLinkConnection
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..MAX)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..MAX)
    BIND-SUPPORT
    UNBIND-SUPPORT
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
  REGISTERED AS { metranRole 2 };

REGISTERED AS { metranRelationshipClass 1 } ;
```

```
NCTPintegrity RELATIONSHIP CLASS
  BEHAVIOUR NCTPintegrityBehavior;
  SUPPORTS ESTABLISH, TERMINATE;

  ROLE nctp
    COMPATIBLE-WITH mNetworkCTP
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
  REGISTERED AS { metranRole 3 }

  ROLE connectivity
    COMPATIBLE-WITH mDelivLinkConnection
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..1)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..1)
    BIND-SUPPORT
    UNBIND-SUPPORT
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1..2)
  REGISTERED AS { metranRole 4 }

  ROLE snc
    COMPATIBLE-WITH mSubNetworkConnection
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..1)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(0..1)
    BIND-SUPPORT
    UNBIND-SUPPORT
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1..2)
  REGISTERED AS { metranRole 5 };

REGISTERED AS { metranRelationshipClass 2 } ;

PNOs RELATIONSHIP CLASS
  BEHAVIOUR PNOsBehavior;
  SUPPORTS ESTABLISH, TERMINATE;

  ROLE requesting
    COMPATIBLE-WITH top
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    BIND-SUPPORT
    UNBIND-SUPPORT
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
  REGISTERED AS { metranRole 6 }

  ROLE respondingPNO
    COMPATIBLE-WITH top
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
    BIND-SUPPORT
    UNBIND-SUPPORT
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
  REGISTERED AS { metranRole 7 }

  ROLE 3rdPartyPNOs
    COMPATIBLE-WITH top
    PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1..MAX)
    REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1..MAX)
    BIND-SUPPORT
    UNBIND-SUPPORT
    PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
  REGISTERED AS { metranRole 8 };

REGISTERED AS { metranRelationshipClass 3 } ;
```

```
    PathMgmt RELATIONSHIP CLASS
      BEHAVIOUR PathMgmtBehavior;
      SUPPORTS ESTABLISH, TERMINATE;

      ROLE cnx
        COMPATIBLE-WITH top
        PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1..MAX)
        REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1..MAX)
        BIND-SUPPORT
        UNBIND-SUPPORT
        PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
      REGISTERED AS { metranRole 9 }

      ROLE path
        COMPATIBLE-WITH top
        PERMITTED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
        REQUIRED-ROLE-CARDINALITY-CONSTRAINT INTEGER(1)
        BIND-SUPPORT
        UNBIND-SUPPORT
        PERMITTED-RELATIONSHIP-CARDINALITY-CONSTRAINT INTEGER(1)
      REGISTERED AS { metranRole 10 };

