Université de Montréal

Un cadre formel pour le développement orienté aspect : modélisation et vérification des interactions dues aux aspects

par
Farida Mostefaoui

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Août, 2008

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Un cadre formel pour le développement orienté aspect : modélisation et vérification des interactions dues aux aspects

présentée par:

Farida Mostefaoui

a été évaluée par un jury composé des personnes suivantes:

Guy Lapalme
président-rapporteur

Julie Vachon
directrice de recherche

Hanifa Boucheneb
membre du jury

Jörg Kienzle
examinateur externe

Alain Vincent
représentant du doyen de la FES

RÉSUMÉ

Un des principaux objectifs du génie logiciel est d'assurer la qualité des logiciels produits en proposant des solutions rigoureuses et performantes aux problèmes de développement communément rencontrés. La qualité d'un logiciel se définit typiquement par sa lisibilité, sa facilité de compréhension, sa réutilisabilité, son évolutivité, sa fiabilité et sa maintenabilité. Afin d'atteindre ce but, le développement logiciel peut heureusement s'appuyer sur des techniques, des méthodes et des outils éprouvés. Certaines solutions encore à l'étude promettent bien sûr de venir s'ajouter à ce lot. La programmation par aspects compte justement parmi les approches émergentes auxquelles les chercheurs en génie logiciel ont porté une attention particulière ces dernières années.

Par rapport aux approches de programmation traditionnelles, le paradigme par aspects préconise une programmation modulaire qui permet une meilleure séparation des préoccupations d'un système non seulement principales (services et fonctions) mais également secondaires et transverses. Il propose, en effet, de décomposer les programmes non seulement en unités modulaires propres aux préoccupations principales, mais aussi en unités modulaires spécifiques aux préoccupations transverses, appelées communément *aspect*, très souvent gérées de façon inadéquate dans la programmation traditionnelle. Cette modularité permet ainsi d'offrir de nouvelles perspectives quant à lisibilité, la compréhension, la traçabilité, l'évolution et la réutilisation des systèmes logiciels produits.

Dans le cadre de ce travail de recherche, notre objectif principal est de doter le paradigme aspect d'une approche de développement rigoureuse. Suivant les objectifs fondamentaux du paradigme par aspects, non seulement la programmation mais également la modélisation et la vérification devraient permettre une meilleure séparation des préoccupations en vue d'assurer la rigueur, la fiabilité et la correction des logiciels produits. Une des difficultés inhérentes au développement par

aspects réside dans la façon de composer ces aspects. En plus de miser sur une technique de tissage efficace, on souhaite que les aspects n'interfèrent pas entre eux et ne viennent pas compromettre la correction du système de base suite à des interactions sournoises et non prévues. Étant donné la nature des problèmes de composition redoutés, nous croyons qu'une approche de développement par aspects aurait grand intérêt à s'appuyer sur l'emploi de méthodes formelles. Pour confirmer ce point de vue, la présente thèse vise à montrer qu'une approche de développement par aspects intégrant une discipline de modélisation et de vérification formelle permet de détecter de façon précoce (i.e. avant même le tissage par le compilateur) les interférences indésirables qu'une analyse statique du code source lui seul n'aurait su révéler. Plus concrètement, le travail que nous présentons ici propose une notation de modélisation orientée aspect étayée d'une sémantique formelle et d'un outil de vérification. Cet outil permet précisément de détecter les interférences entre les aspects et leurs effets indésirables sur les propriétés essentielles du système de base.

Mots clés : Paradigme aspect, Modélisation, UML, Vérification formelle, Réseaux de Petri colorés, COOPN/ 2, Alloy.

ABSTRACT

One of the most challenging tasks of software engineering is to produce high quality software products. The term quality refers to attributes that the software system should possess such as readability, reusability, evolution, maintainability and reliability. To achieve these goals, the software engineering discipline proposes techniques, methods, and tools. One of the most recent techniques that have emerged in this context is the one based on the aspect paradigm.

The aspect oriented programming approach contrasts with traditional programming approaches. It advocates a better modularity based on the good separation of all the system functionalities: main functionalities (functions and services) are separated from secondary and crosscutting functionalities. Indeed, this paradigm proposes program decomposition not only into main modules describing main functionalities but also into modules specific to crosscutting functionalities commonly known as aspects. These functionalities are most often badly managed in the traditional programming approaches. Hence, the modularity provided by the aspect paradigm allows a better readability, understandability, traceability, evolution and reusability of software products.

In this research work, the main objective is to endow the aspect paradigm with a concise development approach. In such an approach, the modeling and the verification processes should allow a better separation of concerns to guarantee the reliability and the correction of the produced software. Amongst the challenges encountered in a typical aspect-oriented development approach is the coherent composition of the aspects. Indeed, aspects put together in a system need not to interact with each other in an unpredictable way, thus harming the correctness of the underlying system. Aiming at avoiding inconsistency due to careless composition of aspects, we advocate using formal methods. As a matter of fact, this thesis demonstrates that an aspect-oriented development approach, where stringent

modeling and formal verification are integrated, can expose unwanted interactions between aspects at early stages, even more efficiently than source code static analysis. To achieve the said demonstration, this work introduces an aspect-oriented modeling notation, together with a formal semantics, and a tool that has the ability to uncover the interactions between the aspects and their adverse affects on the key properties of the underlying system.

Keywords: Aspect paradigm, Modeling, UML, Formal verification, Colored Petri nets, COOPN/2, Alloy.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES APPENDICES

REMERCIEMENTS

En premier lieu, je tiens à remercier chaleureusement ma directrice de thèse Julie Vachon pour l'encadrement de mon travail de recherche ainsi que pour son apport tant au niveau des connaissances qu'au niveau humain. Je la remercie également pour ses encouragements et son soutien constant tout au long de cette thèse.

Je remercie humblement Guy Lapalme pour m'avoir fait honneur de présider ce jury de thèse et Hanifa Boucheneb pour avoir accepté d'y participer.

Je remercie également Jörg Kienzle pour avoir accepté la tâche d'examinateur externe et ainsi d'avoir bien voulu lire et commenter ce manuscrit.

J'associe à ces remerciements Alain Vincent pour avoir accepté de participer à mon jury à titre de représentant du doyen.

Enfin, accomplir une thèse en étant mère de trois jeunes enfants est l'une des tâches des plus exigeantes. Je tiens à remercier tout particulièrement, mes enfants Ferhat, Azzeddine et Sara pour leur amour, patience et soutien. Ils ont dû me supporter durant toutes les étapes de cette thèse, toutes aussi difficiles les unes que les autres. Je leur dédie ce modeste travail tout en espérant que son aboutissement leur serve d'exemple dans leur vie future : ne jamais abandonner et toujours aller de l'avant.

CHAPITRE 1

INTRODUCTION

Un des objectifs majeurs recherché par la discipline du génie logiciel est de favoriser et d'appuyer une ingénierie des systèmes d'information de *bonne* qualité c'est-à-dire le développement de logiciels lisibles, compréhensibles, réutilisables, évolutifs, maintenables et fiables. Pour atteindre cet objectif, de nombreux modèles (objets, composants, etc.) et méthodes de développement (processus unifié, MDA, etc.) ont été adoptés ces dernières années. Cependant, si ces modèles et méthodes ont des avantages certains, il n'en demeure pas moins que l'augmentation de la masse et de la diversité des informations à traiter, la complexité croissante des systèmes informatiques et leur incessante évolution rendent le processus de développement plus difficile, plus coûteux et moins fiable, affectant ainsi la qualité des logiciels produits. Ainsi, l'ingénierie des systèmes d'information se tourne actuellement, de plus en plus, vers de nouvelles techniques et méthodes de développement facilitant l'évolution et favorisant la réutilisation et la maintenance des systèmes. C'est dans ce contexte qu'on à vu émerger les approches de développement basées sur le *paradigme aspect.*

1.1 Contexte : le paradigme aspect

Le paradigme aspect [KLM+97] est une nouvelle technique de programmation modulaire offrant une meilleure compréhension, une plus grande réutilisabilité et une maintenance aisée des systèmes logiciels. Il permet de prendre en charge les *préoccupations transverses*, une réalité souvent gérée de façon inadéquate dans les modèles de programmation classique (procéduraux, orientés objet, etc.). Une préoccupation est dite transverse dans un système logiciel si son implémentation impacte plusieurs modules. Par exemple, dans le cas de la programmation orientée objet,

ces préoccupations transverses peuvent représenter les fonctionnalités inter-classes (c'est-à-dire des connexions complexes entre divers objets) ou encore les besoins non fonctionnels (comme la sécurité, la persistance des données, ...), etc. Les approches de programmation classique, dont l'approche objet, ne fournissent pas de support direct permettant une représentation explicite et localisée de telles préoccupations transverses. En effet, le code de ces préoccupations se retrouve généralement décliné et éclaté dans la description des différents composants constituant le système, engendrant ainsi le problème de *dispersion et d'enchevêtrement de code* qui rend difficile la compréhension, la réutilisabilité, la maintenabilité des artefacts logiciels. Pour pallier ces insuffisances, la programmation par aspects permet de séparer *proprement* toutes les préoccupations (ou fonctionnalités) d'un système qu'elles soient *de base* ou *transverses*. Elle propose, en effet, de décomposer les programmes non seulement en unités modulaires propres aux préoccupations de base, mais aussi en unités modulaires spécifiques aux préoccupations transverses. Les premières sont représentées par des composants de base constituant le *système de base* et les secondes sont décrites par des composants secondaires appelés *aspects*. Pour obtenir le système final, un mécanisme de composition appelé *tissage* permet ensuite de composer les préoccupations transverses dans le système de base à des points bien définis appelés *points de jointure*. En proposant ainsi de retarder, jusqu'à la phase de tissage, l'intégration dans le système de base des préoccupations transverses, l'approche aspect relâche donc le couplage entre les différentes préoccupations de base et transverses, offrant ainsi de nouvelles perspectives quant à la lisibilité, la compréhension, la traçabilité, l'évolution et la réutilisation de ces préoccupations. En outre, la montée en puissance, à pas soutenus, des langages de programmation orientés aspect, tel que le très populaire AspectJ [KHH+01], a grandement contribué à l'émergence de ce paradigme.

1.2 Problèmes et enjeux

La communauté du paradigme aspect est en pleine croissance. Ce nouveau paradigme ne cesse d'attirer l'attention des chercheurs et des développeurs. Toutefois, en dépit de sa manifeste popularité, ce paradigme est encore loin de devenir le paradigme adopté par les développeurs. En effet, s'il est vrai qu'un large éventail d'outils et de langages de programmation (AspectJ [Asp02,Lad03], Hyper/j [OT00], JAC [PSD+04], AspeCtC [GJ08], AspectC++ [SGSP02], Aspectual Caml [TMY], etc.) est offert au niveau de l'implantation, il n'existe de nos jours, aucun standard pour une méthodologie de développement ni pour un langage de modélisation afin de soutenir les développeurs de la communauté aspect dans leur tâche. Un premier enjeu pour cette communauté est donc d'offrir aux développeurs et utilisateurs de ce paradigme une méthodologie et un langage de modélisation standardisé. Cette méthodologie devra permettre la spécification, la visualisation, la construction et la documentation des artefacts du système, tel que c'est le cas pour l'approche orientée objet avec la méthode UP (Unified Process) [Sco02] qui utilise le langage UML [Obj05]. Cependant, doter le paradigme aspect d'un langage de modélisation unifié est un défi majeur. En effet, les langages orientés aspect existants sont basés sur des approches différentes, telles que l'approche asymétrique [KHH+01], la séparation multidimensionnelle des préoccupations [TOHS99] (appelée communément, par opposition à la première, approche symétrique[1]) et les filtres de composition [AT98]. Chacune de ces approches propose sa propre solution et offre de nouveaux concepts et mécanismes, qui permettent une meilleure modularisation des préoccupations transverses par rapport aux approches classiques de développement. Ces concepts et mécanismes permettent, en fait, une décomposition et une recomposition efficaces des programmes, tout en gardant bien séparées et encapsulées les différentes préoccupations transverses ou de base d'un système. Ainsi,

[1] Le terme symétrique (par opposition à asymétrique) est relatif au type de composition utilisé.

chaque langage orienté aspect a sa façon particulière de prendre en charge et d'exposer les concepts de la programmation par aspects pour le développeur. Déjà, la plupart des travaux existant sur la modélisation par aspects, sont surtout dédiés à l'approche de programmation, dite asymétrique, incarnée par le langage le plus populaire AspectJ [Asp02], car celui-ci est doté d'une sémantique opérationnelle très intuitive et naturelle. Cependant, malgré la multitude des propositions concernant la modélisation par aspects (l'atelier AOM (*Aspect Oriented Modelling*) qui s'organise dans le cadre des conférences AOSD et MODELS, est spécialement dédié à ce créneau de recherche), un standard de modélisation n'a pas encore été proposé et adopté par la communauté aspect.

Par ailleurs, un autre enjeu du paradigme aspect, réside dans le processus de composition du système final, en soulevant des problèmes tels que : comment recomposer les aspects pour former un système global cohérent ? Comment savoir si un aspect réalise bien les fonctionnalités pour lesquelles il est conçu ? Comment savoir si les fonctionnalités d'un aspect n'entrent pas en conflit avec les fonctionnalités du système de base, ou encore avec les fonctionnalités des autres aspects, une fois composées ensemble ? Comment savoir si le système final reste cohérent après la composition des aspects ? Il convient de noter, que plusieurs ateliers (*workshops*) sont régulièrement organisés sur les thèmes entourant ces questions. Mentionnons notamment, par exemple, l'atelier FOAL (*Fondations Of Aspect Oriented langages*) organisé conjointement avec la conférence AOSD ou encore l'atelier *Aspects, Dependencies, and Interactions* organisé dans le cadre de la conférence *ECOOP*.

La composition des aspects est en fait un problème majeur, car une *mauvaise* composition peut conduire à un système erroné. En effet, les aspects peuvent violer la cohérence du système initial après leur composition, eu égard aux interactions et conflits qui peuvent exister entre les aspects eux-mêmes et/ou entre les aspects et le système de base. D'une part, les aspects à composer à un même point de jointure peuvent être conflictuels, d'autre part, l'insertion d'un aspect peut violer

l'état cohérent du programme à ce point de jointure. Par ailleurs, ce problème est d'autant plus susceptible de survenir dans le cas où plusieurs aspects sont concurrents à un même point de jointure (point de composition). Effectivement, si aucun ordre d'exécution des aspects n'est défini, ceux-ci vont être exécutés de façon non déterministe. Or, une simple composition par concaténation des différents aspects n'est certainement pas suffisante, vu les relations de dépendance qui peuvent exister entre eux. De nos jours, les développeurs aspect ne disposent que de notions rudimentaires pour la composition des aspects. Ils ont la responsabilité d'identifier les interactions entre les aspects conflictuels, de résoudre manuellement les conflits et d'implémenter le code de la résolution des conflits sans aucun support dédié.

1.3 Objectifs de la thèse

Sur la base des constats évoqués ci-dessus, nous nous intéressons plus particulièrement dans le cadre de cette thèse à la mise au point d'une approche pour la modélisation et la vérification des systèmes par aspects, qui assure évidemment une *bonne* composition des aspects. La modélisation et la vérification sont deux activités importantes dans tout processus de développement rigoureux (figure 1.1). Particulièrement, dans le cas du développement par aspects, la modélisation et la vérification devraient permettre une meilleure séparation des préoccupations assurant la rigueur, la fiabilité et la correction des logiciels produits, permettant ainsi d'atteindre les objectifs visés par ce paradigme.

Ainsi, notre premier objectif porte sur la définition d'un langage de modélisation pour le paradigme aspect, afin de prendre en charge les aspects durant tout le cycle de développement. Entre autres, ce langage doit pouvoir assurer la traçabilité des aspects entre toutes les phases de développement ; il doit aussi pouvoir spécifier clairement la composition des aspects. Pour ce faire, nous proposons d'étendre UML et de lui définir un nouveau profil (que nous appelons Aspect-UML [VM04a, MV06a]),

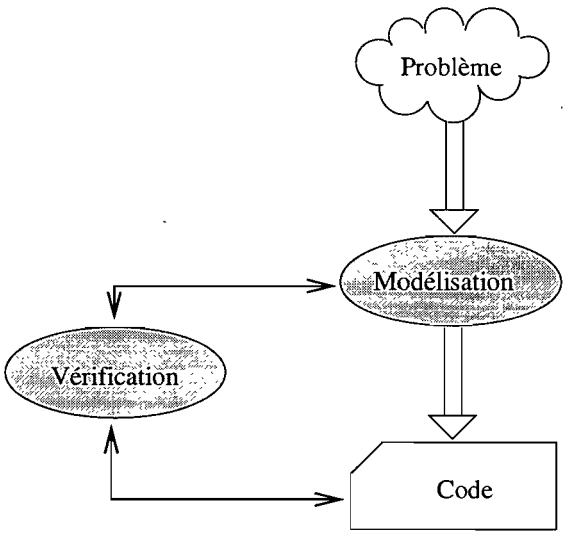


FIG. 1.1 – Processus de développement.

pour la modélisation orientée aspect. Ce profil définit des stéréotypes pour décrire les principaux concepts des langages par aspects et introduit un formalisme de contraintes pour décrire la sémantique relative à la composition des aspects (cf. chapire 3). Nous pensons, en effet, que le langage UML adopté par la communauté objet est un choix approprié pour le paradigme aspect. Cette idée est très naturelle, du fait que la technique de programmation orientée aspect est une extension, voire, une amélioration, de la programmation orientée objet. Aussi, du fait de sa normalisation, l'utilisation d'UML est indispensable comme outil de structuration et de communication pour les développeurs, d'autant plus qu'il existe des outils riches et conviviaux le mettant en oeuvre.

Tel que schématisée par le figure 1.2, l'approche de modélisation, que nous proposons, vise à prendre en charge le concept de séparation des préoccupations de la phase de la spécification des besoins jusqu'à la conception. Notre approche propose un modèle dans chacune des trois vues : cas d'utilisation, structurelle et comportementale.

Par ailleurs, pour une meilleure prise en charge des aspects durant le cycle

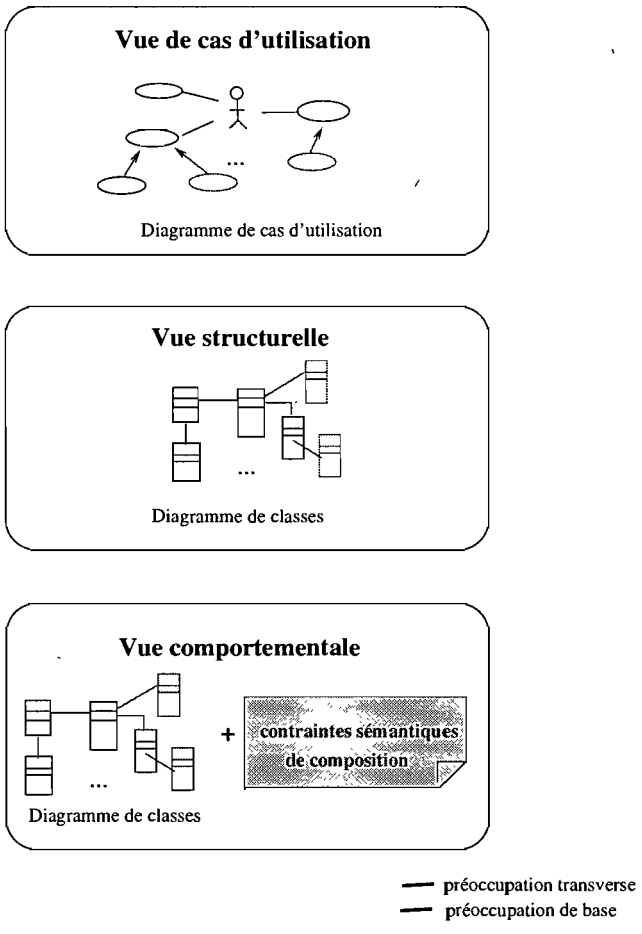


FIG. 1.2 – Méthodologie de modélisation avec notre profil Aspect-UML.

de développement, il convient de s'intéresser non seulement à leur représentation et à leur description au niveau des modèles, mais également à la vérification de leur composition, et ce afin de former un système final cohérent. L'approche de modélisation par aspects que nous proposons, à travers le profil Aspect-UML, est rigoureuse et offre des informations sémantiques précises quant à la composition des aspects. En particulier, ces informations sémantiques peuvent tout à fait servir à la vérification formelle de la composition des aspects. Pour ce faire, nous devons expliciter formellement la composition et le tissage des aspects.

Notre second objectif, dans le cadre de cette thèse, est donc de proposer une

formalisation pour notre profil Aspect-UML, où le tissage des aspects est explicitement décrit. Notre but n'est évidement pas d'introduire un formalisme de plus dans la littérature abondante des modèles formels, mais plutôt d'utiliser un formalisme déjà existant.

Dans un premier temps, nous proposons une formalisation de Aspect-UML en termes de réseaux de Petri [Pet81]. Le choix des réseaux de Petri est motivé par les raisons suivantes : (1) ils ont une sémantique formelle et précise bien adaptée à la description des systèmes séquentiels et concurrents ; (2) ils sont particulièrement appropriés pour exprimer la composition des flots de contrôle (tel que la séquence, le choix non déterministe, ...) ; (3) ils offrent une représentation graphique (qui permet d'enrichir et de compléter la représentation graphique déjà fournie par Aspect-UML).

Notons, toutefois, que le recours aux réseaux de Petri, pour la formalisation des modèles Aspect-UML est d'autant plus pertinent si le type de réseaux choisi dispose d'outils pour la vérification et l'analyse automatique. Pour cela, notre choix s'est porté plus particulièrement sur les réseaux de Petri colorés (rdPc) [Jen97] qui disposent d'un outil de simulation et d'analyse automatique qui a fait ses preuves, soit CPN Tools [KCJ98]. Nous proposons ainsi une traduction de Aspect-UML vers le formalisme rdPc. Cette traduction permet entre autres de décrire les préoccupations de base et les préoccupations transverses comme des réseaux de Petri et ensuite de décrire les processus de composition et de tissage à l'aide d'opérateurs de composition de réseaux de Petri que nous définissons. Il convient cependant de noter que les rdPc font appel au langage fonctionnel ML [MTHM96] pour la spécification des données. Nous pensons, en fait, que la disparité des paradigmes aspect et fonctionnel est un inconvénient pour le choix des rdPc, car le processus de traduction de Aspect-UML vers le formalisme des rdPc devra considérer un passage d'un monde Aspect/Objet vers un monde fonctionnel. Passage que nous estimons être assez complexe, délicat et non sans compromis.

Cette complexité de la traduction nous amène, dans un second temps, à considérer les réseaux de Petri objet, tels qu'incarnés par le formalisme COOPN/2 [Bib97]. Il nous semble, en effet, plus naturel d'opter pour ce type de réseaux de Petri dits orientés objet plutôt qu'*à la ML*, pour pallier la différence des paradigmes aspect et fonctionnel. Nous pensons que la traduction, dans ce cas-ci, est plus naturelle, simple et directe. Or, le gain en abstraction et en facilité d'expression se fait ici au détriment de la complexité de l'analyse et de la vérification des systèmes décrits. À notre connaissance, il ne semble pas y avoir d'outils de vérification satisfaisants et performants pour ce type de réseaux.

Bref, les réseaux de Petri orientés objet nous permettent de définir une sémantique claire et naturelle de Aspect-UML mais nous offrent peu de mécanismes de vérification automatique. Pour s'affranchir de cette limite, nous avons pensé, dans une troisième étape, à l'adoption d'un nouveau formalisme comme modèle cible pour la traduction de notre profil Aspect-UML. Notre choix s'est porté cette fois-ci sur Alloy [Jac06], un langage de modélisation structurelle basé sur la logique du premier ordre. Alloy a été conçu pour la spécification formelle et l'analyse des modèles orientés objet. Il a été développé à partir d'idées inspirées de Z [Spi92] et des différentes tentatives de formaliser la modélisation objet. Notre choix d'Alloy est principalement motivé par le fait qu'il est un langage de spécification de style déclaratif tout comme Aspect-UML (notons que les réseaux de Petri sont de type opérationnels). Aussi, Alloy permet une vérification automatique par résolution de contraintes et non par énumération des états tel que c'est le cas dans les réseaux de Petri.

Plus précisément, (1) Alloy est doté d'une sémantique mathématique simple et uniforme et d'une syntaxe facile (tout est basé sur la notion de *relation*, il n'offre pas de constructions spéciales pour définir les machines d'états, les traces, la synchronisation, la concurrence, etc.) ; (2) il est assez expressif pour décrire la composition des flots de contrôle, même s'il ne dispose pas directement d'opérateurs dédiés ;

(3) il offre à la fois des représentations textuelles et graphiques ; (4) il définit des concepts et des mécanismes qui peuvent facilement simuler les concepts du paradigme aspect/objet ; (5) il est soutenu par un outil [All] d'analyse sémantique efficace et entièrement automatisé.

Ainsi, dans le cadre de cette thèse nous proposons une traduction formelle des modèles Aspect-UML non seulement en termes de réseaux de Petri mais également une formalisation de Aspect-UML vers Alloy. Pour ce faire, nous définissons formellement une fonction de traduction, entièrement automatisable, qui prend en entrée un modèle Aspect-UML et retourne en sortie une spécification Alloy sémantiquement équivalente et surtout vérifiable. Cette traduction est bien sûr cohérente avec celle présentée en termes de réseaux de Petri. Nous pensons, néanmoins, que la traduction vers Alloy est bien plus naturelle et plus souple que celle vers les rdPc, vu les raisons déjà citées plus haut (et résumées dans la table 1.1, page 12).

Une fois le profil Aspect-UML doté d'une sémantique dans un formalisme disposant de mécanismes de vérification automatique (figure 1.3), il nous est à présent possible d'atteindre notre objectif de vérification de la composition des aspects. Rappelons cependant que parmi les sémantiques proposées seuls les rdPc et Alloy sont soutenus par des outils de vérification.

Bien que le modèle des réseaux de Petri jouisse d'une renommée incontestable (c'est, en effet, un formalisme qui a fait ses preuves), nous pensons néanmoins que l'utilisation de l'outil Alloy est bien plus profitable pour nous quant à nos objectifs de vérification. En fait, tout comme la majorité des outils de vérification utilisant la méthode de vérification basée sur les modèles [GCDH+01], l'outil CPN Tools supportant les rdPc se voit rapidement confronté au problème *d'explosion d'états*. L'outil Alloy, quant à lui, contourne cette limite. D'une part, Alloy procède pour la vérification par résolution de contraintes plutôt que par énumération des états. D'autre part, il ne considère que des modèles de petite taille et ce en fixant au préalable la portée (c'est-à-dire le nombre d'éléments pour chaque type de données)

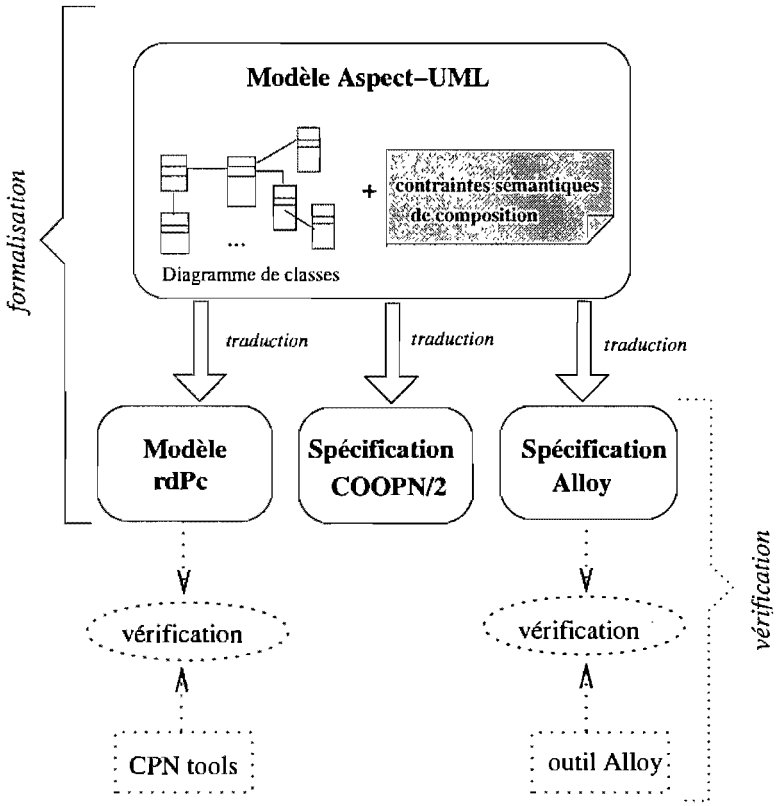


FIG. 1.3 – Sémantiques proposées pour le profil Aspect-UML.

des modèles à analyser[2]. L'analyse des modèles avec Alloy repose, en effet, sur l'hypothèse fondamentale *de la petite portée* (*the small scope hypothesis*) [Jac06]. Cette hypothèse stipule que les réponses négatives d'une analyse ont tendance à apparaître dans les petites instances de modèles. L'absence d'erreurs dans les petites instances peut ainsi renforcer et augmenter notre assurance quant à la correction effective du modèle. Certes, l'analyse avec Alloy est consistante (ne retourne jamais de faux positifs) mais demeure incomplète[3] (puisqu'elle ne considère que les modèles dont la taille ne dépasse pas celle fixée par l'utilisateur).

En optant pour Alloy, nous avantageons donc une vérification finie sans explosion d'états mais incomplète au lieu d'une vérification complète mais sujette au

[2]La portée est définie soit par l'utilisateur, soit fixée par défaut.

[3]Notons, cependant, que l'analyse est complète à l'intérieur de la portée choisie.

problème d'explosion comme c'est le cas avec les rdPc (voir tableau comparatif 1.1, page 12). Cela dit, nous ne pensons pas que cette incomplétude de la vérification avec Alloy soit un inconvénient majeur à notre approche. En effet, selon [Jac06] la plupart des défauts et des erreurs dans les modèles peuvent être déjà observés dans les petites instances, puisque ces défauts sont très souvent le résultat de certains éléments traités et développés incorrectement dès le départ.

Bien que nous privilégions la vérification de nos modèles Aspect-UML avec Alloy (dans le cadre de cette thèse), il est opportun de noter que le cadre formel (figure 1.4, page 13) que nous proposons pour le développement par aspects laisse la voie ouverte à d'autres pratiques de simulation et de vérification formelle des systèmes par aspects et ce à travers le formalisme des réseaux de Petri.

Finalement, le tableau 1.1 donné à la page 12 résume les avantages et les inconvénients des trois sémantiques proposées. Les symboles utilisés pour la légende devront être interprétés ainsi : $\surd$: supporté, ($\surd$) : indirectement supporté (simulable), - : non supporté.

	rdPc	COOPN/2	Alloy
sémantique formelle	$\surd$	$\surd$	$\surd$
expression de la composition	$\surd$	$\surd$	($\surd$)
représentation graphique	$\surd$	$\surd$	$\surd$
orientation aspect/objet	-	$\surd$	($\surd$)
outil de vérification	$\surd$	-	$\surd$
limitation du problème d'explosion	-	-	$\surd$
complétude de la vérification	$\surd$	$\surd$	-

TAB. 1.1 – Comparasion des modèles formels proposés pour la formalisation de Aspect-UML.

1.4 Étude de cas : Un système de téléphonie à base de services

Afin d'illustrer le bien fondé de notre approche, nous prêtons notre cadre formel de développement orienté aspect à la modélisation et à la vérification d'une

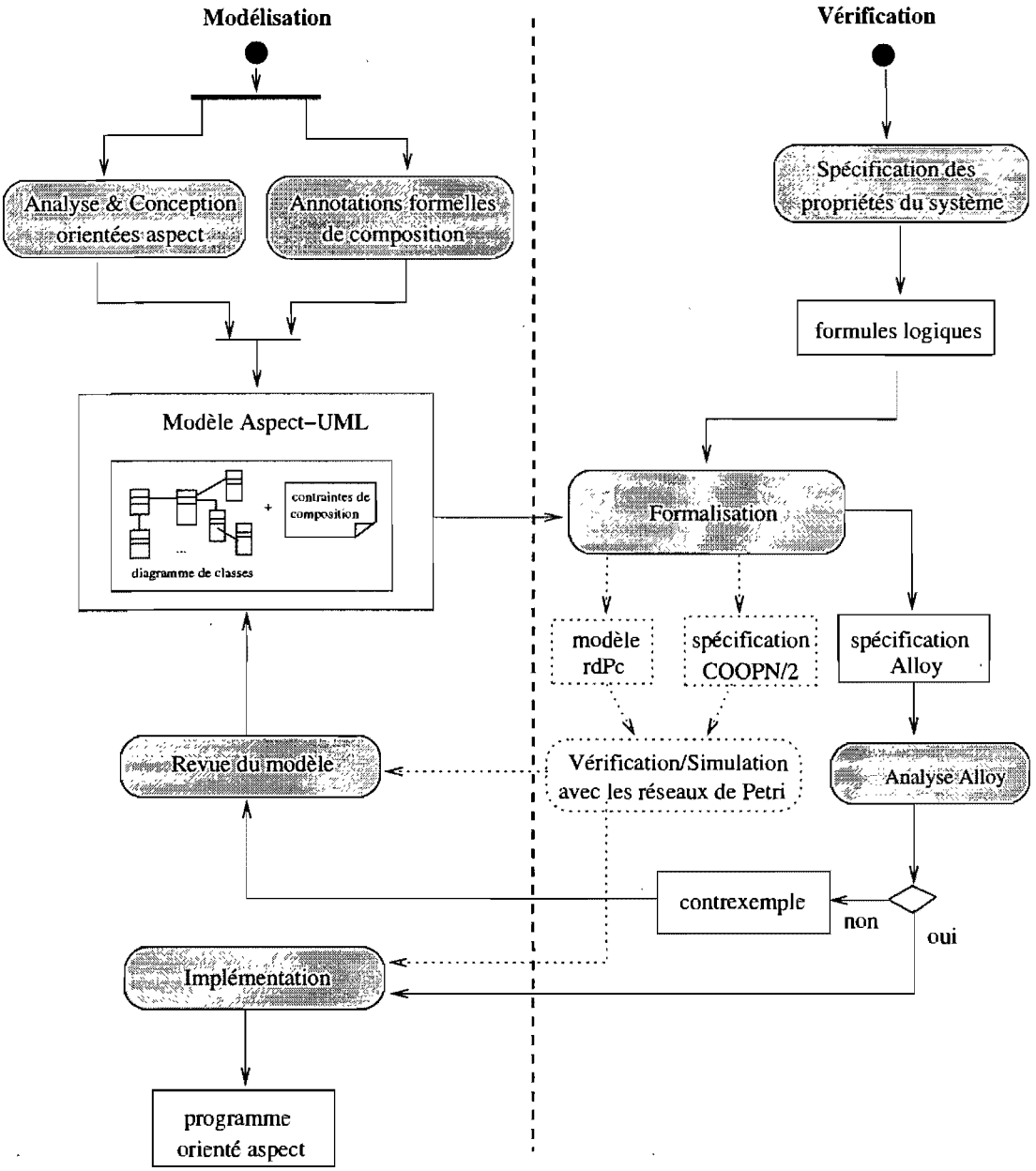


FIG. 1.4 – Un cadre formel pour le développement orienté aspect.

application de téléphonie conçue à base de services. Dans une telle application, le noyau du système (appelé aussi système de base) permet uniquement de gérer les connexions téléphoniques entre les abonnés (initiation et rupture des communications). Un service téléphonique, quant à lui, est un composant logiciel qui réalise une fonctionnalité additionnelle (par rapport au noyau) telle que le transfert d'appels, la messagerie vocale, le blocage d'appels, etc. Les services sont généralement développés et testés à part de façon isolée du reste du système, le plus souvent par des développeurs différents. Ils sont ensuite rajoutés de façon incrémentale au système de téléphonie, durant les différentes phases du cycle de développement. Notons finalement que les services sont parfois optionnels ; ils peuvent être activés ou désactivés dynamiquement.

Plus formellement, un système de téléphonie à base des services S_1, S_2, S_3, ... a la forme $B + F_1 + F_2 + F_3 + \ldots$, où B est la description de base, chaque F_i est la fonctionnalité décrivant le service S_i, et "+" dénote l'opération de composition des fonctionnalités.

Cependant, quand plusieurs services sont ajoutés à un système, il peut y avoir des *interactions* entre les différentes fonctionnalités décrivant les services. Si certaines interactions peuvent parfois être bénignes, en général les interactions peuvent être gravement dommageables et compromettantes pour le développement du système et/ou les attentes de l'utilisateur. L'interaction de services est définie dans [BDC+88] comme suit : *une interaction de services est un comportement où un service inhibe ou déroute l'exécution prévue d'un autre service ou d'une autre fonctionnalité, ou crée un dilemme d'exécution entre les fonctionnalités des services.*

Comme exemple d'interaction, considérons le cas suivant. Si un usager donné souscrit en même temps au service de messagerie vocale si la ligne est occupée et au service d'interruption d'appel si la ligne est occupée, alors que se passera-t-il si cet usager reçoit un nouvel appel alors qu'il est en cours de communication ? Dans ce cas, le système se retrouvera face à un dilemme quant au choix du service à

exécuter.

Un autre cas d'interaction plus subtile est le suivant. Si un usager (Bob) souscrit au service de transfert d'appel, en transférant tous ses appels vers un autre usager (Alice), et si Alice souscrit au service de blocage de tous les appels venant d'un autre usager (Marie), alors que se passera-t-il si Marie appelle Bob ? Dans ce cas-ci, l'appel de Marie, déstiné à Bob, sera transféré vers Alice (croyant qu'il s'agit de Bob), ce qui compromet clairement la fonctionnalité du service de blocage d'appel, et contrarira sérieusement l'usager (Alice). Ce scénario d'interaction impliquant les deux services de blocage d'appels et de transfert d'appels est schématisé par la figure 1.5 (les étapes (1) et (5) du scénario montrent clairement que la fonctionnalité du service de blocage d'appel est inhibée).

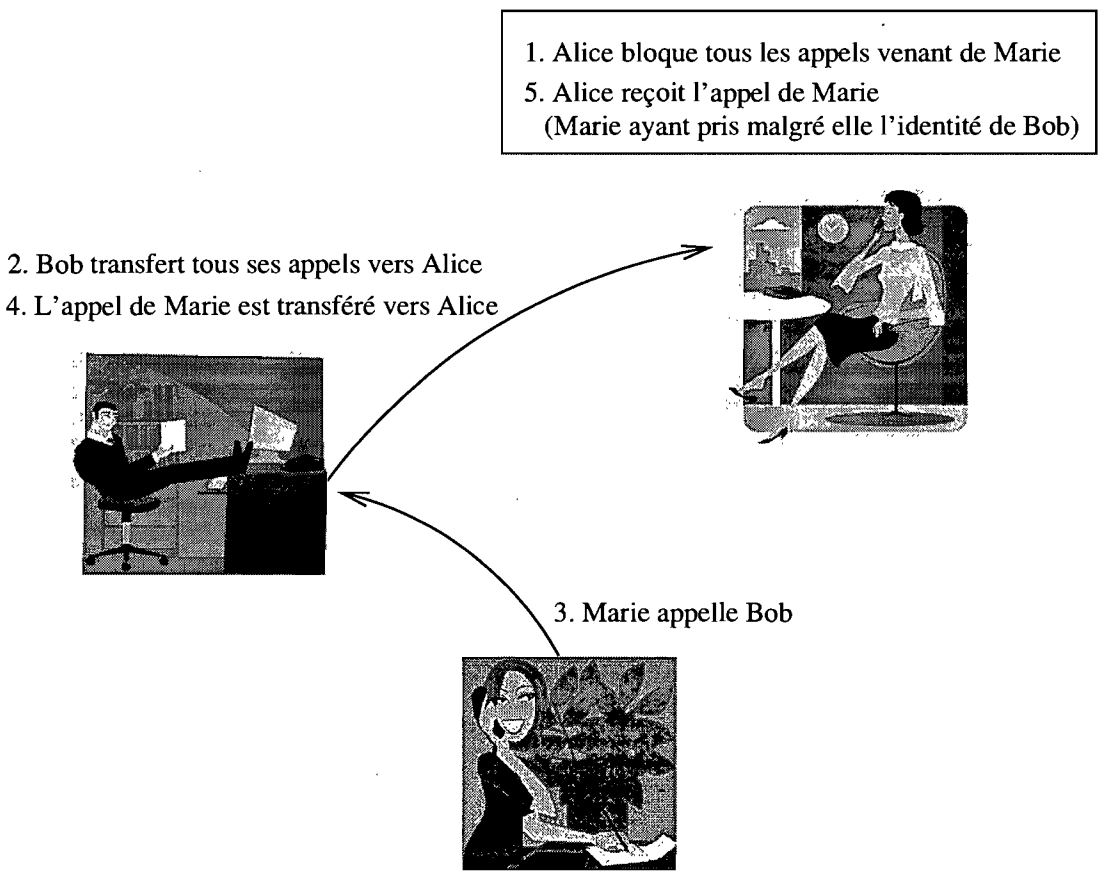


FIG. 1.5 – Scénario décrivant une interaction entre services.

Les interactions entre services peuvent être très difficiles à analyser et à résoudre, surtout si plus d'un service doit être rajouté à la description de base. En effet, avec le nombre sans cesse croissant des services, il y a une explosion combinatoire du nombre de scénarios susceptibles de comporter des interactions. En général, ni les inspections manuelles, ni les simples tests n'offrent de solution efficace et flexible à la détection de ces interactions nuisibles. Des approches plus précises et plus rigoureuses s'attaquant à ce problème sont alors nécessaires.

Dans le cadre de cette thèse, nous considérons l'application de téléphonie à base de services schématisée à la figure 1.6. Le système de base permet de gérer les connexions du réseau téléphonique qui consiste en des appareils téléphoniques, des clients à la fois propriétaires des téléphones et usagers du réseau et un contrôleur de réseau. À cette description de base, nous souhaitons intégrer les sept services additionnels donnés par la figure. Notons que ces services sont inspirés de [Zav04] et [Asp02].

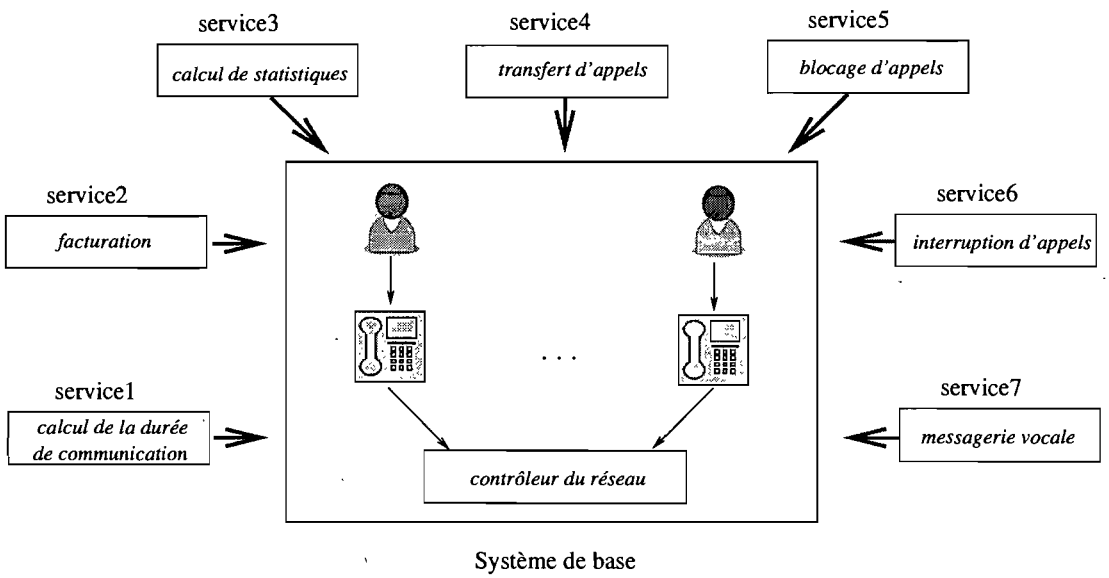


FIG. 1.6 – Une application de téléphonie à base de services.

Le choix de cette étude de cas pour notre travail est pertinent à plus d'un titre.

En effet, deux raisons au moins peuvent être évoquées pour une telle étude. Nous les explicitons dans ce qui suit.

1. La programmation orientée aspect se prête bien à l'implémentation des systèmes de téléphonie à base de services. L'approche aspect permet en fait de garantir la séparation effective des fonctionnalités décrivant les services et de la description de base du système. Elle fournit un support direct pour une représentation explicite de ceux-ci, contribuant à garder visible et isolée dans une unité modulaire chaque service de l'application. D'autre part, les concepts de *service* et d'*aspect* sont liés dans le sens où ils servent tous les deux à ajouter des fonctionnalités possiblement transversales à un système de base.
2. En proposant une implémentation orientée aspect aux systèmes de téléphonie à base de services, nous ramenons le problème d'interaction de services au problème d'interaction entre aspects, c'est-à-dire notre sujet de prédilection.

1.5 Contributions majeures

Dans cette thèse, nous proposons un cadre formel pour le développement orienté aspect afin de prendre en charge la modélisation et la vérification des interactions dues aux aspects. Nous illustrons l'emploi de notre approche de développement sur une étude de cas décrivant une application réelle de téléphonie à base de services. Plus précisément, nos contributions de recherche dans le cadre de cette thèse sont les suivantes :

1. *Définition d'un profil UML pour la modélisation des systèmes orientés aspect* [VM04a, MV06a]. Nous définissons un profil appelé Aspect-UML qui (1) prend en charge le concept d'aspect durant le cycle de développement à travers les vues de cas d'utilisation, structurelle et comportementale ; (2) assure la traçabilité des aspects entre toutes les phases de développement grâce au raffinement successif de modèles ; et (3) spécifie clairement la composition

des aspects à l'aide des contraintes sémantiques.

2. *Définition d'une sémantique formelle du profil Aspect-UML en termes de réseaux de Petri* [MV05, MV06a, MV06b]. Nous proposons d'abord une traduction de Aspect-UML vers le formalisme des réseaux de Petri colorés (rdPc) [MV05, MV06a]. Dans cette traduction nous décrivons, en particulier, les aspects et les fonctionnalités de base d'un modèle Aspect-UML comme des rdPc. Ensuite pour former le système global, les rdPc ainsi obtenus sont composés et tissés à l'aide d'opérateurs de composition que nous définissons. Cette formalisation permet de doter le paradigme aspect d'une sémantique formelle précise permettant la simulation et la vérification de propriétés intéressantes des systèmes par aspects.

   Nous proposons aussi une formalisation du profil Aspect-UML en réseau de Petri orienté objet [MV06b], en traduisant un modèle Aspect-UML vers une spécification COOPN/2. En plus d'offrir une sémantique naturelle et intuitive au paradigme aspect, cette formalisation démontre une application intéressante des réseaux de Petri orientés objet et en particulier du formalisme COOPN/2.

3. *Formalisation du profil Aspect-UML en Alloy* [MV07a]. Nous proposons une formalisation entièrement automatisable qui permet de traduire un modèle Aspect-UML vers une spécification Alloy. La traduction inclut notamment la spécification en Alloy des éléments statiques d'un modèle Aspect-UML, de ses éléments comportementaux et des processus de composition et de tissage des aspects.

4. *Vérification des interactions dues aux aspects* [MV07b, MV07a]. Nous présentons une approche pour la vérification des interactions dues aux aspects en utilisant l'outil Alloy. Cette approche suppose que le système de base et les aspects sont tous individuellement corrects et focalise sur la détection d'er-

reurs résultant des interactions dues aux aspects. Plus précisément, elle vise à révéler des problèmes d'interférence importants, telles que la violation des propriétés locales et la violation des propriétés globales. L'approche proposée consiste alors à définir dans le langage Alloy un patron générique pour chaque type de propriétés à vérifier (locale contre globale). Pour la vérification d'une propriété donnée, il suffit alors d'instancier le patron correspondant et de l'exécuter avec l'analyseur Alloy.

5. *Détection et résolution du problème d'interaction entre services téléphoniques* [MV07a]. L'étude de cas de la téléphonie à base de services, que nous proposons pour illustrer notre approche, contribue largement à la résolution du problème crucial des interactions entre services connu dans les systèmes de téléphonie. En effet, l'approche aspect peut être une solution intéressante pour la modélisation, la vérification et l'implémentation de genre de systèmes, en associant les services au concept d'aspects.

1.6 Organisation de la thèse

Le deuxième chapitre présente un état de l'art de la modélisation et de la vérification dans le cadre du développement orienté aspect. Après avoir introduit le paradigme aspect, ce chapitre présente les principales approches proposées actuellement pour la modélisation et la vérification des systèmes par aspects. La notion d'interactions dues aux aspects y est également présentée. Les quelques travaux ayant décrit et classifié ces interactions sont aussi recensés et présentés.

Le troisième chapitre traite de la modélisation orientée aspect. Il vise à définir un profil UML pour le paradigme aspect. D'abord, ce chapitre présente brièvement le langage unifié UML et ses mécanismes d'extension. Ensuite, notre profil Aspect-UML y est présenté, en décrivant ses trois vues de cas d'utilisation, structurelle et comportementale. Aussi, l'approche de modélisation avec Aspect-UML y est

illustrée sur un cas restreint de l'application de téléphonie à base de services.

Le quatrième chapitre considère la formalisation du profil Aspect-UML en termes de réseaux de Petri. Il rappelle dans une première étape le formalisme des réseaux de Petri, en particulier les réseaux de Petri colorés et les réseaux de Petri objet (COOPN/2). Par la suite, la traduction de Aspect-UML vers les réseaux de Petri colorés y est présentée et détaillée, de même que la traduction vers les réseaux orientés objet. Dans les deux cas, la traduction proposée est expliquée et illustrée sur une étude de cas.

Le cinquième chapitre est consacré à la formalisation du profil Aspect-UML en Alloy. Il présente d'abord l'outil d'analyse Alloy. Il introduit ensuite le langage de spécification Alloy et illustre ses différents concepts sur un exemple simple. Par la suite, il décrit par des règles formelles, en des étapes claires et successives, la traduction d'un modèle Aspect-UML vers une spécification Alloy, tout en l'illustrant sur notre étude de cas.

Le sixième chapitre traite de la vérification formelle des interactions dues aux aspects. Après, avoir décrit l'application complète de téléphonie à base de services, ce chapitre illustre en détail l'application de notre cadre formel de développement à cette étude de cas. Ce chapitre met notamment l'accent sur l'aspect vérification, en relevant des cas d'interaction intéressants entre les différents services considérés.

Finalement, la conclusion de ce rapport rappelle nos contributions et explicite les principales perspectives de ce travail de recherche.

CHAPITRE 2

ÉTAT DE L'ART

2.1 Introduction

Ce chapitre présente un état de l'art sur la modélisation et la vérification des systèmes par aspects. Puisque nous nous intéressons plus particulièrement aux interactions dues aux aspects, ce chapitre recense et présente aussi les quelques travaux ayant décrit et classifié ces interactions. Dans un premier temps, nous présentons d'abord le paradigme aspect et nous définissons brièvement ses principaux concepts. Dans les sections qui suivent, nous résumons les principales approches proposées dans la littérature pour la classification des interactions dues aux aspects, la modélisation orientée aspect, la composition des aspects dans les modèles orientés aspect et la vérification des interactions dues aux aspects. Notamment, nous mettons en avant l'utilité de ces approches et montrons les problèmes et limites de leur utilisation.

2.2 Le paradigme aspect

Le paradigme aspect utilise le principe de la séparation des préoccupations afin d'offrir une meilleure organisation des programmes, par rapport aux approches classiques de développement. En effet, dans les approches classiques, cette organisation est le plus souvent mal adaptée pour la représentation explicite de certaines préoccupations des systèmes, dites préoccupations *transverses*. La mise en oeuvre de ces préoccupations donne souvent lieu à une dispersion et un enchevêtrement de code au sein de l'application qui rendent difficile, entre autres, l'évolution du système et la réutilisation de ses composants. Par essence, le paradigme aspect permet la programmation séparée et modulaire des préoccupations transverses. Il propose, en

effet, de décomposer les programmes non seulement en unités modulaires propres aux préoccupations de base, mais aussi en unités modulaires spécifiques aux préoccupations transverses. De plus, ce paradigme offre des moyens facilitant la composition et l'intégration des diverses unités afin de produire des systèmes mieux organisés, plus compréhensibles et offrant une meilleure réutilisabilité.

Plusieurs idées de solutions ont émergé ces dernières années pour la séparation des préoccupations, dans le paradigme aspect. Ces solutions expriment clairement comment décomposer les préoccupations d'une application et comment les recomposer ensuite pour former un système global. Comme solutions nous citons : les filtres de composition [AT98], la séparation multidimensionnelle des préoccupations [TOHS99] et l'approche asymétrique [KLM+97]. À noter que, dans ce travail, nous considérons le paradigme aspect selon l'approche asymétrique. En fait, il s'agit de l'approche la plus commune et celle adoptée par les concepteurs des langages orientés aspect les plus connus tels que AspectJ.

L'approche asymétrique du paradigme aspect est basée sur le *modèle du point de jointure* (ou *join point model*). Cette approche utilise les concepts de composant primaire et de composant secondaire, où les composants primaires forment les préoccupations de base et les composants secondaires sont les préoccupations transverses ou aspects. Un mécanisme de composition appelé *tissage* (*weaving*), permet de lier les préoccupations transverses aux composants de base à des points bien définis appelés *points de jointure* (ou *join points*). Comme nous le verrons plus loin, la structure des composants secondaires (aspects) est différente de la structure des composants primaires (classes). De plus, la composition par point de jointure n'est possible qu'entre un composant primaire et un composant de base. C'est ce qui vaut à cette approche l'appellation dite *asymétrique*. Le lecteur peut se référer à [Lad02c, Lad02a, Lad02b, Lad03] pour une introduction à l'approche asymétrique du paradigme aspect et au langage AspectJ.

Les trois étapes de développement de l'approche asymétrique sont principale-

ment les suivantes.

1. Décomposition aspectuelle : décomposer les besoins de façon à identifier les préoccupations de base et les préoccupations transverses ;
2. Implémentation des préoccupations : implémenter chaque préoccupation séparément ;
3. Recomposition aspectuelle : intégrer les préoccupations transverses dans les préoccupations de base afin de constituer le système final.

2.2.1 Principaux concepts du paradigme aspect

Afin de comprendre les principes de la technique aspect, nous présentons dans ce qui suit les concepts et mécanismes de base introduits par les langages de programmation orientés aspect. Il s'agit d'un ensemble d'éléments permettant, d'une part, de définir le code relatif aux préoccupations transverses (destinés à augmenter ou à modifier le code des éléments de base), et de préciser, d'autre part, la manière (*c-à-d* où, quand et comment) dont ce code sera composé avec le code de base de l'application.

Point de jointure (*Join point*). L'un des éléments les plus importants dans la technique de programmation par aspects est la notion de point de jointure. Un point de jointure est un point d'exécution pouvant se rattacher à un appel de méthode, à l'exécution d'une méthode, à l'invocation d'un constructeur, au traitement d'une exception, ou à un autre point identifié dans le flot d'exécution d'un programme.

Point de coupure (*Pointcut*). Une fois les points de jointure déterminés, ceux-ci peuvent être assemblés et déclarés dans un point de coupure (appelé également point de recouvrement). Un point de coupure est plus qu'un contenant pour les points de jointure ; il montre directement comment une préoccupation va entrecouper l'application de base. Il s'agit d'un prédicat construit sur la base d'un

ou plusieurs points de jointure, et éventuellement, d'autres paramètres précisant le contexte d'exécution. Ce contexte définit les données du point de jointure. Notamment, il peut contenir l'objet dont on invoque la méthode au point de jointure et les arguments de l'appel.

Advice. Le partenaire naturel d'un point de coupure est l'advice. En fait, un advice est toujours attaché à un point de coupure. C'est un fragment de code à exécuter quand l'application atteint les points de jointure déclarés dans le point de coupure. Le corps d'un advice se comporte comme le corps d'une méthode. Cependant, l'advice peut être exécuté *avant*, *après* ou *à la place* des points de jointure contenus dans le point de coupure (attaché à cet advice), s'il est précédé respectivement par un des labels *before*, *after* ou *around*.

Aspect. L'unité de modularisation d'une préoccupation transverse est appelée *aspect*. La définition d'un aspect peut contenir, comme la définition d'une classe, des déclarations d'attributs et des définitions de méthodes propres à l'aspect. Distinct du concept de classe, l'aspect contient aussi des définitions de points de coupure et d'advices.

La figure 2.1 montre un exemple d'aspect tiré de [Lad03]. Le code AspectJ donné décrit, entre autres, un aspect appelé *LoggingAspect* réalisant une tâche de journalisation avant chaque appel de la méthode *credit* de la classe *Account*. Le code montre la syntaxe AspectJ des concepts d'aspect, d'advice, de point de coupure. En particulier, la syntaxe du point de coupure *creditOperation* définit le point de jointure concerné par cet aspect, en l'occurrence l'appel de la méthode *void Account.credit(float)* et les paramètres précisant le contexte d'exécution de l'advice. En effet, ce contexte précise que l'objet cible correspond à la valeur de la variable *a*, et la valeur de l'argument de la méthode affectée (*credit*) correspond à la valeur de la variable *m*. Ces valeurs sont transmises respectivement aux paramètres *account* et *amount* de l'advice, lors de l'exécution.

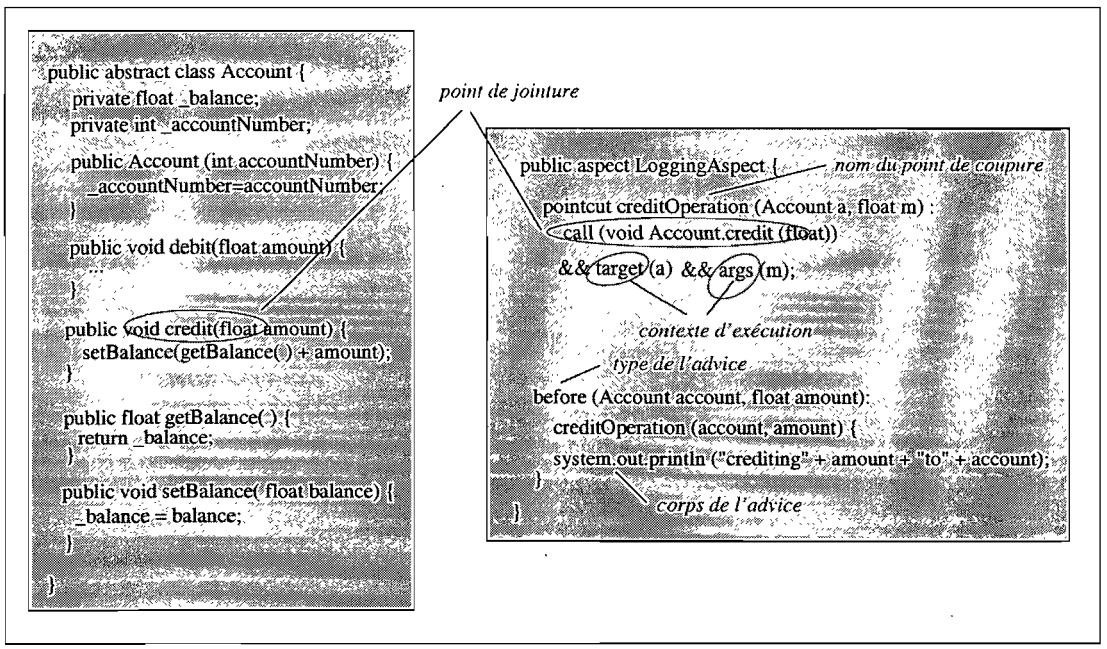


FIG. 2.1 – Exemple d'aspect écrit en AspectJ.

2.3 Classification des interactions dues aux aspects

Le paradigme aspect promeut la séparation des préoccupations et offre des avantages incontestables. Toutefois, le paradigme étant relativement jeune, plusieurs questions et problèmes restent à éclaircir tels que : comment identifier et décomposer les préoccupations transversales ? Comment recomposer ces préoccupations pour former un système global cohérent ? La façon de composer les aspects demeure une tâche délicate, car une mauvaise composition peut induire des comportements indésirables. En effet, des interactions conflictuelles peuvent se manifester entre les aspects eux-mêmes et/ou entre les aspects et le système de base. D'une part, des dépendances non apparentes peuvent s'établir entre certains aspects. D'autre part, les effets d'un aspect peuvent nuire ou empêcher l'exécution d'un autre. Ainsi, les interactions entre aspects nécessitent une analyse complexe et laborieuse pour les développeurs, surtout qu'actuellement il n'existe pas d'outils matures pour l'aide à la détection et résolution des interactions potentiellement conflictuelles. Afin de

mieux comprendre la façon dont les aspects interagissent et de mieux cerner le problème des interactions indésirables entre aspects, cette section est consacrée à la revue de littérature des quelques travaux qui se sont penchés sur ce problème et qui proposent une classification de ces interactions. Notons que les critères de classification varient d'une approche à l'autre. Certaines approches s'intéressent à classer les interactions elles-mêmes, par contre, d'autres s'intéressent à classer les aspects selon le type d'interaction qu'ils induisent. Par ailleurs, les auteurs adoptent différents points de vue pour la classification. Certains classent les interactions du point de vue de la spécification des besoins, afin de documenter les exigences d'un système, alors que d'autres considèrent le point de vue de l'implémentation, afin d'identifier et d'analyser les interactions présentes dans le code.

2.3.1 Classification de Katz *et al.* [KG99]

Ce travail est le premier à proposer une classification des interactions entre aspects. Les auteurs définissent trois catégories d'aspects : les aspects *spectatifs*, les aspects *regulatifs* et les aspects *invasifs*. Les aspects spectatifs sont ceux qui n'influencent pas les valeurs des variables du système de base. Quant aux *regulatifs*, ils peuvent affecter seulement le flot de contrôle du système de base (en court-circuitant par exemple certaines séquences d'exécution). Les aspects *invasifs* peuvent changer les valeurs des variables du système.

2.3.2 Classification de Clifton *et al.* [CL02c, CL02b, Cli05]

Une autre classification, du point de vue de l'implémentation, similaire à celle de [KG99] est la proposition de Clifton *et al.* Les auteurs suggèrent de classer les aspects en *observeurs* (ou *spectateurs*) et *assistants*.

Les observeurs sont les aspects qui ne changent pas la spécification des méthodes du système de base où ils sont tissés. Ici, le terme observation est utilisé afin de spécifier que la méthode est simplement observée par l'aspect, elle ne requiert, en fait, au-

cune information de cet aspect pour son exécution. Contrairement aux observeurs, les aspects dits assistants sont ceux qui changent et modifient la spécification du système de base.

2.3.3 Classification de Rinard *et al.* [RSB04]

La classification proposée par Rinard *et al.* offre un niveau de granularité plus fin que les précédentes. Ce travail s'intéresse à définir une relation entre un advice et une méthode, et ce en considérant la façon dont ceux-ci partagent le contexte d'exécution du système. Ainsi, un advice et une méthode sont dits *orthogonaux*, s'ils accèdent à des variables disjointes. Ils sont dits *indépendants*, si l'un n'accède jamais en écriture à une variable que l'autre peut lire ou écrire. Ils sont reliés par une relation d'*observation* si l'advice peut lire les variables que la méthode peut écrire. Inversement, ils sont reliés par une relation d'*actuation* si l'advice peut écrire les variables que la méthode peut lire. Finalement, un advice et une méthode sont liés par une relation d'*interférence* si les deux peuvent accéder en écriture à la même variable.

2.3.4 Classification de Kienzle *et al.* [KYX03]

Une proposition informelle pour classer les interactions entre aspects est celle de Kienzle *et al.* Ce travail s'intéresse aux dépendances entre aspects, sachant qu'un aspect est défini en se basant sur les services qu'il fournit, les services qu'il requiert des autres aspects et les services qu'il supprime. Les auteurs classent les interactions entre aspects selon les deux critères : les dépendances et le mécanisme d'activation. Ils distinguent alors trois types de dépendances : *orthogonale*, *unidirectionnelle* (*préservante* contre *modifiante*) et *circulaire*.

Les aspects dits orthogonaux sont des aspects qui offrent des services complètement indépendants. Un aspect unidirectionnel préservant fournit de nouveaux services en se basant sur des services offerts par d'autres aspects, sans toutefois

les altérer. Un aspect unidirectionnel modifiant remplace ou modifie les services offerts par d'autres aspects. Les aspects circulaires sont mutuellement dépendants, c'est-à-dire, le premier aspect requiert les services offerts par le second aspect, qui à son tour ne peut offrir ses services qu'avec l'aide du premier. Un exemple d'une telle dépendance est présenté dans [KG02]. Cependant, dans [KG02], les auteurs montrent que considérer chaque aspect d'une dépendance circulaire de façon isolée, n'a pas de signification. Il est en fait préférable et plus simple de les traiter comme un seul aspect. D'autre part, le critère du mécanisme d'activation distingue deux types d'aspects : les aspects *autonomes* qui peuvent agir seuls (par exemple périodiquement), et les aspects *déclenchés* qui sont activés par d'autres parties de l'application.

2.3.5 Classification de Brito *et al.* [BM04]

Ce travail introduit les concepts de *contribution positive* et de *contribution négative* entre les aspects au niveau de la spécification des besoins. Un aspect contribue positivement s'il renforce les services du système de base. Inversement, il contribue négativement s'il restreint les services du système de base. Cette approche vise à attribuer des priorités aux aspects, en considérant plus prioritaires les aspects qui contribuent positivement. En outre, cette approche considère que les situations de conflits entre aspects à un point de jointure peuvent survenir si les aspects contribuent négativement (les aspects qui contribuent positivement ne causent, en fait, aucun problème).

2.3.6 Classification de Khan *et al.* [KR06]

Ce travail donne une classification des dépendances entre les besoins d'un système. Les auteurs proposent une approche de découpage multidimensionnelle des préoccupations qui facilite la compréhension des dépendances entre les besoins. Les coupes proposées définissent quatre catégories d'exigences : *temporelles*, *condition-*

nelles, *liées aux règle d'affaires* ou *liées aux tâches à effectuer*. Un besoin appartient à une coupe temporelle s'il spécifie une propriété temporelle (comme par exemple l'action a doit être faite au temps t). Il appartient à la coupe conditionnelle s'il dépend de l'exécution d'un critère particulier (c-*à*-d tant que le critère est faux le besoin ne s'applique pas). Un besoin est dans la coupe de règles d'affaires s'il décrit des politiques organisationnelles, des décisions ou des structures. Il est dans une coupe orientée tâche, s'il dépend d'une réponse de l'utilisateur ou d'une entrée externe.

Les coupes des besoins peuvent être intra ou interdépendantes. Si une coupe est dépendante d'une autre coupe à l'intérieur d'un même besoin alors on parle d'*intradépendance*. Par contre, on parle d'*interdépendance* si la coupe d'un besoin dépend d'une coupe d'un autre besoin. En outre, une coupe d'un besoin peut résulter d'une autre coupe, elle peut utiliser ou être basée sur une coupe précédente, ou encore elle peut se produire simultanément qu'une autre coupe.

2.3.7 Classification de Zhang *et al.* [ZCdBG07]

La proposition de Zhang *et al.* décrit une approche pour identifier les interactions entre aspects et minimiser les situations de conflits lors du tissage. Elle est assez similaire à celle de Kienzle *et al.* [KYX03]. Les auteurs proposent trois types de relation de précédence entre aspects : la relation *follows*, la relation *hidden by* et la relation *dependent on*.

Si un aspect A *follows* un aspect B, alors l'aspect B est plus prioritaire que l'aspect A. Si un aspect A est *hidden by* un aspect B, alors l'aspect B désactive et annule l'aspect A quand ils sont appliqués au même point de jointure. Finalement, si un aspect A *dependent on* un aspect B alors l'aspect A ne peut être exécuté qu'en présence de l'aspect B.

2.3.8 Classification de Sanen *et al.* [STW+06, SLR+06]

La proposition de Sanen *et al.* décrit une approche pour classer et documenter les interactions entre aspects. Dans le même esprit que [BM04], les auteurs font remarquer que les interactions peuvent être positives tout comme elles peuvent être négatives. Ainsi, leur système de classification reflète cette distinction. Les quatre types d'interactions mis en évidence sont : *le conflit*, *la dépendance*, *le renforcement* et *l'exclusion mutuelle*. Ces types permettent de documenter les interactions en spécifiant l'effet qu'elles devraient avoir.

Le conflit saisit la situation d'interférence sémantique entre aspects. Il y a interférence si un aspect qui s'exécutait correctement en isolation, ne fonctionne plus correctement lorsque composé avec un second aspect. Dans ce cas, ce second aspect influence négativement l'exécution du premier aspect. Une interaction de dépendance décrit la situation où un aspect requiert explicitement un autre aspect pour son exécution. Quant au renforcement, ce type d'interaction survient quand un aspect influence l'exécution correcte d'un autre aspect positivement (par exemple en ajoutant des fonctionnalités supplémentaires et complémentaires). Finalement, l'exclusion mutuelle est l'effet d'une interaction entre aspects (implémentant des services similaires) qui n'active que l'un ou l'autre de ces aspects.

2.3.9 Discussion

Les travaux pour la classification des interactions entre aspects, présentés dans cette section, offrent un panorama intéressant des différentes façons de catégoriser les interactions entre aspects. Cependant, les critères choisis pour la classification varient d'une approche à l'autre et sont difficilement unifiables. Certaines approches s'intéressent aux interactions entre les aspects et le système de base, d'autres s'intéressent aux interactions entre les aspects eux-mêmes, d'autres encore s'intéressent aux interactions entre les besoins d'un système. Par ailleurs, les travaux présentés

s'intéressent aux interactions entre aspects à différents niveaux. Certains auteurs classent les interactions du point de vue de la spécification des besoins, afin de documenter les exigences d'un système, alors que d'autres considèrent le point de vue de l'implémentation, afin d'identifier et d'analyser les interactions présentes dans le code. Le tableau 2.1 résume les différentes approches selon le type d'interaction et le niveau considérés. Rappelons que les travaux qui se sont penchés sur l'analyse des interactions dues aux aspects et la détection des conflits (que ce soit du point de vue syntaxique ou sémantique) seront présentés à la section 2.6 de ce chapitre.

	type	niveau
Katz *et al.* [KG99]	aspect/système de base	implémentation
Clifton *et al.* [Cli05]	aspect/système de base	implémentation
Rinard *et al.* [RSB04]	advice/méthode	implémentation
Kienzle *et al.* [KYX03]	aspect/aspect	spécification des besoins
Brito *et al.* [BM04]	aspect/aspect	spécification des besoins
Khan *et al.* [KR06]	besoin/besoin	spécification des besoins
Zhang *et al.* [ZCdBG07]	aspect/aspect	spécification des besoins
Sanen *et al.* [SLR+06]	aspect/aspect	spécification des besoins

TAB. 2.1 – Résumé des approches pour la classification des interactions entre aspects.

Nous pensons qu'il serait fort utile est intéressant d'unifier ces classifications en définissant une taxonomie des interactions entre aspects. Cette taxonomie permettrait de cataloguer et de classer les différentes définitions associées aux interactions entre aspects. Elle servirait de point de référence pour comparer les nombreux travaux portant sur ce sujet.

2.4 Modélisation orientée aspect

La programmation par aspects est encore jeune et nouvelle, elle ne dispose pas encore d'outils ou de langages de modélisation matures à l'instar de la programmation orientée objet. Entre autres, plusieurs chercheurs travaillent à l'intégration du

paradigme aspect au niveau de la modélisation. Bien que plusieurs se soient penchés sur le sujet aucun langage de modélisation pour le paradigme aspect ne s'est clairement démarqué. La plupart des travaux sur la modélisation par aspects s'inspirent très fortement de la modélisation par objets. Ainsi, ce nouveau paradigme hérite de tous les acquis de la technologie objet. L'un des acquis incontestable de l'objet est le langage unifié de modélisation par objets UML [Obj05]. L'utilisation d'UML comme assise à la modélisation par aspects est très naturelle, sachant que la programmation par aspects se veut une extension voir une amélioration de la programmation par objets. La plupart des langages de programmation par aspects proposés actuellement sont tous construits autour de l'orienté objet : ils sont soit des extensions soit des évolutions des langages orientés objet.

Un nombre important d'approches de modélisation par aspects a vu le jour afin de soutenir le processus de développement par aspects appelé AOSD (*Aspect Oriented Software Development*). Les techniques AOSD doivent fournir des moyens systématiques pour (1) l'identification, (2) la modularité, (3) la représentation et (4) la composition des préoccupations transverses. Le rapport de la communauté AOSD Europe [CRS+05] dresse une revue des différentes approches de modélisation proposées dans la littérature pour le paradigme aspect.

Cette section présente et discute des approches de modélisation les plus significatives et ce dans le but de motiver et de soutenir notre proposition, c'est-à-dire celle que nous présenterons dans le chapitre suivant. La synthèse de ces propositions, dont les principales caractéristiques sont résumées dans le tableau 2.2, page 40, est basée sur les propriétés suivantes :

- **Portée.** La portée d'une proposition détermine la phase du cycle de développement considérée par celle-ci : la phase d'analyse et de spécification des besoins, ou la phase de conception.
- **Modèle support**. Si une proposition consiste à définir un ensemble de concepts et de relations spécifiques à un modèle ou à un langage de pro-

grammation par aspects particulier (comme AspectJ par exemple), on parle dans ce cas de modèle support spécifique, sinon si la proposition est indépendante d'un modèle ou d'un langage de programmation en particulier, le modèle support est alors dit général.

- **Exhaustivité**. Une proposition est complète si elle modélise l'ensemble complet des concepts et relations de son modèle ou langage support, sinon si elle ne considère que les principaux éléments de cet ensemble, elle est dite partielle.
- **Vue**. Pour représenter un système, il convient le plus souvent de s'intéresser non seulement à la modélisation de sa structure statique, mais aussi à la modélisation de sa partie dynamique. Une proposition peut s'intéresser exclusivement à la vue structurelle ou comportementale d'un système, comme elle peut considérer les deux vues.
- **Extension**. La plupart des approches de modélisation existantes proposent, pour la définition des concepts aspect, d'étendre le langage UML. Cependant, elles se distinguent principalement par la définition qu'elles donnent au concept d'aspect lui-même. Selon les approches, celui-ci peut être représenté par une classe, un package, un composant, etc.

Pour présenter ces approches nous les regroupons selon la modélisation qu'elles proposent au concept d'aspect qui est l'unité de modularisation d'une préoccupation transverse.

2.4.1 Aspect comme cas d'utilisation

Jacobson dans [Jac03, JN05] présente une idée très intuitive pour modéliser les concepts du paradigme aspect par les cas d'utilisation. La correspondance entre les concepts de l'un et de l'autre est très naturelle. Cette correspondance s'appuie sur l'idée de base que les cas d'utilisation doivent être maintenus séparés, durant tout le cycle de vie du logiciel, de l'acquisition des besoins jusqu'au test, et ce

même quand ils entrecoupent plusieurs composants. Selon l'auteur Jacobson, la technique aspect semble être la solution idéale pour maintenir cette séparation. En utilisant les aspects, l'approche de modélisation par cas d'utilisation évoluera de façon substantielle. En effet, maintenir les cas d'utilisation séparés (même ceux reliés par des relations d'extension), durant toutes les phases de développement jusqu'à l'exécution, sera réalisable grâce aux principes de la programmation orientée aspect et à ses langages. Jacobson propose ainsi de modéliser les aspects par des extensions de cas d'utilisation et les points de jointure par des points d'extension. Dans ce cas, un point de coupure correspond à un ensemble de points d'extension rattachés à une réalisation de cas d'utilisation. Quant à l'advice, il correspond au comportement à réaliser dans une extension de cas d'utilisation.

2.4.2 Aspect comme stéréotype de classe UML

L'une des premières propositions de modélisation des aspects est celle de Suzuki et Yamamoto [SY99], elle intègre les aspects au niveau de la phase de conception pour la modélisation de systèmes réalisés plus précisément à l'aide d'AspectJ. Elle étend le langage UML par la définition de stéréotypes, et propose une notation pour la visualisation des aspects. Cette extension consiste notamment en la définition d'un stéréotype particulier de classe (la métaclasse Class du métamodèle d'UML) nommée « Aspect », pour modéliser les aspects. Une classe « Aspect » peut ainsi contenir un nombre arbitraire d'attributs et d'opérations et peut participer dans plusieurs associations, généralisations et avoir différentes relations de dépendance. Les advices sont définis comme étant des stéréotypes d'opérations de nom « weave ». Les auteurs proposent de modéliser la relation liant un aspect à une classe de base par une relation de dépendance de type abstraction (abstraction dependency relationship) stéréotypée par « realize ». Cette approche propose aussi de montrer le résultat statique du tissage par la représentation de la structure des classes finales du système à modéliser, en utilisant des packages contenant, respec-

tivement, les classes de base avec leurs structures originales et les aspects qui les entrecoupent et les classes après tissage avec leurs nouvelles structures. Cette proposition est, en fait, une première tentative de modélisation des aspects spécifique à un langage particulier de programmation (AspectJ). Elle est néanmoins insuffisante, car seulement quelques concepts d'AspectJ sont modélisés (la proposition reste muette quant à la modélisation des points de coupure) et de plus certains éléments de modélisation offerts sont inappropriés aux aspects (telle que la relation de réalisation pour modéliser une relation d'entrecoupement, car une réalisation est une relation entre une spécification et l'élément qui implémente cette spécification, alors que la relation d'entrecoupement entre aspects et classes a une sémantique complètement différente.

Plusieurs autres tentatives de modélisation des aspects, dont le modèle support est AspectJ, ont été proposées ensuite. Nous pouvons citer par exemple le travail de Stein *et al.* [SHU02] qui propose des éléments de modélisation pour l'ensemble des concepts et mécanismes offerts par AspectJ, dans le moindre des détails. En particulier, les auteurs proposent de modéliser un aspect par un stéréotype particulier de classe UML (de la metaclasse Class) et les points de coupure et les advices par des stéréotypes d'opérations (de la métaclasse Operation). Quant à la relation d'entrecoupement, elle est modélisée par une relation « crosscut » sans toutefois préciser de quelle relation UML il s'agit. Néanmoins, les auteurs l'associent plus à la relation « extend » d'UML plutôt qu'à ses différentes relations de dépendance. Par ailleurs, pour la représentation dynamique d'un système à base d'AspectJ, les auteurs proposent, en plus, de modéliser les points de jointure par des liens UML et par de nouveaux stéréotypes (tels que les pseudo-opérations).

Une autre proposition qui offre une couverture complète des concepts aspect, mais moins exhaustive que la précédente, est celle d'Aldawud *et al.* [AEB03]. Celle-ci se concentre, exclusivement, sur la représentation structurelle d'un système donné, où un aspect est modélisé par un stéréotype particulier de classe UML, les

points de coupure et les advices par des stéréotypes d'opérations et la relation d'entrecoupement par un stéréotype de la relation d'association d'UML (de la métaclasse Association). Par ailleurs, contrairement à la proposition précédente qui est dédiée spécifiquement à la modélisation des systèmes écrits en AspectJ, celle-ci ne vise pas une implémentation dans un langage spécifique.

Ces deux dernières propositions s'accordent, toutefois, pour la définition d'un point de coupure par un stéréotype particulier d'opération UML. À notre avis, cette modélisation est inadéquate; En fait un point de coupure est un conteneur pour un ensemble de points de jointure, il ne définit aucune opération. Selon nous, cette façon de modéliser ne respecte pas la définition d'un point de coupure.

À la différence de toutes les propositions discutées précédemment, l'approche de Fuentes *et al.* [FS06] propose de fournir un langage générique de modélisation par aspects complet, notamment, pour la représentation structurelle des systèmes indépendamment d'un domaine d'application particulier. Pour ce faire, les auteurs proposent de définir un aspect par un stéréotype de classe ou de composant, un point de jointure par un stéréotype « hook » associé à la méthode identifiée comme point de jointure et un advice par un stéréotype d'opération. Par ailleurs, la relation d'entrecoupement est modélisée soit à travers les points de coupure soit directement par des associations de liaison entre le système de base et les aspects. Dans le premier cas, les points de coupure sont modélisés textuellement (contraintes) ou graphiquement (diagrammes) et dans le second cas des diagrammes comportementaux doivent être élaborés pour spécifier les associations de liaison entre le système de base et les aspects. Si la généricité de cette approche est intéressante, elle présente toutefois quelques inconvénients : elle reste vague concernant la modélisation de l'entrecoupement. D'une part, elle ne précise pas exactement comment se fait la modélisation textuelle et graphique des points de coupure et d'autre part elle n'explique pas la modélisation des diagrammes comportementaux.

2.4.3 Aspect comme stéréotype de package UML

Les propositions, citées plus haut, s'accordent pour la définition d'un aspect par un stéréotype particulier de classe, alors que d'autres propositions étendant UML par stéréotypage considèrent un aspect comme étant un stéréotype particulier de package (de la métaclasse Package). Parmi ces propositions, nous citons le travail de Bash et Sanchez [BS03] qui couvre les concepts d'AspectJ et qui s'intéresse (comme dans le cas de la proposition de Stein) à la définition d'éléments de modélisation par aspects nécessaires à la représentation de la partie dynamique d'un système réalisé à l'aide d'AspectJ. Dans cette proposition, chaque aspect est encapsulé dans son propre package. La structure et les différents comportements de l'aspect peuvent être modélisés dans ce package, en incluant les diagrammes de classes, les diagrammes d'interactions et tout autre moyen permettant de spécifier ses fonctionnalités. Chaque package encapsulant un aspect porte le stéréotype « aspect ». Le diagramme de packages indique explicitement les points de jointure où les packages aspect entrecoupent le système principal. Ces points de jointure sont schématisés et définis à l'extérieur des packages aspects et des packages composants (ces points n'appartiennent en fait à aucun package exclusivement mais sont utilisés pour indiquer la relation d'entrecoupement entre un aspect et un composant). Des liens relient les points de jointure vers les packages aspects et les packages composants. Ces liens décrivent la relation de dépendance. A l'intérieur d'un package aspect, un diagramme de classes est utilisé pour montrer les classes entrecoupées par cet aspect et un diagramme de séquences est défini pour modéliser le comportement de l'aspect lui-même. Par ailleurs, les diagrammes de classes dans le package principal du système ne modélisent pas les aspects ; ce sont les diagrammes d'interactions du package principal qui les mettent en évidence.

Une autre proposition qui considère un aspect comme étant un stéréotype de package UML est celle de Clarke [CB05]. L'auteur propose une approche, appelée

"Theme", indépendante d'un modèle ou d'un langage de programmation par aspects en particulier, pour le développement orienté aspect, plus particulièrement en ce qui concerne les phases d'analyse (Theme/Doc) et de la conception (Theme/UML). D'une part, Theme/Doc aborde le problème de l'identification des aspects dans la phase d'analyse des besoins. Il propose pour cela de représenter les relations entre les fonctionnalités d'un système à l'aide d'un graphe d'analyse (il n'introduit et ne propose aucune modélisation spécifique pour les différents concepts du paradigme aspect). D'autre part, au niveau de la conception, Theme/UML permet de modéliser les fonctionnalités (les themes) d'un système et de spécifier comment celles-ci doivent être composées. Chaque thème décrit une vue du système, où seules les parties pertinentes à une fonctionnalité donnée sont montrées. On distingue deux types de thèmes : les thèmes de base, qui peuvent partager des structures et des comportements avec d'autres thèmes de base, et les thèmes transverses (ou les aspects) dont les comportements se superposent aux fonctionnalités des thèmes de base. Avec Theme/UML, un theme est modélisé par un stéréotype de package UML nommé « theme ». Chaque package « theme » contient le diagramme de classes et le diagramme de séquences décrivant la structure et les fonctionnalités du theme modélisé. De plus, un package « theme » décrivant un aspect est paramétré ; les paramètres représentent les points de jointure entrecoupés par l'aspect.

Notons que ces deux propositions qui modélisent un aspect par un stéréotype de package UML, restent imprécises quant à la modélisation des points de coupure et des advices. En effet, le comportement des aspects est modélisé de façon explicite dans les diagrammes de séquences.

2.4.4 Aspect comme stéréotype de composant UML

Certaines approches de modélisation choisissent de modéliser un aspect par un stéréotype de composant UML. Nous citons par exemple la proposition de Bara *et al.* [BGL04] qui présente un modèle conceptuel pour les aspects et dresse un

rapprochement entre les concepts de ce modèle et les concepts d'UML. Pour ce faire, les auteurs proposent un diagramme de composants et un diagramme aspectuel de classes pour la représentation structurelle des systèmes. Dans le diagramme de composants, les fonctionnalités de base du système sont encapsulées dans un composant principal et les aspects sont encapsulés dans un composant stéréotypé « aspect ». Le diagramme de composants modélise aussi le concept de tissage en l'encapsulant dans un composant stéréotypé « weaver ». L'idée est de gérer les points de jointure dans le composant weaver, à l'extérieur des aspects et du composant de base ; ceux-ci sont représentés comme des ports au niveau des différents composants. D'autre part, le diagramme aspectuel de classes permet de modéliser les relations entre les aspects et le système de base. Dans ce diagramme, les classes sont connectées à travers leurs ports respectifs à une classe association qui a le stéréotype « aspect ». Cette classe est responsable d'apporter les changements à chacune des classes auxquelles elle est reliée, elle définit entre autres les points de coupure et les advices des différents aspects. Notons cependant, que les auteurs restent vagues et imprécis quant à la modélisation donnée aux points de coupure, aux advices et à la relation d'entrecoupement.

En outre, la proposition de Fuentes *et al.* [FS06] (décrite à la page 2.4.2) permet aussi de considérer un aspect comme un stéréotype de composant UML.

2.4.5 Discussion

De manière générale l'analyse des travaux évalués dans le tableau 2.2 montre que :

- Toutes les propositions de modélisation par aspects que nous avons recensées s'appuient sur UML, le langage standard de modélisation par objets, plus précisément par la définition de nouveaux stéréotypes.
- La plupart des propositions sont générales et indépendantes d'un langage de programmation particulier, à l'exception de celles de Suzuki *et al.*, Stein

	Portée	**Modèle support**	**Exhaustivité**	**Vue**	Extension
Jacobson [JN05]	analyse	général	complète	cas d'utilisation	cas d'utilisation
Suzuki *et al.* [SY99]	conception	spécifique (AspectJ)	partielle	structurelle	classe
Stein *et al.* [SHU02]	conception	spécifique (AspectJ)	complète	structurelle/ comportementale	classe
Aldawud *et al.* [AEB03]	conception	général	complète	structurelle	classe
Fuentes *et al.* [FS06]	conception	général	partielle	structurelle	classe/ composant
Bash *et al.* [BS03]	conception	spécifique (AspectJ)	partielle	structurelle/ comportementale	package
Clarke [CB05]	analyse/ conception	général	partielle	structurelle/ comportementale	package
Barra *et al.* [BGL04]	conception	général	complète	structurelle	composant

TAB. 2.2 – Caractéristiques des principales approches de modélisation par aspects.

et al. et Bash *et al.* qui sont étroitement liées au langage AspectJ. Nous pensons, en fait, qu'il convient de définir un langage de modélisation général, permettant de spécifier des aspects à un niveau d'abstraction plus élevé, de manière indépendante des différents langages d'implémentation.

- La plupart des propositions de modélisation par aspects sont dédiées à la phase de conception, sauf celle de Jacobson qui couvre exclusivement la phase d'analyse et celle de Clarke qui couvre les deux phases d'analyse et de conception. De manière générale, ces propositions (à l'exception de Stein *et al.*, Bash *et al.* et Clarke) se concentrent sur la représentation des structures des systèmes, mais très rarement sur la représentation de leurs comportements.
- Souvent, les concepts fondamentaux des modèles supports ne sont pas complètement spécifiés. En effet, certains de ces concepts (tels que point de coupure et advice) qui forment les briques de base de toute modélisation par aspects sont omis dans certaines propositions (Suzuki *et al.*, Fuentes *et al.*, Bash *et al.* et Clarke). Par ailleurs, les éléments de modélisation nouvellement introduits pour spécifier de tels concepts sont parfois inappropriés : relation de réalisation pour la définition de la relation d'entrecoupement (Suzuki *et al.*), opération pour la spécification d'un point de coupure (Stein *et al.* et Aldawud *et al.*), etc. ou encore imprécis : la relation « crosscut » qui modélise l'entrecoupement (Stein *et al.*). Nous sommes convaincus qu'un bon langage de modélisation doit offrir une sémantique claire et précise de ces concepts fondamentaux.