    REGISTERED AS { metranRelationshipClass 4 } ;
```

Index

Bibliographie

[A.C93] A.Clemm. Incorporating Relationships into OSI Management Information. In *2nd IEEE Network Management and Control Workshop*, Tarrytown, NY, USA, sep 1993.

[ACM+96] Ricardo Anido, Ana Cavalli, Toma Macavei, Luiz Paula Lima, Marylèbe Clattin, and Marc Phalippou. Engendrer des tests pour un vrai protocole grâce à des techniques éprouvées de vérification. In *CFIP*, 1996.

[ACS] Service Definition for the Association Control Service Element, Addendum 1: Peer-entity Authentication during Association Establishment, ISO/IEC 8649 1988/DAD1.

[ASN] ASN.1 Free Value Tool. Available at ftp://osi.ncsl.nist.gov/pub/osikit/.

[B81] B. Berthomieux B. Algebraic specification of communication protocols. Technical report, CNRS/LAAS and USC/Information Science Institut Research Report, April 1981.

[Bae93] Brigitte Baer. A Conformance Testing Approach for Managed Objects. In *4th IFIP/IEEE International Workshop on Distributed System Operations and Management*, Long Branch, New Jersey, USA, oct 1993.

[Bap95] S. Bapat. *Object-Oriented Networks: Models for Architecture, Operations and Management*. Prentice Hall, 1995.

[BBG96] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Coopn/2 : A specification language for distributed systems eng ineering. Technical Report 167, Software Engineering Laboratory, Swiss Federal Institute of Te chnology, Lausanne, Switzerland, 1996. available at ftp://lglftp.epfl.ch/pub/Paper s/biber-TR96-167.ps.

[Bei95] B. Beizer. *Black-Box testing*. John Wiley & sons, Inc, 1995.

[BG94] Paulo Borba and Joseph A. Goguen. An Operational Semantics for FOOPS. Technical Report PRG-TR-16-94, Programming Research Group, Nov 1994. 61pp, Available at ftp://ftp.comlab.ox.ac.uk/pub/Documents/techreports/TR-16-94.ps.Z.

[BM94] A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, pages 885–899, june 1994.

[Bri87a] E. Brinskma. Lotos specifications, their implementations abd their tests. In *PSTV VI*, 1987.

[Bri87b] E. Brinskma. On the existence of canonical testers. In *Memorandum inf-87-5, University of Twente*, 1987.

[Bri88] E. Brinskma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*. North-Holland, 1988.

[BTV91] E. Brinskma, J. Tretmans, and L. Verhaard. A framework for test case selection. In B. Jonsson, J. Parrow, and P. Pehrson, editors, *Protocol Specification, Testing and Verification XI*. North-Holland, 1991.

[Ca89] S. Chakravarthy and al. A research project in active, time-constrained database management (final report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Massachusetts, August 1989.

[Cav96] A. Cavali. Testing methods for sdl systems. In *Computer Networks and ISDN Systems*, 1996.

[CDD+89] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: an Object-Oriented Extension to Z. In S.T. Vuong, editor, *Proc. 2nd Int. Conf. on Formal Description Techniques for Communication Protocols and Distributed Systems*, pages 401–420. North-Holland, 1989.

[CGPT95] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking a test generation tool with verification techniques. In *in Proc. of IWPTS*, Evry- France, 1995.

[Che76] P. P-S. Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[Che96] Cristophe Chevrier. *Test de conformité de protocoles de communication modèle de fautes et génération automatique de séquences de test*. PhD thesis, Université de Bordeaux I, 1996.

[CMIa] Management Information Protocol Specification - Common Management Information Protocol, ISO/IEC 9596-1, ITU X.711.

[CMIb] Management Information Service Definition - Common Management Information Service Definition, ISO/IEC 9595, ITU X.710.

[CPRZ89] L. Clarke, A. Podgurski, D. Ridchardson, and S. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, pages 1318–1331, november 1989.

[CR91] W. Clinger and J. Rees. *Revised*^{4} Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), 1991. Available at http://www.cs.indiana.edu/scheme-repository/doc/standards/r4rs.ps.gz.