Sur la base des remarques évoquées plus haut, notre motivation principale, dans le cadre de cette thèse, est de proposer une approche de modélisation par aspects qui (1) couvre les deux phases d'analyse et de conception, tout en se concentrant sur la représentation des structures des systèmes et sur la représentation de leurs comportements, (2) est générale c'est-à-dire indépendante d'un langage de programmation par aspects en particulier, (3) est complète dans le sens où elle couvre

les principaux concepts du paradigme aspect, tout en rapprochant leur modélisation de leur définition commune. Par ailleurs, notre approche devra aussi disposer de moyens permettant, dans un contexte plus large, l'expression de la composition des aspects et ce afin de nous concentrer plus particulièrement, dans notre travail, sur la vérification de cette composition dans les systèmes par aspects.

2.5 Composition des modèles orientés aspect

Comme le souligne le rapport [CRS+05] de la communauté de développement par aspects (AOSD) consacré aux approches de modélisation pour le paradigme aspect, très peu d'approches offrent un support pour la composition des aspects dans les modèles de conception. Ce support consiste, entre autres, à définir des directives prescriptives pour la composition d'un modèle aspect dans un modèle de base, et ce afin d'aboutir à un modèle final tissé. Dans les quelques travaux proposés en littérature, le tissage d'un aspect dans un modèle de base est vu comme une transformation du modèle de base considéré.

L'approche Theme proposée par Clarke [CB05] (brièvement décrite à la page 2.4.3) introduit la notion de relation de composition pour les themes. Cette notion spécifie clairement comment les themes doivent être composés. Ces relations de composition permettent d'identifier les éléments de modèle se chevauchant, de spécifier comment les modèles doivent être composés (avec l'opération de fusion ou avec l'opération d'annulation) et de définir des priorités entre les themes aspects.

Les travaux de Straw *et al.* [RGF+06] s'intéressent aussi à la composition des modèles de conception. Les deux modèles aspect et primaire à composer sont exprimés sous forme de diagramme de classes UML. Lors de la composition, des conflits et incohérence peuvent apparaître dans le modèle tissé. Pour prévenir ces problèmes, les auteurs se basent sur des directives de composition permettant, entre autres, l'ajout et la suppression d'éléments, le remplacement d'un élément par un

autre et la définition d'un ordre de composition des aspects.

Dans le même esprit que les deux précédentes, le travail de Fuentes [FS07] définit un processus pour la composition de modèles UML orientés aspect. Ce processus est défini comme une chaine de transformations de modèle où les conflits entre aspects qui s'exécutent à un même point de jointure sont résolus en attribuant des priorités aux aspects.

Nous admettons que ces propositions sont intéressantes car, d'une part, elles offrent un modèle de conception pour le système final tissé et d'autre part elles définissent des directives prescriptives de composition permettant de prévenir d'une certaine façon l'incohérence et les conflits dans les systèmes tissés. De telles approches pour la composition de modèles par aspects présentent, cependant, l'inconvénient de laisser à la charge du concepteur l'identification et la résolution des incohérences et des conflits. Nous pensons, en effet, que la définition de ces directives de composition ne suffit pas pour assurer la cohérence effective du système final. Nous soutenons plus une approche par vérification formelle. Pour ce faire, nous proposons dans le cadre de notre travail, d'expliciter la composition de nos modèles de conception dans un modèle tissé formel. Pour cela, nous proposons de traduire nos modèles de conception (de base et aspect) vers des modèles formels équivalents et ensuite de tisser les modèles formels obtenus pour aboutir à un modèle tissé formel. C'est sur ce modèle final obtenu que seront vérifiées formellement la composition des aspects et la cohérence du système.

2.6 Vérification des interactions

Dans le paradigme aspect, le tissage des aspects dans le système de base doit préserver les propriétés du système initial et garantir la correction du système final (ou composé), c'est-à-dire assurer une composition des aspects sans conflits qui évite des situations imprévues, où le comportement prévu du système est corrompu.

Ainsi, pour assurer une composition correcte des aspects dans les systèmes tissés, certaines approches utilisent les techniques d'analyse et de vérification formelle pour vérifier les interactions dues aux aspects. Ce sujet est relativement récent et par conséquent la littérature en parle peu. Cette section recense néanmoins les quelques solutions apportées pour parvenir à cette vérification. La synthèse de ces solutions que nous résumons dans le tableau 2.3, page 50, est basée essentiellement sur les caractéristiques suivantes :

- **Abstraction.** Pour vérifier un système, il convient le plus souvent de le décrire par une spécification. Dans une approche d'abstraction dite backward, cette spécification est obtenue à partir du code, alors que dans une approche d'abstraction dite forward, la spécification est obtenue à partir d'un modèle conceptuel. Une solution peut s'intéresser alors à vérifier une spécification obtenue dans un processus forward ou dans un processus backward.
- **Technique**. La technique d'une solution détermine la technique utilisée pour parvenir à la vérification. Nous distinguons : l'analyse des traces, le model-checking, la preuve de théorèmes, etc.
- **Couverture**. Une solution peut considérer la vérification du système complet comme elle peut se concentrer exclusivement sur la vérification de certaines parties du système. Nous distinguons alors des solutions de couverture totale et des solutions de couverture partielle.
- **Type**. Le type détermine le type de vérification considéré par la solution. Si la solution consiste à analyser des propriétés syntaxiques alors le type de la vérification est syntaxique, sinon si la vérification porte sur les propriétés sémantiques du système alors le type est dit sémantique. Notons que les solutions qui se concentrent sur la vérification du code sont celles qui proposent des vérifications syntaxiques car, en fait, l'information sémantique est moins présente dans le code. Inversement, les solutions qui portent sur la vérification des modèles se focalisent plus sur des vérifications sémantiques.

- **Support**. Certaines solutions offrent des outils et/ou de la documentation pour atteindre les objectifs de vérification. Dans ce cas, la solution offre un support pour la vérification.

Pour présenter les solutions nous proposons de les regrouper selon la technique de vérification utilisée.

2.6.1 Analyse statique des traces

Dans [SKB03], Storzer *et al.* s'intéressent à analyser l'impact d'un aspect sur le programme de base. Pour ce faire, les auteurs proposent une analyse statique des traces décrivant les changements internes du programme. L'approche est basée sur la comparaison de deux traces pour un seul test, avec des entrées identiques. La première trace est générée par le système de base (sans les aspects), alors que l'autre trace est générée par le programme composé (programme de base et aspects). La solution consiste à évaluer l'ensemble D des différences entre les deux traces. En analysant simplement cet ensemble D le programmeur peut aisément détecter les parties du système altérées par les aspects. L'analyse de différences des traces montre, en fait, quel impact a l'introduction d'un aspect dans un programme de base donné.

Toujours en adoptant la technique d'analyse des traces, Sereni et de Moor [SdM03] proposent une solution pour identifier les aspects qui n'altèrent pas le programme de base. Cependant (à la différence de la proposition de Strozer *et al.*) dans ce cas-ci, les traces du programme à analyser sont générées à partir du graphe d'appels des méthodes (présentes dans le code du programme) et de l'expression des points de coupure qui décrivent les points de jointure.

2.6.2 Analyse statique des pointeurs

Rinard *et al.* [RSB04] proposent une solution pour la vérification syntaxique des interactions entre les aspects et le système de base. Rappelons, que dans ce

même travail, les auteurs présentent aussi, une classification des interactions entre un advice et une méthode (cf. section 2.3.3, page 27). Pour vérifier tous les types d'interactions (lecture/lecture, lecture/écriture, écriture/écriture) catalogués dans leur système de classification, les auteurs proposent d'utiliser l'analyse statique des pointeurs. Cette analyse permet d'extraire une spécification des champs que l'exécution de chaque advice ou méthode peut accéder.

2.6.3 Analyse statique par découpage de programmes

La proposition de Balzarotti *et al.* [BCM05] vise à appliquer les techniques de découpage de programmes ou *program slicing* pour identifier les interactions entre les aspects et déterminer précisément la partie du programme concernée par ces interactions. Les auteurs implémentent un système de découpage des programmes [1] écrit en AspectJ, où l'identification des interactions antre aspects consiste à vérifier si les noeuds d'une coupe d'un aspect apparaissent dans la coupe d'un autre aspect, autrement dit, il y a absence d'interactions si les coupes sont complètement disjointes.

2.6.4 Preuve inductive

Très peu de travaux ont considéré les preuves inductives pour la vérification des systèmes par aspects. L'idée de base a été décrite initialement par Devreux dans [Dev03] et Sipma dans [Sip03]. Elle consiste à vérifier la correction d'un aspect en utilisant la spécification dite *par contrat*. Dans une telle spécification, un couple (*assume*, *guarantee*) est associé à chaque aspect pour spécifier, d'une part, les hypothèses (assume) sur le contexte (point de jointure) et d'autre part, les propriétés qui doivent être satisfaites (guarantee) par l'aspect. La guarantee d'un aspect est alors correcte si elle est prouvée à partir de l'hypothèse du point de join-

[1] Plus précisément, le découpage est réalisé sur le code exécutable des programmes tissés.

ture où il est appliqué et du code de son advice en utilisant un système de preuves inductives tel que PVS [ORS92].

Une autre proposition pour la vérification des systèmes par aspects qui utilise la spécification *par contrats* (ou *assume-guarantee*) est celle de Goldman et Katz [GK06, GK07]. Cependant, à la différence de la précédente qui s'intéresse particulièrement à la vérification de la spécification d'un aspect, celle-ci suggère une vérification modulaire et générique des systèmes orientés aspect. Le prototype proposé permet de vérifier que pour n'importe quelle machine à états décrivant un système de base et satisfaisant les hypothèses d'un aspect donné, la machine à états tissée satisfait les propriétés désirées. Une seule machine à états générique est construite à partir des hypothèses de l'aspect (*c-à-d* le descripteur du point de coupure et la machine à états de l'advice [2]) et est vérifiée pour la propriété désirée. Ainsi, quand l'aspect devra être tissé dans un système de base particulier, il suffit d'établir que la machine à états de base satisfait les hypothèses de l'aspect. Cette approche vise surtout à garantir que le tissage des aspects génériques à travers différents programmes de base est correct. D'ailleurs, elle ne s'attaque ni au cas du tissage de plusieurs aspects à un même point de jointure, ni à la détection des éventuels conflits qui peuvent en découler.

2.6.5 Model-checking

D'autres tentatives pour la vérification formelle des systèmes par aspects exploitent les techniques de model-checking. Nous pouvons citer par exemple, le travail de Denaro et Monga [DM01] dont l'idée est de vérifier les propriétés des aspects indépendamment de leurs environnements d'exécution, comme par exemple, vérifier l'absence de blocage pour un aspect de synchronisation. L'approche proposée prend en entrée le code de l'aspect à vérifier ; celui-ci est analysé, afin de dériver un

[2] L'approche considère seulement les aspects qui contiennent un seul advice.

modèle PROMELA. D'autre part, la propriété à vérifier est exprimée en formule temporelle LTL. Ensuite, le model-checker SPIN [Hol97] permet de vérifier si la formule est bien satisfaite par le modèle.

Nelson *et al.* [NCA01] présentent une approche utilisant le model-checking pour vérifier les propriétés des systèmes composés de plusieurs préoccupations transverses. Les préoccupations sont modélisées comme des processus concurrents spécifiés par des systèmes de transitions étiquetées. Les préoccupations peuvent être composées au moyen des opérateurs *merge* et *override*. Les propriétés du système composé sont vérifiées en utilisant le model-checker LTSA [MK99]. Un autre travail traitant du model-checking des systèmes orientés aspect est proposé par Katz [Kat04]. L'auteur s'intéresse, plus particulièrement, à vérifier la correction des aspects génériques, une fois tissés dans un système de base particulier. Le model-checker prototype proposé permet de vérifier que les nouvelles propriétés décrites par les aspects génériques sont vraies dans le système tissé, et que les propriétés désirées du système de base sont maintenues.

Nous distinguons, par ailleurs, une autre proposition de vérification des systèmes orientés aspect par model-checking, il s'agit de celle présentée par Krishnamurthi *et al.* [KFG04]. À la différence des deux travaux discutés précédemment qui proposent une vérification totale du système entier tissé, celle-ci préconise une vérification modulaire. Elle propose, en effet, de vérifier que les invariants d'états valides dans le système de base initial sont aussi vrais dans les aspects ajoutés au système. Pour ce faire, les auteurs proposent un prototype qui génère d'abord un automate à états finis à partir du code source d'un aspect pour ensuite le vérifier par model-checking. Les propriétés vérifiées se limitant aux invariants d'états, il n'est pas nécessaire de vérifier tout le système après le tissage de l'aspect, si on assure que le système de base vérifiait initialement les invariants en question. Ceci est vrai pour autant que l'aspect tissé soit sans effet de bord (comme par exemple désactivation d'une fonctionnalité). En fait, l'approche telle que présentée

ne spécifie pas comment les ressources (variables) sont partagées lors du tissage du système d'états décrivant l'aspect au système d'états décrivant le système de base. Autrement dit, la sémantique du tissage n'est pas explicitement définie dans ce travail.

2.6.6 Discussion

De manière générale l'analyse des travaux évalués dans le tableau 2.3 montre que :

- Toutes les propositions que nous avons recensées pour la vérification des systèmes par aspects s'inscrivent dans l'approche d'abstraction dite backward. En effet, ces propositions se concentrent sur la vérification des spécifications générées à partir du code des systèmes et non à partir des modèles de conception.
- Certaines de ces propositions (tels que celles de Storzer *et al.*, Sereni *et al.*, Rinard *et al.* et Balzarotti) suggèrent une vérification de type syntaxique. Ceci tient principalement du fait qu'elles sont axées sur l'analyse syntaxique du code ou encore sur l'analyse des traces des programmes orientés aspect. Ce genre d'analyse sert, en fait, à s'assurer de l'absence d'interférence dans les systèmes par aspects, en vérifiant que tous les aspects sont des observeurs (qui ne modifient pas le système de base). Cette analyse n'est cependant pas concluante dans le cas où les aspects ne sont pas des observeurs (aspects qui modifient le système de base). En effet, l'analyse syntaxique peut seulement révéler que l'aspect modifie effectivement le système de base, sans pour autant préciser si la modification est désirable ou non.
- Les propositions qui permettent de révéler les incohérences sémantiques dues aux interactions entre aspects sont celles utilisant la vérification formelle (comme le model-checking ou la preuve inductive). De manière générale, ces propositions se concentrent sur la vérification des invariants d'états du sys-

Approche	Abstraction	Technique	Couverture	Type	Support
Storzer *et al.* [SK]	backward	analyse de traces	partielle	syntaxique	outil
Sereni *et al.* [SdM03]	backward	analyse de traces	partielle	syntaxique	-
Rinard *et al.* [RSB04]	backward	analyse de pointeurs	complète	syntaxique	outil
Balzarotti *et al.* [BCM05]	backward	découpage de programme	partielle	syntaxique	outil
Devreux [Dev03]	backward	preuve inductive	partielle	sémantique	-
Goldman *et al.* [GK07]	backward	preuve inductive	partielle	sémantique	outil
Denaro *and al.* [DM01]	backward	model-checking	partielle	sémantique	-
Nelson *and al.* [NCA01]	backward	model-checking	complète	sémantique	outil
Katz [Kat04]	backward	model-checking	complète	sémantique	outil
Krishnamurthi *et al.* [KFG04]	backward	model-checking	partielle	sémantique	outil

TAB. 2.3 – Classification des méthodes de vérification des systèmes par aspects.

tème augmenté par les aspects. Seules les propositions de Denaro *et al.* et Devreux, font exception en s'intéressant à vérifier les propriétés d'un aspect donné.

- Par ailleurs, certaines solutions (comme Nelson *et al.* et Katz) utilisant la vérification formelle sont de couverture totale. Nous admettons que ces propositions sont intéressantes, car elles permettent de détecter le plus de conflits et d'incohérence dues aux interactions entre aspects. Elles présentent toutefois l'inconvénient d'être rapidement confrontées au problème d'explosion d'états puisque le système tissé est traité en entier en une seule fois.
- En revanche, les solutions de couverture partielle utilisant la vérification formelle des invariants d'états des systèmes tissés (telles que Krishnamurthi *et al.* et Goldman *et al.*) proposent une vérification modulaire pour répondre au problème d'explosion d'états. Si la modularité de ces approches est intéressante, elles présentent toutefois quelques inconvénients. En effet, la solution proposée par krishnamurthi *et al.* ne traite pas le cas des aspects susceptibles de modifier le système de base. D'autre part, la solution proposée par Goldman *et al.* ne s'attaque pas au cas des aspects conflictuels leur approche ne considérant qu'un seul aspect par point de jointure.

Notre objectif principal, dans le cadre de cette thèse, est de pouvoir vérifier les interactions dues aux aspects et de détecter les incohérences qui en résultent. Nous voulons, en particulier, vérifier ces interactions au niveau des modèles de conception des systèmes par d'aspects (réalisés plus précisément avec notre profil de modélisation orienté aspect). Ceci nous permettra de révéler les incohérences du systèmes à un haut niveau d'abstraction, bien avant l'implémentation. Nous proposons, pour ce faire, une approche de vérification pour les systèmes par aspects qui (1) s'inscrit dans un processus d'abstraction dit forward, (2) utilise la vérification formelle, (3) est de couverture partielle, c'est-à-dire se concentre sur les éléments du système reliés à la composition des aspects, (4) révèle les incohérences sémantiques du

système tissé dues aux interactions entre aspects, (5) ne se limite pas aux aspects observeurs, c'est-à-dire qu'elle traite même les aspects qui modifient le système de base.

2.7 Conclusion

La plupart des enjeux et problèmes liés à l'analyse et la conception par aspects (telles la composition des aspects et la vérification des interactions dues aux aspects) demeure des questions ouvertes. La diversité des propositions et des travaux effectués dans ce cadre contribue à élargir le champ des investigations et des possibilités dans l'élaboration de nouveaux modèles et de nouvelles techniques de vérification pour les systèmes par aspects. Dans ce chapitre, nous avons, en effet, présenté un éventail d'approches de modélisation par aspects et de méthodes de vérification pour les systèmes par aspects, en mettant en avant leur utilité et montrant les problèmes et limites de leur utilisation.

CHAPITRE 3

LE PROFIL ASPECT-UML

La modélisation en génie logiciel est une phase importante et nécessaire dans le processus de développement pour aller du problème vers sa réalisation. En effet, la modélisation permet de décrire une spécification semi-formelle (ou formelle) du problème à résoudre. Par contre, la programmation en est sa réalisation physique. Ces deux phases de développement : modélisation et programmation sont évidemment intimement liées : l'aspect programmation découle directement de la modélisation. La modélisation correspond aux questions quoi ? et comment ? Il faut une méthodologie et une démarche rigoureuse pour répondre à ces questions. La rigueur induit le choix d'une notation claire qui ne doit pas prêter à confusion.

Dans ce chapitre, nous présentons notre approche de modélisation pour le paradigme aspect. À l'instar de beaucoup d'autres propositions, notre approche de modélisation s'appuie sur le langage unifié de modélisation par objets UML [Obj05]. En effet, notre proposition pour la modélisation orientée aspect utilise les extensions d'UML et définit un profil UML appelé Aspect-UML qui permet de prendre en charge les phases de développement d'un système orienté aspect. Avant de donner les détails de notre proposition, nous commencerons par présenter, d'abord, brièvement UML, ainsi que ses mécanismes d'extension. Ensuite, nous consacrerons les sections suivantes à la description de notre profil de modélisation Aspect-UML ; nous présenterons la vue de cas d'utilisation, la vue structurelle et la vue comportementale. Suivra ensuite une discussion où nous comparerons notre proposition aux différentes approches de modélisation recensées dans la littérature et présentées à la section 2.4 du chapitre 2. Nous terminerons ce chapitre par une conclusion.

3.1 Le langage de modélisation UML

UML (Unified Modeling Language) [Obj05, JRB99] est un langage graphique de modélisation des données et des traitements; il est né de la fusion des trois méthodes qui ont le plus influencé la modélisation objet : OMT (Object Modeling Technique) [RBE+91], Booch [Boo91] et OOSE (Object-Oriented Software Engineering) [JCJd92]. Depuis sa normalisation par l'OMG (Object Management Group), UML est devenu le standard de modélisation objet. C'est un langage général et intuitif pouvant être facilement compris, dont l'expressivité permet de décrire des systèmes à la fois très complexes et très gros. C'est une notation textuelle et graphique pour la modélisation qui permet de décrire un système selon différentes vues complémentaires. UML est composé de 13 types de diagramme qui sont hiérarchiquement dépendants et qui se complètent. Combinés, les différents types de diagramme de UML offrent des vues complémentaires des aspects statiques et dynamiques du système. UML n'étant pas une méthode, l'utilisation de ces diagrammes est laissée à l'appréciation de l'utilisateur, cependant le diagramme de classes est généralement considéré comme l'élément central d'UML.

Dans notre approche de modélisation, nous nous somme intéressées à deux types de diagrammes : le diagramme de cas d'utilisation et le diagramme de classes. Nous les décrivons brièvement dans ce qui suit.

3.1.1 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation est utilisé pour donner une vision globale du comportement fonctionnel du système. Ce diagramme définit les acteurs et les cas d'utilisation d'un système ainsi que les liens et les dépendances qui existent entre eux.

Un acteur est une entité externe au système qui interagit avec lui et qui bénéficie d'un de ses services.

Un cas d'utilisation est un service offert par le système, et dont l'effet profite à un acteur. Les cas d'utilisation décrivent des fonctionnalités qui peuvent être indépendantes ou non. Chaque cas d'utilisation fournit un ou plusieurs scénario(s) qui expriment comment le système doit interagir avec les acteurs afin de réaliser un objectif spécifique. En plus, du diagramme de cas d'utilisation qui fournit une vue de haut niveau de tous les cas d'utilisation, plusieurs méthodes de développement (basées sur les cas d'utilisation) définissent un template textuel que les développeurs doivent documenter quand ils élaborent un cas d'utilisation, afin de définir son (ses) scénario(s).

Par ailleurs, dans un diagramme de cas d'utilisation il est possible de définir des relations entre les cas d'utilisation. L'inclusion et l'extension sont deux types de relation de dépendance que l'on peut spécifier entre les cas. Une relation d'inclusion entre un cas d'utilisation A et un cas d'utilisation B signifie que le scénario spécifié par B est inséré dans le scénario spécifié par A, comme un fragment de ce dernier. L'exécution du scénario de B peut avoir un effet sur l'exécution du scénario de A. On parle de *cas de base* (cas A) et de *cas d'inclusion* (cas B). Par contre, la relation d'extension entre un cas d'utilisation A et un cas d'utilisation B tel que B étend A signifie que le service décrit par B est un scénario alternatif du service de A. Ce mécanisme permet de regrouper les flots d'événements exceptionnels hors de la séquence principale d'événements pour gagner en clarté. On commence avec un cas d'utilisation de base, et on l'étend au besoin par d'autres fonctionnalités sans avoir à changer le cas de base. On parle de *cas d'extension* (cas B) et de *cas de base* (cas A). Les relations entre les cas d'utilisation sont notées comme des flèches de dépendance avec les mots-clés « include » pour l'inclusion et « extend » pour l'extension.

3.1.2 Diagramme de classes

Le diagramme de classes offre une vue statique du système. Il permet de représenter les classes, ainsi que les différentes relations entre celles-ci. Une classe est définie par un nom, des attributs et des méthodes. UML permet de définir différents types de relations entre les classes :

- **Association** Une association est une relation entre deux classes (association binaire) ou plus (association n-aire) qui décrit les liens sémantiques entre leurs instances. En autres, les instances peuvent utiliser ces liens pour interagir. On peut documenter une association en lui attribuant un nom, des noms de rôle pour les instances participantes, des multiplicités et des informations concernant sa navigabilité.
- **Généralisation/Spécialisation** La généralisation décrit une relation entre une classe (super-classe) et une classe spécialisée (sous-classe). Cette relation permet à la sous-classe d'hériter et/ou redéfinir des attributs et des méthodes de la super-classe. Dans certains langages comme Java, la généralisation/spécialisation est réalisée par le mécanisme d'héritage.
- **Dépendance** Une dépendance est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle. Elle indique que la modification de la cible a un impact sur la source.
- **Réalisation** La réalisation est une relation entre une spécification et son implémentation. Par exemple, une interface est une spécification qui peut être réalisée par une (ou plusieurs) classe(s) ; chaque classe doit alors implémenter les opérations de l'interface.

Une note est un commentaire placé sur un diagramme. Elle est rattachée généralement à un élément du diagramme plutôt qu'à un diagramme. Les notes peuvent être nécessaires pour préciser les intentions du concepteur, ou encore pour introduire des commentaires, des observations ou des explications à propos d'un élément

particulier du diagramme. Le cas échéant, une même note peut être attachée à plusieurs éléments du diagramme.

3.1.3 Mécanismes d'extension UML

A priori, UML est un langage universel destiné à la modélisation de tout type de systèmes orientés objet. Cependant certains domaines spécifiques sont caractérisés par des concepts qui nécessitent l'ajout de nouveaux types d'éléments de modélisation supplémentaires. Afin de satisfaire et d'adapter UML aux besoins spécifiques de ces systèmes, l'OMG a introduit depuis la version UML 1.3 des mécanismes dits d'extension. Ces mécanismes sont les contraintes, les valeurs étiquetées (*tagged values*) et les stéréotypes. Les contraintes et les valeurs étiquetées servent à enrichir les propriétés des éléments de modélisation ; par contre les stéréotypes enrichissent le méta-modèle d'UML par l'ajout de nouveaux éléments de modélisation.

- Une contrainte est une restriction sémantique représentée par une expression textuelle et rattachée à un élément du modèle (ou une liste d'éléments). Elle peut être décrite dans une notation mathématique, un langage de contraintes comme OCL [Obj04], un langage de programmation ou encore un langage informel (naturel).
- Une valeur étiquetée (ou *tagged value*) est une paire, une étiquette (*tag*) et une valeur (*value*), qui enregistre une information à propos d'un élément. Les valeurs étiquetées sont utilisées pour sauvegarder des informations arbitraires sur les éléments du modèle. Elles sont particulièrement utiles pour sauvegarder les informations relatives à la gestion de projet, telles que : date de création d'un élément, état d'un élément, etc.
- Un stéréotype est un nouvel élément du modèle, défini à partir d'un élément existant. Il est nécessairement rattaché à une classe de base dans le méta-modèle. Les informations contenues dans le stéréotype sont les mêmes que celles de l'élément de base, mais sa sémantique et son usage sont différents.

Un stéréotype peut aussi utiliser les valeurs étiquetées afin de sauvegarder des propriétés additionnelles qui ne sont pas supportées par l'élément de base. Quand un stéréotype est appliqué à un élément du modèle (instance de la méta-classe sur laquelle le stéréotype est défini), cet élément sera annoté par « nom du stéréotype ».

3.1.4 Profil UML

La notion de profil UML est apparue dans le standard UML 1.3, comme un moyen permettant de structurer et regrouper les extensions apportées à UML pour les besoins de modélisation dans un domaine spécifique. Un profil UML est donc une spécialisation du méta-modèle UML pour un domaine d'utilisation particulier. Plusieurs profils ont déjà été standardisés par l'OMG, tels que le profil UML pour les systèmes temps réel [Obj02c], le profil UML pour les applications d'entreprise distribuées (EDOC) [Obj02b] et aussi des profils UML pour les plateformes EJB [Rat01] et CORBA [Obj02a]. Les profils UML peuvent hériter d'autres profils, être dépendants, ou encore être regroupés. En plus de ces profils standardisés, les utilisateurs peuvent aussi définir leurs propres profils. Un modèle UML est construit sous un profil particulier, c'est-à-dire qu'en plus des éléments standard de UML, il fait appel à des contraintes, valeurs étiquetées et/ou stéréotypes qui apportent une sémantique spécifique à certains éléments de modélisation.

Dans le cadre de notre travail, nous avons défini un profil Aspect-UML [VM04a, MV06a] qui regroupe un certain nombre d'extensions que nous avons introduites afin de considérer les spécificités du paradigme aspect. Nous consacrons la suite de ce chapitre à la présentation de ce profil.

3.2 Étude de cas

L'exemple d'application que nous utilisons pour illustrer notre méthodologie de développement avec le profil Aspect-UML, est une implémentation orientée aspect d'une application de téléphonie à base de services (voir section 1.4 du chapitre 1). Dans cette application, le système de base permet de gérer les connexions téléphoniques entre les abonnés. Le réseau téléphonique consiste en des appareils téléphoniques (ou *devices*), des clients à la fois propriétaires des téléphones et usagers du réseau et un contrôleur de réseau qui permet de gérer les connexions entre les clients (figure 1.6, 16). Les usagers du système peuvent initier et rompre des communications téléphoniques via leurs téléphones. La modélisation du système de base doit donc voir à représenter les clients, les devices et les connexions.

À ce système de base, on peut ajouter un grand nombre de fonctionnalités additionnelles afin d'étendre la gamme de services téléphoniques offerts aux usagers. Rappelons qu'un service téléphonique est un incrément de fonctionnalité qui vient s'ajouter à la description de base du système. Ainsi un système organisé en services sera composé d'une description de base et de différentes fonctionnalités F_1, F_2, ... où chaque F_i décrit le service S_i. Comme nous l'avons souligné et expliqué à la section 1.4, les services téléphoniques peuvent tout à fait être implémentés au moyen d'aspects. En effet, les concepts de *service* et d'*aspect* sont liés dans le sens où ils servent tous les deux à ajouter des fonctionnalités à un système de base. La programmation orientée aspect se prête bien à l'implémentation des systèmes de téléphonie conçus à base de services.

Pour les besoins de ce chapitre, nous limitons l'étude de cas à l'introduction des services pour le calcul de la durée des communications et de leur facturation aux clients (la modélisation de l'étude de cas complète sera décrite au chapitre 6). La fonctionnalité de calcul de la durée d'une communication intervient au début et à la fin de celle-ci, elle permet de calculer et de sauvegarder le temps de connexion pour

chaque communication. Par contre, la fonctionnalité de facturation est réalisée à la fin de la communication. La facture du client qui a initié la connexion (l'appelant) est alors mise à jour.

3.3 Le profil Aspect-UML

L'idée, fortement présente dans la communauté de développement par aspects et répandue par la plupart des articles de recherche, est de concevoir un langage de modélisation basé sur UML, pour le développement par aspects. Cette idée est très naturelle, du fait que la technique de programmation orientée aspect soit une extension, voire une amélioration, de la programmation orientée objet. Ces deux paradigmes sont appelés à coexister.

Notre proposition Aspect-UML [VM04a, MV06a], pour la modélisation orientée aspect, est une extension d'UML. En fait, nous avons conçu Aspect-UML comme un profil UML, dont les stéréotypes décrivent les principaux concepts des langages par aspects (les aspects, les advices, les points de coupure et les points de jointure) et les contraintes décrivent la sémantique relative à la composition des aspects. L'approche de modélisation, que nous proposons, vise à prendre en charge le concept de séparation des préoccupations de la phase de la spécification des besoins jusqu'à la conception. Ainsi, dans le cadre de notre travail, nous nous intéressons à élaborer la vue de cas d'utilisation, la vue structurelle et la vue comportementale.

La vue de cas d'utilisation donne une vue globale des fonctionnalités du système en incluant les aspects. Quant à la vue structurelle, elle est composée d'un diagramme de classes du système représentant les entités du domaine, les contrôleurs, les interfaces ainsi que les aspects. Ce diagramme nous permet de voir explicitement la structure transverse des aspects et leurs dépendances. Par ailleurs, la vue comportementale permet de représenter le contexte d'exécution des points de jointure et des advices, en y précisant les états des objets qui interagissent. Cette vue

dépeint une superposition des annotations sémantiques sur le diagramme de classes où à chaque opération (point de jointure et advice) est associée une spécification comportementale, sous forme de pré et de postconditions. Nous reviendrons plus en détail sur ces annotations, lorsque nous aborderons la vue comportementale.

Dans la suite de cette section, nous présentons notre approche de modélisation par aspects, nous détaillons le contenu de chacune de ses vues en les expliquant sur l'étude de cas.

3.3.1 Vue de cas d'utilisation

Selon AORE (Aspect Oriented Requirements Engineering) [ABC+05], un aspect au niveau de la phase de spécification des besoins est une *propriété à large spectre* (*broadly-scoped property* en anglais), représentée par un seul besoin ou un ensemble cohérent de besoins (comme par exemple le besoin de sécurité qui englobe les besoins d'authentification et d'autorisation), qui affecte plusieurs autres besoins dans le système. La phase de la spécification des besoins, doit donc fournir des moyens systématiques pour l'identification, la modularité, la représentation et la composition de ces exigences transversales, qu'elles soient fonctionnelles ou non fonctionnelles.

Dans notre approche, nous proposons d'adapter la vue de cas d'utilisation pour prendre en charge la modularité, la représentation et la composition des propriétés transversales que sont les aspects.

Dans UML 2.0, la vue de cas d'utilisation, réservée à la description des exigences fonctionnelles essentielles, n'est pas destinée à saisir les besoins non fonctionnels, tels que, les contraintes de qualité, les contraintes de conception et les contraintes d'utilisation. En général, ces besoins sont documentés séparément et regroupés dans une spécification sous le nom d'exigences supplémentaires [MB01] (*supplementary specifications*).

Notre première idée présentée dans [VM04a] est de traiter les besoins supplémen-

taires comme des citoyens de première classe, en les considérant comme des cas d'utilisation spéciaux, les élevant ainsi au haut niveau des cas d'utilisation. Ceci, amène les développeurs à les prendre en charge précocement, de manière plus disciplinée et plus concise. Dans [VM04a], nous avons présenté les aspects comme un moyen pour saisir ces besoins particuliers, et nous avons montré comment les modéliser et les inclure dans le diagramme de cas d'utilisation. Nous avons ensuite développé cette idée dans [MV05, MV06a] et proposé une méthodologie de développement par aspects où les aspects sont considérés dès la phase de spécification des besoins et sont pris en charge tout au long du cycle de développement.

Ainsi, nous proposons de représenter les aspects comme des cas d'utilisation spéciaux et leur composition par des liens de dépendance de type « extension ». Un aspect est donc modélisé comme un type particulier de cas d'utilisation d'extension, annoté du stéréotype « `Aspect` ». Les points de coupure sont assimilés à des points d'extension spéciaux auquels on attribue le stéréotype « `pointcut` ». Un point d'extension étant ainsi défini comme un élément de *RedefinableElement* du métamodèle UML [Obj05], un point de coupure représente donc les locations où les aspects peuvent être tissés. Pour modéliser le comportement transverse des aspects (cas d'utilisation d'extension) et pour montrer leur insertion dans les cas d'utilisation de base, la relation stéréotypée « `crosscuts` », élément de *DirectedRelationship* du métamodèle UML [Obj05], est utilisée. Par conséquent, les cas d'utilisation de base sont explicitement étendus de façon modulaire aux points indiqués.

Remarquons, cependant, que l'introduction au niveau des diagrammes de cas d'utilisation des stéréotypes « `Aspect` », « `pointcut` » et « `crosscuts` » n'ajoute rien à la sémantique de base de UML. Ce sont simplement des éléments syntaxiques qui nous permettent de mettre en évidence les fonctionnalités transverses dans ce type de diagrammes.

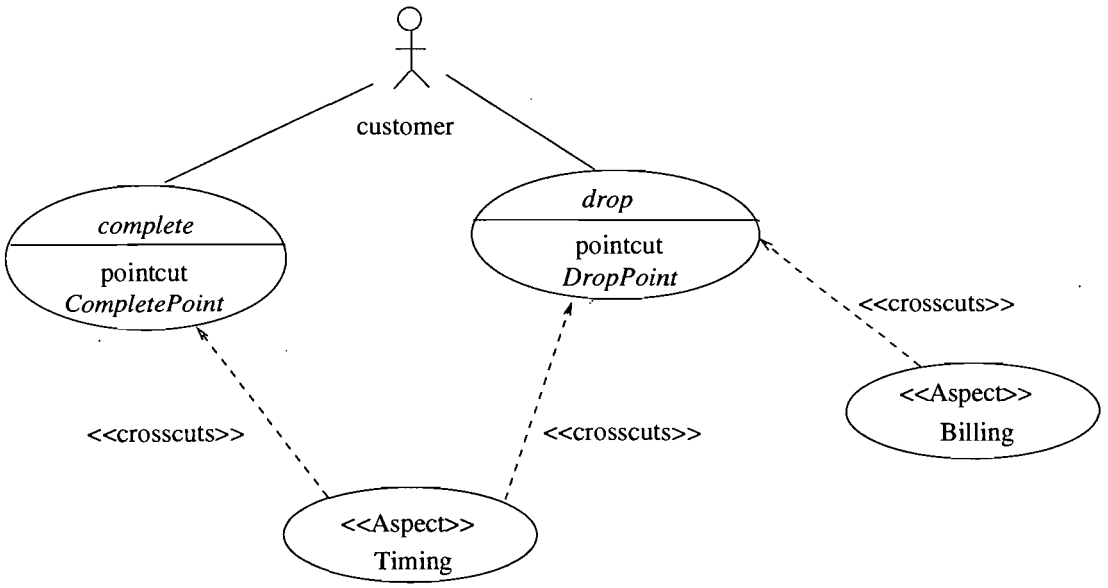


FIG. 3.1 – Diagramme de cas d'utilisation pour l'application de téléphonie.

3.3.1.1 Application à l'étude de cas

La figure 3.1 montre le diagramme Aspect-UML de cas d'utilisation de l'application de téléphonie.

Dans cette application, les fonctionnalités de base du système de téléphonie concernent principalement la gestion des appels téléphoniques, tels que *initier un appel* et *rompre un appel.* Sachant que les cas d'utilisation saisissent et décrivent les exigences fonctionnelles, alors il est tout à fait naturel de voir ces fonctionnalités représentées respectivement par les cas d'utilisation de base *complete* et *drop*, tels que représentés par la figure3.1.

Comme nous l'avons expliqué à la section 3.3.1, si on souhaite intégrer à l'application de téléphonie les deux fonctionnalités transverses suivantes : *calcul de la durée d'une communication* et *facturation des communications pour un client*, le développeur peut le faire en utilisant la notion d'aspects. Nous introduisons donc deux aspects *Timing* et *Billing* respectivement pour les deux fonctionnalités citées.

La fonctionnalité de *calcul de la durée de communication* est composée de deux

sous-fonctionnalités *calcul du temps de début de communication* et *calcul du temps de fin de communication* qui interviennent respectivement au début et à la fin de chaque communication. Par contre, la fonctionnalité de *facturation* doit être réalisée seulement à la fin de chaque communication. Ces deux fonctionnalités sont transverses, elles entrecoupent les deux fonctionnalités de base que sont *initier une communication* et *rompre une communication.* Ces aspects permettent de renforcer le système de base avec ces deux nouvelles fonctionnalités sans altérer son fonctionnement initial. Dans la vue de cas d'utilisation proposée pour le système de téléphonie, les deux aspects *Timing* et *Billing* sont représentés respectivement par les cas d'utilisation d'extension *Timing* et *Billing*, spécifiant le comportement additionnel à exécuter afin d'accomplir les besoins visés par la fonctionnalité en question.

Comme le montre la figure 3.1, pour modéliser la dépendance de ces deux aspects et leur insertion dans les cas d'utilisation de base *complete* et *drop*, la relation « `crosscuts` » est utilisée. Les cas d'utilisation de base seront donc étendus aux points de coupure indiqués *CompletePoint* et *DropPoint.*

3.3.1.2 Templates pour les cas d'utilisation

Les templates servent à documenter les cas d'utilisation en décrivant leurs scénarios [Coc00]. Dans notre méthodologie, afin de distinguer les cas d'utilisation de base et les cas d'utilisation transverses, nous avons défini pour chaque type de cas d'utilisation un template qui contient ses informations pertinentes.

1. **Templates pour les cas d'utilisation de base.** Le template décrivant le scénario d'un cas d'utilisation de base contient entre autre : le nom du cas d'utilisation, ses acteurs, les pré et postconditions, le scénario principal, éventuellement les scénarios secondaires et les points de coupure où seront éventuellement intégrées des fonctionnalités additionnelles. La table 3.1 décrit le scénario pour le cas d'utilisation de base *complete* pour initier une

communication.

2. **Template pour les cas d'utilisation transverses.** Un cas d'utilisation transverse peut définir un ou plusieurs scénarios d'extension correspondant aux différents points de coupure qu'il entrecoupe. Ainsi, le template documentant un cas d'utilisation transverse définit non pas un scénario principal mais un segment décrivant le scénario d'extension pour chaque point de coupure. Ce template contient aussi le nom du cas d'utilisation transverse précédé du stéréotype « `aspect` », ses acteurs, les pré et postconditions, le nom des points de coupure où sera inséré le cas d'utilisation transverse et éventuellement des points de coupure appartenant à ce cas transverses. La table 3.2 donne la documentation décrivant le scénario du cas d'utilisation transverse *Timing.*

Use case :	complete
Level :	But utilisateur
Actors :	Customer
Preconditions :	S'assurer que la ligne téléphonique est disponible, aucune communication n'est établie sur la ligne de l'appelant.
Postconditions :	Assurer qu'une connexion est bien établie.
Main scenario :	1. L'abonné appelant accède à son appareil téléphonique pour former le numéro de destination désiré. 2. L'abonné vérifie la tonalité et compose le numéro de destination. 3. Le système se connecte à la destination et établit la communication.
Pointcuts :	CompletePoint

TAB. 3.1 – Documentation du scénario décrivant le cas d'utilisation de base *complete.*

3. **Template pour les points de coupure.** Afin d'améliorer la modularité dans la vue de cas d'utilisation, les points de coupure sont décrits et dé-

Use case :	« aspect » Timing
Crosscut pointcuts	CompletePoint, DropPoint
Level :	Besoin
Preconditions :	La connexion est établie.
Postconditions :	La durée de la connexion est calculée et sauvegardée.
Segment 1 :	<CompletePoint> 1. Le système accède au Timer de la connexion en cours. 2. Le temps de début de connexion est alors calculé. 3. Le système sauvegarde ce temps de début de connexion.
Segment 2 :	<DropPoint> 1. L'utilisateur rompt la connexion. 2. Le système accède au timer de la connexion qui vient d'être rompue, afin de lire le temps de début de communication. 3. Le temps de fin de communication est calculé. 4. Le système calcule et sauvegarde la durée de la connexion.
Pointcuts :	none

TAB. 3.2 – Documentation du scénario du cas d'utilisation transverse *Timing*.

finis séparément au moyen de templates spécifiques. Chaque point de coupure est décrit par : son nom, une description textuelle et une liste de références correspondant à des étapes dans les cas d'utilisation contenant le point de coupure. Ces références sont définies par la séquence : <nom du cas d'utilisation>[numéro de l'étape]. Éventuellement, on peut spécifier aussi une condition rattachée au point de coupure. Cette condition détermine si le scénario d'extension sera effectué ou non quand le point de coupure sera atteint.

La table 3.3 dresse la documentation spécifiant le point de coupure *CompletePoint*.

Pointcut :	CompletePoint
Textual description :	Ce point définit le début d'une communication.
References :	<complete>[step3]

TAB. 3.3 – Documentation décrivant le point de coupure *CompletePoint*.

3.3.2 Vue structurelle

Le diagramme de classes est utilisé pour décrire la structure des éléments logiques du système. Cette vue doit intégrer les entités du domaine, les contrôleurs, les interfaces ainsi que les éléments décrivant les aspects.

Nous proposons de modéliser le concept d'aspect, décrit dans la vue de cas d'utilisation par une sorte de cas d'utilisation d'extension, par un stéréotype de classe UML appelé « `Aspect` ». Quant aux points de coupure, assimilés à des points d'extension dans la vue de cas d'utilisation, nous les modélisons par des interfaces UML annotées par le stéréotype « `Pointcut` ». Une interface « `Pointcut` » définit un ensemble de points de jointure qu'elle encapsule ainsi qu'une opération à exécuter quand un de ses points de jointure est atteint. Par ailleurs, une relation « `crosscuts` » permet de relier une interface « `Pointcut` » avec chacun de ses points de jointure ; un point de jointure étant évidemment une opération définie dans une classe ou un aspect. De même, une relation de « `réalisation` » relie une interface « `pointcut` » avec chaque aspect qui l'implémente. Tout aspect implémentant une interface « `Pointcut` » fournit une méthode appelée *advice* qui sera exécutée aux points de jointure définis par l'interface « `Pointcut` ». L'advice est annoté avec l'un des stéréotypes *before*, *after* ou *around* selon qu'il doit être exécuté respectivement avant, après ou à la place des points de jointure.

Les aspects sont donc représentés par des classes spéciales qui implémentent une ou plusieurs interfaces constituant les points de coupure où les aspects doivent précisément intervenir en invoquant l'advice approprié. Cette modélisation a l'avantage de maximiser la réutilisabilité de l'aspect en le déchargeant de la définition des points de jointure et de fournir une interface entre un aspect et les classes et/ou

aspects qu'il entrecoupe. De plus, elle a l'avantage de montrer et de mettre en évidence la concurrence et les conflits directs entre les aspects. En effet, des aspects sont potentiellement conflictuels s'ils réalisent une même interface « `Pointcut` » et s'ils définissent un advice de même type.

3.3.2.1 Application à l'étude de cas

La figure 3.2, page 68, montre le diagramme de classes de base de l'application de téléphonie mettant en évidence les éléments de domaine du système de base, à savoir, la classe *Connection*, la classe *Customer* et la classe *Device*.

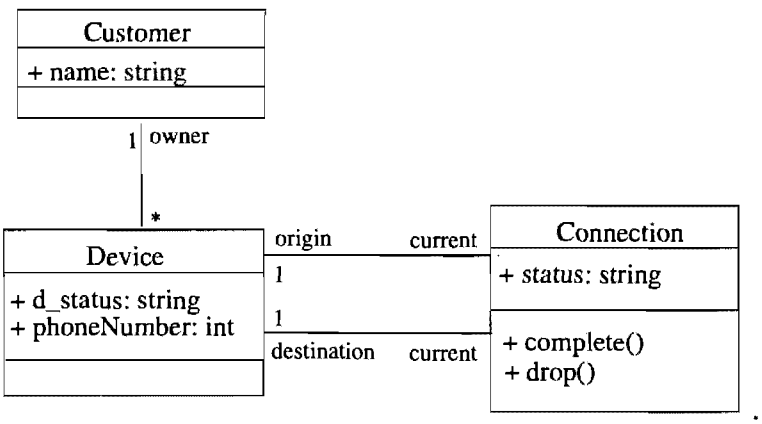


FIG. 3.2 – Diagramme de classes pour l'application de base de téléphonie.

À ce diagramme de base on désire intégrer les exigences des fonctionnalités supplémentaires, décrites par les cas d'utilisation *Timing* et *Billing* de la figure 3.1, page 63.

La figure 3.3, page 69, montre, comment les éléments conceptuels requis par les nouvelles fonctionnalités *Timing* et *Billing* sont introduits dans le diagramme de classes UML. L'élaboration de ce diagramme de classes se fait à partir du modèle de la vue de cas d'utilisation. En effet, les cas d'utilisation d'extension *Timing* et *Billing* sont saisis et réalisés par des classes nommées respectivement *Timing*

et *Billing* portant le stéréotype « `Aspect` ». Quant aux points de coupure *CompletePoint* et *DropPoint*, ils sont modélisés par des interfaces portant le stéréotype « `Pointcut` ». Pour mettre en évidence et modéliser l'entrecoupement de la classe *Connection* par les aspects *Timing* et *Billing*, au niveau des deux méthodes (points de jointure) *complete()* et *drop()* (via les interfaces respectives), la relation « `crosscuts` » est utilisée. Les interfaces *CompletePoint* et *DropPoint* contiennent donc chacune une opération abstraite nommée respectivement *opComplete* et *opDrop* dont les implémentations respectives dans les aspects, c'est-à-dire les advices, doivent être exécutées quand un de leurs points de jointure est atteint.

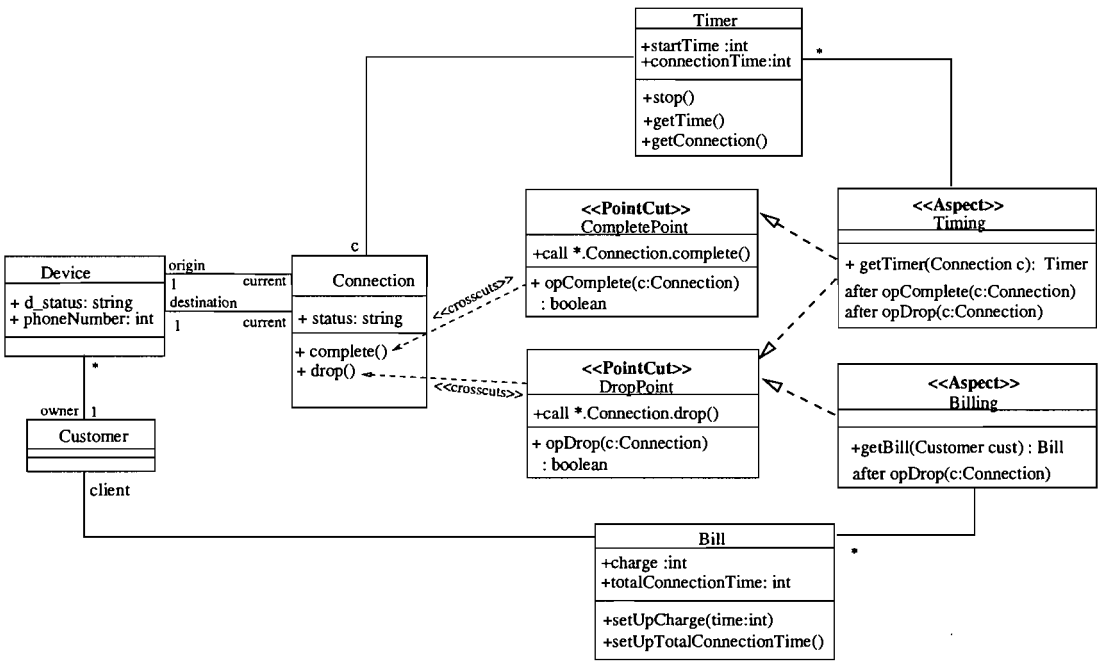


FIG. 3.3 – Diagramme de classes pour l'application de téléphonie.

1. **Aspect Timing** : Cet aspect est utilisé pour calculer la durée d'une communication. Pour exécuter ses fonctionnalités, l'aspect *Timing* utilise deux advices qui sont *Timing.opComplete* et *Timing.opDrop*. L'advice *opComplete* est exécuté après chaque début de connexion, afin de sauvegarder le temps de

début de communication. D'un autre côté, l'advice *opDrop* est exécuté après chaque fin de connexion, pour sauvegarder le temps de fin de communication. La différence entre ces deux temps donne la durée de communication. Afin de gérer la notion de temps, une nouvelle classe appelée *Timer* est définie et est reliée à l'aspect *Timing*, Une méthode *getTimer(c)* est alors définie dans l'aspect Timing, afin d'accéder à l'objet Timer de la connexion *c*. En effet, les advices *Timing.opComplete* et *Timing.opDrop* font appel respectivement aux méthodes *start()* et *stop()* de l'objet *Timer* pour calculer les temps de début et de fin de la communication en cours.

L'aspect *Timing* est donc exécuté au début et à la fin de chaque connexion. Par conséquent, il réalise les deux interfaces « `Pointcut` » *CompletePoint* et *DropPoint*, qui entrecoupent la classe *Connection* aux points de jointure respectifs à l'appel de la méthode *complete()* et à l'appel de la méthode *drop()*.

2. **Aspect Billing :** Cet aspect est utilisé pour mettre à jour la facture du client. Ceci est réalisé par l'advice *Billing.opDrop* exécuté à chaque fin de connexion. En effet, à chaque fin de connexion le montant de la communication est calculé et est rajouté à la facture du client. Une classe *Bill* est reliée à l'aspect *Billing* afin de pouvoir accéder aux factures des clients et de les mettre à jour.

   L'aspect *Billing* est donc exécuté à la fin de chaque connexion, réalisant ainsi l'interface *DropPoint* qui elle entrecoupe la classe *Connection* à l'appel de la méthode *drop()*.

3.3.3 Vue comportementale

L'objectif final de notre travail est d'assurer la correction de la composition des aspects dans le système final, c'est à dire après le tissage des aspects dans le système de base. Plus précisément, nous nous intéressons à vérifier les interactions entre aspects dans le système final. La vérification que nous comptons effectuer

ne concerne que la composition des aspects et non la correction du modèle en entier. Par conséquent, la vue comportementale de notre profil Aspect-UML doit permettre de représenter les propriétés caractérisant le contexte et les contraintes de composition des aspects. Un modèle de la vue comportementale est obtenu à partir du modèle de la vue structurelle. En fait, le développeur est invité à enrichir le diagramme de classes d'un certain nombre de contraintes formelles, formant ainsi le diagramme de la vue comportementale. Les quatre sortes de contraintes requises par un tel modèle sont décrites plus bas. Ces contraintes peuvent être spécifiées directement sur le diagramme de classes au moyen d'une note portant le stéréotype « `Aspect-UML Constraint` » qu'on rattache à l'élément contraint. On peut également regrouper les contraintes dans un fichier texte indépendant. Le diagramme de la figure 3.4 décrit la vue comportementale de l'application de téléphonie, où le diagramme de classes est enrichi avec les contraintes de composition.

3.3.3.1 Contraintes de précédence

Il est possible que plusieurs aspects s'entrecoupent à un même point de jointure et que tous proposent un advice de la même sorte (`before` ou `after`). Sur un diagramme de classes Aspect-UML, on peut identifier ces aspects en repérant les advices de la même sorte qui implémentent une même interface « `Pointcut` ». C'est effectivement le cas des aspects *Timing* et *Billing* de l'application de téléphonie qui implémentent tous deux des advices `after` au point de coupure *DropPoint*. Il est toutefois possible d'ordonner l'exécution de ces aspects en définissant une relation de précédence entre eux. Pour ce faire, il suffit d'ajouter au diagramme de classes une contrainte globale de la forme $aspect_1 < aspect_2$, où "<" est une relation d'ordre partiel. Dans le cas de l'application de téléphonie, puisqu'il importe de calculer la durée d'un appel pour pouvoir le facturer, le développeur peut définir la contrainte $Timing < Billing$.

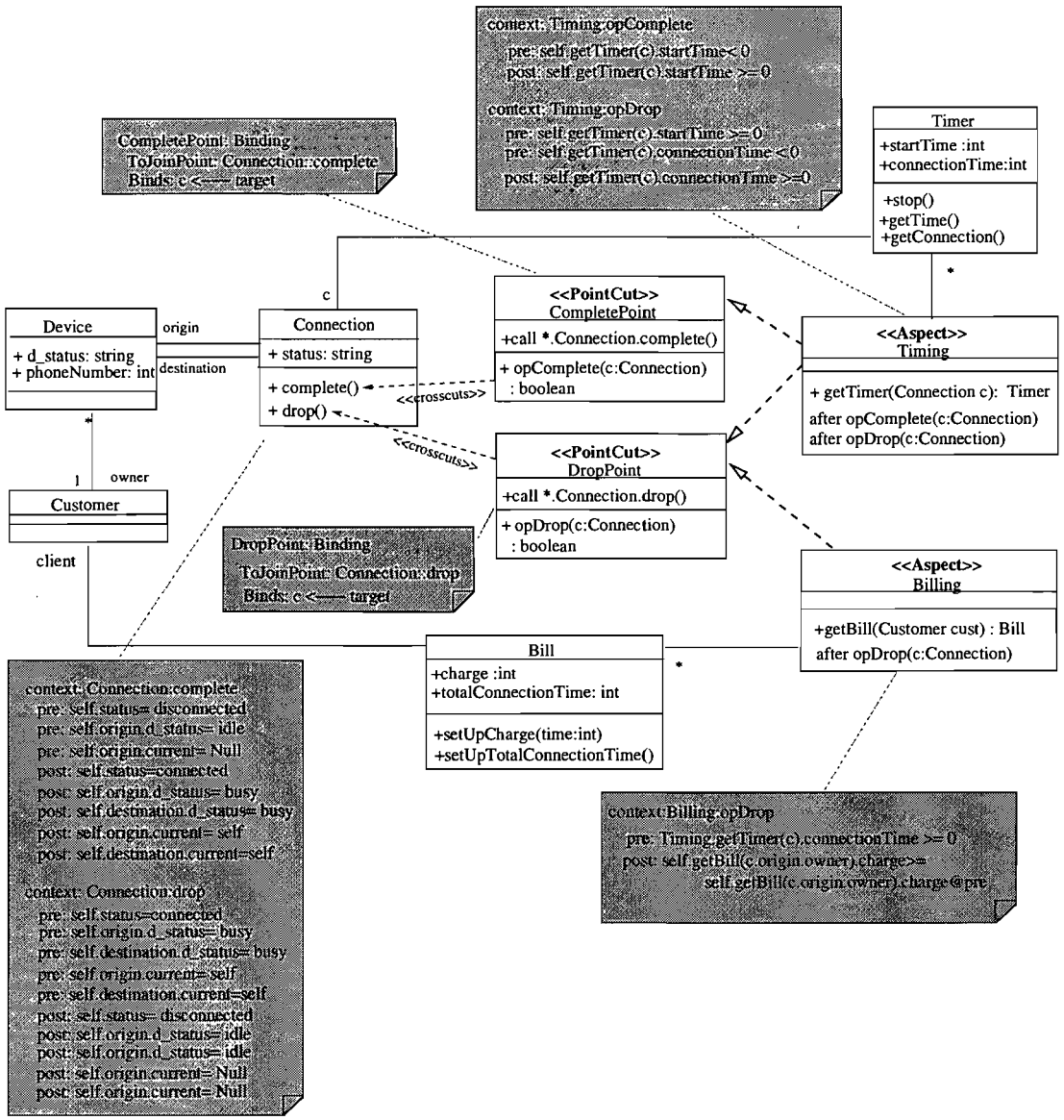


FIG. 3.4 – Modèle Aspect-UML décrivant la vue comportementale de l'application de téléphonie.

3.3.3.2 Spécification du passage du contexte d'un point de jointure à un advice

Un advice peut nécessiter l'accès aux données d'un point de jointure. Le point de coupure permet d'exposer le *contexte* d'un point de jointure. Ce contexte contient notamment l'objet dont on invoque la méthode au point de jointure et les arguments de l'appel. Le passage du contexte est décrit au moyen d'une contrainte Aspect-UML spécifiant la façon dont les paramètres formels du point de coupure sont liés aux éléments du contexte.

<pointcut> : : Binding

ToJoinPoint : $< jointPointName >$

Binds : $< pcArg >_1 \leftarrow < context\_elem >, \ldots, < pcArg >_n \leftarrow < context\_elem >$;

Dans cette contrainte, $< pcArg >_1, \ldots, < pcArg >_n$ sont les paramètres du point de coupure et *context_elem* est soit un paramètre du point de jointure, soit la variable *target* qui retourne une référence sur l'objet appelé. Dans notre exemple, la contrainte suivante spécifie le passage du contexte du point de jointure *Connection* :: *drop*() au point de coupure *DropPoint* :

DropPoint : : Binding

ToJoinPoint : *Connection* :: *drop*()

Binds : $c \leftarrow target$

3.3.3.3 Spécification des points de jointure

Pour focaliser la vérification sur la correction de la composition des aspects, nous faisons l'hypothèse que le modèle initial (c'est-à-dire le modèle de base et le modèle de chacun des aspects pris séparément) est correct et que nous disposons de la spécification de certains fragments de ce modèle. Entre autre, il est attendu que le développeur fournisse, dans le modèle Aspect-UML, la spécification des méthodes constituant des points de jointure. Cette spécification est exprimée sous forme de pré et postconditions et est notée de la façon suivante :

context nom_classe : : nom_méthode()

pre : $condition_1, \ldots$

post : $condition_2, \ldots$

Le contexte est défini par le nom de la méthode (préfixé par le nom de la classe à laquelle elle appartient), constituant le point de jointure. Les conditions sont des expressions booléennes construites sur les types de base (integer, real, string, boolean), les attributs, les paramètres formels, les opérateurs [1] et les associations du modèle (on fait référence à une relation d'association en utilisant la notation pointée, tout comme pour les attributs ordinaires). Une précondition exprime une propriété nécessaire à l'exécution de la méthode, alors qu'une postcondition exprime une propriété assurée suite à l'exécution de la méthode. Pour l'application de téléphonie, la spécification du point de jointure *Connection* :: *drop*() est définie comme suit :

context Connection : : drop()

pre : $self.status = connected$

pre : $self.origin.d\_status = busy$

pre : $self.destination.d\_status = busy$

pre : $self.origin.current = self$

pre : $self.destination.current = self$

post : $self.status = disconnected$

post : $self.origin.d\_status = idle$

post : $self.destination.d\_status = idle$

post : $self.origin.current = Null$

post : $self.destination.current = Null$

3.3.3.4 Spécification des advices

Pour vérifier la composition d'un aspect, nous devons connaître la spécification de ses advices. Nous pourrons alors vérifier que le tissage des advices n'entrave pas

[1] Entre autres, seuls les opérateurs sans effet de bord sont considérés.

la correction du système de base. Nous faisons l'hypothèse que le corps d'un advice respecte effectivement la spécification qui en est donnée. Cette spécification est formulée de la même façon que pour les points de jointure. Cette fois, le contexte est cependant de la forme *nom_aspect* : :*nom_advice()*. L'exemple suivant décrit la spécification de l'advice *opDrop()* de l'aspect *Billing* de l'application de téléphonie[2].

context Billing : : opDrop(c : Connection)

pre : $Timing.getTimer(c).connectionTime \geq 0$

post : $getBill(c.origin.owner).charge > getBill(c.origin.owner).charge_{@pre}$

Nous pouvons assurer la sûreté du système après le tissage d'un advice en appliquant les règles de contravariance et de covariance. Ces règles stipulent respectivement que les préconditions de l'advice doivent être plus faibles que les préconditions du point de jointure et que les postconditions de l'advice doivent être plus fortes que (*c-à-d* doivent impliquer) les postconditions du point de jointure. S'il y a possibilité de contradiction entre les préconditions d'un advice et celles d'un de ses points de jointure, ou entre leurs postconditions, la vérification devra signaler une erreur.

Par ailleurs, afin de solutionner le *frame problem* [3] [MH69] inhérent aux spécifications formelles basées sur les pré et postconditions nous associons aux spécifications la sémantique des *frames* (*frame semantics*) suivante : *Toute entité n'apparaissant pas dans les pré et post conditions d'une opération est considérée comme inchangée lors de l'exécution de l'opération.*

[2]Dans la contrainte qui suit, @pre est une construction empruntée à OCL (Object Constraint language) [Obj04] utilisée pour faire référence à la valeur antérieure d'un élément.

[3]En inteligence artificielle, le *frame problem* a été initialement formulé comme étant le problème d'exprimer en logique un domaine dynamique sans spécifier explicitement quelles conditions ne sont pas affectées par une action.

3.4 Discussion

Au niveau de la phase d'analyse des besoins, nous avons présenté, dans notre travail, une idée très intuitive pour étendre les cas d'utilisation afin de prendre en charge les aspects. Notons que cette approche facilite la composition et ceci grâce aux relations *extend* et *include*. À l'instar de [Jac03, JN05] nous considérons un aspect comme un cas d'utilisation d'extension spécial. Cependant, pour modéliser la nature transverse des advices, nous avons défini deux éléments UML stéréotypés qui sont l'élément « `Pointcut` » et la relation « `crosscuts` », au lieu d'utiliser les éléments traditionnels de UML qui sont respectivement le `point d'extension` et la relation `extend`. Par ailleurs, la vue de cas d'utilisation de notre profil Aspect-UML est enrichie avec des templates spécifiques décrivant les cas d'utilisation de base, les cas d'utilisation d'extension et les points de coupure.

Par ailleurs, la vue structurelle de notre profil Aspect-UML, se distingue notablement des autres propositions [SY99, SHU02, AEB03, CL02a] recensées dans la littérature et présentées dans la section 2.4. En effet, même si on y retrouve cette similitude de représenter un aspect par un stéréotype de classe UML, notre proposition se distingue, d'une part, par la façon dont elle définit dans UML les autres concepts aspect à savoir, point de coupure, point de jointure et advice, et d'autre part, par sa façon de modéliser et de mettre en évidence l'entrecoupement en le montrant graphiquement. En fait, la plupart des approches à l'instar de [SHU02, AEB03] modélisent les points de coupure par des stéréotypes d'opérations UML ; or, un point de coupure est un conteneur pour un ensemble de points de jointure. A notre sens, cette façon de modéliser ne met pas visuellement en évidence l'entrecoupement réalisé par le point de coupure. Nous approuvons plus une modélisation d'un point de coupure par une interface. Modéliser un point de coupure par une interface et un point de jointure par un attribut de cette interface tel que nous l'avons proposé dans [VM04a, MV06a] nous semble un choix pertinent car

cela d'une part, reflète bien la sémantique de ces deux concepts et, d'autre part, offre une meilleure modularité des modèles orientés aspects. De plus, cela permet de faire ressortir et de mettre en évidence les interactions directes entre aspects en montrant graphiquement les aspects implémentant les mêmes interfaces et offrant des advices de même type.

De plus, nous avons proposé une vue comportementale qui enrichit le diagramme de classes par des annotations sémantiques, telles que les pré et les postconditions pour spécifier le comportement des points de jointure et des advices. À notre connaissance, même si l'idée d'utiliser les annotations UML dans les modèles orientés aspect a déjà été suggérée dans la littérature [JPWG02,NBA04], ces annotations n'ont pas été élaborées et exploitées pour la vérification formelle de modèles.

3.5 Conclusion

À notre sens, l'idéal pour une approche de modélisation par aspects est de couvrir toutes les phases de développement de logiciels, depuis l'acquisition des besoins jusqu'au test. Nous pensons, que les travaux de recherche en cours doivent trouver une convergence vers un langage de modélisation unifié orienté aspect, comme cela a été le cas pour le paradigme objet. Dans le cadre de notre travail, nous avons défini une approche de modélisation par aspects qui regroupe un certain nombre de critères que nous jugeons essentiels pour notre objectif de vérification, à savoir :

- Prendre en charge les aspects dès l'analyse puis tout au long de la conception, pour faciliter leur implémentation ultérieure.
- Proposer une modélisation des concepts du paradigme aspect qui soit la plus fidèle à leur sémantique communément reconnue.
- Grâce aux annotations, pouvoir extraire d'un modèle Aspect-UML un modèle formel qui peut être vérifié formellement.

CHAPITRE 4

SÉMANTIQUE DE ASPECT-UML EN RÉSEAUX DE PETRI

4.1 Introduction

La modélisation avec le profil Aspect-UML que nous proposons demeure conviviale tout en étant rigoureuse. Bien que d'un haut niveau d'abstraction, les modèles Aspect-UML fournissent des informations sémantiques décrivant la composition des aspects, informations dont nous profiterons pour la vérification de cette composition. Cependant, il demeure difficile de s'assurer d'une composition *correcte* des aspects sans une formalisation de leur tissage. En effet, pour vérifier la composition des aspects, leur tissage doit être exprimé dans un modèle qui a des fondements mathématiques rigoureux et une sémantique précise. Nous nous concentrerons, dans ce chapitre, à extraire un modèle mathématique à partir d'un modèle Aspect-UML, où le tissage est explicité formellement. Ainsi, en se basant sur ce modèle sémantique, la composition des aspects spécifiés dans le modèle Aspect-UML correspondant peut être vérifiée formellement.

Il existe un grand nombre de modèles mathématiques permettant de décrire le comportement d'un système. Dans notre cas, nous proposons dans [MV05, MV06a, MV06b] d'exprimer ce modèle formel à l'aide de réseaux de Petri [Pet81]. Les réseaux de Petri constituent un formalisme mathématique bien adapté à la description des systèmes séquentiels et concurrents. Ils sont particulièrement appropriés pour exprimer la composition des flots de contrôle (tel que la séquence, le choix non déterministe,...). De plus, ils offrent une représentation graphique intéressante. Nous avons choisi, en particulier, des réseaux de Petri de haut niveau, que sont les réseaux de Petri colorés [Jen97] et les réseaux de Petri orientés objet tels qu'incarnés par le formalisme COOPN/2 [Bib97] pour doter notre profil Aspect-UML

d'une sémantique formelle. En effet, d'une part, les réseaux de Petri colorés disposent d'un outil de simulation et d'analyse automatique qui a fait ses preuves, soit CPN Tools [KCJ98]. D'autre part, l'orientation objet des réseaux de Petri incarnés par COOPN/2 est proche de l'orientation objet/aspect de notre profil Aspect-UML.

La première section de ce chapitre sera consacrée à la présentation des réseaux de Petri, plus particulièrement nous présenterons les réseaux de Petri colorés [Jen97]. La section suivante sera dédiée à la formalisation de Aspect-UML en termes de réseaux de Petri colorés. Nous y détaillerons la construction des réseaux de Petri colorés qui décrivent la sémantique d'un modèle Aspect-UML. Ensuite, suivra une section consacrée aux réseaux de Petri orientés objet, où nous présenterons le formalisme COOPN/2. Puis, nous expliquerons comment traduire un modèle Aspect-UML vers une spécification COOPN/2. La section suivante présentera brièvement l'approche de vérification à suivre pour identifier les erreurs liées à la composition et au tissage des aspects dans un modèle Aspect-UML décrit à l'aide de réseau de Petri (colorés ou orientés objet). Suivra, par la suite, une discussion, où nous identifierons les limites et les avantages de chacun des formalismes présentés par rapport à notre objectif.

4.2 Les réseaux de Petri

Le formalisme des réseaux de Petri [Pet81, Bra82], introduits pour la première fois par Carl Adam Petri [Pet62], permet la modélisation de processus dans un cadre formel. Ils permettent de modéliser la concurrence et le partage des ressources. Un réseau de Petri est un graphe bipartite orienté. Les noeuds sont appelés places (représentées par des cercles) ou transitions (représentées par des rectangles) et sont connectés par des arcs valués (néanmoins deux places ne peuvent pas être reliées entre elles, ni deux transitions). La valuation des arcs en entrée permet d'indiquer les conditions d'activation d'une transition alors que la valuation des arcs en sortie

détermine l'effet de l'action représentée par la transition. Les places peuvent contenir des jetons représentant généralement des ressources disponibles. La figure 4.1 représente le réseau de Petri modélisant l'exemple du producteur/consommateur où un producteur produit et place les messages dans un buffer et deux consommateurs prennent et consomment les messages produits.

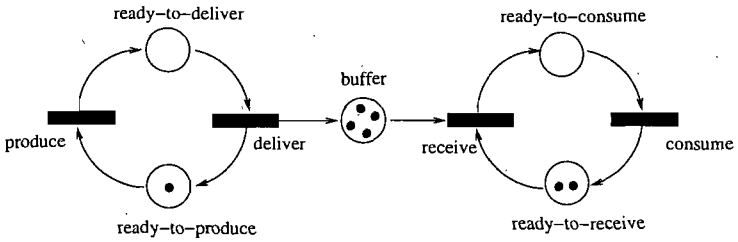


FIG. 4.1 – Réseau de Petri modélisant l'exemple du producteur/consommateur.

Un réseau de Petri évolue à chaque franchissement de transition : selon la valuation des arcs correspondants, des jetons sont pris dans les places en entrée de cette transition et envoyés dans les places en sortie de cette transition. Le franchissement d'une transition est une opération indivisible qui est rendue possible s'il y a suffisamment de jetons dans les places d'entrée. L'exécution d'un réseau de Petri n'est donc pas déterministe, car plusieurs transitions peuvent être franchissables à un instant donné. Dans ce cas, le choix de la transition à tirer est non déterministe et l'évolution du réseau de Petri l'est donc aussi. Une définition plus complète des réseaux de Petri peut être trouvée dans [Mur89] qui est un "tutoriel" sur les réseaux de Petri.

Depuis leur apparition, plusieurs extensions (présentées dans [Dia01]) ont été proposées pour les réseaux de Petri, afin de prendre en charge l'évolution des besoins de modélisation de données complexes. Une classification des différents réseaux issus de ces extensions est présentée dans [BdC92]. Nous retrouvons par exemple les réseaux colorés [Jen97], les réseaux temporisés [FGM98], les réseaux stochastiques [Emm88], les réseaux orientés objet [Bib97], etc. Notons, cependant, que le

gain en abstraction et en facilité d'expression des réseaux de Petri de haut niveau se fait en général au détriment de la complexité de l'analyse et de la vérification. En effet, la vérification des réseaux de Petri de haut niveau est bien plus difficile et plus complexe que la vérification des réseaux de Petri de base. Dans notre travail, nous nous intéressons particulièrement aux réseaux de Petri colorés et aux réseaux de Petri orientés objet.

4.3 Les réseaux de Petri colorés

Les réseaux de Petri colorés (rdPc) [Jen97,KCJ98] ou CPN (Colored Petri Nets), ou encore réseaux colorés, offrent une possibilité d'expression plus compacte et plus puissante que les réseaux de Petri ordinaires. Dans un réseau coloré, les jetons sont de haut niveau et porteurs d'information au lieu d'être anonymes comme dans les réseaux ordinaires. Les places sont donc marquées par des jetons structurés et typés auxquels une information est rattachée. Dans la terminologie des réseaux de Petri colorés, les types associés aux places sont appelés des domaines de couleur et un élément d'un domaine de couleur est appelé une couleur. Le domaine de couleur d'une place détermine le type de données que peut contenir cette place.

Un état d'un réseau de Petri coloré est appelé marquage. Il définit le nombre de jetons contenus dans chaque place à un moment donné et la valeur associée à chacun de ces jetons. Le marquage d'une place est décrit par un multi-ensemble [1] de valeurs de jetons : ainsi une place peut avoir plusieurs jetons de même valeur. Le marquage initial détermine l'état initial du réseau.

Les actions d'un réseau coloré sont représentées par des transitions. Le franchissement d'une transition consomme des jetons des places connectées aux arcs entrant et produit des jetons dans les places connectées aux arcs sortant, changeant ainsi le marquage du réseau. Le nombre de jetons consommés et produits par le

[1] Intuitivement, un multi-ensemble est un ensemble ou un élément peut apparaître plus d'une fois.

franchissement d'une transition, ainsi que la valeur de ces jetons sont déterminés par les expressions sur les arcs. En effet, à chaque arc reliant une place à une transition (et inversement) est associée une expression. Cette expression est évaluée à un multi-ensemble dont le type est celui de la place adjacente à l'arc.

Afin de franchir une transition, il faut évaluer au préalable les expressions de ses arcs entrants. Pour cela, il faut assigner aux variables apparaissant dans chaque expression les valeurs des jetons tirés de la place d'entrée. Les valeurs des jetons consommés par la transition sont déterminées par l'évaluation des arcs entrants. Par ailleurs, les valeurs des jetons produits par la transition sont déterminées par l'évaluation des expressions des arcs sortants. En outre, des gardes peuvent être associées aux transitions. Une garde est une expression booléenne qui doit être vraie pour que la transition soit franchissable.

La définition formelle d'un réseau de Petri coloré, tirée de [Jen97], est donnée par la définition 4.3.1. Il est cependant nécessaire de fixer au préalable la notation utilisée :

- Tous les éléments de type T sont dénotés par le nom T lui même ;
- Le type de la variable v est dénoté $Type(v)$;
- Le type d'une expression $expr$ est dénoté par $Type(expr)$;
- L'ensemble des variables de l'expression $expr$ est dénoté par $Var(expr)$;
- Pour un ensemble M donné, M_{MS} représente le multi-ensemble sur M.

Definition 4.3.1 (Réseau de Petri coloré). *Un réseau de Petri coloré est un tuple noté* $CPN = (\Sigma, P, T, R, C, G, E, I)$, *où :*

- Σ *est un ensemble fini et non vide de* ***types****, aussi appelés domaines de couleurs ;*
- P *est un ensemble fini de* ***places****;*
- T *est un ensemble fini de* ***transitions*** *tel que* $P \cap T = \emptyset$ *;*
- $R \subseteq (P \times T) \cup (T \times P)$ *est un ensemble d'arcs orientés ;*
- $C : P \rightarrow \Sigma$ *est une fonction de coloriage qui associe à chaque place* p *de* P

un domaine de couleurs $C(p)$;

- *G est une fonction de garde pour les transitions, définie de T vers les expressions, telle que : $\forall t \in T : [Type(G(t)) = Boolean \wedge Type(Var(G(t))) \subseteq \Sigma]$;*
- *E est une fonction d'expression pour les arcs, définie de R vers les expressions telle que : $\forall r \in R : [Type(E(r)) = C(p)_{MS} \wedge Type(Var(E(r))) \subseteq \Sigma]$, où p est la place incidente à l'arc r* [2] ;
- *I est une fonction d'initialisation pour les places telle que : $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$.*

La figure 4.2 montre un réseau de Petri coloré ayant deux places P_1 et P_2 de type entier et une place P_3 dont le type est le produit cartésien d'entiers. Aussi, une garde est associée à la transition T.

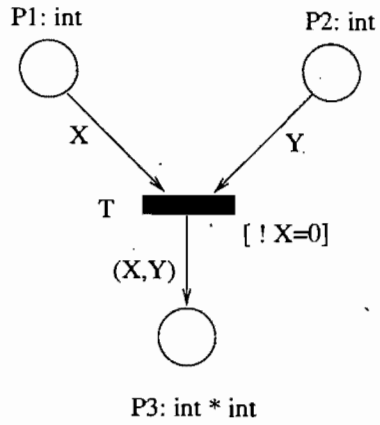


FIG. 4.2 – Exemple d'un réseau de Petri coloré.

4.4 Sémantique de Aspect-UML en termes de réseaux de Petri colorés

Dans notre travail [MV05, MV06a], nous décrivons la sémantique de Aspect-UML en termes de réseaux de Petri colorés. Dans cette section, nous présentons les

[2] Autrement dit, chaque arc r est étiqueté par une expression évaluée à un multi-ensemble dont le type est celui de la place adjacente à r. De plus, le type de chaque variable apparaissant dans l'expression de r est dans Σ.

étapes successives pour la construction de ces réseaux de Petri.

Un modèle Aspect-UML décrit, en particulier, un ensemble de *données* (objets et types simples) manipulées par un *flot d'exécution*, lui-même contraint par la composition d'aspects aux points de jointure spécifiés. La sémantique verra donc à traduire le flot d'exécution par une séquence de transitions et les données par des jetons colorés dans les places d'un rdPc. Les pré et postconditions du modèle Aspect-UML seront traduites par des contraintes qu'on associera aux arcs et aux transitions de ce réseau. Cette traduction ne prétend pas donner une sémantique à UML : elle ne couvre que les éléments du modèle Aspect-UML pertinents à la vérification de la composition des aspects.

La suite de cette section présente d'abord la traduction sémantique de la partie statique d'un modèle Aspect-UML (section 4.4.1). Nous expliquons comment les types de données et les classes sont transposés dans le monde fonctionnel de ML [MTHM96], utilisé par les rdPc pour la spécification des données. Nous définissons ensuite l'état d'un système Aspect-UML comme le marquage d'un réseau de Petri coloré à l'aide de valeurs symboliques représentées par des jetons colorés. Quant aux contraintes du modèle Aspect-UML, elles sont traduites en fonctions ML agissant sur ces valeurs symboliques.

Dans un second temps, nous présentons la traduction sémantique de la partie dynamique du modèle Aspect-UML. Nous voyons comment construire le rdPc décrivant le comportement d'un point de jointure et d'un advice (section 4.4.2). Ensuite, nous expliquons comment composer les rdPc afin de décrire la composition et le tissage des advices à un point de jointure (sections 4.4.4 et 4.4.5). En l'occurrence, nous formalisons également le passage du contexte défini par les points de coupure (section 4.4.3).

Finalement, les étapes de la construction du rdPc décrivant la sémantique d'un modèle Aspect-UML sont résumées à la section 4.4.6.

4.4.1 Traduction sémantique de la partie statique d'un modèle Aspect-UML

Dans un modèle Aspect-UML, les opérations et les contraintes sont définies sur des objets et/ou des données de type simple (integer, boolean, date, ...). Or, le domaine sémantique que nous considérons n'est pas orienté objet : en effet, le formalisme des rdPc que nous utilisons est basé sur le paradigme fonctionnel. Pour faciliter la simulation des concepts orientés objet dans le paradigme fonctionnel, nous nous permettons de restreindre notre traduction sémantique aux modèles Aspect-UML ne faisant usage que de l'héritage simple dit *par extension* (*c-à-d* sans recours au polymorphisme et à l'invocation dynamique).

4.4.1.1 Représentation des types de données dans la sémantique

Les types de données simples (e.g. `integer`, `real`, etc.) et les types énumérés (e.g. `boolean`) utilisés dans un modèle Aspect-UML sont traduits par des types construits ou prédéfinis du langage fonctionnel ML [MTHM96] (`int`, `real`, `bool`, etc.), puisque les réseaux de Petri colorés font appel à ML pour la spécification des données. Par conséquent, à chaque type t utilisé dans un modèle Aspect-UML est associé un type ML t-`Color` dans la sémantique.

Il est à noter que certains types de données, tels `int-Color` ou `real-Color`, dénotent des ensembles infinis de valeurs. Dès lors, un système utilisant ces types voit rapidement exploser le nombre de ses états potentiels. Une solution consiste à décrire les états du système au moyen de valeurs symboliques (plus abstraites) au lieu d'étayer l'infinité de combinaisons des valeurs concrètes correspondantes. Chaque valeur symbolique est une approximation d'un ensemble de valeurs concrètes. Les opérations sur les valeurs symboliques sont donc elles-aussi des approximations de celles qui seraient effectuées sur les valeurs concrètes. On choisit ces approximations de telle sorte que l'incertitude relative n'engendre pas d'imprécision ou d'inexac-

titude des propriétés à démontrer. Bien qu'on y perde en précision absolue, cette approche nous permet de gagner en efficacité tout en garantissant l'objectif de vérification.

Dans le cas qui nous intéresse, on choisit de représenter les valeurs associées aux variables de type `integer` (resp. `real`) par des expressions symboliques représentant une union d'intervalles sur les entiers (resp. les réels). Par exemple, si x est une variable entière devant satisfaire les conditions $0 < x \leq 5$ ou $x \geq 25$, on lui associera la valeur symbolique suivante : $\{\ ]0,5], [25,\infty[\ \}$. Pour ce faire, on introduit un type `intSet` (resp. `realSet`) pour représenter ces intervalles ainsi que des opérations pour les manipuler.

4.4.1.2 Représentation des classes dans la sémantique

Pour pallier les différences de paradigmes entre Aspect-UML et les rdPc basés sur des types algébriques (à la ML), nous proposons de simuler les principaux concepts orientés objets par des mécanismes de la programmation fonctionnelle. Telle que présentée et résumée par la table 4.1, notre approche demeure simple et n'a donc pas la prétention de couvrir tous les concepts du paradigme objet.

4.4.1.3 Représentation des états d'un modèle Aspect-UML

L'état global d'un modèle Aspect-UML est décrit par l'état de son environnement E_{env} et l'état de son exécution E_{exec}.

L'état de l'environnement E_{env} est décrit par l'état des instances existant dans le modèle à un moment donné, qu'il s'agisse d'objets appartenant à une classe ou de l'instance unique décrivant l'état d'un aspect. Dans la sémantique, E_{env} sera décrit par le marquage d'un ensemble de places désignées pour contenir les instances des classes et des aspects du modèle.

Plus formellement, considérons *CTypes* l'ensemble des types définis par les classes et les aspects du modèle Aspect-UML, et *DTypes* l'ensemble des types

Concept du paradigme objet	**Concept du paradigme fonctionnel**
Classe C	Définition d'un nouveau type record C.
Attribut A de la classe C	Champ A dans le type record C
Méthode $m(a_1 : T_1, a_n : T_n) : T_r$ de la classe C	Fonction $m : (C, T_1, ..., T_n) \rightarrow T_r$
Identité d'un objet	Champ *Ident* dans tous les types record décrivant une classe ; la valeur de ce champ est gérée comme une clef et permet d'identifier chaque objet de façon unique.
Héritage de la sous-classe B par la super-classe A (par extension)	Définition d'un type *tagged union* pour la sous-classe B. Le type comportera deux tags, l'un associé à un record type A, l'autre associé à un record type B (regroupant les attributs spécifiques à B).

TAB. 4.1 – Simulation des concepts objets dans le paradigme fonctionnel.

de données (prédéfinis ou définis par l'utilisateur) utilisés dans ce modèle. Un environnement est défini par l'ensemble de places $\Pi_{CTypes} = \{P^t | t \in CTypes\}$ tel que chaque place $P^t \in \Pi_{CTypes}$ permet de stocker des valeurs de type t-`Color`.

L'état d'exécution E_{exec} d'un modèle Aspect-UML décrit l'état du flux d'exécution (valeur du compteur ordinal) et les valeurs assignées aux paramètres formels de la prochaine opération à exécuter (en l'occurrence, un point de jointure ou un advice). Dans le modèle sémantique, cette information sera exprimée par le marquage de places spécifiquement créées soit pour identifier les différents points d'exécution du système, soit pour contenir les valeurs actuelles des paramètres. Nous introduirons ces places au moment de construire les rdPc décrivant les opérations associées aux points de jointure et aux advices.

4.4.1.4 Représentation des pré et postconditions

Les pré et postconditions d'un modèle Aspect-UML définissent des contraintes sur les états du système avant et après l'exécution d'une opération (*c-à-d* un point

de jointure ou un advice). Considérons un modèle Aspect-UML dont les opérations sont modélisées par les transitions d'un rdPc et les états par ses marquages. L'exécution d'une opération est donc associée au franchissement de la transition correspondant dans le rdPc. Cette exécution est possible selon Aspect-UML pour autant que les préconditions de l'opération soient préalablement satisfaites. De plus, l'exécution de l'opération doit garantir les postconditions qui lui sont associées. Dans le modèle sémantique (rdPc), on ajoutera donc des expressions sur les arcs entrants dans la transition ainsi qu'une garde pour vérifier que le marquage du rdPc satisfait bien les préconditions de l'opération correspondante. De même des expressions sur les arcs sortants de cette transition permettront de produire le résultat spécifié par les postconditions.

Pour assurer la satisfaction des postconditions d'une opération, on devra trouver, en particulier, une solution à l'ensemble de contraintes apparaissant sur les arcs sortants. Cette solution ne doit pas impliquer de dépendance entre les variables (e.g. $x > y$). En effet, la modélisation en rdPc ne permet pas de maintenir de dépendance entre des valeurs qui sont éventuellement stockées dans des places différentes. Pour éviter ce problème d'interdépendance, nous supposons que chaque postcondition est exprimée en fonction d'au plus une variable.

Tenant compte de cette hypothèse, nous définissons deux fonctions ML destinées à résoudre et vérifier respectivement les pré et postconditions..

1. $satisfyPre(precond; domValues)$: cette fonction prend une liste *precond* d'expressions booléennes et une liste $domValues$ contenant un domaine de valeurs pour chaque variable apparaissant dans *precond*. Elle retourne une liste qui énumère toutes les solutions satisfaisant toutes les formules de *precond*. La liste des solutions est finie puisque les domaines des valeurs sont finis ;
2. $solvePost(postcond; domValues; var)$: cette fonction prend une liste *postcond* d'expressions booléennes et une liste $domValues$ contenant un domaine de valeurs pour chaque variable (différente de var) apparaissant dans *postcond*.

Elle retourne une liste de toutes les valeurs de *var* qui satisfont toutes les formules dans *postcond*. Cette solution pour *var* est donnée sous forme symbolique. Elle ne contient pas de dépendance étant donné l'hypothèse faite concernant la forme des postconditions.