[CW93] Cusack and Wezeman. Deriving Tests for objects specified in Z. In Bowen J.P. and Nicholls J.E., editors, *Z User Workshop*. Springer-Verlag, 1993.

[Dar96] Muriel Daran. *Modélisation des Comportements Erronés du Logiciel et Application à la Validation des Tests par Injection de Fautes*. PhD thesis, LAAS- INP Toulouse, 1996.

[DLS78] R.A. DeMillo, R.J Lipton, and F.G. Sayward. Hints on test data selection: Help for practicing the programmer. *Computer*, 11(4):34–41, 1978.

[EMS97] Rolf Eberhardt, Sandro Mazziotta, and Dominique Sidou. Design and testing of information models in a virtual environment. In *The Fifth IFIP/IEEE International Symposium on Integrated Network Management "Integrated Management in a Virtual World"*, San Diego, CA, USA, may 1997. available at http://www.eurecom.fr/~tims/papers/im97.ps.gz.

[Eri] Erisoft. Imtools.

[EWO95] Methodology for Testing Conformance to Managed Objects, EWOS PT N 028-Draft 9 bis , 1995.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of Conference on Computer-Aided Verification (CAV'96)*, New Brunswick, New Jersey, USA, 1996.

[FJJV96a] J.C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming, Special Issue on Industrial Relevant Applications of Formal Analysis Techniques*, 1996.

[FJJV96b] J.C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of conformance test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of Conference on Computer-Aided Verification (CAV'96)*, New Brunswick, New Jersey, USA, 1996.

[FRM] Basic Reference Model - Part 4: Management Framework, ISO/IEC 7498-4 : 1989(E), 1989, ITU X.700.

[Fuc92] Norbert E. Fuchs. Specifications are (preferably) executable. Technical Report 92, University of Zurich (CS Dept.), 1992. Available at ftp://ftp.ifi.unizh.ch/pub/techreports/.

[G8596] Management of the Transport Network – Application of the ODP Framework, ITU-T G851-01, 1996.

[GDM] Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects, ISO/IEC 10165-4, ITU X.722.

[Gen95] Guy Genilloud. *Towards a distributed architecture for systems management*. PhD thesis, Ecole Politechnique Fédérale de Lausanne, December 1995.

[GG75] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pages 156–173, june 1975.

[GK96] A. Guerrouat and H. Konig. Automation od test case derivation in respect to test purposes. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Testing of Communicating Systems, Proc. of IFIP International Worshop on the Testing of Communicating Systems*. Chapman & Hall, 1996.

[God95] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems* – An Approach to the State Exploision Problem. PhD thesis, Université de Liège, Faculté des Sciences Appliquées, 1995. Available at http://www.montefiore.ulg.ac.be/services/verif/papers/thesis.ps.Z.

[God97] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Principles of Programming Languages*. ACM, 1997.

[GRM] ISO/IEC JTC 1/SC 21, ITU X.725 – Information Technology – Open System Interconnection – Data Management and Open Distributed Processing – Structure of Management Information – Part 7 : General Relationship Model.

[GW91] Godefroid and Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *CAV'91 Springer Verlag,LNCS 575*, 1991.

[HC90] I.J. Hayes and Jones C.B. Specifications are not (necessarily) executable. Technical Report 90-148, University of Manchester (CS Dept.), 1990. Available at ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-89-12-1.ps.Z.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HW92] Eric N. Hanson and Jennifer Widom. An Overview of Production Rules in Database Systems. Technical report, University of Florida (CIS), 1992. Available at ftp://ftp.cis.ufl.edu/cis/tech-reports/tr92/tr92-031.ps.

[ISO87a] ISO. *Information Processing – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*. ISO International Standard 8824, 1987.

[ISO87b] ISO. *Information Processing – Open Systems Interconnection – Specification of Basic Encoding Rules for ASN.1*. ISO International Standard 8825, 1987.