4.4.2 Modélisation des points de jointure et des advices

Les points de jointure et les advices sont des opérations dont la sémantique est donnée par les pré et postconditions spécifiées dans le modèle Aspect-UML. Chaque opération est modélisée dans un rdPc par une transition pouvant modifier l'état du système (E_{env} et E_{exec}).

Le rdPc décrivant une opération *op* est donc construit à partir des informations données dans le modèle Aspect-UML, à savoir les arguments de l'opération, la classe (ou l'aspect) qui contient l'opération, les pré et postconditions associées à cette opération, mais aussi la description de l'environnement et du flot d'exécution.

Soit une opération *op* dont la signature est $op(a_0 : t_0, a_1 : t_1, ..., a_n : t_n) : t_r$ et qui décrit un point de jointure (ou un advice) défini dans une classe (ou un aspect) de type t_0. (À noter que la modélisation de *op* dans le paradigme fonctionnel utilise le paramètre a_0 pour référencer l'objet visé par l'invocation de *op*). Soit $Args_{op} = \{a_1 : t_1, ..., a_n : t_n\}$ les paramètres formelles de l'opération *op*.

Soient respectivement $preCond^{op}$ et $postCond^{op}$ l'ensemble de toutes les pré-conditions et postconditions de *op*. Soit $var(preCond^{op})$ l'ensemble de toutes les variables apparaissant dans $preCond^{op}$. Soit aussi $var(postCond^{op})$ l'ensemble de toutes les variables sur lesquelles portent les postconditions de $postCond^{op}$.

Étant donné un environnement composé d'un ensemble de places Π_{CTypes}, la modélisation en rdPc de *op* est notée $rdPc(op)$ et contient les éléments suivants :

Places et transitions

- une transition T^{op} représentant l'opération ;

- une place $P^{op}_{a_i}$ pour chaque paramètre formel $a_i \in Arg_{op}$. (A) Si le type t_i de a_i est un type de données ($t_i \in DTypes$) alors la place $P^{op}_{a_i}$ porte la couleur t_i-`Color` associée aux valeurs de type correspondant dans ML. (B) Si le type t_i de a_i est un type défini par une classe ou un aspect ($t_i \in CTypes$), alors la place $P^{op}_{a_i}$ porte la couleur `ObjId-Color` qu'on associe aux identités d'instances. En particulier, on aura systématiquement a_0 = `target` et donc une place $P^{op}_{\texttt{target}}$ contenant l'identité de l'objet visé par l'invocation de *op*;
- un sous-ensemble de places de l'environnement $\Pi^{op} = \{P^{t_i} \in \Pi_{CTypes} \mid o_i : t_i \in var(preCond^{op}) \cup var(postCond^{op}) \wedge t_i \in Ctypes\}$ contenant les instances référencées dans les pré et postconditions de *op* (soit directement par les arguments de *op* ou indirectement par des appels de méthodes ou encore par des accès aux champs des arguments de *op*).
- deux places P^{op}_{in} et P^{op}_{out} utilisées pour modéliser le début et la fin d'exécution de l'opération *op*. Ces places contiennent des valeurs de type `fUnit-Color` telles les constantes `ok`, `weaveBegin` et `weaveEnd`.

Création des arcs

Le rdPc modélisant l'opération *op* d'une classe (ou aspect) de type t_0 contient les arcs suivants :

- Pour tout $a_i : t_i \in Args_{op}$: Si $a_i \in var(preCond^{op}) \cup var(postCond^{op})$, il est alors nécessaire de consulter l'argument a_i pour vérifier les préconditions de l'opération ou assurer ses postconditions. Pour ce faire, un arc bidirectionnel[3] reliant la place $P^{op}_{a_i}$ et la transition T^{op} est créé et étiqueté par une nouvelle variable x_i. De plus, si $t_i \in Ctypes$, x_i n'est donc pas une valeur mais une référence sur un objet. On doit aller chercher l'objet référencé par a_i et dont l'identité nous est effectivement donnée par x_i. C'est pourquoi on ajoute un

[3]Les arguments sont passés par valeur : ils sont simplement consultés puis remis dans leur place. Les effets définis par les postconditions de *op* n'ont pas de conséquence sur la valeur des arguments mais ont plutôt un impact sur les objets éventuellement référencés par eux.

arc reliant la place $P^{t_i} \in \Pi^{op}$ à la transition T^{op} en l'étiquetant par une nouvelle variable $xObj_i$.

Par ailleurs, pour que $xObj_i$ corresponde bien à l'objet dont l'identité est x_i, on ajoute la contrainte $getId(xObj_i) = x_i$ à l'ensemble $CondSet^{op}$ de contraintes venant compléter les préconditions de *op*. Finalement, afin de limiter le domaine de résolution des pré et postconditions, on définit un ensemble D^{op} décrivant le domaine des variables pouvant apparaître dans les pré et postconditions de *op*. On aura donc $(a_i, xObj_i) \in D^{op}$ si $t_i \in Ctypes$ et $(a_i, x_i) \in D^{op}$ si $t_i \in Dtypes$;

- Pour tout $a_i : t_i \in Args^{op}$: Si $a_i \in var(postCond^{op})$ et $t_i \in Ctypes$, c'est que l'objet référencé par a_i est contraint suite à l'exécution de *op* (autrement dit, il existe au moins une postcondition de *op* portant sur a_i). La nouvelle valeur contrainte de l'objet référencé par a_i est calculée par la fonction $solvePost(postCond^{op}, D_{op}, a_i)$. On crée donc un arc de T^{op} à P^{t_i} étiqueté par $solvePost(postCond^{op}, D_{op}, a_i)$;
- Pour tout $o_i : t_i \in var(preCond^{op}) \cup var(postCond^{op}) \mid t_i \in Ctypes \wedge o_i \neq a_j \in Args^{op}$, $\forall j = 1..n$, il est nécessaire de consulter l'instance référencée par o_i pour vérifier les préconditions de l'opération ou pour assurer ses postconditions. Pour ce faire, on ajoute un arc reliant la place $P^{t_i} \in \Pi^{op}$ à la transition T^{op} en l'étiquetant par une nouvelle variable $oObj_i$.

  Pour que $oObj_i$ corresponde bien à l'objet référencé par o_i (dans les pré et/ou postconditions de *op*), on ajoute la contrainte $getId(oObj_i) = getObjId(o_i, preCond^{op} \cup postCond^{op})$ [4] à l'ensemble $CondSet^{op}$ de contraintes venant compléter les préconditions de *op*. Finalement, on ajoute $(o_i, oObj_i) \in D^{op}$ à l'ensemble D^{op} décrivant le domaine des variables pouvant apparaître dans les pré et postconditions de *op*.

[4] La fonction $getObjId(o_i, preCond^{op} \cup postCond^{op})$ permet de récupérer l'identité de l'objet o_i apparaissant dans les conditions $preCond^{op} \cup postCond^{op}$

Si $o_i : t_i \notin var(postCond^{op})$ alors l'objet est simplement consulté pour vérifier des préconditions et puis remis à sa place. Dans ce cas, l'arc reliant la place P^{t_i} à la transition T^{op} est donc bidirectionnel.

Si $o_i : t_i \in var(postCond^{op})$, c'est que l'objet référencé par o_i est contraint suite à l'exécution de *op* (autrement dit, il existe au moins une postcondition de *op* portant sur o_i). La nouvelle valeur contrainte de l'objet référencé par o_i est calculée par la fonction $solvePost(postCond^{op}, D_{op}, o_i)$. On crée donc un arc de T^{op} à P^{t_i} étiqueté par $solvePost(postCond^{op}, D_{op}, a_i)$;

- Pour représenter le flot d'exécution traversant *op*, on doit finalement ajouter un arc de la place P^{op}_{in} à T^{op}, et un autre de T^{op} à P^{op}_{out}. Ces arcs sont étiquetés par la constante ok de façon à permettre le transfert normal du flot.

Ajout de gardes

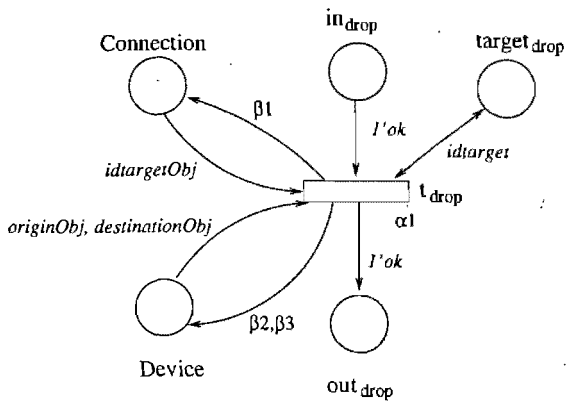
Finalement afin de forcer l'évaluation des préconditions de *op*, on munit la transition T^{op} d'une garde $satisfyPre(precond^{op} \cup CondSet^{op}; D^{op})$ qui vérifie s'il existe effectivement une solution, étant donné D^{op}, satisfaisant toutes les préconditions.

Revenons à l'application de téléphonie présentée au chapitre précédent, dont le modèle Aspect-UML est donné par la figure 3.4, page 72.

La figure 4.3 présente le rdPc modélisant le point de jointure *Connection.drop*(). D'autre part, les figures 4.4 et 4.5 montrent respectivement les rdPc modélisant les advices $Timing.opDrop(c : Connection)$ et $Billing.opDrop(c : Connection)$. À noter que les places *Connection*, *Device*, *Timing*, *Timer*, *Billing* et *Bill*, présentent dans les réseaux de Petri, représentent des places appartenant à l'environnement Π_{CTypes}.

Dans la figure 4.3, la transition t_{drop} est reliée aux places *Connection* et *Device* afin de vérifier les préconditions de *drop* et d'assurer ses postconditions, qui portent non seulemnt sur l'objet *target* mais aussi sur les instances référencées par *origin* et *destination* dans les pré et postconditions.

De même, la place *Timer* présente dans la figure 4.4 est utilisée pour vérifier



où:

$\alpha 1 = satisfyPre(\ pre^{drop}, D^{drop})$

$\beta 1 = solvePost(postCond^{drop}, D^{drop}, target)$

$\beta 2 = solvePost(postCond^{drop}, D^{drop}, origin)$

$\beta 3 = solvePost(postCond^{drop}, D^{drop}, destination)$

$D^{drop} = \{(\ target, idtargetObj\ ), (origin, originObj), (destination, destinationObj)\}$

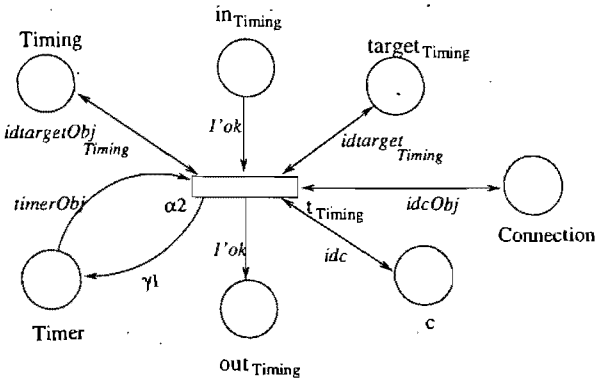
$pre^{drop} = preCond^{drop} \cup \{(\ getId(idtargetObj) = idtarget\ )\} \cup$
$\{(\ getId(originObj) = getObjId(origin, preCond^{drop} \cup postCond^{drop}\ ))\} \cup$
$\{(\ getId(destinationObj) = getObjId(destination, preCond^{drop} \cup postCond^{drop}\ ))\}$

FIG. 4.3 – rdPc modélisant le point de jointure *Connection.drop*().

les préconditions et d'assurer les postconditions de l'advice $Timing.opDrop(c : Connection)$.

Notons aussi, que dans la figure 4.5, les place *Timing*, *Timer*, *Bill* et *Device* sont reliées à la transition $t_{Billing}$. En effet, *Timing* et *Timer* sont utilisées pour vérifier les préconditions de l'advice $Billing.opDrop(c : Connection)$ et *Bill* et *Device* sont utilisées pour assurer ses postconditions.

Par ailleurs, notons que l'expression pre^{drop} (resp. pre^{Timing}, $pre^{Billing}$) représente l'ensemble de toutes les contraintes à satisfaire, c'est-à-dire celles spécifiées par les préconditions de *Connection.drop*() (resp. *Timing.opDrop*(), *Billing.opDrop*()), soit $preCond^{drop}$ (resp. $preCond^{Timing}$, $preCond^{Billing}$) et celles induites par sa mo-



où:

$\alpha 2 = satisfyPre(\ pre^{Timing},\ D^{Timing}\ )$

$\gamma 1 = solvePost(postCond^{Timing},\ D^{Timing},\ getTimer(c)\ )$

$D^{Timing} = \{(target_{Timing}, idtargetObj_{Timing}),\ (c,\ idcObj),\ (getTimer(c),\ timerObj)\}$

$pre^{Timing} = preCond^{Timing} \cup \{(getId(idcObj) = idc)\} \cup$

$\{(getId(idtargetObj_{Timing}) = idtarget_{Timing})\} \cup$

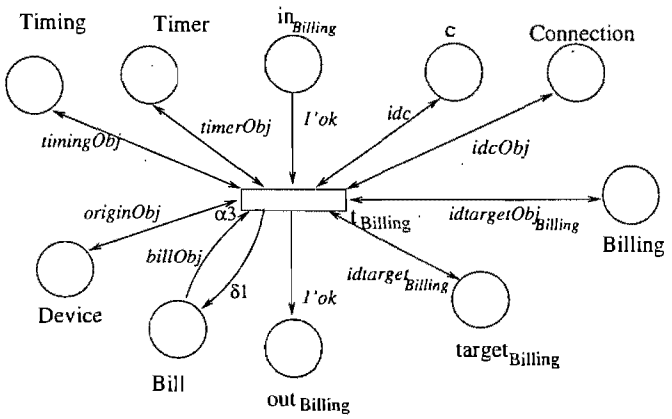
$\{(getId(timerObj) = getObjId(getTimer(c),\ preCond^{Timing} \cup postCond^{Timing})\}$

FIG. 4.4 – rdPc modélisant l'advice $Timing.opDrop(c : Connection)$.

délisation en rdPc, soit $CondSet^{drop}$ (resp. $CondSet^{Timing}$, $CondSet^{Billing}$). Ces contraintes sont évaluées sur les valeurs (éventuellement symboliques) des arguments de $Connection.drop()$ (resp. $Timing.opDrop()$, $Billing.opDrop()$) qui sont réunies dans D^{drop} (resp. D^{Timing}, $D^{Billing}$).

4.4.3 Modélisation des points de coupure

Les points de coupure ont pour principale fonction de regrouper certains points de jointure et d'en exposer le contexte pour le passer aux advices qui doivent s'y tisser. Soit PC un point de coupure associé à un ensemble de points de jointure $PJ^{PC} = \{pj_i | 1 \leq i \leq n\}$. Les paramètres formels d'un point de coupure PC sont représentés par les variables $Param^{PC} = \{v_1 : t_1, ..., v_m : t_m\}$, alors que ceux du point de jointure pj_i sont donnés par $Args^{pj_i} = \{a_1 : t'_1, ..., a_n : t'_n\}$.



où:

$\alpha 3 = satisfyPre(\ pre^{Billing}, D^{Billing}\ )$;

$\delta 1 = solvePost(\ postCond^{Billing}, D^{Billing}, getBill(c.origin.owner))$

$D^{Billing} = \{(taget_{Billing} idtargetObj_{Billing}), (c,idcObj), (Timing,timingObj),$
$(origin,originObj), (getBill(c.origin.owner),billObj)\}$

$pre^{Billing} = preCond^{Billing} \cup \{(getId(idcObj)=idc)\} \cup$
$\{(getId(idtargetObj_{Billing})=idtarget_{Billing})\} \cup$
$\{(getId(timingObj)=getObjId(Timing, preCond^{Billing} \cup postCond^{Billing})\} \cup$
$\{(getId(originObj)=getObjId(origin, preCond^{Billing} \cup postCond^{Billing})\} \cup$
$\{(getId(billObj)=getObjId(getBill(c.origin.owner), preCond^{Billing} \cup postCond^{Billing})\}$
$\{(getId(timerObj)=getObjId(getTimer(c), preCond^{Billing})\}$

FIG. 4.5 – rdPc modélisant l'advice $Billing.opDrop(c : Connection)$.

Dans le modèle Aspect-UML, chaque $pj_i \in PJ^{PC}$ est associé à une contrainte $\Gamma(pj_i, PC) \subseteq Param^{PC} \times Args^{pj_i}$ qui définit le passage du contexte de pj_i à PC. Ce contexte est composé par les valeurs des arguments et/ou par l'identité de l'objet visé par l'invocation du point de jointure pj_i. Concrètement, ce contexte est transmis aux paramètres de l'advice implémentant PC lors de l'invocation de pj.

Pour décrire la liaison du contexte d'un point de jointure à un advice adv, *via* un point de coupure PC, on associe à chaque $pj \in PJ^{PC}$ une fonction $bind_{pj,PC}(adv) \subseteq places(rdPc(adv)) \times places(rdPc(pj))$. Traduisant la contrainte $\Gamma(pj_i, PC)$, cette

fonction calcule la relation entre les places du rdPc de l'advice adv et celles du rdPc décrivant le point de jointure pj de PC. Soit un advice adv et ses paramètres formels Arg_{adv}, la fonction $bind_{pj,PC}(adv)$ est définie comme suit :

- $(P^{adv}_{x_k}, P^{pj}_{y}) \in bind_{pj,PC}(adv)$, si $v_k \leftarrow y \in \Gamma(pj_i, PC)$, $x_k \in Args^{adv}, y \in Args(pj)$ et $k \in \{1, ..., m\}$;
- $(P^{adv}_{x_k}, P^{pj}_{\texttt{target}}) \in bind_{pj,PC}(adv)$, si $v_k \leftarrow \texttt{target} \in \Gamma(pj_i, PC)$, $x_k \in Args^{adv}$ et $k \in \{1, ..., m\}$.

Soit $PC_{\phi,\psi}$, un point de coupure implémenté par un ensemble ϕ d'advices et composé d'un ensemble ψ de points de jointure. La sémantique de $PC_{\phi,\psi}$ décrit la fusion des places des rdPc des points de jointure de ψ à celles du rdPc des advices qui implémentent PC :

$$sem(PC_{\phi,\psi}) = \{(pj, b) | adv \in \phi, pj \in \psi, b = bind_{pj,PC}(adv)\}$$

4.4.4 Composition des advices aux points de jointure

À un point de jointure donné plusieurs advices peuvent être candidats à l'exécution. Le cas échéant, ces advices sont statiquement conflictuels s'ils sont de la même catégorie (`before` ou `after`). Selon la modélisation Aspect-UML, une relation de précédence peut ou non avoir été définie entre les aspects contenant les advices. Dans le premier cas, il suffit de composer séquentiellement les advices selon l'ordre prescrit. Dans le second cas, ne pouvant faire d'hypothèse sur l'ordre d'exécution, nous composons les advices par entrelacement non déterministe.

Soient deux advices conflictuels Ad_1 et Ad_2 (à un point de jointure donné) et les aspects Asp_1 et Asp_2 qui les encapsulent respectivement. Pour composer ces advices au même point de jointure, nous utilisons soit l'opération de composition séquentielle soit l'opération d'entrelacement non déterministe.

4.4.4.1 Composition séquentielle

Si une relation de précédence est définie entre les aspects Asp_1 et Asp_2 dans le modèle Aspect-UML, et si cette relation détermine que $Asp_1 < Asp_2$, alors la composition des réseaux de Petri colorés décrivant respectivement Ad_1 et Ad_2, se note $rdPc(Ad_1) \odot rdPc(Ad_2)$ et suit le patron de composition séquentielle indiqué à la figure 4.6. Ce patron exprime que l'exécution de $rdPc(Ad_1)$ est suivie immédiatement par l'exécution de $rdPc(Ad_2)$, et que $rdPc(Ad_2)$ ne peut pas s'exécuter avant $rdPc(Ad_1)$.

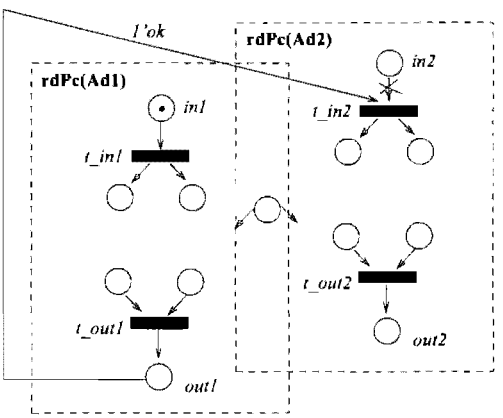


FIG. 4.6 – Patron de composition séquentielle.

4.4.4.2 Entrelacement non déterministe

Si aucune relation de précédence n'est définie entre les aspects Asp_1 et Asp_2, dans le modèle Aspect-UML, alors la composition des réseaux de Petri colorés décrivant respectivement Ad_1 et Ad_2 se note $rdPc(Ad_1) \oplus rdPc(Ad_2)$ et suit le patron d'entrelacement non déterministe indiqué à la figure 4.7. Ce patron exprime que les deux rdPc, $rdPc(Ad_1)$ et $rdPc(Ad_2)$, doivent s'exécuter en séquence, dans un ordre ou dans l'autre. Notons que les arcs, les places et les transitions utilisés

pour exprimer la composition sont dessinés en rouge.

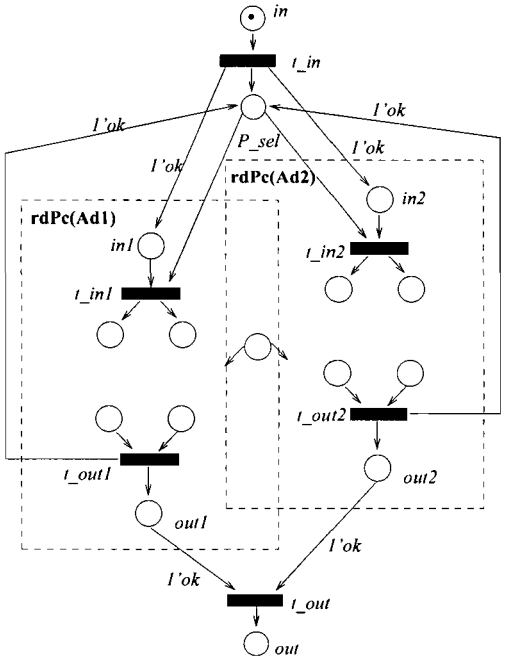


FIG. 4.7 – Patron d'entrelacement non déterministe.

La composition de deux advices Ad_1 et Ad_2 à un même point de jointure pj définit un nouvel advice composite, noté Ad. Pour passer le contexte du point de jointure pj au nouvel advice composite Ad, il importe de définir la nouvelle fonction de liaison de contexte $bind_{pj}(Ad)$:

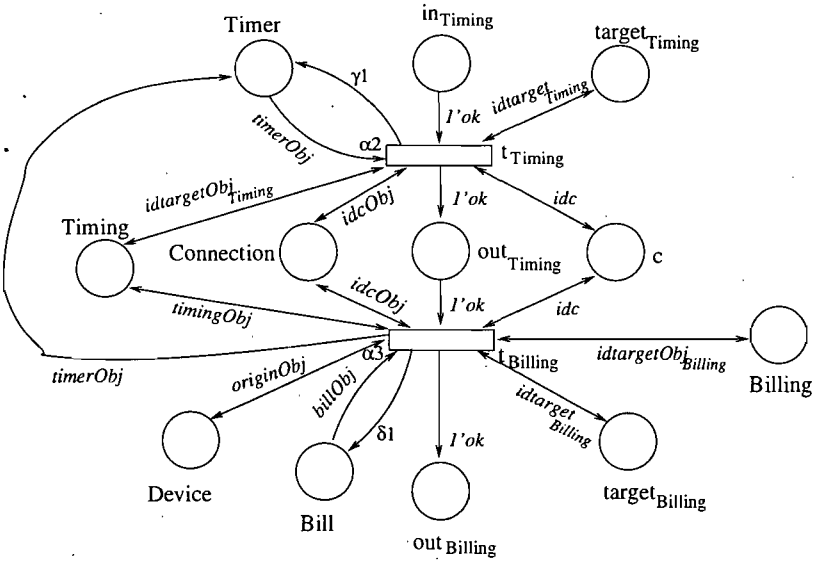
Soit PC_1 le point de coupure contenant pj implémenté par l'advice Ad_1 et soit PC_2 le point de coupure contenant pj implémenté par l'advice Ad_2. Le passage du contexte du point de jointure pj à l'advice Ad (composé par Ad_1 et Ad_2), noté $bind_{pj}(Ad)$, est obtenu en faisant l'union[5] des fonctions de binding qui lient respectivement l'advice Ad_1 au point de jointure pj *via* PC_1 et l'advice Ad_2 au

[5]On suppose ici que les places représentant les arguments des advices portent des noms distincts.

point de jointure pj *via* PC_2.

$$bind_{pj}(Ad) = bind_{pj,PC_1}(Ad_1) \cup bind_{pj,PC_2}(Ad_2)$$

Dans le cas de l'application de téléphonie, les advices $Timing.opDrop(c : Connection)$ et $Billing.opDrop(c : Connection)$ doivent être composés séquentiellement si on suppose qu'une relation de précédence est définie explicitement entre eux ($Timing < Billing$). La figure 4.8 montre le rdPc résultant de la composition séquentielle des rdPc modélisant ces deux advices, c'est-à-dire $rdPc(Timing.opdrop(...)) \odot rdPc(Billing.opdrop(...))$.



où:

$\alpha 2$ = *satisfyPre(pre* ^{Timing}, D^{Timing});

$\alpha 3$ = *satisfyPre(pre* ^{Billing}, $D^{Billing}$);

$\gamma 1$ = *solvePost(postCond* ^{Timing}, D^{Timing}, *getTimer(c))* ;

$\delta 1$ = *solvePost(postCond* ^{Billing}, $D^{Billing}$, *getBill(c.origin.owner))*

FIG. 4.8 – Composition séquentielle des advices $Timing.opDrop(c : Connection)$ et $Billing.opDrop(c : Connection)$.

4.4.5 Tissage des advices aux points de jointure

Une fois les advices conflictuels composés, il suffit maintenant de les lier aux points de jointure spécifiés. Nous appelons cette opération *tissage*. Nous distinguons *le tissage avant* et *le tissage après* selon la catégorie de l'advice considéré.

4.4.5.1 Tissage avant

L'opération de *tissage avant* permet d'insérer le rdPc modélisant un advice (composite ou non) au tout début du rdPc modélisant un point de jointure. Pour cela, une transition t_{begin} est créée pour intercepter le jeton dans la place d'entrée du point de jointure et le mettre dans la place d'entrée de l'advice. Une seconde transition t_{end} est créée pour remettre le jeton produit dans la place de sortie de l'advice dans la place d'entrée du point de jointure pour qu'il puisse reprendre son exécution. Soit pj un point de jointure, soit Ad un advice à tisser avant le point de jointure pj et soit $bind_{pj}(Ad)$ la fonction de binding liant le contexte de pj à celui de Ad. Le *tissage avant* du rdPc de l'advice Ad au point de jointure pj, noté $rdPc(Ad) \dagger_{\texttt{before}} rdPc(pj)$, est réalisé tel que décrit ci-après et tel qu'illustré par la figure 4.9.

- Pour modéliser le passage des paramètres selon la liaison de contexte spécifiée, on procède à la fusion des places du rdPc de l'advice à celles du rdPc du point de jointure selon la fonction de binding donnée. Ainsi, $\forall(x, y) \in bind_{pj}(Ad)$, on fusionne la place x du point de jointure pj à la place y de l'advice Ad;
- Pour modéliser le transfert du flot d'exécution du point de jointure vers l'advice et le récupérer ensuite, on procède comme suit : (1) On crée une transition t_{begin} reliée à la place in^{pj} par un arc entrant étiqueté par la constante `ok`, et reliée à la place in^{ad} par un arc sortant étiqueté aussi par la constante `ok`. (2) On crée une transition t_{end} reliée à la place out^{ad} par un arc entrant étiqueté par `ok`, et reliée à la place in^{pj} par un arc sortant étiqueté par `weaveEnd` (une

constante du type `fUnit`). (3) On change l'étiquette qui se trouve sur l'arc reliant in^{pj} à t_{in}^{pj} et on la remplace par `weaveEnd`.

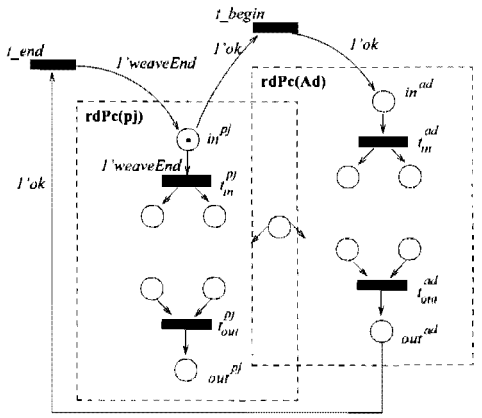


FIG. 4.9 – Patron de tissage avant.

4.4.5.2 Tissage après

L'opération de *tissage après* permet d'insérer le rdPc modélisant un advice tout à la fin du rdPc modélisant un point de jointure. Pour cela, une transition t_{begin} est créée pour intercepter le jeton produit dans la place de sortie du point de jointure et le mettre dans la place d'entrée de l'advice. Une seconde transition t_{end} est créée pour transférer le jeton produit dans la place de sortie de l'advice dans la place de sortie du point de jointure, qui ainsi termine son exécution. Soit pj un point de jointure, soit Ad un advice à tisser après le point de jointure pj et soit $bind_{pj}(Ad)$ la fonction de binding liant le contexte de pj à celui de Ad. Le *tissage après* du rdPc de l'advice Ad au point de jointure pj, noté $rdPc(Ad) \dagger_{\texttt{after}} rdPc(pj)$, est réalisé tel qu'indiqué ci-après et tel qu'illustré par la figure 4.10.

- On réalise la fusion des places dans les deux rdPc pour modéliser la liaison de contexte selon la fonction de binding donnée, tel que : $\forall (x, y) \in bind_{pj}(Ad)$,

on fusionne la place x du point de jointure à la place y de l'advice ;

- (1) On crée une transition t_{begin} reliée à la place in^{ad} par un arc sortant étiqueté par `ok` et reliée à la place out^{pj} par un arc entrant étiqueté par `weaveBegin`. (2) On crée une transition t_{end} reliée à la place out^{ad} par un arc entrant étiqueté par `ok` et reliée à la place out^{pj} par un arc sortant étiqueté par `ok`. (3) On change l'étiquette qui se trouve sur l'arc reliant t^{pj}_{out} à out^{pj} et on la remplace par `weaveBegin`.

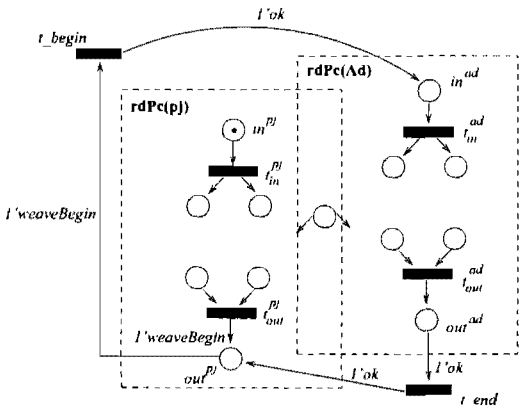


FIG. 4.10 – Patron de tissage après.

Revenons à l'exemple de l'application de téléphonie. La figure 4.11 illustre le *tissage après* de l'advice composite $rdPc(Timing.opDrop(...)) \odot rdPc(Billing.opDrop(...))$ au point de jointure modélisé par $rdPc(Connection.drop())$. Notons que la place $target_{drop}$ de $rdPc(connection.drop)$, dessinée en pointillées, a été fusionnée avec la place c de $rdPc(Timing.opDrop) \odot rdPc(Billing.opDrop)$, selon la relation de liaison de contexte définie dans le modèle Aspect-UML.

4.4.6 Construction de la sémantique d'un modèle Aspect-UML

Soit un modèle Aspect-UML composé d'un ensemble de points de jointure $PJset$, d'un ensemble de points de coupure $PCset$, d'un ensemble d'advices $Adset$

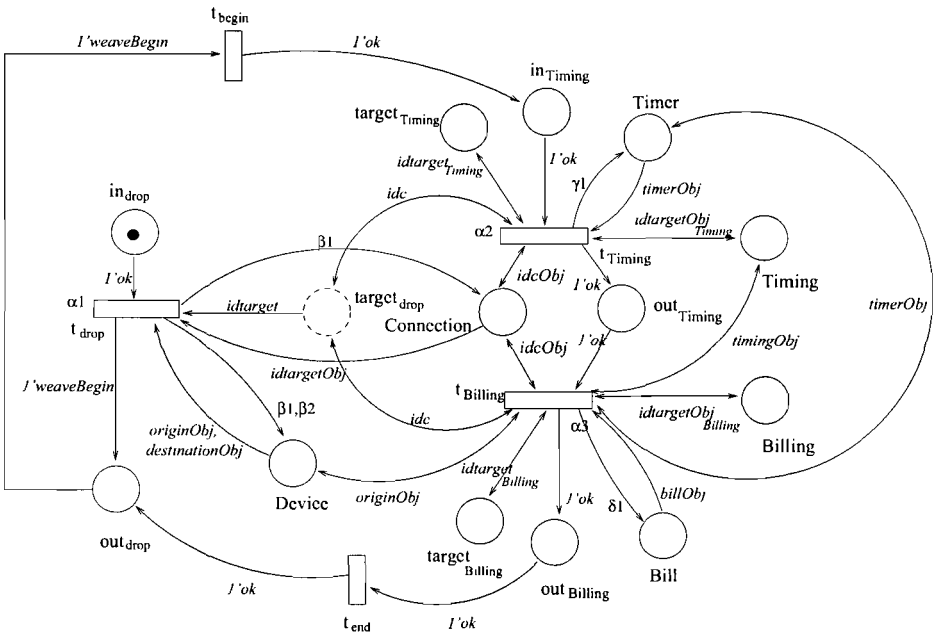


FIG. 4.11 – Le tissage avant d'un advice composite avec un point de jointure.

et d'un ensemble d'aspects $Aspects$. On considère également :

- une fonction $getPJ(pc)$ qui retourne l'ensemble des points de jointure d'un point de coupure $pc \in PCset$;
- une fonction $getPC(pj)$ qui retourne l'ensemble des points de coupure contenant $pj \in PJset$;
- une fonction $getAd(pc, \tau)$ qui retourne les advices de catégorie τ (avec $\tau \in \{\texttt{before}, \texttt{after}\}$) et qui implémente le point de coupure pc ;
- une fonction $aspectOf(ad)$ qui retourne l'aspect $a \in Aspects$ auquel appartient $ad \in Adset$.

Finalement, on dispose d'une relation de précédence "<" sur les aspects ainsi que des contraintes de binding $\Gamma(pj, pc)$ entre les arguments d'un point de jointure pj et ceux d'un point de coupure pc.

La méthode de construction de la sémantique d'un modèle Aspect-UML est la suivante :

1. On construit le réseau $rdPc(ad)$ de chaque advice $ad \in Adset$ et le réseau $rdPc(pj)$ de chaque point de jointure $pj \in PJset$ (section 4.4.2).

2. Pour chaque point de coupure $pc \in PCset$ tel que $getPJ(pc) = \psi$ et $getAd(pc, \texttt{before}) \cup getAd(pc, \texttt{after}) = \phi$, on calcule la sémantique de $sem(pc_{\phi,\psi})$ selon les contraintes de passage de contexte $\Gamma(pj, pc)_{pj \in \psi}$ indiquées (section 4.4.3).

3. Pour chaque point de jointure $pj \in PJset$, on détermine les advices qui entrecoupent pj *avant* et tous ceux qui l'entrecoupent *après*. Ainsi on construit l'ensemble $Advices(\texttt{before})^{pj} = \bigcup_{pc_i \in getPC(pj)} getAd(pc_i, \texttt{before})$. On construit l'ensemble $Advices(\texttt{after})^{pj}$ de façon similaire.

4. Pour chaque point de jointure $pj \in PJset$, on trie les deux ensembles d'advices $Advices(\texttt{before})^{pj}$ et $Advices(\texttt{after})^{pj}$ selon la relation de précédence définie entre les aspects du modèle. Ainsi on ordonne les advices selon un ordre strict partiel "$<_{ad}$" tel que $Ad_1 <_{ad} Ad_2$ ssi $aspectOf(Ad_1) < aspectOf(Ad_2)$. On dira que Ad_1 et Ad_2 sont indépendants, noté $Ad_1 \perp_{ad} Ad_2$, ssi $Ad_1 \not<_{ad} Ad_2$ and $Ad_2 \not<_{ad} Ad_1$.
Dans chaque ensemble trié, on compose d'abord les advices indépendants par entrelacement non déterministe. Ainsi pour chaque paire d'advices $Ad_1, Ad_2 \in Advices(\tau)_i^{pj}$ ($\tau \in \{\texttt{before, after}\}$ tels que $Ad_1 \perp_{ad} Ad_2$, on construit un nouvel advice $Ad = rdPc(Ad_1) \oplus rdPc(Ad_2)$ et on calcule le nouvel ensemble d'advices :

$$Advices(\tau)_{i+1}^{pj} = Advices(\tau)_i^{pj} \cup \{Ad\} \backslash \{Ad_1, Ad_2\}$$

Le nouvel advice Ad se voit attribué la même priorité que Ad_1 et Ad_2.
Une fois tous les advices de même priorité composés, on combine les advices de priorités différentes par composition séquentielle, en respectant l'ordre de précédence et en tenant compte du type de tissage (*avant* ou *après*). Ainsi pour tout $Ad_1, Ad_2 \in Advices(\tau)_i^{pj}$ tel que $Ad_1 <_{ad} Ad_2$, on construit un nouvel advice

$Ad = rdPc(Ad_1) \odot rdPc(Ad_2)$ si τ =`before` (ou composé dans l'ordre inverse si τ =`after`) dont la priorité est celle de Ad_1. Comme précédemment, l'ensemble d'advices est itérativement mis à jour pour prendre en compte le nouvel advice composite.

Les opérations de composition sont appliquées récursivement jusqu'à ce que $Advices(\tau)_i^{pj}$ (pour chaque $\tau \in \{\texttt{before}, \texttt{after}\}$) contienne au plus un advice que l'on notera $AdResult(\tau)^{pj}$ (section 4.4.4).

5. Pour chaque $pj \in PJset$, pour chaque nouvel advice composite $AdResult(\tau)^{pj}$ ($\tau \in \{\texttt{before}, \texttt{after}\}$) obtenu, on calcule la fonction de liaison de contexte $bind_{pj}(AdResult(\tau)^{pj})$ qui lie cet advice composite au point de jointure pj, *via* les points de coupure de $getPC(pj)$. Pour ce faire, on doit calculer l'union de tous les bindings des advices composant $AdResult(\tau)^{pj}$.

$$bind_{pj}(AdResult(\tau)^{pj}) = \bigcup_{\substack{\forall a, pc \text{ tel que} \\ a \in (Advices(\tau)_0^{pj} \cap getAd(pc)) \\ \text{et } (pj, b) \in sem(pc)}} b$$

6. Pour chaque $pj \in PJset$, on construit par tissage un nouveau réseau (section 4.4.5).

$$AdResult(\tau)^{pj} \dagger_\tau rdPc(pj) \qquad (\text{pour } \tau = \texttt{before} \text{ et } \tau = \texttt{after})$$

L'ensemble des rdPc ainsi obtenus constitue la sémantique du modèle Aspect-UML. On prendra soin de réduire les rdPc en éliminant entre autres les places non connectées à une transition. Il s'agira de places décrivant des arguments non utilisés et éventuellement des places de Π_{CTypes} associées au type de ces arguments.

4.5 Les réseaux de Petri orientés objet : COOPN/2

COOPN/2 [Bib97] (Concurrent Object-Oriented Petri Nets) est un langage de modélisation orienté objet appartenant à la catégorie des réseaux de Petri de haut niveau. Il permet de prendre en charge les concepts orientés objet tel que l'héritage et le sous-typage. COOPN/2 est basé sur deux formalismes que sont les algèbres partielles à sortes ordonnées [GM92] et les réseaux algébriques [Rei91], dans lesquels les places et les jetons traditionnels des réseaux de Petri sont des valeurs algébriques. D'une part, les spécifications algébriques sont utilisées pour décrire les structures de données d'un système. D'autre part, les réseaux de Petri servent à modéliser son comportement et ses aspects relatifs à la concurrence. Un problème propre à ces deux formalismes est leur manque de capacité de structuration. COOPN/2 a donc adopté l'approche orientée objet qui considère un système comme une collection d'objets indépendants qui collaborent et interagissent afin d'accomplir les fonctionnalités du système.

4.5.1 Les objets

Un objet est une entité indépendante possédant un état interne et offrant des services au monde extérieur. COOPN/2 considère un objet comme un ensemble de variables d'états et/ou d'attributs encapsulés qui peuvent être utilisés par les méthodes. Un objet COOPN/2 est défini comme un réseau algébrique encapsulé dans lequel les places représentent l'état interne de l'objet et contiennent des multi-ensembles de valeurs algébriques, alors que les transitions modélisent son comportement. Les transitions sont composées d'une précondition et d'une postcondition représentant respectivement les valeurs consommées et produites dans les places d'un objet. COOPN/2 offre deux types de transitions : les transitions externes (ou méthodes) et les transitions internes (ou invisibles). Les méthodes correspondent aux services offerts à l'extérieur, tandis que, les transitions internes décrivent le

comportement interne des objets ou ses réactions aux stimuli extérieurs. Graphiquement, comme montré par la figure 4.12 (tirée de [Bib97]), les objets sont représentés par de grands ovales contenant les places illustrées par de petits cercles. Les transitions internes sont représentées par des rectangles blancs et les méthodes par des rectangles noirs.

4.5.2 Interactions entre objets

La coopération entre objets est réalisée au moyen de synchronisations. Quand un objet nécessite un service, il demande à être synchronisé avec la méthode de l'objet fournisseur de ce service. Dans COOPN/2, les transitions ont la capacité de requérir les services d'autres objets au moyen d'une expression de synchronisation. Ces expressions de synchronisation peuvent utiliser trois opérateurs :"//" pour la simultanéité, ".." pour la séquence et "+" pour le non-déterminisme. Un objet demandeur de services a donc la possibilité d'exprimer, à l'aide de ces opérateurs, la façon dont il désire se synchroniser avec un ou plusieurs partenaires. Par exemple, l'expression de synchronisation m *with* $(m_1..m_2)//(m_3+m_4)$ signifie que la transition m a lieu si $(m_1..m_2)//(m_3+m_4)$ peut être exécutée par le système ce qui veut dire que m_1 suivie immédiatement par m_2 est exécutée simultanément soit avec m_3 soit avec m_4.

Graphiquement, comme montré par la figure 4.12, la synchronisation entre les transitions est schématisée par des lignes discontinues. Dans l'exemple du producteur/consommateur modélisé dans la figure 4.12, la méthode *deliver* de la classe *Producer* doit se synchroniser avec la méthode *put* de $Buffer$ et la méthode *receive* de la classe *Consumer* doit se synchroniser avec la méthode *get* de $Buffer$.

4.5.3 Spécification COOPN/2

Une spécification COOPN/2 est une collection d'objets communicants. Formellement, elle est constituée de deux sortes de modules : les modules de type de

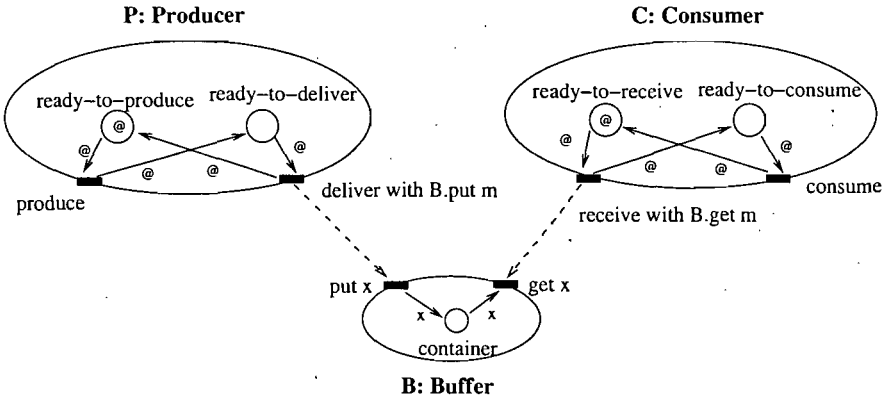


FIG. 4.12 – Représentation graphique du producteur/consommateur modélisé en COOPN/2 [Bib97].

données abstrait ou ADT (Abstract Data Type) et les modules de classe.