[iso92a] iso. ESTELLE : A Formal Description Technique based on Extended State Transition Model , 1992.

[ISO92b] ISO/IEC IS 9646. Information Technology - Open Systems Interconnection - Conformance Testing, Methodologie and Framework, ISO/IEC IS 9646, 1992.

[IT92a] ITU-T. ITU-T Recommendation X.290-92: OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications - General Concepts., 1992.

[IT92b] ITU-T. ITU-T Recommendation X.291-92: OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications - Abstract Test Suite Specification., 1992.

[IT92c] ITU-T. ITU-T Recommendation X.292-92: OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications - The Tree and Tabular Combined Notation., 1992.

[IT92d] ITU-T. ITU-T Recommendation X.293-92: OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications - Test Realization., 1992.

[IT92e] ITU-T. ITU-T Recommendation X.294-92: OSI Conformance Testing Methodology and Framework for Protocol Recommendations for CCITT Applications - Requirements on Test Laboratories and Clients for the Conformance Assessment Process., 1992.

[IT92f] ITU-T. ITU-T Recommendation Z.100-92: Specification and Description Language, 1992.

[IT92g] ITU-T. ITU-T Recommendation Z.120-92: Message Sequence Charts , 1992.

[Jav] Java RMI - Remote Method Invocation. available at http://www.javasoft.com/.

[JKS91] Hannu-Matti Jarvinen and Reino Kurki-Suonio. DisCo Specification Language: Marriage of Action and Objects. In *Proc. of 11th International Conference on Distributed Computing Systems*, Arlington, Texas, may 1991. IEEE Computer Society Press. Available at http://www.cs.tut.fi/laitos/DisCo/DisCo-english.fm.html.

[JM97] T. Jéron and P. Morel. Abstraction, τ-réduction et déterminisation à la volée : application à la génération de test. In *Proc. of CFIP*, 1997.

[Kab95] Paul Kaboré. *Une approche de test de conformité des systèmes d'administration de réseaux.* . PhD thesis, Université Henri Poincaré, Nancy I, Centre de Recherche en Informatique de Nancy (CRIN), 1995.

[Kil93] H. Kilov. Information Modeling and Object-Z: Specifying Generic reusable Associations. In *Proc. of NGIST'93*, 1993. Haifa, Israel.

[KMS96] Haim Kilov, Helen Mogill, and Ian Simmonds. *Object Oriented Behavior Specifications*, chapter Invariants in the Trenches, pages 77–100. Kluwer Academic Publishers, 1996.

[KR94] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach*. Object-Oriented Series. Prentice Hall, 1994.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, Palo Alto, California, December 1991.

[Led91] Guy Leduc. Relations d'implémentation et transformations autorisées d'une spécification lotos. *Réseaux et Informatique Répartie*, 1991.

[LL95] R. Lai and W. Leung. Industrial and academic protocol testing. In *Computer Networks and ISDN Systems*, 1995.

[LOT] LOTOS: A Formal Description Technique based on the Temporal Ordering of Observable Behaviour, ISO IEC 8807 (1987).

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, August 1996.

[Lyo96] J.L. Lyons. ARIANE 5, flight 501, Failure Report by the Inquiry Borad, 1996.

[Maz86] Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1987, ed. Grzegorz Rozenberg, LNCS 266; Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, LNCS 254-255, 1987, LNCS 188 (1984), LNCS 340 (1988), LNCS 483 (1991)*. Lecture Notes in Computer Science, 1986.

[Maz95] Sandro Mazziotta. Formalisation et Simulation de comportement pour le test d'objets gérés. In *Premier Colloque Francophone sur La gestion de Réseau et de Services*, Paris, FRANCE, sep 1995. Available at http://www.eurecom.fr/~tims/papers/gres95-paper.ps.gz.

[Mil80] Robin Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MS96a] Sandro Mazziotta and Dominique Sidou. A *Scheme*-based Toolkit for the Fast Prototyping of TMN-systems. submitted to Seventh International Workshop on Distributed Systems : Operations & Management, 1996.

[MS96b] Sandro Mazziotta and Dominique Sidou. Guidelines for the Specification of Managed Object Behaviors with TIMS. In *ECOOP96 Workshop : Use of Object-Oriented technology for Network Design and Management*, Linz, Austria, July 1996.