Module ADT. Les modules ADT servent à décrire les structures de données de la spécification. Les algèbres de sortes ordonnées sont utilisées pour décrire ce type de modules. L'interface d'un module ADT introduit un ensemble de sortes ainsi qu'un ensemble de constructeurs et d'opérations sur ces sortes. Le corps du module ADT contient les axiomes décrivant les propriétés des opérations définies par le module.

Module de classe. Un module de classe est une sorte de réseau algébrique servant à décrire un groupe d'objets possédant un état et fournissant des services à l'extérieur. Tout comme un module ADT, un module de classe contient (1) une interface (`Interface`) qui décrit la partie du module visible de l'extérieur (*c-à-d* tous les éléments du module qui sont accessibles) et (2) un corps (`Body`) qui définit les éléments locaux non visibles de l'extérieur. Un module de classe contient les champs suivants :

- **Type** définit explicitement le type des objets ;
- **Objects** définit les noms des objets créés statiquement ;
- **Methods** décrit les noms des méthodes avec leur profil ;
- **Places** énumère les places de la classe avec leur type ;
- **Axioms** regroupe la description des transitions de la classe. Ces descriptions

sont appelées formules comportementales et sont de la forme :

Event [**with** *Sync*] : : [*Cond* => *Pre* − > *Post*], où :

Event est le nom de la transition ; *Syn* est une expression de synchronisation optionnelle ; *Cond* est une condition optionnelle sur les valeurs algébriques ; *Pre* et *Post* sont les pré et postconditions de la transition.

```
Class Producer;
Interface
  Use BlackTocken, Message, Buffer;
  Type producer;
  Object P : producer;
  Method produce, deliver;
Body
  Places
    ready–to–produce _, ready–to–deliver_ : blackTocken;
  Initial
    ready–to–produce @ ;
Axioms
  produce : : ready–to–produce @ –> ready–to–deliver @;
  deliver with B.put m : : ready–to–deliver @ –> ready–to–produce @;
  where m : message, B : buffer;
End Producer;
```

FIG. 4.13 – Spécification COOPN/2 de la classe *Producer* de l'exmple du producteur/consommateur [Bib97].

La spécification de la figure 4.13 (tirée de [Bib97]) décrit le module de classe de la classe *Producer* de l'exemple du producteur/consommateur (figure 4.12). Entre autres, la spécification indique la valeur initiale de la place $ready - to - produce$ (cf. champ **Initial**) et les axiomes décrivant les méthodes *produce* et *deliver*. L'axiome décrivant la méthode *produce*, indique que pour s'exécuter cette méthode consomme un jeton de la place $ready - to - produce$ et produit un jeton dans la place $ready - to - deliver$. Quant à l'axiome décrivant la méthode *deliver*, il indique que pour s'exécuter la méthode *deliver* doit se synchroniser avec la méthode *put* de la classe $Buffer$. Notons que l'expression de synchronosation utilisée, dans ce

cas, ne comporte aucun opérateur de synchronisation puisque seule la méthode *put* est impliquée dans la synchronisation de la méthode *deliver*.

En COOPN/2 chaque objet possède une identité propre qui lui est attribué lors de sa création. Le mot réservé `self` dénote l'identité de l'objet courant. La création dynamique d'objets s'effectue par la synchronisation avec une méthode particulière `create`. L'invocation de `o.create` a donc pour effet la création d'un objet du type de la variable `o`. Une méthode de création peut également être définie sous la rubrique `Creation` du module de classe.

Par ailleurs, COOPN/2 distingue le notion d'héritage de celle de sous-typage. Ces deux relations sont à déclarer explicitement dans les modules de classe. La relation de sous-typage est à définir sous la rubrique `Subtype`, alors que la relation d'héritage est à définir sous la section `Inherit`. Cette section permet de déclarer les modules hérités et les modifications à effectuer comme le renommage et la redéfinition.

4.6 Sémantique de Aspect-UML en COOPN/2

Rappelons que la traduction vers les rdPc que nous avons présentée à la section 4.4 fait appel au langage fonctionnel ML [MTHM96] pour la spécification des données. Dans cette traduction, nous avons dû considérer un passage d'un monde objet/aspect vers un monde fonctionnel; un passage qui malheureusement est assez complexe et délicat. Cette complexité de la traduction, nous amène, dans un second temps, à considérer les réseaux de Petri orientés objet, tels qu'incarnés par le formalisme COOPN/2 [Bib97]. Il nous semble, en fait, plus naturel d'opter pour ce type de réseaux de Petri dits orientés objet plutôt qu'*à la ML*, pour pallier la différence des paradigmes objet/aspect et fonctionnel. En effet, étant orienté objet, COOPN/2 nous permettra d'exprimer plus facilement les interactions et la communication entre les objets et les aspects tissés d'un modèle Aspect-UML.

Dans cette section, nous présentons la construction d'une spécification COOPN/2 qui formalise un modèle Aspect-UML, telle que nous l'avons proposée sommairement dans [MV06b]. Rappelons que pour la construction de cette spécification, nous disposons déjà dans le langage COOPN/2 des concepts de module ADT, module de classe et des opérations de synchronisation.

4.6.1 Spécification des types de données abstraits

Chaque type de structure de données (`integer`, `boolean`, etc.) utilisé dans un modèle Aspect-UML est décrit au moyen d'un module ADT dans la spécification COOPN/2. Cependant comme déjà discuté à la section 4.4.1.1, certains types de données abstraits (comme `integer`), dénotent des ensembles infinis de valeur, et font ainsi exploser rapidement le nombre des états des systèmes les utilisant. Pour limiter le problème d'explosion, nous recourrons aussi à la représentation symbolique comme expliqué à la section 4.4.1.1. Cette solution consiste à décrire les états du système au moyen de valeurs symboliques au lieu d'étayer l'infinité de combinaisons des valeurs concrètes correspondantes. Ainsi, pour représenter symboliquement ces types de données, nous introduisons un module générique de type de données abstrait appelé `SymbolicData` qui définit une liste d'intervalles. La spécification de cet ADT s'appuie sur l'arithmétique des intervalles [Moo77].

Une autre structure de données, nécessaire lorsque nous manipulons des réseaux de Petri, est le classique jeton noir anonyme. Ce type appelé `BlackToken` est dénoté par la constante @.

4.6.2 Spécification des classes et des aspects

Un modèle Aspect-UML est traduit vers une spécification COOPN/2 dans laquelle le tissage des aspects au niveau des points de jointure est décrit explicitement. Donc, avant de décrire les classes et les aspects, expliquons d'abord comment se fait le processus de tissage.

Pour expliciter le tissage, dans la spécification COOPN/2, nous faisons appel au patron de conception *Pont*(*Bridge design pattern*) [GHJV95]. Ce patron, dont la structure est montrée par la figure 4.14, propose de découpler une abstraction (`Abstraction`) de son implémentation (`Implementor`), afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre. Ce patron permet de modifier à loisir l'implémentation d'une opération voire même de choisir l'implémentation désirée lors de l'exécution.

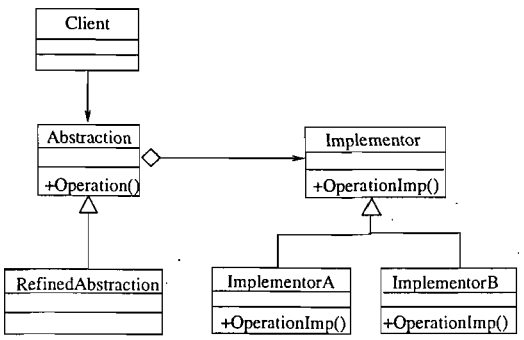


FIG. 4.14 – Structure du patron de conception *Pont*.

Le tissage doit être explicité au niveau de chaque point de jointure. Ainsi, chaque classe/aspect du modèle Aspect-UML contenant un ou plusieurs points de jointure sera traduite vers COOPN/2 tout en la restructurant de façon à réaliser le patron de conception *Pont* de la façon suivante : le tissage des advices se fait au niveau de la couche abstraite (`Abstraction`) alors que le comportement original de la classe est maintenu par la couche implémentation (`Implementor`). Les clients interagissent avec la couche abstraite qui, à son tour, utilise les services de la couche implémentation pour offrir les services demandés, tout en invoquant les advices qui ont été tissés.

Une classe (ou un aspect) C d'un modèle Aspect-UML sera traduite en COOPN/2 ainsi :

- Si C ne contient aucun point de jointure alors elle sera décrite par un module de classe de même nom, où les transitions externes sont les méthodes (et/ou

advices) de la classe/aspect et les places correspondent aux attributs de la classe/aspect.

- Si C contient un ou plusieurs points de jointure alors elle sera traduite vers deux modules de classe *AbstractC* et *CState* qui spécifient respectivement la couche abstraction et la couche implémentation de C réalisant le patron *Pont*. Étant donné que le comportement original de C est maintenu par *CState* alors les transitions et les places de ce module doivent spécifier respectivement les méthodes (et/ou advices) et les attributs de C. Par ailleurs, comme le module *AbstractC* spécifie le tissage aux points de jointure alors ses transitions doivent spécifier les points de jointure de C. Un exemple d'une telle traduction est donné plus loin dans la section 4.6.4.
- Les axiomes décrivent le comportement des opérations de la classe/aspect C. En particulier, les pré et postconditions associées aux opérations (points de jointure et advices) dans le modèle Aspect-UML pourront naturellement être exprimées dans les axiomes décrivant ces opérations (des exemples de tels axiomes sont donnés dans la spécification COOPN/2 décrivant l'exemple de téléphonie (figure 4.16, page 116)).

4.6.3 Tissage des advices à un point de jointure

À un point de jointure, plusieurs advices peuvent être candidats à exécution. Ces advices sont statiquement conflictuels s'ils sont de même catégorie (`before` ou `after`). Selon notre approche de modélisation Aspect-UML, une relation de précédence peut être définie entre les aspects contenants ces advices. Ainsi, si une telle relation est définie alors les advices doivent être tissés séquentiellement au point de jointure selon l'ordre spécifié. Autrement, les advices sont tissés de façon non déterministe l'un après l'autre. Le tissage en séquence et le tissage non déterministe sont exprimés respectivement au moyen des opérateurs de synchronisation ". ." et "||" offerts par la langage COOPN/2 (cf. section 4.5). En fait, une expression de

synchronisation est ajoutée à chaque axiome décrivant un point de jointure. Ces expressions de synchronisation décrivent (1) les advices impliqués dans le tissage et (2) les opérateurs de synchronisation requis. Remarquons, que ces expressions de synchronisation sont définies dans les classes du modèle réalisant la couche abstraction (du patron *Pont*), puisque c'est à ce niveau que ce fait le tissage (voir l'exemple de la figure 4.15, page 114).

4.6.4 Application à l'exemple de téléphonie

Pour expliquer la traduction d'un modèle Aspect-UML vers une spécification COOPN/2, considérons notre exemple de l'application de téléphonie dont le modèle Aspect-UML est donné par la figure 3.4, page 72.

Les types de donnés utilisés dans ce modèle Aspect-UML sont traduits vers des modules ADT et les classes, les aspects et les interfaces « Pointcut » sont traduits vers des modules de classe (voir l'annexe I pour la spécification complète en COOPN/2 de cette étude de cas).

```
Class AbstractConnection;
Interface
  Use NullConnection;
  Type abstractConnection;
  Subtype abstractConnection < nullConnection;
  Methods complete, drop;
  Creation create-connection;
Body
  Use ConnectionState, Timing_Aspect, Billing_Aspect;
  Places
    state_ : connectionState;
Axioms
  complete with s.complete . . Timing.opComplete self : : state s -> state s ;
  drop with s.drop . . (Timing.opComplete self , . Billing.opDrop self) : : state s -> state s ;
  create-connection with s.create-connectionState self : : -> state s, state s, state s;
  where s : connectionState;
End AbstractConnection;
```

FIG. 4.15 – Spécification COOPN/2 du module de classe *AbstractConnection*.

Remarquons que le modèle Aspect-UML de l'application de téléphonie définit deux points de jointure `complete()` et `drop()` qui sont contenus dans la classe `Connection`. Pour expliciter le tissage au niveau de ces deux points de jointure lors de la traduction vers COOPN/2, la classe `Connection` sera restructurée afin de réaliser le patron de conception *Pont*. D'une part, nous définissons le module de classe `AbstractConnection` pour réaliser la couche abstraction de la classe `Connection`. C'est avec cette couche que les clients interagissent et c'est ce module `AbstractConnection` qui est entrecoupé par les aspects `Timing` et `Billing`. D'autre part, nous introduisons le module de classe `ConnectionState` pour réaliser la couche implémentation de `Connection`. C'est lui qui se charge de fournir les services de la classe `Connection`, il n'est accessible que depuis la couche abstraite, c'est-à-dire le module `AbstractConnection`. La spécification COOPN/2 des deux modules de classe `AbstractConnection` et `ConnectionState` est donnée respectivement par les figures 4.15 et 4.16.

Graphiquement, les différents objets de l'application de téléphonie sont schématisés par la figure 4.17. Notons que par souci de clarté et de lisibilité, seules les synchronisations de la méthode `drop()` sont montrées. En particulier, les pré et postconditions du point de jointure `drop()` sont spécifiées, elles montrent en fait, les conditions d'activation et l'effet de la transition exécutant ce point de jointure. En effet, pour vérifier les préconditions et assurer les postconditions du point de jointure `drop` (telles que déclarées dans le modèle Aspect-UML), la transition `drop()` de l'objet `s :ConnectionState` doit se synchroniser en séquence, d'abord avec les méthodes `getD_status()` et `getCurrent` des instances représentant l'origine et la destination de l'objet *s* (*c-à-d* la connexion en cours) visé par l'invocation de `drop()` et ensuite avec les méthodes `setD_status()` et `setCurrent` de ces mêmes instances.

Par ailleurs, notons que la transition `drop()` du module de classe `AbstractConnection` doit exprimer les synchronisations nécessaires pour le tissage des aspects

```
Class ConnectionState;
Interface
  Use AbstractConnection;
  Type connectionState;
  Methods complete, drop;
  Creation create-connectionState_ : abstractConnection;
Body
  Use Device, Status, D_Status ;
  Places
    connect_ : abstractConnection ;
    status_ : status ;
    origin_, destination_ : device ;
Axioms
  complete with (o.getD_status idle || o.getCurrent Null) . .
                (o.setD_status busy || o.setCurrent c || d.setD_status busy || d.setCurrent c)
       : : => origin o, destination d, status disconnected, connect c
              -> origin o, destination d, status connected, connect c ;
  drop with (o.getD_status busy || o.getCurrent c || d.getD_status busy || d.getCurrent c ) . .
            (o.setD_status idle || o.setCurrent Null || d.setD_status idle || d.setCurrent Null)
       : : => origin o, destination d, status connected, connect c
              -> origin o, destination d, status disconnected, connect c ;
  create-connectionState c : : -> status disconnected, connect c ;
  where o, d : device, c: abstractConnection ;
End ConnectionState;
```

FIG. 4.16 – Spécification COOPN/2 du module de classe *ConnectionState*.

Timing et Billing. Remarquons l'opération de synchronisation utilisée pour accomplir le processus de tissage. En supposant qu'une relation de précédence, telle que `Timing < Billing`, est explicitement définie dans le modèle Aspect-UML, les advices `Timing,opDrop()` et `Billing.opDrop()` sont synchronisés en séquence avec la transition `drop()`. En outre, rappelons que le passage de contexte décrit par les contraintes du modèle Aspect-UML est implémenté au moyen du mécanisme d'unification fourni par le langage COOPN/2. En effet, les paramètres formels et effectifs sont unifiés lors de la synchronisation. La spécification COOPN/2 complète pour cette étude de cas est présentée à l'annexe I.

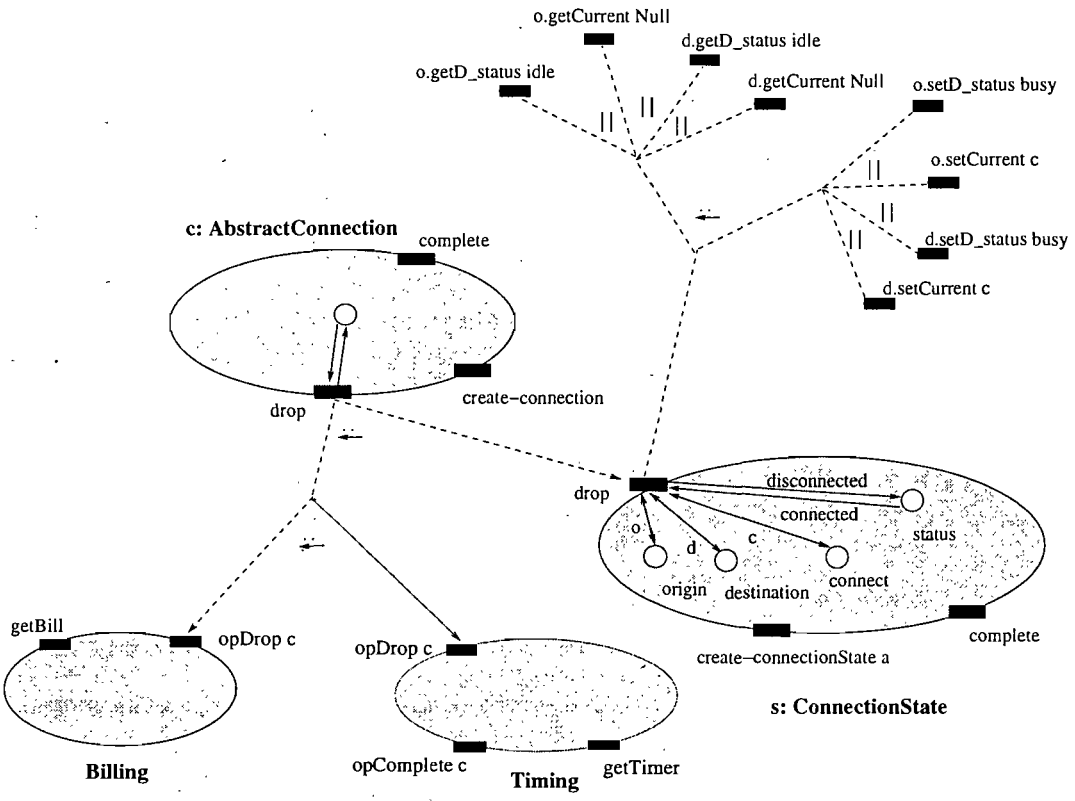


FIG. 4.17 – Synchronisations des objets pour réaliser le tissage au point de jointure `drop()`.

4.7 Vérification de la composition et du tissage des aspects

Nous nous intéressons à identifier les erreurs liées à la composition et au tissage des aspects dans un modèle Aspect-UML décrit à l'aide de réseaux de Petri. Le but étant de fournir un outil de détection d'erreurs signalant aux utilisateurs les possibles anomalies dans la composition et le tissage des aspects.

4.7.1 Vérification des réseaux de Petri colorés

Soit $\delta : M \rightarrow rdPc$ une fonction sémantique qui prend un modèle Aspect-UML et retourne le réseau de Petri coloré correspondant (tel qu'expliqué à la section 4.4). Si l'on souhaite vérifier que les interactions des aspects présents dans le modèle

Aspect-UML ne causent pas d'interférences (effets indésirables) on procède comme suit : (1) la construction du réseau de Petri coloré $\delta(M)$ décrivant le modèle Aspect-UML M à vérifier, (2) la formulation en logique temporelle des propriétés que le système devrait vérifier même après l'ajout d'aspects, (3) la génération du graphe de marquage du rdPc et son éventuelle optimisation, puis (4) la détection automatique d'anomalies dans la composition des aspects en vérifiant, par analyse et/ou model-checking [BBF+01], les propriétés sur le graphe de marquages.

Etant donné un ensemble de propriétés à vérifier, la correction d'un modèle Aspect-UML est ici directement dépendante du tissage des aspects dans le système de base et de leur composition. Nous portons notre attention sur deux types d'erreurs pouvant compromettre la correction du système de base :

1. *La violation d'une propriété globale du système de base.* Nous entendons par propriété globale une propriété décrivant un invariant d'état du système. Ces propriétés sont formulées en logique temporelle, puis vérifiées par model-checking sur le graphe de marquages du rdPc.
2. *La violation d'une propriété locale du système.* Une propriété locale est relative à une opération (tel qu'un point de jointure ou un advice), elle est exprimé par les pré et postconditions de l'opération. La violation d'une telle propriété engendrera nécessairement un état de blocage indésirable dans le réseau de Petri. L'ajout d'un aspect peut effectivement bloquer le système que ce soit à cause d'une interférence avec le système de base due au tissage, ou d'un problème de composition avec un autre aspect. Les états de blocage peuvent être détectés par l'analyse du rdPc avec l'outil CPN Tools.

Dans le cas de l'application de téléphonie, une propriété à vérifier peut être la suivante : *lorsqu'une connexion est terminée, elle doit immédiatement passer à l'état déconnecté*, autrement dit, la post-condition $status = disconnected$ au point de jointure *Connection* :: *drop*() doit toujours être vérifiée, même après le tissage des aspects *Timing* et *Billing*. Cette propriété peut être décrite en logique tempo-

relle et vérifiée par model-checking sur le graphe des marquages du rdPc. Quant au problème des blocages indésirables, on peut les repérer en appliquant les techniques de détection de blocages propres aux réseaux de Petri. L'utilisateur devra préalablement avoir défini un prédicat permettant de distinguer les états bloquants acceptables de ceux qui sont indésirables. Par exemple, dans notre application de téléphonie, si la contrainte de précédence $Timing < Billing$ n'est pas définie alors le rdPc obtenu aurait effectivement présenté un état de blocage indésirable que l'algorithme de détection aurait heureusement révélé. En effet, une des exécutions possibles du rdPc, où *Billing* est exécuté avant *Timing*, aurait été bloquante étant donné l'impossibilité de franchir la transition $t_{Billing}$ avec une valeur nulle du temps de connexion (voir le réseau de Petri coloré, de la figure 4.11, page 103, décrivant le tissage de *Timing* et *Billing* au point de jointure *drop*()).

4.7.2 Vérification de la spécification COOPN/2

Tel que nous l'avons déjà mentionné, les réseaux de Petri orientés objet incarnés par le formalisme COOPN/2 ne disposent pas de nos jours d'outil de vérification automatisé. Néanmoins, on peut exploiter la sémantique dénotationnelle de COOPN/2, telle que décrite dans [Bib97], pour atteindre nos objectifs de vérification. En effet dans [Bib97], l'auteur décrit la sémantique d'une spécification COOPN/2 sous forme de système de transitions étiqueté. Ainsi, une fois ce système de transitions construit à partir de la spécification COOPN/2 décrivant un modèle Aspect-UML, celui-ci peut être vérifié automatiquement pour détecter les erreurs dues à la composition et au tissage des aspects. Comme dans le cas des réseaux de Petri colorés, la vérification des propriétés globales se fait par model-checking sur le système de transitions obtenu. De même, la vérification des propriétés locales se fait par détection des états de blocage indésirables dans le système de transitions.

4.8 Discussion

Beaucoup d'efforts dans la communauté Aspect ont été consentis afin de doter le paradigme aspect d'une sémantique formelle. Par exemple, [And01] utilise les algèbres de processus pour donner un fondement formel à la programmation orientée aspect ; [RJ03, WZL03] proposent une sémantique opérationnelle afin de donner une signification formelle aux différents points de coupure et advices définis dans AspectJ ; [WKD04] propose une sémantique dénotationnelle pour un sous-ensemble de AspectJ ; [Lam02] formalise les points de jointure correspondant à l'interception d'appels de méthodes ; et [DFS04] propose un langage expressif et formel pour la définition de l'entrecoupement. Bien que ces modèles formels puissent tout à fait servir de bases pour décrire formellement les systèmes par aspects, à notre connaissance la plupart des sémantiques proposées ne traitent pas le problème épineux lié aux interactions indésirables entre aspects, et ne fournissent pas réellement de méthodologie de vérification formelle pour sa résolution. En revanche, la sémantique que nous proposons offre un cadre formel intéressant permettant de raisonner sur la composition des aspects, la vérification formelle des interactions entre aspects et la détection des conflits. En effet, la formalisation de Aspect-UML en termes de réseaux de Petri ouvre la voie à la simulation et au model-checking de propriétés intéressantes dans les systèmes par aspects.

Nous avons proposé, dans un premier temps, d'exprimer ce modèle sémantique à l'aide de réseaux de Petri colorés. En plus de disposer de plusieurs outils de simulation et de techniques d'analyse automatique, les réseaux colorés s'avèrent aussi bien adaptés à la description des systèmes séquentiels que des systèmes concurrents. Nous avons choisi ce formalisme pour modéliser le tissage et la composition des aspects. L'avantage majeur que représente l'existance d'outils automatiques de vérification pour ce type de réseaux, a fortement guidé notre choix, vu que notre objectif derrière cette sémantique est la vérification de la composition des

aspects. En effet, nous avons privilégié le choix d'un type de réseaux de Petri colorés supporté par un outil de simulation et d'analyse qui a fait ses preuves, soit CPN Tools [KCJ98]. Par ailleurs, il est utile de rappeler que ces réseaux font appel au langage fonctionnel ML [MTHM96] pour la spécification des données. Cependant, vue la disparité des paradigmes objet/aspect et fonctionnel, la traduction de Aspect-UML vers le formalisme des rdPc tel qu'expliquée à la section 4.4 est assez complexe et délicate.

Cette complexité de la traduction nous a amené, dans un second temps, au choix du formalisme COOPN/2 pour définir la sémantique de Aspect-UML. Il nous a, en fait, semblé plus naturel d'opter pour ce type de réseaux de Petri dits orientés objet plutôt qu'*à la ML*, pour pallier la différence des paradigmes objet/aspect et fonctionnel. En effet, tel qu'expliqué à la section 4.6 la traduction est simple et directe, vu l'orientation objet des deux langages origine (Aspect-UML) et cible (COOPN/2). Or le gain en abstraction et en facilité d'expression se fait ici au détriment de la complexité de l'analyse et de la vérification des systèmes décrits. À notre connaissance, il ne semble pas y avoir d'outils de vérification satisfaisants et performants pour ces réseaux.

Dans la traduction proposée vers les réseaux de Petri, que ce soit vers les réseaux colorés ou les réseaux dits orientés objet, nous avons proposé des abstractions quant à la représentation des données (fc. section 4.4.1.1) afin de limiter le problème d'explosion d'états. Nous avons, en fait, choisi des approximations (abstractions) de telle sorte que l'incertitude induite n'engendre pas d'imprécision ou d'inexactitude des propriétés à démontrer. Bien qu'on y perde en précision absolue, cette approche nous permet jusqu'à un certain degré d'atteindre l'objectif de vérification en utilisant les réseaux de Petri colorés. En effet, nous pensons que malgré les abstractions proposées, la vérification de grands systèmes verra facilement exploser leur nombre d'états.

En résumé, les réseaux de Petri nous permettent de définir une sémantique

claire et naturelle de Aspect-UML, mais malheureusement ils sont vite confrontés au problème d'explosion d'états lors de la vérification automatique. Par conséquent, pour contourner cette limite et pour parvenir à nos objectifs de vérification, nous avons opté, dans le cadre de notre travail, pour l'utilisation de l'outil Alloy [All] ; un analyseur de modèles basé sur la logique du premier ordre. Les deux prochains chapitres seront consacrés respectivement à la traduction de Aspect-UML vers Alloy et à la vérification des interactions entre aspects en utilisant Alloy.

4.9 Conclusion

Dans ce chapitre nous avons proposé une formalisation de notre profil Aspect-UML en termes de réseaux de Petri. Les réseaux de Petri étant un modèle formel avec des fondements mathématiques rigoureux et une sémantique précise, nous offrons ainsi un cadre formel à notre méthodologie de développement. En effet, en dotant Aspect-UML de cette sémantique, nous ouvrons la voie à la simulation et à la vérification formelle de propriétés intéressantes des systèmes par aspects. Cependant, comme nous l'avons discuté plus haut, les deux types de réseaux de Petri choisis (colorés et orientés objet) présentent tous les deux des avantages essentiels et des inconvénients, hélas, limitatifs quant à notre objectif de recherche, qui se focalise sur la vérification des interactions dans les systèmes par aspects. Nous proposons, ainsi, dans le chapitre suivant de formaliser notre profil Aspect-UML dans le langage Alloy [Jac06] et ce afin de mettre à profit les principaux avantages offerts par l'outil Alloy [All] dans la vérification des interactions dues aux aspects.

CHAPITRE 5

FORMALISATION DE ASPECT-UML EN ALLOY

5.1 Introduction

Dans le chapitre précédent, nous avons proposé une traduction de notre profil Aspect-UML en réseaux de Petri. La traduction proposée sert de base pour la formalisation des concepts Aspect et pour la définition d'une sémantique claire et naturelle de Aspect-UML. Cependant, une limite que nous avons soulevée à l'utilisation des réseaux de Petri concerne la vérification automatique. En effet, d'une part, tout comme la majorité des outils de vérification utilisant la méthode de vérification basée sur les modèles, l'outil CPN Tools supportant les réseaux de Petri colorés se voit rapidement confronté au problème d'explosion d'états. D'autre part, les réseaux de Petri orientés objet ne sont pas soutenus par un outil de vérification automatique. Ainsi, pour contourner cette limite et pour atteindre nos objectifs de vérification, nous avons pensé à utiliser l'outil Alloy [All, Jac06]. En effet, étant conçu pour la spécification formelle et se prêtant bien à l'analyse des modèles par objets, Alloy nous a paru rapidement comme étant l'outil approprié pour accomplir nos objectifs de vérification formelle, à savoir la vérification des interactions entre aspects dans les modèles Aspect-UML. Par ailleurs, Alloy est un langage de style déclaratif tout comme Aspect-UML qui nous offre des possibilités de vérification intéressantes. Il dispose, en effet, d'un outil de vérification qui permet de contourner le problème d'explosion d'état. D'une part, il procède pour la vérification par résolution de contraintes plutôt que par énumération des états. D'autre part, il permet à l'utilisateur de limiter la taille des modèles à analyser.

Plus précisément, Alloy est doté d'une sémantique mathématique simple et uniforme et d'une syntaxe facile (tout est basé sur la notion de *relation* ; il n'offre pas

de constructions spéciales pour définir les machines d'états, les traces, la synchronisation, la concurrence, etc.). Même s'il n'est pas orienté objet, il définit des concepts et des mécanismes qui peuvent facilement simuler les concepts du paradigme aspect/objet. De plus, il est assez expressif pour décrire la composition des flots de contrôle, même s'il ne dispose pas directement d'opérateurs dédiés. D'autre part, il est soutenu par un outil [All] d'analyse sémantique efficace et entièrement automatisée, qui offre à la fois des représentations textuelles et graphiques. Par ailleurs, selon [Jac02] "*Alloy est un langage assez expressif pour saisir les propriétés les plus complexes tout en demeurant commode et adéquat pour une analyse efficace*".

En nous inspirant de notre traduction de Aspect-UML vers les réseaux de Petri exposée au chapitre précédent, nous proposons dans ce chapitre la traduction formelle de Aspect-UML vers Alloy. Nous y présenterons d'abord le langage formel Alloy ainsi que son analyseur associé. Ensuite, nous donnerons la syntaxe abstraite de Alloy et la syntaxe abstraite de Aspect-UML, sur la base desquelles sera définie la traduction. Par la suite, les étapes de traduction d'un modèle Aspect-UML vers une spécification Alloy seront expliquées, présentées sous forme de règles formelles et illustrées sur notre exemple d'application de téléphonie déjà utilisé dans les deux chapitres précédents.

5.2 Le langage de modélisation Alloy

Alloy [All, Jac02, Jac06] est un langage de modélisation structurelle basé sur la logique du premier ordre. Conçu pour la spécification formelle et se prêtant bien à l'analyse des modèles par objets, Alloy[1] a été développé à partir d'idées inspirées de Z [Spi92], et les différentes tentatives de formaliser la modélisation objet (une comparaison de Alloy et de Z se trouve dans [Jac99]). Alloy est un langage de modélisation, offrant à la fois des représentations textuelles et graphiques. Il est

[1] Alloy a été développé par le *Software Design Group* du MIT.

doté d'une sémantique mathématique simple et uniforme et d'une syntaxe naturelle à utiliser. De plus, il est soutenu par un outil d'analyse sémantique efficace et entièrement automatisé.

5.2.1 Analyseur Alloy

L'analyseur Alloy est un outil basé sur les algorithmes de satisfiabilité des solveurs SAT (tel que Berkmin, mChaff, zChaff, ou RelSAT). Pour analyser un modèle, Alloy convertit l'ensemble des formules logiques le décrivant en une forme normale conjonctive (CNF) [Jac00]. L'expression résultante forme une instance du problème de satisfiabilité (SAT). L'analyseur utilise alors un solveur SAT pour rechercher de façon exhaustive une solution pour la formule SAT. Si le solveur trouve une solution à la formule, l'analyseur Alloy la reconvertit en Alloy et rapporte le modèle résultant à l'utilisateur.

Contrairement à la majorité des outils de vérification utilisant la méthode de vérification basée sur les modèles, l'outil Alloy ne se voit pas confronté au problème d'explosion d'états. D'une part, il procède pour la vérification par résolution de contraintes plutôt que par énumération des états. D'autre part, il ne considère que des modèles de petite taille et ce, en fixant au préalable la portée [2] (*c-à-d* le nombre d'atomes pour chaque type de données) des modèles à analyser. L'analyse des modèles avec Alloy repose, en effet, sur l'hypothèse fondamentale *de la petite portée* (*the small scope hypothesis*) [Jac06]. Cette hypothèse stipule que les réponses négatives d'une analyse ont tendance à apparaître déjà dans les petites instances de modèles. L'absence d'erreurs dans les petites instances peut ainsi renforcer et augmenter notre assurance quant à la correction effective du modèle. En effet, selon [Jac06] la plupart des défauts dans les modèles peuvent être déjà observés dans les petites instances, puisque ces défauts sont très souvent le résultat de certains

[2] La portée est définie soit par l'utilisateur, soit fixée par défaut.

éléments traités et développés incorrectement dès le départ. Même si ces détails appartiennent à une grande instance, Alloy trouvera, en fait, ces défauts déjà dans les petites instances. La stratégie recommandée lors de l'utilisation de l'analyseur Alloy consiste à toujours commencer avec une petite portée d'analyse et de l'augmenter graduellement jusqu'à ce qu'une faute soit trouvée ou jusqu'à ce qu'une approximation satisfaïsante soit atteinte. Alloy supporte les deux types d'analyse automatique suivantes :

1. **La simulation.** Elle est utilisée pour démontrer la satisfiabilité d'un prédicat donné en l'exécutant avec la commande *run* de l'analyseur. Les exécutions d'un prédicat génèrent des exemples d'instances qui satisfont les contraintes imposées par le prédicat. Autrement dit, *run* tente de trouver une instance de modèle qui satisfait le prédicat. Si l'exécution d'un prédicat ne produit aucune instance, alors soit que le prédicat est toujours faux (non valide) peu importe le modèle ou alors que le modèle est sur-contraint (*c-à-d* il y a des contradictions dans ses contraintes).
2. **La vérification.** Elle est utilisée pour prouver la validité des assertions en essayant de générer des contre-exemples. La commande *check* de l'analyseur Alloy, est utilisée pour exécuter les assertions. Autrement dit, *check* tente de vérifier que toutes les instances du modèle satisfont l'assertion. Si une assertion n'est pas valide, son exécution génère un contrexemple.

Comme nous l'avons expliqué plus haut, ces deux types d'analyse sont effectués à l'intérieur d'une portée définie par l'utilisateur. Quand l'analyseur Alloy trouve un contrexemple alors l'assertion est nécessairement fausse (violée), autrement, aucune conclusion ne peut être tirée au sujet de sa validité sur des modèles de taille supérieure. L'utilisateur peut toutefois augmenter la taille des instances pour une prochaine analyse.

5.2.2 Éléments d'un modèle Alloy

Une spécification Alloy est structurée en modules simples qui peuvent être réutilisables. Chaque module contient une séquence de fragments appelés *paragraphes* qui sont utilisés pour définir les types de base, les contraintes et les commandes d'un modèle Alloy. Dans ce qui suit, nous présentons ces différents paragraphes. Dans cette section, nous considérons un exemple simple de spécification (tiré de [Jac06]) décrivant un annuaire électronique afin d'illustrer les principaux concepts du langage et leur syntaxe.

5.2.2.1 Signatures

Les signatures décrivent les données de l'univers à modéliser et sa structure. Une signature définit un type de base et une collection de relations appelées champs. Les relations représentent des ensembles de tuples d'atomes des types de base. Par exemple, pour spécifier notre annuaire téléphonique, nous commençons par définir les ensembles d'atomes qui seront utilisés pour dénoter les noms et les adresses. Dans Alloy, de tels ensembles d'atomes sont définis en utilisant des signatures comme suit :

sig Name, Addr { }

À chaque signature déclarée est implicitement associé un type de base qui peut être utilisé dans d'autres déclarations. Ainsi, disposant de types pour les noms et les adresses, nous pouvons à présent spécifier ce qui constitue un annuaire. Sachant qu'un annuaire consiste en un mapping des noms vers les adresses, nous introduisons une nouvelle signature appelée `Book` contenant un champ `addr` qui relie les noms aux adresses.

sig Book { addr : Name $->$ *lone* Addr }

En fait, addr est une relation ternaire qui relie les annuaires (Book), les noms (Name) et les adresses (Addr). $(b, n, a) \in$ addr signifie que, selon addr, le nom n est relié à l'adresse a dans l'annuaire b. Le mot clé lone dans la déclaration indique la multiplicité de la relation. Dans ce cas, chaque nom est relié à au plus une adresse.

Par ailleurs, Alloy permet la définition de sous-signatures comme des extensions de signatures de base déjà définies. Les sous-signatures définissent des sous-types qui peuvent être aussi étendus à leur tour. Comme spécifié dans la définition qui suit, BookWithSpamAlert est un sous-ensemble de Book. En fait, BookWithSpamAlert est un annuaire spécial qui étend la définition d'un annuaire en ajoutant un ensemble d'adresses considérées comme des spams.

sig BookWithSpamAlert *extends* Book { spams : *set* Addr }

5.2.2.2 Faits

Un fait (*fact*) peut restreindre explicitement les relations définies par les signatures et ainsi limiter les valeurs possibles qu'elles peuvent contenir. Dans Alloy, un fait est un invariant et est donc toujours vrai, *c-à-d* les faits sont satisfaits par toutes les instances du modèle. Des faits conflictuels engendrent un modèle incohérent, qui n'a donc aucune instance possible pour les types spécifiés. Un exemple de fait est donné ci-après. Ce fait peut être utilisé pour indiquer que deux personnes distinctes ne peuvent avoir des adresses identiques dans un annuaire.

fact { *all* b : Book | *all* r,r' : Addr | r = r' => b.addr.r = b.addr.r' }

5.2.2.3 Prédicats et fonctions

La structure des domaines de données est spécifiée en Alloy en utilisant les signatures et les faits. Quant aux opérations, elles sont généralement modélisées en

utilisant les prédicats et les fonctions. Un prédicat est une contrainte réutilisable, dont le corps est une formule qui retourne `vrai` ou `faux`. Par contre, une fonction est une expression réutilisable qui retourne un résultat de type spécifique.

Dans notre exemple de spécification d'un annuaire téléphonique, nous considérons la définition du prédicat *add* suivante pour modéliser l'ajout d'une nouvelle adresse à un annuaire :

pred add(b,b' :Book, n : Name, a : Addr) { b'.addr = b.addr + n-> a }

Le prédicat *add* a quatre paramètres indiquant respectivement l'état de l'annuaire avant l'ajout de l'adresse, l'état de cet annuaire après et le nom et l'adresse à ajouter à l'annuaire. Ce prédicat est vrai si la relation addr dans le nouvel annuaire est égale à la relation addr dans l'ancien annuaire augmentée d'un nouveau tuple contenant le nom n et la nouvelle adresse a.

Toujours dans la spécification de l'annuaire téléphonique, la fonction *lookup* suivante permet de rechercher un nom dans un annuaire.

fun lookup(b : Book, n : Name) : *set* Addr {(b.addr).n}

Le corps de la fonction *lookup* est une expression qui indique que le résultat d'une recherche est égal à l'ensemble des adresses qui sont des images du nom n par la relation *addr* de l'annuaire b.

5.2.2.4 Assertions

Les assertions sont utilisées pour vérifier si un modèle satisfait des contraintes spécifiques. Les assertions sont donc des contraintes (conjectures, hypothèses) que l'on souhaite vérifier sur un modèle et qui doivent être satisfaites par toutes les instances de modèle. L'analyseur Alloy peut être utilisé pour vérifier la validité des assertions. Si une assertion n'est pas valide, son exécution génère un contre-exemple.

Par exemple, l'assertion suivante indique que l'ajout d'une entrée pour un nom n dans l'annuaire ne doit pas affecter le résultat de la fonction de recherche *lookup* pour un autre nom $n' \neq n$:

```
assert addLocal { all b,b' : Book, n,n' : Name, a : Addr |
        add(b,b',n,a) && n ! = n' => lookup(b,n') = lookup(b',n') }
```

5.3 Syntaxe abstraite de Alloy

Afin de décrire la traduction de Aspect-UML vers Alloy à l'aide de règles formelles, nous donnons au préalable, dans cette section, la syntaxe abstraite du langage Alloy. Quant à la syntaxe abstraite de Aspect-UML, la section suivante lui sera consacrée. Notons que pour décrire les syntaxes abstraites, la notation $T-$typé est utilisée. Soit T un ensemble, un ensemble $T-$typé A est une famille d'ensembles indexés par T, on écrit $A = (A_t)_{t \in T}$.

Tout au long de ce chapitre, nous considérons les ensembles disjoints d'éléments syntaxiques suivants : **S**, **R**, **Var**, **Fc**, **Pr**, **Fn** et **Ass** correspondant respectivement aux ensembles de tous les noms de signatures (types de base), de relations, de variables, de faits, de predicats, de fonctions et d'assertions.

Definition 5.3.1 (Signature). *Une signature sur* $\boldsymbol{S}$ *et* $\boldsymbol{R}$ *est un triplet* $sig^s = \langle \{s\}, \leq^s, R^s \rangle$, *où :*

- s *est le nom de type décrit par la signature ;*
- $\leq^s$ *est un ordre partiel décrivant la relation de sous-typage de* s *par rapport aux autres types sur* $\boldsymbol{S}$ *;*
- $R^s = (R_{s,w})_{w \in \boldsymbol{S}^*}$ *est un ensemble* $(\{s\} \times \boldsymbol{S}^*)$*-typé fini de noms de relations de* $\boldsymbol{R}$.

La signature globale d'une spécification Alloy, notée SIG, regroupe les types, les relations de sous-typage et les relations déclarés dans chaque signature.

Definition 5.3.2 (Signature globale). *Soit* $S = \{sig^{s_i} = \langle\{s_i\}, \leq^{s_i}, R^{s_i}\rangle | 1 \leq i \leq n\}$ *un ensemble de signatures. La signature globale de* S *est :*

$$SIG_S = \langle \cup_{1\leq i\leq n}\{s_i\}, fermeture - transitive(\cup_{1\leq i\leq n} \leq^{s_i}), \cup_{1\leq i\leq n} R^{s_i}\rangle.$$

Avant de donner la syntaxe des faits, des prédicats, des fonctions et des assertions, donnons d'abord celle des expressions (définition 5.3.3) et des formules (définition 5.3.4) Alloy qui les forment.