[Nah95] R. Nahm. *Conformance Testing based on Formal Description Techniques and Message Sequence Charts*. Ph.D. thesis, Univesity of Bern, March 1995.

[NGH93] R. Nahm, J. Grabowski, and D. Hogrefe. Test case generation for temporal properties. Technical Report IAM-93-013, Institut fur Informatik, Universitat of Bern, 1993.

[NMF92] Ensembles: Concepts and Format, 1992. Network Mangement Forum.

[NS95] E. Najm and J.-B. Stephani. A formal Semantics of the ODP Computational Model. *Computer Networks and ISDN Systems*, 27(8):1305–1329, 1995.

[ODM95] ISO/IEC JTC 1/SC 21 – Information Retrieval – Transfer and MAnagement for OSI Management, 1995.

[OMG96] Common Object Request Broker Architecture, 1996. Available at http://www.omg.org.

[P4096] EURESCOM Project P408. Pan-european tmn - experiemnts and field trial support, deliverable 5, specifications of the xcoop interface for sdh network management, 1996.

[Pha94] Marc Phalippou. *Relation d'implantation et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Université de Bordeaux, 1994.

[PMB+95] G. Pavlou, K. McCarthy, S. Bhatti, G. Knight, and Simon Walton. The OSIMIS Platform : Making OSI Management Simple. In Chapman & Hall, editor, *Integrated Network Management IV*, pages 480–493, 1995.

[RC96] O. Rafiq and A. Cavali. Special issue on protocol testing. In *Computer Networks and ISDN Systems*, December 1996.

[Rei85] W. Reisig. *Petri Nets*. Springer Verlag, 1985.

[RM-a] Basic Reference Model of ODP – Part 1: Overview and Guide to Use of the Reference Model, ISO 10746-1, ITU X.901.

[RM-b] Basic Reference Model of ODP – Part 2: Foundations, ISO 10746-2, ITU X.902.

[RM-c] Basic Reference Model of ODP – Part 3: Architecture, ISO 10746-3, ITU X.903.

[RM-d] Basic Reference Model of ODP – Part 4: Architectural Semantics, ISO 10746-4, ITU X.904.

[ROR91] M. T. Rose, J.P. Onions, and C.J. Robins. The ISO development environment: User's manual, version 8.0. Technical report, PSI, July 1991. Available at ftp://ftp.psi.net.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, pages 367–375, April 1985.

[Sid97] Dominique Sidou. *Validation of Functional Behavior Specifications of Distributed Object Frameworks* . PhD thesis, Ecole Politechnique Fédérale de Lausanne, November 1997.

[SME95] Dominique Sidou, Sandro Mazziotta, and Rolf Eberhardt. TIMS : a TMN-based Information Model Simulator, Principles and Application to a Simple Case Study. In *Sixth International Workshop on Distributed Systems : Operations & Management*, Ottawa - Canada, 1995. IFIP / IEEE. Available at http://www.eurecom.fr/~tims/papers/dsom95-paper.ps.gz.

[Spi89] J.M. Spivey. *The Z Notation*. Prentice Hall, 1989.

[TIN94a] TINA Computational Modelling Concepts, 1994. Available at http://www.tinac.com.

[TIN94b] TINA Information Modelling Concepts, 1994. Available at http://www.tinac.com.

[TMN92a] Principles for a Telecommunications Management Network, ITU Recommendation M.3010, 1992.

[TMN92b] Generic Network Information Model, ITU Recommendation M.3100, 1992.

[TR96] A. Touag and A. Rouger. Generation de tests a partir de specifications lds basée sur une construction partielle de l'arbre d'accessibilité. In *CFIP*, 1996.

[Tre92] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, Universiteit Twente, 1992.

[Val90] Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1987, ed. Grzegorz Rozenberg, LNCS 266*. Springer Verlag, 1990.

[VAM96] Vernadat, P. Azema, and F. Michel. Covering step graph. In *Application and Theory of Petri Nets*, 1996.

[VvB82] Verhijen and van Bekkum. Niam: An information analysis method. In *IFIP WG 8.2 Working Conference on Comparative Review of Information Systems Design Methodologies*, New York, 1982. IFIP, North-Holland.

[Wez90] C.D. Wezeman. The co–op method for compositional derivation of conformance tester. In *PSTV IV*, 1990.

[WG93] Wolper and Godefroid. Partial-order methods for temporal verification. In *Concur'93 LNCS 575*, 1993.