Definition 5.3.3 (Expression). *Soit* $SIG = \langle S, \leq^S, R\rangle$ *une signature globale. Soient* Fn *un ensemble de noms de fonctions de* ***Fn*** *et* Var *un ensemble de noms de variables de* ***Var****. Les expressions expr de l'ensemble de toutes les expressions (sur* SIG, Fn, Var*) noté* $\mathbf{Expr}_{SIG,Fn,Var}$ *sont construites selon la syntaxe suivante :*

expr : := r | v | *f(expr,...,expr)* | ***none*** | *expr binop expr* | *unop expr* | ***Int*** *intExpr*

binop : := + | & | - | · | ->

unop : := ~ | ^

intExpr : := number | ***int*** *expr* | *intExpr intOp intExpr*

intOp : := + | -

number = 0 | 1 | ..

où : $r \in S \cup R$, $v \in Var$, $f \in Fn$, ***none*** *est la relation dénotant l'ensemble vide.*

Alloy comporte deux types d'expressions : les expressions relationnelles et les expressions entières. Cependant, Alloy n'étant pas destiné au calcul numérique, il n'implémente pas toute l'arithmétique des entiers. Dans la définition 5.3.3, les opérateurs (dérivés de binop) "+", "&" et "-" sont les opérateurs standards des ensembles, respectivement, l'union, l'intersection et la différence. Les opérateurs "." et "->" représentent respectivement la composition relationnelle (jointure) et le produit cartésien. Quant aux opérateurs "~" et "^", ce sont les opérateurs relationnels standard pour la transposition et la fermeture transitive, définies sur les relations binaires. D'autre part, *intExpr* représente une expression dont la valeur est un entier. Une telle expression est obtenue soit à partir d'un littéral entier, en combinant

d'autres expressions entières (par des opérateurs usuels "+" et "-"), ou encore en appliquant l'opérateurs "**int**" à une expression relationnelle Alloy . "**int**" est appliqué à une expression relationnelle dénotant un scalaire et retourne la valeur entière de l'expression. Inversement, une valeur relationnelle peut être obtenue à partir d'un entier en appliquant l'opérateur **Int**.

La syntaxe des formules Alloy est donnée par la définition 5.3.4. Cependant, par souci de simplicité, ici nous ne considérons qu'un sous-ensemble du langage Alloy consistant en des formules simples. Les autres formules plus complexes peuvent se ramener à ces formules de base plus simples par des équivalences logiques.

Definition 5.3.4 (Formule). *Soit SIG une signature globale. Soient Pr un ensemble de noms prédicats de* $\boldsymbol{Pr}$*, Fn un ensemble de noms de fonctions de* $\boldsymbol{Fn}$ *et Var un ensemble de noms de variables de* $\boldsymbol{Var}$*.*

Les formules formula de l'ensemble de toutes les formules (sur SIG, Pr, Fn, Var) noté $\boldsymbol{Form}_{SIG,Pr,Fn,Var}$ *sont construites selon la syntaxe suivante :*

formula : := *elemFormula* | *compFormula* | *quantFormula* | *intFormula*

elemFormula : := *p(expr,...,expr)* | *expr* **in** *expr* | *expr* = *expr*

compFormula : := *! formula* | *formula* &&& *formula*

quantFormula : := **all** *var* : *expr* | *formula*

intFormula : := *intExpr intCompOp intExpr*

intCompOp : := < | > | = | =< | >=

où $p \in Pr$, $v \in Var$ *et* $expr \in \boldsymbol{Expr}_{SIG,Fn,Var}$.

Une fois les expressions et les formules défninies, nous donnons ci-après les définitions pour les faits, les assertions, les prédicats et les fonctions.

Definition 5.3.5 (Fait). *Soit* $SIG = \langle S, \leq, R \rangle$ *une signature globale. Un fait, sur SIG, est décrit par un tuple* $\langle f, \varphi^f \rangle$*, tel que* $f \in \boldsymbol{Fc}$ *est le nom du fait et* φ^f *est une formule de* $\boldsymbol{Form}_{SIG,Pr,Fn,Var}$.

Definition 5.3.6 (Assertion). *Soit $SIG = \langle S, \leq, R\rangle$ une signature globale. Une assertion sur SIG est décrite par un tuple $\langle a, \varphi^a\rangle$, tel que $a \in \boldsymbol{Ass}$ est le nom de l'assertion et φ^a est une formule de $\boldsymbol{Form}_{SIG,Pr,Fn,Var}$.*

Definition 5.3.7 (Prédicat). *Soit $SIG = \langle S, \leq, R\rangle$ une signature globale. Un prédicat sur SIG est décrit par un tuple $\langle p, param, \varphi^p\rangle$, tel que $p \in (Pr_{t_1,...,t_n})_{t_i \in (S \cup \{Int\})^*}$, avec $Pr \subset \boldsymbol{Pr}$, est le nom du prédicat; $param = (x_1, ..., x_n)$ est une liste de paramètres tels que $x_i \in (Var)_{t_i \in (S\cup\{Int\})^*}$, avec $(Var)_{t_i} \subset \boldsymbol{Var}$, et φ^p est une formule de $\boldsymbol{Form}_{SIG,Pr,Fn,Var}$.*

Definition 5.3.8 (Fonction). *Soit $SIG = \langle S, \leq, R\rangle$ une signature globale. Une fonction sur SIG est décrite par un tuple $\langle g, param, \varphi^g\rangle$, tel que $g \in (Fn_{t_1,...,t_n,t})_{t_i,t \in (S\cup\{Int\})^*}$ (où $Fn \subset \boldsymbol{Fn}$), est le nom de la fonction; $param = (x_1, ..., x_n)$ est une liste de paramètres formels tels que $x_i \in (Var)_{t_i \in S\cup\{Int\}^*}$ (où $(Var)_{t_i} \subset \boldsymbol{Var}$), et φ^g est une expression de $(E)_t$, où $(E)_t \in \boldsymbol{Expr}_{SIG,Fn,Var}$.*

5.4 Syntaxe abstraite de Aspect-UML

Tout au long de la suite de ce chapitre, nous considérons pour la syntaxe de Aspect-UML, un univers composé des ensembles disjoints **T**, **V**, **M**, **A**, **X**, **F** représentant respectivement les ensembles de tous les noms de types, d'attributs, de méthodes, de points de coupure, de variables et de fonctions. De plus, on distingue l'ensemble des types de données $\mathbf{T}^D$ de l'ensemble des types définis par les classes $\mathbf{T}^C$ et ceux définis par les aspects $\mathbf{T}^A$ (tel que $\mathbf{T} = \mathbf{T}^D \cup \mathbf{T}^C \cup \mathbf{T}^A$ et $\mathbf{T}^D \cap \mathbf{T}^C = \emptyset$, $\mathbf{T}^D \cap \mathbf{T}^A = \emptyset$ et $\mathbf{T}^C \cap \mathbf{T}^A = \emptyset$).

5.4.1 Signatures des types de données et interfaces des classes/aspects

Chaque type de données simple d'un modèle Aspect-UML est décrit par une signature de type de données abstrait notée Σ, sur **T** et **F**.

Definition 5.4.1 (Signature d'un type de données). *La signature d'un type de données (sur $\boldsymbol{T}$ et $\boldsymbol{F}$) est un triplet $\Sigma = \langle \{t\}, \leq^D, F\rangle$ où :*

- *$t \in \boldsymbol{T}^D$ est le nom du type de données;*

méthodes, de points de coupure, de variables et de fonctions. De plus, on distingue l'ensemble des types de données $\mathbf{T}^D$ de l'ensemble des types définis par les classes $\mathbf{T}^C$ et ceux définis par les aspects $\mathbf{T}^A$ (tel que $\mathbf{T} = \mathbf{T}^D \cup \mathbf{T}^C \cup \mathbf{T}^A$ et $\mathbf{T}^D \cap \mathbf{T}^C = \emptyset$, $\mathbf{T}^D \cap \mathbf{T}^A = \emptyset$ et $\mathbf{T}^C \cap \mathbf{T}^A = \emptyset$).

5.4.1 Signatures des types de données et interfaces des classes/aspects

Chaque type de données simple d'un modèle Aspect-UML est décrit par une signature de type de données abstrait notée Σ, sur $\mathbf{T}$ et $\mathbf{F}$.

Definition 5.4.1 (Signature d'un type de données). *La signature d'un type de données (sur $\boldsymbol{T}$ et $\boldsymbol{F}$) est un triplet $\Sigma = \langle \{t\}, \leq^D, F \rangle$ où :*

- *$t \in \boldsymbol{T}^D$ est le nom du type de données ;*
- *$\leq^D \subseteq (\{t\} \times T^D) \cup (T^D \times \{t\})$ est un ordre partiel décrivant la relation de sous-typage du type t par rapport aux autres types de données ;*
- *$F = (F_{w,t'})_{w \in \boldsymbol{T}^*, t' \in \boldsymbol{T}}$ est un ensemble de noms de fonctions ($\boldsymbol{T}^*$, $\boldsymbol{T}$)-typé (tel que w contient t ou $t' = t$).*

De façon similaire, une interface de classe (ou d'aspect) sur $\mathbf{T}$ et $\mathbf{M}$) est notée Ω, et est définie comme suit :

Definition 5.4.2 (Interface de classe/aspect). *Soit $\boldsymbol{T}^{CA} = \boldsymbol{T}^C \cup \boldsymbol{T}^A$. Une interface sur $\boldsymbol{T}$ et $\boldsymbol{M}$ est un triplet $\Omega = \langle \{c\}, \leq^C, M \rangle$ où :*

- *$c \in \boldsymbol{T}^{CA}$ est le nom du type de la classe ou de l'aspect ;*
- *$\leq^C \subseteq (\{c\} \times \boldsymbol{T}^{CA}) \cup (\boldsymbol{T}^{CA} \times \{c\})$ est un ordre partiel décrivant la relation de sous-typage du type c par rapport aux autres types ;*
- *$M = (M_{c,w,r})_{c \in \boldsymbol{T}^{CA}, w \in \boldsymbol{T}^*, r \in \boldsymbol{T}}$ est un ensemble ($\{c\} * \boldsymbol{T}^* * \boldsymbol{T}$)-typé fini de noms de méthodes de $\boldsymbol{M}$.*

Chaque interface de classe/aspect $\Omega = \langle \{c\}, \leq^C, M \rangle$ entraîne la définition d'une signature d'un type de données abstrait $\Sigma_\Omega = \langle \{c\}, \leq^C, F_\Omega \rangle$ dans laquelle F_Ω contient les opérations nécessaires pour gérer les sous-types, la création de références d'objets, etc.

ensemble d'interfaces Ω construit une relation de sous-typage globale notée $\leq_{\Sigma,\Omega}$.

Definition 5.4.3 (Signature globale et interface globale). *Soit* $\mathbf{\Sigma} = (\Sigma_i)_{1\leq i\leq n}$ *un ensemble de signatures de types de données abstraits et soit* $\mathbf{\Omega} = (\Omega_j)_{1\leq j\leq m}$ *un ensemble d'interfaces tels que* $\Sigma_i = \langle \{t_i\}, \leq_i^D, F_i\rangle$ *et* $\Omega_j = \langle \{c_j\}, \leq_j^C, M_j\rangle$.
La signature globale sur $\mathbf{\Sigma}$ *et* $\mathbf{\Omega}$ *est :*

$$\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}} = \langle \bigcup_{1\leq i\leq n} \{t_i\} \cup \bigcup_{1\leq j\leq m} \{c_j\},\ \leq_{\Sigma,\Omega},\ \bigcup_{1\leq i\leq n} F_i \cup \bigcup_{1\leq j\leq m} F_{\Omega_j}\rangle.$$

L'interface globale sur $\mathbf{\Omega}$ *est :*

$$\Omega_{\mathbf{\Omega}} = \langle \bigcup_{1\leq j\leq m} \{c_j\},\ fermeture\text{-}transitive(\bigcup_{1\leq j\leq m} \leq_j^C),\ \bigcup_{1\leq j\leq m} M_j\rangle.$$

5.4.2 Expressions et conditions

Nous définissons ci-après, les expressions et les conditions d'un modèle Aspect-UML qui seront utilisées plus loin dans les définitions des classes et des aspects.

Definition 5.4.4 (Expression). *Soient* $\Sigma = \langle T, \leq, F\rangle$ *la signature globale et* $\Omega = \langle C, \leq^C, M\rangle$ *l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que* Σ *est complète. Soient également,* $V = (V_t)_{t\in T}$ *un ensemble de noms d'attributs de* $\mathbf{V}$ *($V^c \subseteq V, c \in C$, contient les attributs de V appartenant à la classe de type c) et* $X = (X_t)_{t\in T}$ *un ensemble de variables de* $\mathbf{X}$.
L'ensemble $Exp_{\Sigma,\Omega,V,X} = (E_t)_{t\in T}$ *de toutes les expressions sur* Σ, Ω, V, X *est le plus petit ensemble tel que :*

- *Si* $x \in X_t, t \in T$ *alors* $x \in E_t$;
- *Si* $v \in V_t, t \in T$ *alors* $self.v \in E_t$;
- *Si* $f \in F_{t_1,\ldots,t_n,t}$ *avec* $t_i, t \in T\ \forall i, 1 \leq i \leq n$ *et* $a_i \in E_{t_i} \forall i, 1 \leq i \leq n$ *alors* $f(a_1, \ldots, a_n) \in E_t$;
- *Si* $o \in E_c$ *avec* $c \in C$ *et* $v \in V_t^c$, $t \in T$ *alors* $o.v \in E_t$;
- *Si* $m \in M_{c,t_1,\ldots,t_n,r}$ *avec* $c \in C, t_i \in T\ \forall\ 1 \leq i \leq n, r \in T, o \in E_c$ *et* $a_i \in E_{t_i} \forall\ 1 \leq i \leq n$ *alors* $o.m(a_1, \ldots, a_n) \in E_r$.

Dans la définition 5.4.5, nous donnons la syntaxe simplifiée d'une condition, c'est-à-dire que nous ne considérons que l'égalité entre deux expresssions de même

Definition 5.4.5 (Condition). *Soient $\Sigma = \langle T, \leq, F \rangle$ la signature globale et $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Soient également V un ensemble de noms d'attributs et X un ensemble de variables et soit $Exp_{\Sigma,\Omega,V,X} = (E_t)_{t \in T}$ l'ensemble de toutes les expressions (sur Σ, Ω, V et X). Une condition de l'ensemble des conditions $Cond_{\Sigma,\Omega,V,X}$ sur les expressions de $Exp_{\Sigma,\Omega,V,X}$ est une contrainte cond construite selon la syntaxe suivante :*

$cond ::= elemCond \mid compCond$

$elemCond ::= expr_t \ opComp^t \ expr_t \mid expr_{Integer} \ opComp^{int} \ expr_{Integer}$

$compCond ::= \neg \ cond \mid cond \wedge cond$

$opComp^t ::= = \mid ...$

$opComp^{int} ::= < \mid > \mid \leq \mid \geq$

type et les inégalités entre les entiers. Cette définition peut, toutefois, être complétée par d'autes opérateurs de comparaison comme l'inclusion, l'appartenance, etc. appliqués à d'autres types tels que les réels, les listes, les tableaux, etc.

5.4.3 Classes

Pour décrire les méthodes d'un modèle Aspect-UML, nous utilisons les formules de méthode. Une formule de méthode (telle que donnée par la définition 5.4.6) consiste donc en un nom de méthode, un ensemble de paramètres formels et des pré et postconditions décrivant les propriétés qui doivent être vérifiées avant et après l'exécution de la méthode.

Une classe est vue comme une template à partir de laquelle les objets de la classe sont instanciés. Une classe (telle que donnée par la définition 5.4.7) consiste en une interface de classe, un ensemble de noms d'attributs et un ensemble de formules de méthodes.

Definition 5.4.6 (Formule de méthode). *Soient $\Sigma = \langle T, \leq, F \rangle$ la signature globale et $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Soit également, $V = (V_t)_{t \in T}$ un ensemble de noms d'attributs de $\boldsymbol{V}$. Une formule de méthodes sur Σ, Ω, V et $\boldsymbol{X}$ est un quadruplet $\langle m, p, pre, post \rangle$ où :*

- $m \in M_{c,t_1,...,t_n r}$;
- $p = (p_1, ..., p_n)$ *où* $X = (X_t)_{t \in T}$ *est un ensemble de noms de variables de* $\boldsymbol{X}$ *et* $p_i \in X_{t_i}, \forall i, 1 \leq i \leq n$, *appelées paramètres formels ;*
- $pre, post \subset Cond_{\Sigma,\Omega,V,LX}$, *où* $LX = \{p_1, ..., p_n\}$.

Definition 5.4.7 (Classe). *Soient $\Sigma = \langle T, \leq, F \rangle$ la signature globale et $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Une classe cl (sur Σ, Ω) est un triplet $cl_{\Sigma,\Omega} = \langle \Omega^c, V, \Phi \rangle$, où :*

- $\Omega^c = \langle \{c\}, \leq^c, M^c \rangle$ *est une interface définissant la classe de type* $c \in T^C$;
- $V = (V_t)_{t \in T}$ *est un ensemble de noms d'attributs typés de* $\boldsymbol{V}$;
- Φ *est un ensemble de formules de méthodes sur* Σ, $\Omega \cup \Omega^c$, V, M^c *(et* $\boldsymbol{X}$*).*

5.4.4 Points de jointure et points de coupure

Un point de jointure correspond à un point particulier dans l'exécution d'un programme. Dans notre profil Aspect-UML, nous considérons, entre autres, un point de jointure comme le point correspondant à un appel ou une exécution de méthode. Formellement un point de jointure est défini ainsi :

Definition 5.4.8 (Point de jointure). *Soit $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble d'interfaces. Un point de jointure sur Ω, est une paire $\langle kind, m \rangle$, où :*

- $kind \in \{call, execute, ...\}$;
- $m \in M$.

Un point de coupure est une collection de points de jointure. De plus, un point de coupure a un nom et éventuellement un ensemble de paramètres formels.

Definition 5.4.9 (Point de coupure). *Soient $\Sigma = \langle T, \leq, F \rangle$ la signature globale et $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Un point de coupure sur Σ, Ω est un triplet $ptC_{\Sigma,\Omega} = \langle pc, args, jpSet \rangle$, où :*

- *$pc \in A_{t_1,...,t_n,t_r}$, où $A = (A_{w,t})_{w \in T^*, t \in T}$ est un ensemble typé de noms de points de coupure de $\boldsymbol{A}$;*
- *$args = (x_1, ..., x_n)$, où $X = (X_t)_{t \in T}$ est un ensemble typé de noms de variables de $\boldsymbol{X}$ et $x_i \in X_{t_i}, \forall i, 1 \leq i \leq n$;*
- *$jpSet$ est un ensemble de points de jointure sur Ω.*

5.4.5 Aspects

Les aspects sont syntaxiquement très similaires aux classes. La syntaxe des interfaces d'aspects est la même que celle des interfaces de classes. Le corps d'un aspect contient cependant une construction supplémentaire pour la définition des advices. Les advices ne sont pas déclarés dans les interfaces d'aspects car ils ne peuvent pas être invoqués comme les méthodes. Le contenu des advices est défini dans le corps des aspects par des formules d'advices.

Une formule d'advice est définie comme un formule de méthode, sauf qu'un advice n'a pas de nom. En revanche, il est rattaché à un point de coupure par un label `before` ou `after`.

Definition 5.4.10 (Formule d'advice). *Soient $\Sigma = \langle T, \leq, F \rangle$ la signature globale et $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Soit également P un ensemble de points de coupure sur Σ, Ω et $V = (V_t)_{t \in T}$ un ensemble de noms d'attributs de $\boldsymbol{V}$. Une formule d'advice sur Σ, Ω, P et V est un tuple $\langle \gamma, ptC, args, pre, post \rangle$, où :*

- *$\gamma \in \{before, after\}$;*
- *$ptC = \langle pc, args', jpSet \rangle$ est un point de coupure de P ;*
- *$args = (a_1, ..., a_n)$ où $X = (X_t)_{t \in T}$ est un ensemble de noms de variables de $\boldsymbol{X}$ et $a_i \in X_{t_i}, \forall i, 1 \leq i \leq n$, appelées paramètres formels ;*
- *$pre, post \subset Cond_{\Sigma,\Omega,V,LX}$ où $LX = \{a_1, ..., a_n\}$.*

Definition 5.4.11 (Aspect). *Soient $\Sigma = \langle T, \leq, F \rangle$ la signature globale et $\Omega = \langle C, \leq^C, M \rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Soit P un ensemble de noms de points de coupure de $\boldsymbol{P}$ sur Σ, Ω. Un aspect asp sur Σ, Ω est un tuple $asp_{\Sigma,\Omega,P} = \langle \Omega^a, prio, V, \Phi, \Psi \rangle$, où :*

- *$\Omega^a = \langle \{a\}, \leq^a, M^a \rangle$ est une interface d'aspect définissant l'aspect de type $a \in \boldsymbol{T}^A$;*
- *$prio$ est un ordre partiel sur $\boldsymbol{T}^A$ décrivant la relation de priorité de l'aspect asp par rapport aux autres aspects ;*
- *$V = (V_t)_{t \in T}$ est un ensemble de noms d'attributs typés de $\boldsymbol{V}$;*
- *Φ est un ensemble de formules de méthodes sur $\Sigma, \Omega \cup \Omega^a, V, M^a$;*
- *Ψ est un ensemble de formules d'advices sur $\Sigma, \Omega \cup \Omega^a, V, P$.*

5.4.6 Spécification de base

Une spécification de base $bSpec_{\Sigma,\Omega}$ est une collection de signatures de types de données, de classes et d'aspects, définie formellement par :

Definition 5.4.12 (Spécification de base). *Soit Σ un ensemble de signatures de types de données abstraits et soit Ω un ensemble d'interfaces telle que la signature globale $\Sigma_{\Sigma,\Omega}$ st complète. Soit aussi P un ensemble de noms de points de coupure sur Σ, Ω. Une spécification de base sur Σ, Ω et P est un ensemble de signatures de types de données, de classes et d'aspects, défini comme suit :*
$bSpec_{\Sigma,\Omega,P} = \{\Sigma_i | 1 \leq i \leq n\} \bigcup \{(Cl_{\Sigma,\Omega})_j | 1 \leq j \leq m\} \bigcup \{(Asp_{\Sigma,\Omega,P})_k | 1 \leq k \leq l\}$

La spécification de base $bSpec_{\Sigma,\Omega,P}$ est dite complète si et seulement si Σ et Ω sont respectivement égales à la signature globale et à l'interface globale de la spécification.

5.4.7 Spécification Aspect-UML

Afin de lier le contexte des points de jointure aux paramètres d'un point de coupure, une fonction de liaison de contexte est définie. Dans la définition qui suit, nous définissons formellement la liaison de contexte d'un ensemble X vers un

ensemble Y, qui associe à chaque élément de Y un seul élément de X ou de la variable pré-définie *target*.

Definition 5.4.13 (Liaison de contexte). *Soient $X = (X_{t\prime})_{t\prime \in \boldsymbol{T}}$ et $Y = (Y_t)_{t \in \boldsymbol{T}}$ deux ensembles de variables $\boldsymbol{T}$-typés de $\boldsymbol{X}$. Soit "$\leq$" une relation d'ordre partiel sur $\boldsymbol{T}$.*
Une liaison de contexte de X vers Y étant donné "$\leq$", est une fonction définie comme suit : $binding_{X,Y,\leq} = \{(y, x) \mid y \in Y_t, x \in X_{t\prime} \cup \{target_{t\prime}\}, t\prime \leq t\}$.

Ainsi, un point de coupure lié est défini formellement par :

Definition 5.4.14 (Point de coupure lié). *Soient $\Sigma = \langle T, \leq, F\rangle$ la signature globale et $\Omega = \langle C, \leq^C, M\rangle$ l'interface globale d'un ensemble de signatures et d'un ensemble d'interfaces, telle que Σ est complète. Soit aussi P un ensemble de noms de points de coupure sur Σ, Ω. Soit $bSpec_{\Sigma,\Omega,P}$ une spécification de base complète sur Σ, Ω et P. Un point de coupure lié sur Σ, Ω est un tuple $\langle ptC, B\rangle$, où :*

- *$ptC = \langle pc, args, jpSet\rangle$ est un point de coupure sur Σ, Ω ;*
- *B est une fonction qui associe une fonction de liaison de contexte à chaque point de jointure jp associé au point de coupure pc. Elle est définie comme suit : $B = (jp, binding_{A_{jp}, A_{ptC}, (\leq \cup \leq^C)}) \mid jp \in jpSet\}$,*
  où A_{ptc} est l'ensemble des paramètres formels de ptC et A_{jp} est l'ensemble des paramètres formels du point de jointure $jp = \langle kind, m\rangle$ déclarés dans la formule de méthode définissant m dans la spécification de base $bSpec$.

Dans un modèle Aspect-UML, les points de coupure liés sont implémentés par un ensemble d'aspects. Formellement cette implémentation est donnée ci-après :

Definition 5.4.15 (Relation d'implémentation des points de coupure). *Soit $bSpec_{\Sigma,\Omega,P}$ une spécification de base complète sur Σ, Ω, P. Soit $Aset = \{(Asp_{\Sigma,\Omega,P})_k \mid 1 \leq k \leq l\}$ un ensemble d'aspects de $bSpec_{\Sigma,\Omega,P}$ et soit $PCset = \{(ptC_{\Sigma,\Omega})_i \mid 1 \leq i \leq n\}$ un ensemble de points de coupure liés sur Σ, Ω. Une relation d'implémentation de $PCset_{\Sigma,\Omega}$ par $Aset_{\Sigma,\Omega,P}$ est un ensemble défini par :*
$Impl_{Aset,PCset} = \{(A, pc)$ tel que $A = \langle \Omega^{Asp}, prio, V, \Phi^M, \Phi^A\rangle \in Aset_{\Sigma,\Omega,P}$ et $pc = \langle p, B\rangle \in PCset_{\Sigma,\Omega}\}$.

Finalement, une spécification Aspect-UML notée $Spec_{Aspect-UML}$ consiste en

une spécification de base $bSpec_{\Sigma,\Omega,P}$, où les aspects implémentent des points de coupure liés de $PtcSet_{bSpec}$. Formellement elle est définie ainsi :

Definition 5.4.16 (Spécification Aspect-UML). *Soit $bSpec_{\Sigma,\Omega,P}$ une spécification de base complète sur Σ, Ω, P. Soit $Aset = \{(Asp_{\Sigma,\Omega,P})_k \mid 1 \leq k \leq l\}$ un ensemble d'aspects de $bSpec_{\Sigma,\Omega,P}$ et soit $PCset = \{(ptC_{\Sigma,\Omega})_i \mid 1 \leq i \leq n\}$ un ensemble de points de coupure liés sur Σ, Ω tel que $PCset \subseteq P$. Soit $Impl_{Aset,PCset}$ une relation d'implémentation de $PCset$ par $Aset$. Une spécification Aspect-UML est un tuple défini par :*

$$Spec_{Aspect-UML} = \langle bSpec_{\Sigma,\Omega}, PCset, Impl_{Aset,PCset} \rangle, \quad telque :$$

pour tout $Asp = \langle \Omega^{Asp}, prio, V, \Phi^M, \Phi^A \rangle \in Aset$, $\exists p \in PCset$ tel que $(Asp, p) \in Impl_{Aset,PCset}$.

5.5 Traduction formelle de AspectUML vers Alloy

Dans cette section nous décrivons le processus de traduction d'un modèle Aspect-UML vers une spécification Alloy. Cette traduction est bien sûr cohérente avec celle présentée en termes de réseaux de Petri au chapitre précédent. L'approche de modélisation avec Alloy que nous proposons [MV07b, MV07a] offre une solution de traduction de Aspect-UML vers Alloy à la fois flexible, adaptable et générique. L'idée consiste en les deux points suivants :

1. Alloy étant un langage déclaratif, les champs des signatures ne sont pas modifiables *c-à-d* qu'on ne peut pas leur assigner plusieurs valeurs en fonction du temps. Ils sont, en fait, figés et ne peuvent contenir qu'une seule et unique valeur dans un modèle donné (tout comme les variables mathématiques). La première idée de notre solution consiste alors à simuler une trace de valeurs pour les champs des signatures en modifiant leur type, les forçant ainsi à indiquer des traces de valeurs relationnelles à travers le temps. Pour ce faire, nous ajoutons une signature *Time* au type des champs qui doivent garder trace de plusieurs valeurs. Un champ f avec un type $A \rightarrow B$ deviendra alors une relation dynamique $A \rightarrow B \rightarrow Time$ faisant

correspondre des valeurs de type "A" aux valeurs de type "B" à travers le temps. Comme nous l'expliquerons plus loin, une signature *Time* sera introduite afin de définir un ensemble d'atomes ordonnés. Cette solution est inspirée de [Jac06] qui elle même trouve son origine dans [McC68] ;

2. La deuxième idée de notre solution consiste à réifier les points de jointure et les advices en des signatures Alloy contraintes, plutôt que de les traduire en des prédicats et fonctions Alloy équivalentes. Ceci nous permet de généraliser la composition et le tissage, en les définissant par des contraintes génériques. L'idée générale consiste à traduire chaque point de jointure ou advice en une signature qui étend une signature abstraite (appelée `Operation`) dénotant l'ensemble de toutes les opérations (point de jointure ou advice). Chaque signature concrète étendant `Operation` doit fournir des champs individuels représentant les arguments spécifiques du point de jointure ou de l'advice et l'objet cible. Les pré et les postconditions seront décrites par des faits contraignant les signatures concrètes respectives. La composition et le tissage des advices au point de jointure seront représentés respectivement par les signature `Composition` et `Weaving` déclarées comme des extensions de la signature `Operation`. Les règles générales de composition et de tissage seront exprimées sous forme de faits contraignant respectivement les signatures `Composition` et `Weaving`. Ces règles seront ainsi appliquées à toutes les compositions ou tissages d'advices.

Dans la suite de cette section, nous présentons en détails la traduction d'une spécification Aspect-UML vers une spécification Alloy. Notons que la traduction procède en quatre étapes. En premier lieu, les éléments structurels (classes, aspects, types de données) d'un modèle Aspect-UML sont traduits en Alloy (section 5.5.2). La seconde étape concerne la traduction des spécifications comportementales Aspect-UML (les points de jointure, les advices, les pré et les postconditions qui leurs sont associées et les règles de passage de contexte définies par les points de coupure) (section 5.5.3). La troisième étape explique la traduction de la

composition des advices s'exécutant à un même point de jointure (section 5.5.4). La quatrième étape prend en charge la traduction du processus de tissage, où les advices composés sont tissés aux points de jointure spécifiques (section 5.5.5).

5.5.1 Hypothèses

Tout au long du processus de traduction, nous supposons données les hypothèses suivantes.

Soit $Spec_{Aspect-UML} = \langle bSpec_{\Sigma,\Omega,P}, PCsetImpl \rangle$ une spécification Aspect-UML, sur $\Sigma = \langle T, \leq^D, F \rangle$, $\Omega = \langle C, \leq^c, M \rangle$, à traduire en une spécification Alloy $Spec_{Alloy} = \langle SIG = \langle S, \leq^S, R \rangle, FC, PR, FN, ASS \rangle$, sur S, R, Fc, Pr, Fn et Ass.

La formalisation de la traduction sera présentée dans un style dénotationnel. Nous devons, pour cela, décrire la dénotation d'un élément e du domaine syntaxique de $Spec_{Aspect-UML}$ comme une fonction $\tau(e)$ qui associe à e une valeur du domaine syntaxique de $Spec_{Alloy}$.

La fonction polymorphe $\tau : Spec_{Aspect-UML} \rightarrow Spec_{Alloy}$ sera définie au fur et à mesure que nous élaborerons les différentes étapes de traduction de Aspect-UML vers Alloy.

Des exemples tirés de notre application de téléphonie seront utilisés pour illustrer les différentes étapes de traduction.

5.5.2 Traduction des éléments structurels d'Aspect-UML

Les éléments structurels d'une spécification Aspect-UML sont les classes, les aspects, les types de données (primitifs et énumérés). Ces éléments définissent des ensembles d'objets ou de valeurs. Chacun peut être traduit naturellement en une signature Alloy. Cependant le type primitif `Integer` de Aspect-UML est une exception, il correspondra au (seul) type de données prédéfini `int` de Alloy. Alloy fournit un certain nombre (quoique limité) d'opérateurs arithmétiques et de comparaison qui peuvent être utilisés pour la traduction des expressions de Aspect-UML

contenant des valeurs entières. La traduction des autres éléments structurels est expliquée ci-après.

5.5.2.1 Traduction des types de données

Un type de données énuméré t composé d'un ensemble de litéraux $l_1, l_2, ..., l_n$ est traduit en une signature abstraite Alloy, appelée t, étendue par les signatures (singletons) l_i créées pour chaque littéral.

Par exemple, dans notre application de téléphonie, le type énuméré `Status` permet de qualifier l'état actuel d'une connexion. Sa traduction en Alloy donne la signature suivante :

abstract **sig** Status {}
one **sig** connected, disconnected *extends* Status {}

Les types primitifs de Aspect-UML tels que `Boolean`, `Date`, etc. sont traduits comme s'ils étaient des types énumérés. Les opérations sur les types (énumérés ou primitifs) doivent alors être explicitement traduites en des fonctions Alloy. En effet, Alloy ne fournit aucun autre type de données primitif autre que `int`.

La traduction des types de données abstraits de Aspect-UML vers Alloy est formalisée par les règles formelles présentées à la table 5.1. La règle (1) indique que le type de données `Integer` de Aspect-UML est traduit vers le type prédéfini `Int` de Alloy. Par contre, la règle (2) traduit un nom de type de Aspect-UML autre que `Integer` vers un nom de signature Alloy. Les opérations (qui ne dénotent pas des constantes) sur un type (qu'il soit, Integer ou autre) sont traduites vers des fonctions Alloy (règle (4)). Par contre, les noms d'opérations dénotant des constantes d'un type sont traduits vers des signatures Alloy qui étendent la signature de ce type (règle (5)). La signature d'un type de données de Aspect-UML est traduite vers une signature Alloy (règle (6)). De même, une signature Alloy décrivant chaque constante d'un type est rajoutée à la signature globale SIG de la spécification Alloy

TAB. 5.1 – Règles de traduction pour les types de données abstraits.

Pour tout type $t \in T$ tel que $\Sigma^t = \langle\{t\}, \leq^t, F^t\rangle$ et $F^t = \{f_i(t_1, ..., t_{m_i})_{t_j \in T},\ i = 1..n,\ j = 1..m_i\}$,

nous appliquons les règles suivantes :

$$\tau(t) = \textbf{int} \quad si\ t = Integer \tag{5.1}$$

$$\tau(t) \in S \mid \tau(t) = t \qquad si\ t \neq Integer \tag{5.2}$$

$$\tau(\leq^t) = \leq^t_\tau \subset \leq^S \mid t_1 \leq^t t_2 \Rightarrow \tau(t_1) \leq^t_\tau \tau(t_2) \quad \forall\ t_1, t_2 \in T \tag{5.3}$$

$$\forall\ i = 1..n,\ \tau(f_{i_{(t_1,...,t_{m_i})}}) = f_{i_{(\tau(t_1),...,\tau(t_{m_i}))}} \in Fn \qquad si\ m_i > 1 \tag{5.4}$$

$$\tau(f) \in S \mid \tau(f) = f \wedge \tau(f) \leq^t_\tau \tau(t) \quad si\ m_i = 1 \tag{5.5}$$

$$\tau(\Sigma^t) = sig^t = \langle\{\tau(t)\}, \leq^t_\tau, \emptyset\rangle \in SIG \quad si\ t \neq Integer \tag{5.6}$$

$$\forall\ i = 1..n,\ \langle\{\tau(f)\}, \leq^t_\tau, \emptyset\rangle \in SIG \qquad si\ m_i = 1\} \tag{5.7}$$

(règle(7)). En outre, la relation de sous-typage pour les types de données définis dans une spécification Aspect-UML est traduite en une relation de sous-typage sur les signatures dans Alloy (règle (3)).

5.5.2.2 Traduction des classes et des aspects

Une classe ou un aspect d'un modèle Aspect-UML peut être vue comme un ensemble d'objets partageant une collection commune de déclarations d'attributs. Une classe C (resp. un aspect A) peut donc être traduite en une signature Alloy C (resp. A) contenant un champ f_i tel que $C \rightarrow F_i$ (resp. $A \rightarrow F_i$) pour chaque attribut $f_i : F_i, 1 \leq i \leq n$ déclaré dans C (resp. A). Quant aux méthodes, elles sont traduites vers des fonctions Alloy. En particulier, les invocations de méthodes dans les expressions (pré et postconditions) d'un modèle Aspect-UML sont traduits vers des appels de fonctions. En outre, comme nous l'avons souligné au début de cette section, les méthodes qui sont aussi des points de jointure seront réifiées comme des atomes appartenant à une signature abstraite appelée `Operation`. Notons, toute-

fois, que si une méthode m donnée apparaît dans une expression Aspect-UML, et est en même temps un point de jointure, alors elle sera traduite en :

1. une fonction Alloy f^m à utiliser dans la traduction des expressions ;
2. et une signature Sig^m à utiliser dans la traduction des processus de composition et de tissage des aspects. Nous reviendrons plus en détails sur ce deuxième point dans la prochaine section.

Par ailleurs, étant un langage déclaratif (qui décrit ce qu'un système fait et non comment il le fait), les états intermédiaires ne sont pas naturellement spécifiés par Alloy, puisque les champs sont figés et non modifiables. Dans notre cas, nous devons, en fait, lever cette limitation, afin de pouvoir spécifier comment les advices sont composés dans le système de base et quels sont leurs effets intermédiaires sur les états des objets. Pour ce faire, chaque déclaration de champ (dénotant un attribut de classe/aspect sujet à des modifications) est étendu avec la signature `Time`, comme montré par l'exemple qui suit. Ainsi un champ ne dénote plus une valeur (associée à un objet) mais une trace de valeurs.

```
open util/ordering[Time] as timeorder
sig Time {}
sig Connection {
      status : Status one -> Time,
      origin : Device,
      destination : Device}
```

Dans un modèle Aspect-UML, les attributs apparaissant dans les pré et post-conditions des méthodes et des advices sont typiquement identifiés comme étant ceux sujets à des modifications. C'est ce que nous observons dans la traduction vers Alloy de la classe `Connection`, donnée plus haut. Comme spécifié par les pré et les postconditions (données dans le modèle Aspect-UML de l'application de téléphonie à la figure 3.4, page 72), les connexions peuvent changer d'état (*e.g.* changer de

disconnected à *connected*). Ainsi, le type du champ `status` est augmenté de `Time`. Avant l'ajout de `Time`, `c.status` dénote un seul atome `Status`. À présent, il est devenu une relation, où `c.status.t` dénote l'état de la connexion c au temps t. La dimension `Time` permet aux champs de la signature de garder trace des valeurs des attributs aux différents moments d'une exécution. Le type `Time` est simplement un ensemble d'atomes ordonnés. L'ordonnancement est pris en charge par le module paramétré `ordering` fourni par Alloy. Ce module dont le code est donné en annexe, permet de créer une ordre total linéaire à travers les atomes d'une signature.

TAB. 5.2 – Règles de traduction pour les classes.

Soient Pre et $Post$ respectivement les ensembles de toutes les pré-conditions et de toutes les postconditions de $Spec_{Aspect-UML}$.
Soit $Var(Pre \cup Post)$ l'ensemble de toutes les variables apparaissant dans toutes les (pré/post) conditions de $Pre \cup Post$.

Sachant que $Time \in S$ et $sig^{Time} = \langle \{Time\}, \leq^{Time}, \emptyset \rangle \in SIG$.

Pour chaque classe $cl = \langle \Omega^c, V^c, \Phi^c \rangle$, tel que $\Omega^c = \langle \{c\}, \leq^c, M^c \rangle$ sur Σ, Ω, $V^c = \{v_i : t_i |\ t_i \in T \cup C, i = 1..n\}$ et $M^c = \{m^j_{(c,t_1,\ldots,t_l,r)_{t_k,r \in T \cup C}}, j = 1..q, k = 1..l\}$,

nous appliquons les règles suivantes :

$$\tau(c) \in S \mid \tau(c) = c \tag{5.8}$$

$$\forall i = 1..n,\ \tau(v_i) \in R^c \mid \tau(v_i) = v_i \ \wedge \tau(v_i) : \tau(t_i) \quad si\ v_i \notin Var(Pre \cup Post) \tag{5.9}$$

$$\forall i = 1..n,\ \tau(v_i) \in R^c \mid \tau(v_i) = v_i \ \wedge \tau(v_i) : \tau(t_i) \times Time \quad si\ v_i \in Var(Pre \cup Post) \tag{5.10}$$

$$\tau(\leq^c) = \leq^c_\tau \subset \leq^S \mid c_1 \leq^c c_2 \ \Rightarrow \tau(c_1) \leq^c_\tau \tau(c_2) \quad \forall\ c_1, c_2 \in C \tag{5.11}$$

$$\tau(cl) = \langle \{\tau(c)\}, \leq^c, R^c \rangle \in SIG \tag{5.12}$$

$$\forall j = 1..q,\ \tau(m^j_{(c,t_1,\ldots,t_l,r)}) = m^j_{(\tau(c),\tau(t_1),\ldots,\tau(t_l),\tau(r))} \in Fn \tag{5.13}$$

Formellement, la traduction d'une classe de $Spec_{Aspect-UML}$ de type c vers une signature de $Spec_{Alloy}$, est définie par les règles sémantiques données à la table 5.2.

Les aspects sont traduits de façon similaire. Notons qu'initialement, nous supposons que la signature sig^{Time} définissant les objets de type $Time$ est déjà dans la signature globale SIG de $Spec_{Alloy}$. Le type de la classe c est traduit en un type de signature de même nom dans Alloy (règle (8)) et la classe regroupant des objets de ce type est traduite en une signature de SIG (règle (12)). Chaque attribut de la classe est traduit en un champ de la signature, avec le type correspondant (règle (9)), le type est toutefois augmenté de $Time$ si l'attribut apparaît dans une pré ou postcondition de la spécification Aspect-UML (règle (10)). La règle (11) traduit, quant à elle, la relation de sous-typage défnie sur les classes en une relation de sous-typage définie sur les signatures de $Spec_{Alloy}$. Quant à la traduction des opérations (méthodes et advices) elle sera présentée à la section 5.5.3.1 lorsque nous aborderons la traduction des spécifications comportementales.

5.5.3 Traduction des spécifications comportementales de Aspect-UML

Le comportement d'un modèle Aspect-UML est spécifié par des annotations décrivant les pré et postconditions respectives que les méthodes/advices doivent satisfaire. Certaines informations comportementales sont aussi données par les contraintes de passage de contexte spécifiées par les points de coupure.

5.5.3.1 Traduction des points de jointure et des advices

Rappelons que la traduction que nous proposons vers Alloy a pour but la vérification de la composition des aspects et non la vérification de toute l'application. Nous ne nous intéressons ainsi qu'à la traduction des éléments impliqués dans le tissage des aspects. Les méthodes représentant des points de jointure et les advices implémentant les points de coupure sont les seules opérations d'un modèle Aspect-UML à être impliquées dans le processus de composition et de tissage des aspects. Selon notre approche de modélisation en Alloy, ces opérations sont les seules à être réifiées et traduites en des signatures qui étendent une signature abstraite appelée

`Operation` définie comme suit : La signature `Operation` a deux champs `begin` et

abstract **sig** Operation { begin, end : Time}

`end` utilisés respectivement pour marquer le temps de début et de fin d'une opération. Ces informations seront utilisées ultérieurement pour ordonner l'exécution des opérations et pour identifier l'effet d'une opération sur les états des objets sur un intervalle de temps.

Étant donnée une méthode $myOp(f_1 : t_1, ..., f_n : t_n)$, une signature concrète *myOp* qui étend `Operation` est créée. La signature *myOp* doit aussi déclarer (1) un champ `self` pour sauvegarder l'objet invoqué par la méthode et (2) un champ additionnel f_i de type t_i pour chacun de ses paramètres. Un atome de la signature *myOp* représente une invocation spécifique de la méthode *myOp*.

Étant réifiés en signatures, les méthodes/advices seront traités de façon uniforme en Alloy. Ces opérations peuvent être utilisées dans des champs, dans les déclarations de prédicats et de fonctions. Certaines opérations peuvent être définies comme des extensions d'autres opérations, et ainsi réutiliser les arguments factorisés dans une super-opération. Étant donné que les advices et les points de jointure ont des rôles différents dans le processus de tissage, les opérations représentant des advices doivent être distinctes des opérations décrivant des méthodes (utilisées comme points de jointure). Cette distinction est réalisée par la déclaration suivante des sous-types `Advice` et `JP` (c'est en fait le terme *abstract* de Opeartion qui force la définition d'une partition dans ce cas.).

abstract **sig** JP, Advice *extends* Operation {error : boolean}

fact indivisible {*all* op : JP+Advice | op.end = op.begin.next [3]}

[3] *next* est une relation du module ordering qui donne pour chaque élément de l'ordre considéré (dans notre cas Time) son successeur.

Les signatures JP et Advice regroupent toutes les deux des instances d'opérations impliquées dans le processus de tissage, qui peut ou non résulter en une composition valide. Pour repérer les erreurs de composition potentielles, JP et Advice introduisent un champ error qui vaut vrai si une telle erreur est détectée. La spécification Alloy suppose aussi que les advices et les points de jointure de Aspect-UML sont des opérations atomiques ayant une durée unitaire, où la fin (end) d'une opération est le successeur immédiat de son commencement (begin) dans l'ensemble ordonné Time. Un fait appelé indivisible est alors ajouté pour spécifier ces contraintes comportementales. Les faits qui s'appliquent aux signatures abstraites JP, Advice et Operation introduisent les contraintes génériques que tout point de jointure et tout advice doit respecter.

D'autre part, le comportement spécifique d'une opération doit être spécifié par un fait contraignant la signature concrète (qui étend *JP* ou *Advice*). Dans la notation Aspect-UML, cette spécification est réalisée au moyen des pré et postconditions. L'ensemble des pré et postconditions contraignant une opération *myOp* (point de jointure ou advice) dans un modèle Aspect-UML sera ainsi traduit vers un fait (conjonction de formules) contraignant la signature *myOp*. Cette traduction est assez directe vue la ressemblance syntaxique entre une condition Aspect-UML et une formule Alloy et entre une expression Aspect-UML et une expression Alloy. Néanmoins, une certaine précaution s'impose quant à la traduction de l'accès à un attribut. En effet, avec l'introduction du type *Time* dans les types des attributs apparaissant dans les pré et postconditions, l'accès à un attribut a d'un objet o à un temps t nécessite la consultation du champ de signature $o.a$ et ensuite la navigation à travers cette relation afin de trouver $o.a.t$ (c-$à$-d la valeur de l'attribut au temps t).

La notion de temps est implicitement présente dans un modèle Aspect-UML à travers ses pré et postconditions. En effet, une précondition est une condition qui doit être satisfaite *avant* l'exécution d'une opération et une postcondition est une

condition à assurer *après* l'exécution d'une opération. Ces temps *avant* et *après* l'exécution d'une opération sont respectivement saisis dans notre modélisation Alloy par les champs `begin` et `end` de la signature `Operation`. Par conséquent, une précondition (resp. postcondition) d'une opération Aspect-UML sera traduite vers une formule Alloy à être évaluée au temps `begin` (resp. `end`) de cette opération. De même, une expression Aspect-UML sera naturellement traduite vers une expression Alloy, où l'accès à un attribut v au temps t avant (resp. après) l'exécution d'une opération, sera remplacé par la navigation à travers le champ v et la relation t où $t = begin$ (resp. $t = end$).

Avant de donner les règles formelles de traduction d'une opération (point de jointure ou advice) vers Alloy, définissons d'abord formellement la traduction des expressions et des conditions utilisées dans les pré et postconditiosn associées aux opérations.

Soient $Exp^{Aspect-UML}_{\Sigma,\Omega,V,X}$ et $Cond_{\Sigma,\Omega,V,X}$ respectivement les ensembles de toutes les expressions et de toutes les conditions de $Spec_{Aspect-UML}$.

Soient $Exp^{Alloy}_{SIG,Fn,Var}$ et $Form_{SIG,Pr,Var}$ respectivement les ensembles de toutes les expressions et de toutes les formules de $Spec_{Alloy}$.

La traduction d'une expression Aspect-UML vers une expression Alloy est définie par la fonction $\| \ \|^t$: $Exp^{Aspect-UML}_{\Sigma,\Omega,V,X} \rightarrow Exp^{Alloy}_{SIG,Fn,Var}$ qui découle naturellement de la définition 5.4.4. Elle est définie comme suit :

$$\begin{aligned} \| \ x \ \|^t &= \tau(x) \\ \| \ self.v \ \|^t &= self.(\tau(v).t) \\ \| \ f \ \|^t &= \tau(f) \\ \| \ o.v \ \|^t &= o.(\tau(v).t) \\ \| \ o.m \ \|^t &= \tau(m) \end{aligned}$$

pour toute variable $x \in X$, tout attribut $v \in V$, tout objet $o \in X$, toute fonction $f \in F$ et toute méthode $m \in M$.

D'autre part, la traduction d'une (pré/post) condition Aspect-UML vers une formule Alloy est définie inductivement par la fonction $|\ |^t$: $Cond_{\Sigma,\Omega,V,X} \rightarrow Form_{SIG,Pr,Var}$ qui découle de la définition 5.4.5. Elle est définie comme suit :

$$\begin{aligned} |\neg c|^t &= !\ |c|^t \\ |c \wedge d|^t &= |c|^t\ \&\&\ |d|^t \\ |e_1 = e_2|^t &= \|e_1\|^t = \|e_2\|^t \\ |e_1^{int}\ opCom^{int}\ e_2^{int}|^t &= \|e_1^{int}\|^t\ \tau(opCom^{int})\ \|e_2^{int}\|^t \\ &\ldots \end{aligned}$$

pour toutes conditions $c, d \in Cond_{\Sigma,\Omega,V,X}$, toutes expressions $e_1, e_2, e_1^{int}, e_2^{int} \in Exp_{\Sigma,\Omega,V,X}^{Aspect-UML}$ et où chaque opérateur $opCom^{int} \in \{<, >, \leq, \geq\}$ est traduit vers un opérateur Alloy prédéfini équivalent $\tau(opCom^{int}) \in \{<, >, =<, >=\}$ (voir définition 5.3.3).

Par ailleurs, si la syntaxe abstraite de Aspect-UML définit d'autres opérateurs de comparaison (qui ne sont pas prédéfinis dans Alloy), alors ceux-ci se verront traduits vers des prédicats Alloy, qui prennent comme arguments les expressions à comparer et retournent comme résultat la valeur `Vrai` ou `Faux`.

Formellement, la traduction d'une opération (point de jointure) de $Spec_{Aspect-UML}$, vers une signature de $Spec_{Alloy}$, est définie par les règles sémantiques données à la table 5.3. Notons qu'initialement, les types *Operation*, JP et *Advice* sont introduits dans les types de $Spec_{Alloy}$ et que les signatures Sig^{Op}, Sig^{JP}, Sig^{Ad} définissant respectivement ces types sont toujours incluses dans la signature globale SIG. Aussi, le fait *indivisible* contraignant ces opérations est systématiquement ajouté à l'ensemble des faits FC de $Spec_{Alloy}$. Chaque nom de méthode m correspondant à un point de jointure est traduit en un nom de signature dans Alloy étendant JP (règle (14)). La formule de la méthode m est traduite en une signature Alloy (règle (17)), où chaque argument de m est traduit vers un champ de cette signature (règle (15)). Aussi, un champ appelé $self$ est rajouté à la signature pour dénoter l'objet invoqué

TAB. 5.3 – Règles de traduction pour les points de jointure.

Sachant que :

1. $Operation, JP, Advice \in S \mid JP \leq^S Operation$ et $Advice \leq^S Operation$,
2. $sig^{Op} = \langle \{Operation\}, \leq^{Op}, \{begin:Time, end:Time\} \rangle$,
3. $sig^{JP} = \langle \{JP\}, \leq^{JP}, \{error:boolean\}$,
4. $sig^{Ad} = \langle \{Ad\}, \leq^{Ad}, \{error:boolean\}$,
5. $sig^{Op}, sig^{JP}, sig^{Ad} \in SIG$,
7. $boolean \in S$ et $Sig^{boolean} \in SIG$.
8. $fact^{indivisible} = \langle indivisible, \varphi^{indivisible} \rangle \in FC$ tel que $indivisible \in Fc$ et $\varphi^{indivisible} = \{all\ op : JP + Advice \mid op.end = op.begin.next\}$

Pour chaque point de jointure $jp = \langle kind, m \rangle$ de $Spec_{Aspect-UML}$ tel que $m \in M$ et $\Phi^m = \langle m, args, pre, post \rangle \in \Phi^c$ avec $c = \langle \Omega^c, V, \Phi^c \rangle$; $\Omega^c = \langle \{c\}, \leq^c, M^c \rangle$; $args = \{x_i : t_i \mid x_i \in X,\ t_i \in T \cup C,\ i = 1..n\}$; $pre = \{\phi_j,\ j = 1..m\}$ et $post = \{\psi_k,\ k = 1..l\}$,

nous appliquons les règles suivantes :

$$\tau(m) \in S \mid \tau(m) = m \ \wedge m \leq^m JP \text{ tel que } \leq^m \subset \leq^S \tag{5.14}$$

$$\forall i = 1..n, \tau(x_i) \in R^m \mid \ \tau(x_i) = x_i \ \wedge \tau(x_i) : \tau(t_i) \tag{5.15}$$

$$self : \tau(c) \in R^m \tag{5.16}$$

$$\tau(\Phi^m) = \langle \{\tau(m)\}, \leq^m, R^m \rangle \ \in SIG \tag{5.17}$$

$$\tau(pre \cup post) = \langle fact^m, \varphi^m \rangle \in FC \quad tel\ que \quad fact^m \in Fc\ et \tag{5.18}$$
$$\varphi^m = (all\ o : m \mid (\ \underset{j=1..m}{\&\&}\ \| o.\phi_j \|^{begin})$$
$$=> (\ \underset{k=1..l}{\&\&}\ \| o.\psi_k \|^{end}) \&\& \ (o.error = False)$$
$$else \quad (o.error = True))$$

par la méthode, ce champ a évidement le type correspondant au type de l'objet (règle (16)). Comme nous l'avons déjà expliqué, l'ensemble des pré et postconditions contraignant la méthode m est traduit vers un fait contraignant la signature Sig^m (règle (18)). Ce fait est constitué d'un nom $fact^m$ et d'une formule φ^m. Informellement la formule φ^m exprime ce qui suit : lors de toute invocation du point de jointure m, si les préconditions sont vraies au début de l'exécution de m alors les postconditions seront assurées à la fin de l'exécution et aucune erreur ne sera signalée pour cette invocation, sinon il y a forcément une erreur dans l'exécution.

La traduction des advices vers Alloy se fait de façon similaire. Cependant, n'ayant pas de noms, la règle (14) ne s'applique pas aux advices. Nous avons à la place adopté la convention suivante : un advice est toujours désigné par le nom du point de coupure implémenté par l'advice, concaténé au nom de l'aspect le contenant. Un advice est alors tarduit vers une signature de ce nom.

De notre application de téléphonie, voici un exemple de spécification d'un point de jointure en Alloy. Il s'agit du point de jointure `Connection.complete()`, traduit vers la signature `Complete`.

```
sig Complete extends JP { self : Connection }

fact complete {all o : Complete |
      (o.self.status.begin = disconnected) &&    //preconditions
      (o.self.origin.d_status = idle) &&
      (o.self.origin.current = null)
      => (o.self.status.end = connected) &&    //postconditions
          (o.self.origin.d_status = busy) &&
          (o.self.destination.d_status = busy) &&
          (o.self.origin.current = self) &&
          (o.self.origin.current = self) &&
          (o.error = False)
      else (o.error = True) }    //erreur
```

Le point de jointure décrit par la signature `Complete` satisfait implicitement les contraintes génériques imposées pour tous les atomes `JP`, tout en satisfaisant les pré et postconditions décrites par le fait *complete* contraignant la signature `Complete`. Ce point de jointure a un seul paramètre `self` représentant l'objet *Connection* invoqué par l'appel de méthode. Pour accomplir une opération `Complete` (ligne 10 du fait `complete`), une connexion doit a priori être déconnectée et la ligne de l'appelant (l'origine) doit être libre (lignes 2 à 4 du fait) autrement une erreur est signalée (ligne 11 du fait). À la fin de son exécution, l'opération `Complete` doit

remettre la connexion dans l'état connecté (ligne 5) et son origine et sa destination sont occupées (lignes 6 à 9). Les valeurs associées aux états de la connexion respectivement avant et après son établissement sont saisies par les valeurs des champs `begin` et `end` de l'opération `Complete`.

Le problème des frames (*frame problem*). Le fait d'ajouter le temps aux champs des signatures dans la spécification Alloy nous amène à raisonner avec des domaines dynamiques. En passant d'une instance de modèle au temps t_1 vers une autre instance de modèle au temps t_2, les valeurs des champs contraints (par les pré et les postconditions) se verront affecter les valeurs spécifiées par les contraintes correspondantes, par contre les valeurs des champs non contraints se verront affecter des valeurs aléatoires par le solveur de contraintes Alloy. Ce problème, reconnu initialement en intelligence artificielle, est connu sous le nom du *frame problem* [MH69]. La résolution de ce prolème utilise les *conditions des frames* (*frame conditions*) qui sont des contraintes spécifiant que : "*une opération donnée ne doit changer que les champs spécifiés dans ses pré et postconditions et laisser tous les autres champs inchangés*".

Ainsi, dans notre traduction de Aspect-UML vers Alloy, nous devons spécifier non seulement les champs que chaque opération modifie dans le système (en utilisant les postconditions), mais aussi nous devons spécifier explicitement tous les autres champs qui restent inchangés. Lister explicitement toutes ces conditions de frames pour chaque opération peut être une tâche fastidieuse surtout si le nombre des champs inchangés est important. Une approche plus succinte consiste à ajouter des axiomes (appelés *explanation closure axioms* dans [BMR95]) pour indiquer quelles opérations modifient quels champs. Dans le cas de la traduction des modèles Aspect-UML vers Alloy, cette solution consiste à ajouter un fait indiquant que "*si un champ f a changé, alors une opération (advice ou point de jointure) e a eu lieu*". Cette solution rend le codage des conditions de frames succint, générique et

modulaire. Le patron de ce fait est comme suit :

```
fact unchanged { all t : Time - last | some e : JP + Advice {
                 (! f.t = f.t.next => (e in JP1 && e.error = False)) &&
                 (! g.t = g.t.next => (e in Advice1 && e.error = False))
                 ...}}
```

où f, g,... sont les champs modifiés respectivement par les opérations $JP1$, $Advice1$, ...

TAB. 5.4 – Règles de spécification pour les conditions de frames.

Soient Φ et Ψ respectivement les ensembles de toutes les formules de méthodes et de toutes les formules d'advices de $Spec_{Aspect-UML}$.
Soit V l'ensemble de tous les noms d'attributs de $Spec_{Aspect-UML}$.
Soit $Var(\Phi \cup \Psi)$ la fonction qui retourne l'ensemble de toutes les variables apparaissant dans $\Phi \cup \Psi$.

Pour tout $v_i \in Var(\Phi \cup \Psi)$, $i = 1..n$, tel que $v_i \in V$ et $v_i \in Var(post^{\phi_j})$[1] $\forall j = 1..m$ où $\phi_j \in \Phi \cup \Psi$ avec $\tau(\phi_j) = \langle \{op_j\}, \leq^{op_j}, R^{op_j} \rangle$,

nous définissons la condition des frames suivante :

$$\varphi_i = !(\tau(v_i).t = \tau(v_i).t.next) => (e \in \underset{j=1..m}{+} op_j) \&\& (e.error = False) \quad (5.19)$$

Le fait suivant est alors ajouté à $Spec_{Alloy}$:

$$\begin{aligned} fact^{unchanged} &= \langle unchanged, \varphi^{unchanged} \rangle \in FC \quad telque \quad (5.20) \\ unchanged &\in Fc \quad et \\ \varphi^{unchanged} &= (all\ t : Time - last \mid some\ e : JP + Advice \{ \underset{i=1..n}{\&\&} \varphi_i \}) \end{aligned}$$

[1] $post^{\phi_j}$ est l'ensemble des postconditions de l'opération décrite par ϕ_j.

De façon plus formelle, les conditions de frames sont spécifiées tel qu'indiqué par les règles de la table 5.4. La règle (19) calcule la condition des frames pour chaque attribut apparaissant dans les postconditions d'un modèle Aspect-UML. Cette règle stipule que le champ correspondant à cet attribut ne peut être modifié

que par les opérations l'ayant dans leurs postconditions. À partir de là, le fait unchanged dont la formule correspond à la conjonction de toutes les conditions de frames est rajouté à la spécification Alloy (règle (20)).

Dans le cas de notre application de téléphonie, les postconditions données dans le modèle Aspect-UML de la figure 3.4, page 72, montrent que les attributs `status`, `d_status` et `current` sont mis à jour seulement par les opérations `Complete` et `Drop`, `startTime` et `connectionTime` sont mis à jour respectivement par les opérations `OpCompleteTiming` et `OpDropTiming` et `charge` est mis à jour par l'opération `OpDropBilling`. Ceci se traduit en Alloy par le fait unchanged qui suit.

```
fact unchanged {all t : Time - last | some e : JP + Advice {
  ((! status.t=status.t.next) => (e in Complete + Drop && e.error=False))
  &&
  ((! d_status.t=d_status.t.next) => (e in Complete + Drop && e.error=False))
  &&
  ((! current.t=current.t.next) => (e in Complete + Drop && e.error=False))
  &&
  ((! startTime.t = startTime.t.next) => (e in OpCompleteTiming
    && e.error = False))
  &&
  ((! connectionTime.t = connectionTime.t.next) => (e in OpDropTiming
    && e.error = False))
  &&
  ((! charge.t = charge.t.next) => (e in OpDropBilling && e.error = False))}}
```

5.5.3.2 Traduction des contraintes de passage de contexte

Dans notre profil Aspect-UML, un point de coupure est utilisé pour exposer le contexte d'un certain nombre de points de jointure sous une interface commune. Un aspect est dit implémenter un point de coupure s'il fournit un advice à exécuter soit avant, soit après les points de jointure spécifiés par le point de coupure. Un advice entrecoupant un point de jointure doit être capable d'accéder au contexte dans lequel le point de jointure est exécuté. Les points de coupure sont utilisés pour passer le contexte des points de jointure (*c-à-d* ses paramètres actuels) aux advices de façon générique. Dans les modèles Aspect-UML, chaque point de coupure est

annoté avec un ensemble de fonctions de liaison de contexte décrivant comment relier les arguments de ses points de jointure aux paramètres des advices.

La traduction de ces fonctions de liaison de contexte vers Alloy est définie formellement par la règle (21) donnée à la table 5.5. Cette règele stipule que la traduction consiste, en fait, à créer un prédicat $passCtxtJP_j(jp{:}JP)$ pour chaque point de coupure jp_j forçant le mapping des paramètres x_l du point de jointure jp_j vers les arguments z_l de chaque advice ad_k qui l'entrecoupe.

TAB. 5.5 – Règles de traduction pour les contraintes de passage de contexte.

Pour chaque point de coupure lié $\langle pc_i, B_i\rangle,\ i = 1..n$ de $Spec_{Aspect-UML}$, où $pc_i = \langle p_i, param_i, jpSet_i\rangle$ et $B_i = \{(jp_j, binding^j_{args^{jp_j}, param_i}) \mid jp_j \in jpSet_i,\ j = 1..m\}$
Soit $AdSet^{p_i} = \{\psi^k = \langle \gamma^k, pc_i, args^k, pre^k, post^k\rangle,\ k = 1..s\}$ l'ensemble de toutes les formules d'advices implémentant p_i tel que $args^k = \{z_1, ..., z_r\}$.
Pour chaque $jp_j = \langle kind_j, m_j\rangle \in jpSet_i,\ j = 1..m$ avec $\phi^{m_j} = \langle m_j, arg^{jp_j}, pre^{jp_j}, post^{jp_j}\rangle$ et $binding^j_{args^{jp_j}, param_i} = \{\langle x_l, y_l\rangle, l = 1..r\}$.

Sachant que : $\tau(\phi^{m_j}) = \langle\{m_j\}, \leq^{m_j}, R^{m_j}\rangle,\ j = 1..m$ et
$\tau(\psi^k) = \langle\{ad_k\}, \leq^{ad_k}, R^{ad_k}\rangle,\ k = 1..s.$

Le predicat suivant est ajouté à $Spec_{Alloy}$ **:**

$$Pred^{passCtxtJP_j} = \langle passCtxtJP_j, param, \varphi^{passCtxtJP_j}\rangle \in PR \mid Pred^{passCtxtJP_j} = \tau(binding^j_{args^{jp_j}, param_i}) \qquad (5.21)$$

où : $passCtxtJP_j \in \Pr$
$param = \{jp : JP\}$
$\varphi^{passCtxtJP_j} = (\ \underset{k=1..s}{\&\&}\ (\ \underset{l=1..r}{\&\&}\ (\varphi_l))) \mid$
$\forall l = 1..r \quad \varphi_l = \left\{ \begin{array}{ll} (m_j.\tau(x_l) = ad_k.\tau(z_l)) & (si\ x_l \neq target) \\ (m_j.self = ad_k.\tau(z_l)) & (si\ x_l = target) \end{array} \right\}$

Dans notre application de téléphonie, le prédicat `passCtxtDrop(jp :JP)`, donné ci-après, permet de forcer le passage de contexte du point de jointure `Drop` aux deux advices `OpDropTiming` et `OpDropBilling` qui l'entrecoupent.

```
pred passCtxtDrop( p :PJ ) {
    (Drop.self = OpDropTiming.c) &&
    (Drop.self = OpDropBilling.c)) }
```

5.5.4 Composition des advices

Tout comme pour la traduction de Aspect-UML vers les réseaux de Petri, présentée au chapitre précédent, la traduction vers Alloy doit montrer comment les advices conflictuels à un même point de jointure doivent être composés entre eux avant d'être tissés au modèle de base.

Rappelons que selon le paradigme aspect, quand un point de jointure est atteint lors d'une exécution, un ou plusieurs advices peuvent avoir été déclarés pour s'exécuter à ce point. Des conflits peuvent alors survenir si deux advices ou plus sont de même type `before` ou `after`. Aspect-UML permet aux développeurs de définir une relation de précédence entre aspects. Si une telle relation est définie alors la composition des advices sera faite de façon séquentielle. Autrement, aucun ordre n'est imposé et les advices peuvent être composés dans n'importe quel ordre, de façon non déterministe. Ainsi, à chaque point de jointure, chaque collection d'advices conflictuels de même type (`before` ou `after`) résultera en un seul advice composite à exécuter respectivement avant ou après le point de jointure.

La composition d'advices peut être définie naturellement par une opération récursive. Or Alloy ne permet pas la récursivité dans les prédicats et les fonctions ; par contre, il accepte les relations récursives. Ainsi, tout comme pour les points de jointure et les advices, nous avons choisi de réifier l'opération de composition en une signature abstraite Alloy appelée `Composition` qui étend la signature `Operation` définie à la section précédente. La signature `Composition` introduit deux nouvelles relations récursives : `comp1` qui relie la composition à un advice ou une composition d'advices et `comp2` qui relie la composition à une autre composition. En d'autres termes, une composition est soit un advice, soit une paire de compositions.

Comme nous l'avons mentionné plus haut, nous devons distinguer la composi-

```
abstract sig Composition extends Operation {
        comp1 : Composition + Advice,
        comp2 : lone Composition }
```

tion séquentielle de la composition non déterministe. Ceci peut être fait en définissant deux sous-signatures `SeqComposition` et `NDComposition` qui étendent `Composition`. Le comportement particulier de chaque type de composition est spécifié par un fait contraignant les temps de début et de fin de la composition et de ses composants.

Dans Alloy, les deux types de composition séquentielle et non déterministe sont modélisés respectivement par les signatures `SeqComposition` et `NDComposition` données ci-après.

```
abstract sig SeqComposition extends Composition{ }

fact seqComposition { all s : SeqComposition |
    (s.begin= s.comp1.begin) &&
    (no s.comp2 => s.end = s.comp1.end
        else (s.end = s.comp2.end &&
        lte[s.comp1.end, s.comp2.begin])) }


abstract sig NDComposition extends Composition { }

fact ndComposition { all nd : NDComposition |
    no nd.comp2 => (nd.begin = nd.comp1.begin &&
                    nd.end= nd.comp1.end)
        else ((nd.begin = nd.comp1.begin &&
            nd.end = nd.comp2.end &&
            lte[nd.comp1.end, nd.comp2.begin] )
            | |
            (nd.begin= nd.comp2.begin &&
            nd.end = nd.comp1.end &&
            lte[nd.comp2.end, nd.comp1.begin])) }
```

Ces signatures sont contraintes par des faits imposant les contraintes de coordination. Remarquons l'utilisation du prédicat `lte` (*less than or equal*) pour assurer que la fin d'une opération précède le début de l'opération suivante . Notons que si l'égalité est utilisée à la place, aucun autre entrelacement d'opérations ne sera possible entre ces deux opérations.

Les figures 5.1 et 5.2 schématisent les contraintes de coordination que les types de composition séquentielle et non déterministe doivent satisfaire respectivement.

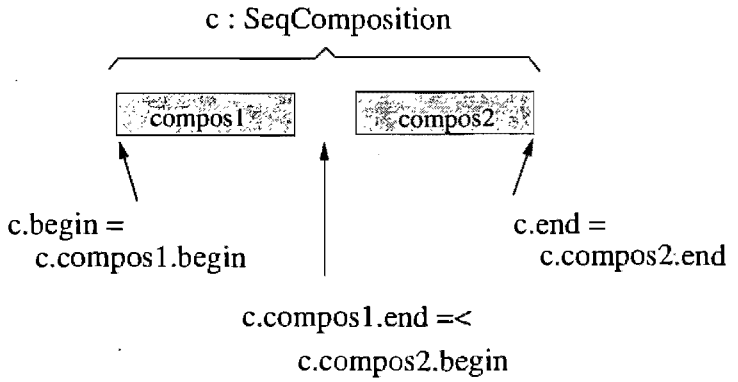


FIG. 5.1 – Contraintes de coordination pour `SeqComposition`.

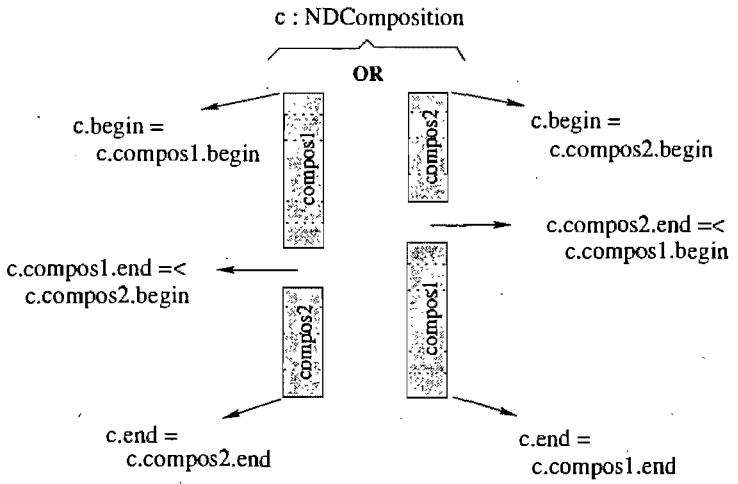


FIG. 5.2 – Contraintes de coordination pour `NDComposition`.

Étant données ces signatures, les ensembles conflictuels d'advices à exécuter à

un même point de jointure peuvent être traduits facilement en Alloy.

Soit $AdSet^p_\gamma$ l'ensemble des advices (de type γ implémentant le point de coupure p) partiellement ordonné [4] par la relation de précédence *prio* définie sur les aspects. Soit $L_i^{p,\gamma} = \{ad_1^i, .., ad_{m_i}^i\}$ pour $i = 1..n$, toutes les listes des advices ordonnés (*c-à-d* soit $m_i = 1$, soit $ad_j^i < ad_{j+1}^i$ pour chaque $j = 1, ..m_i - 1$ et $(ad_{j'}^{i'} \not< ad_j^i) \wedge (ad_j^i \not< ad_{j'}^{i'})$ $\forall i \neq i', \forall j' = 1..m_{i'}$) de $AdSet^p_\gamma$ tel que pour chaque $i = 1..n$ et pour chaque $j = 1..m_i$, ad_j^i est traduit en une signature Alloy de type ad_j^i étendant `Advice`. La traduction de la composition des advices de $AdSet^p_\gamma$ en Alloy consiste alors à :

1. créer une collection de signatures $seq_1^i, ...seq_{m_i}^i$ étendant `SeqComposition`, pour chaque $i = 1..n$;
2. créer une collection de signatures $nd_1, ...nd_n$ étendant `NDComposition` ;
3. ajouter des faits pour contraindre les signatures seq_j^i, $1 \leq i \leq n$, $1 \leq j \leq m_i$ et les signatures $nd_i, 1 \leq i \leq n$ à se comporter comme si elles étaient composées dans une structure ayant la forme suivante :

$$
nd_1 = \left\{ + \quad seq_1^1 = \underbrace{\underbrace{\underbrace{\mathbf{ad_1^1}; \mathbf{ad_2^1}; ...; \underbrace{\mathbf{ad_{m_1}^1}; \emptyset}_{seq_{m_1}^1}}_{seq_2^1}}_{seq_1^1}} \right.
$$

$$
\left. nd_2 = \left\{ + \quad seq_1^2 = \underbrace{\underbrace{\underbrace{\mathbf{ad_1^2}; \mathbf{ad_2^2}; ...; \underbrace{\mathbf{ad_{m_2}^2}; \emptyset}_{seq_{m_2}^2}}_{seq_2^2}}_{seq_1^2}} \right. \right.
$$

$$
\left. \left. nd_3 = \left\{ + \ldots \quad \vdots \right. \right. \right.
$$

$$
\left. \left. \left. nd_n = \left\{ \begin{array}{l} seq_1^n = \underbrace{\underbrace{\underbrace{\mathbf{ad_1^n}; \mathbf{ad_2^n}; ...; \underbrace{\mathbf{ad_{m_n}^n}; \emptyset}_{seq_{m_n}^n}}_{seq_2^n}}_{seq_1^n}} \\ \\ \emptyset \end{array} \right. \right. \right. \right. \right\}
$$

[4] L'ordre est décroissant sur *prio* si $\gamma = before$, *c-à-d* $ad_2 < ad_1$ ssi $(A_1, A_2) \in prio$, où A_1 et A_2 sont les aspects contenant ad_1 et ad_2 respectivement. Par contre, l'ordre est croissant si $\gamma = after$, *c-à-d* $ad_1 < ad_2$ ssi $(A_1, A_2) \in prio$.

Pour cela, à chaque signature seq^i_j, $i = 1..n$, $j = 1..m_i - 1$ est ajoutée la contrainte $seq^i_j.comp1 = ad^i_j$ && $seq^i_j.comp2 = seq^i_{j+1}$. Puisqu'elle termine la séquence, la signature $seq^i_{m_i}$ est contrainte par $seq^i_{m_i}.comp1 = ad^i_{m_i}$ && (no $seq^i_{m_i}.comp2$).

De façon similaire, à chaque signature $nd_i, i = 1..n-1$ est ajoutée la contrainte $nd_i.comp1 = seq^1_i$ && $nd_i.comp2 = nd_{i+1}$. Aussi, puisqu'elle termine la séquence, nd_n est contrainte par $nd_n.comp1 = seq^n_1$ && $nd_n.comp2 = \emptyset$.

Les signatures seq^i_1, $i = 1..n$ décrivent les compositions séquentielles des advices ordonnés de $AdSet^p_\gamma$, et la signature nd_1 dénote les compositions non déterministes des advices non ordonnés de $AdSet^p_\gamma$. Les faits contraignant `NDComposition` et `SeqComposition` assurent que nd_1 décrit tous les entrelacements possibles des advices de $AdSet^p_\gamma$ selon la relation de précédence déclarée. Cette spécification de la composition des advices en Alloy est présentée formellement à la table 5.6.

Notons que l'algorithme présenté traite les cas d'exception suivants :

1. si tous les advices sont ordonnés de façon séquentielle (cas où n=1), alors dans ce cas aucune signature nd_i de type `NDComposition` ne sera créée ;
2. si une liste $L^{p,\gamma}_i$ donnée contient un seul élément (cas où $m_i = 1$), alors pour cette liste aucune signature seq^i_j ne sera créée.

Ce traitement d'exception, nous permet surtout de limiter le nombre de signatures de type `Composition` dans la spécification Alloy.

Dans notre application de téléphonie, la signature montrée ci-après décrit la composition non déterministe des advices `OpDropTiming` et `OpDropBilling` (nous avons éliminé l'ensemble vide à la fin de la composition non déterministe et dans chaque composition séquentielle). Il est cependant important de noter que, selon notre approche de modélisation, la composition séquentielle de ces deux advices

TAB. 5.6 – Spécification de la composition des advices.

Pour chaque point de coupure $pc = \langle p, param, jpSet \rangle$, soit $AdSet^p_\gamma$ l'ensemble de tous les advices de type γ implémentant pc.
Soient $L_i^{p,\gamma} = \{ad^i_j, j = 1,..,m^i\},\ i = 1..n$ les listes des advices ordonnés de $AdSet^p_\gamma$ tels que $(ad^i_j < ad^i_{j+1}\ \forall j = 1..m^i{-}1) \wedge ((ad^i_j \not< ad^{i'}_{j'}) \wedge (ad^{i'}_{j'} \not< ad^i_j)\ \forall i \neq i', j' = 1..m^{i'})$

Sachant que :
1. $\langle \{Composition\}, \leq^{comp}, \{comp1 : Composition{+}Advice, comp2 : Composition\} \rangle \in SIG \mid Composition \leq^{comp} Operation$;
2. $\langle \{SeqComposition\}, \leq^{seq}, \emptyset \rangle$, $\langle \{NDComposition\}, \leq^{nd}, \emptyset \rangle \in SIG \mid SeqComposition \leq^{seq} Composition \wedge ndComposition \leq^{nd} Composition$;
3. $\langle \{f^{seqComposition}, \varphi^{seq}\}$[1] $\langle \{f^{ndComposition}, \varphi^{nd}\}$[2] $\in FC$;
4. $\tau(ad^i_j) = \langle \{ad^i_j\}, \leq^{ad^i_j}, R^{ad^i_j} \rangle \in SIG \mid ad^i_j \leq^{ad^i_j} Advice,\ \forall 1 = 1..n,\ j = 1..m^i$;

on applique l'algorithme suivant :

Si $n = 1$ **alors**
  pour $j{=}1..m^1$ créer
    $Sig^{seq^1_j} = \langle \{seq^1_j\}, \leq^{seq^1_j}, \emptyset \rangle \in SIG \mid seq^1_j \leq^{seq} SeqComposition$
    $fact^{seq^1_j} = \langle f^{seq^1_j}, \varphi^{seq^1_j} \rangle \in FC \mid f^{seq^1_j} \in Fc$ et
    $\varphi^{seq^1_j} = \begin{cases} \{all\ s : seq^1_j \mid s.compos1 = ad^1_j \&\& s.compos2 = seq^1_{j+1}\}\ (si\ j \neq m^1) \\ \{all\ s : seq^1_j \mid s.compos1 = ad^1_j \&\& no\ s.compos2\}\ (si\ j = m^1) \end{cases}$
sinon
  pour $i = 1..n$ créer
    $Sig^{nd_i} = \langle \{nd_i\}, \leq^{nd_i}, \emptyset \rangle \in SIG \mid nd_i \leq^{nd} NDComposition$
    $fact^{nd_i} = \langle f^{nd_i}, \varphi^{nd_i} \rangle \in FC \mid f^{nd_i} \in Fc$ et
    si $m_i = 1$ **alors**
      $\varphi^{nd_i} = \begin{cases} \{all\ c : nd_i \mid c.compos1 = ad^i_{m_i}\ \&\&\ c.compos2 = nd_{i+1}\}\ (si\ \ i \neq n) \\ \{all\ c : nd_i \mid c.compos1 = ad^i_{m_i}\ \&\&\ no\ c.compos2\}\ (si\ \ i = n) \end{cases}$
    sinon
      pour $j{=}1..m^1$ créer
        $Sig^{seq^i_j} = \langle \{seq^i_j\}, \leq^{seq^i_j}, \emptyset \rangle \in SIG \mid seq^i_j \leq^{seq} SeqComposition$
        $fact^{seq^i_j} = \langle f^{seq^i_j}, \varphi^{seq^i_j} \rangle \in FC \mid f^{seq^i_j} \in Fc$ et
        $\varphi^{seq^i_j} = \begin{cases} \{all\ s : seq^i_j \mid s.compos1 = ad^i_j \&\& s.compos2 = seq^i_{j+1}\}\ (si\ j \neq m^i) \\ \{all\ s : seq^i_j \mid s.compos1 = ad^i_j \&\& no\ s.compos2\}\ (si\ j = m^i) \end{cases}$
      $\varphi^{nd_i} = \begin{cases} \{all\ n : nd_i \mid n.compos1 = seq^i_1\ \&\&\ n.compos2 = nd_{i+1}\}\ (si\ \ i \neq n) \\ \{all\ n : nd_i \mid n.compos1 = seq^i_1\ \&\&\ no\ n.compos2\}\ (si\ \ i = n) \end{cases}$

[1] $f^{seqComposition}$ et φ^{seq} sont respectivement le nom et la formule décrivant le fait contraignant la signature $SeqComposition$.
[2] $f^{ndComposition}$ et φ^{nd} sont respectivement le nom et la formule décrivant le fait contraignant la signature $NDComposition$.

ne diffère de celle donnée ci-bas (pour la séquence non déterministe) que par la modification du nom de la signature mère. En fait, pour modéliser la composition séquentielle de ces deux advices, il suffit de supprimer le nom de la signature `NDComposition` et de le remplacer par `SeqComposition`.

```
sig Compos extends NDComposition { }
fact compos { all s : Compos |
                    s.comp1= OpDropTiming &&
                    s.comp2=OpDropBilling }
```

5.5.5 Tissage

Une fois composés, les advices doivent être tissés aux points de jointure. En suivant la même approche de réification, nous définissons une signature `Weaving` étendant `Operation` pour décrire l'opération de tissage des advices aux points de jointure. Le processus de tissage nécessite trois paramètres que nous définissons comme champs dans la signature `Weaving` : le point de jointure (`jp :one JP`) et deux compositions d'advices à tisser respectivement avant et/ou après ce point de jointure (`beforeAdvice, afterAdvice :` *`lone`* `Composition`). Notons le type *lone* de ces deux champs, pour signifier qu'un advice de type `before` et/ou un advice de type `after` peut ou non être présent à ce point de jointure.

```
abstract sig Weaving extends Operation {
    jp : one JP, beforeAdvice, afterAdvice : lone Composition }
```

La signature `Weaving` est contrainte par le fait `mustWeave` afin d'imposer qu'il y ait au moins une composition d'advices à tisser (soit avant, soit après le point de

```
fact mustWeave { all w :Weaving |
      w.beforeAdvice!=none | | w.afterAdvice!= none ) }

fact weavingBefore { all w :Weaving |
      w.afterAdvice=none => (w.begin=w.beforeAdvice.begin &&
      w.end=w.jp.end && w.beforeAdvice.end=w.jp.begin ) }

fact weavingAfter { all w :Weaving |
      w.beforeAdvice =none => (w.begin=w.jp.begin &&
      w.end=w.afterAdvice.end && w.jp.end=w.afterAdvice.begin )}

fact weavingBeforeAfter { all w :Weaving |
      ( w.beforeAdvice!= none && w.afterAdvice!=none )=>
            ( w.begin = w.beforeAdvice.begin &&
             w.end = w.afterAdvice.end &&
             w.beforeAdvice.end=w.jp.begin &&
             w.jp.end=w.afterAdvice.begin ) }
```

jointure). Les faits `weavingBefore`, `weavingAfter`, `weavingBeforeAfter` décrivent respectivement les trois possibilités d'exécution de l'opération de tissage : (1) tissage avant le point de jointure, (2) tissage après le point de jointure et (3) tissage avant et après le point de jointure.

Ces faits coordonnent l'exécution séquentielle de l'advice composite `before` (si présent) suivie par l'opération du point de jointure, et par l'advice composite `after` (si présent). Encore dans ce cas, la coordination entre les opérations est rendue possible grâce aux champs `begin` et `end` du super-type `Operation`. La formalisation du processus de tissage aux points de jointure (d'un point de coupure donné) est définie par les règles de la table 5.7. Notons que tel que précisé par la règle (23), lors du tissage à un point de jointure jp_i, le passage de contexte du point de jointure aux advices qui s'y tissent est réalisé grâce au prédicat $\texttt{passCtxtJP}_i\texttt{(w.jp)}$ déjà défini (c.f. table 5.5, page 158).

TAB. 5.7 – Spécification du processus de tissage.

Soit $pc = \langle p, param, jpSet \rangle$ un point de coupure, tel que $AdSet^{p}_{before}$ et $Adset^{p}_{after}$ sont respectivement, les ensemble d'advices respectivement de type `before` et de type `after` implémentant pc.

Sachant que :
1. $\langle \{Weaving\}, \leq^{w}, \{jp : JP, beforeAdvice, afterAdvice : Composition\} \rangle \in SIG \mid Weaving \leq^{w} Operation$;
2. $\langle f^{must}, \varphi^{must} \rangle^{1} \langle f^{before}, \varphi^{before} \rangle^{1} \langle f^{after}, \varphi^{after} \rangle^{1} \langle f^{beforeAfter}, \varphi^{beforeAfter} \rangle^{1} \in FC$;
3. nd^{before} et nd^{after} sont les signatures décrivant la composition des advices de $AdSet^{p}_{before}$, $AdSet^{p}_{after}$ respectivement ; (table 5.6)

Pour chaque point de jointure $jp_i = \langle kind_i, m_i \rangle \in jpSet$, décrit par la formule ϕ_i tel que $\tau(\phi_i) = \langle \{m\}, \leq^{m}, R^{m} \rangle \in SIG$;

on créé :

$$Sig^{weaving^{p}_{i}} = \langle \{w^{p}_{i}\}, \leq^{w^{p}_{i}}, \emptyset \rangle \in SIG \mid weaving^{p}_{i} \leq^{weaving^{p}_{i}} Weaving \quad (5.22)$$

$$fact^{weaving^{p}_{i}} = \langle f^{weaving^{p}_{i}}, \varphi^{weaving^{p}_{i}} \rangle \in FC \mid \quad (5.23)$$

où :
$$f^{weaving^{p}_{i}} \in Fc$$
$$\varphi^{weaving^{p}_{i}} = \{all\ w : weaving^{p}_{i} \mid w.jp = m\ \&\&\ passCtxtJP_i(w.jp)\ w.adviceBefore = nd^{before} \&\&, w.adviceAfter = nd^{after}\}$$

[1] f^{must}, f^{before}, f^{after} et $f^{beforeAfter}$ sont respectivement les noms des faits $mustWeave$, $weavingBefore$, $weavingAfter$ et $weavingBeforeAfter$ contraignant la signature $Weaving$.

Dans notre application de téléphonie, le tissage de l'advice composite `Compos` au point de jointure `Drop` est définie par la signature `WeavingAfterDrop`. Cette signature est contrainte par le fait `weavingAfterDrop` afin de spécifier les paramètres actuels de l'opération de tissage et les contraintes de passage de contexte pour le point de jointure `Drop`.

Chaque point de jointure dans le modèle Aspect-UML de l'application de téléphonie sera traité de la même façon que le point de jointure `Drop`. Ce traitement implique (1) la composition des advices en réutilisant les signatures abstraites `SeqComposition` et `NDComposition`, et (2) le tissage des advices composites au

```
sig WeavingAfterDrop extends Weaving { }
fact weavingAfterDrop { all s : WeavingAfterDrop |
                    ( w.jp= Drop) && passCtxtDrop(w.jp) &&
                    ( w.beforeAdvice=none ) && ( w.afterAdvice=Compos) }
```

point de jointure en utilisant la signature `Weaving`.

5.6 Discussion

Certains travaux dans la littérature ont déjà considéré Alloy comme langage cible pour la traduction de leurs systèmes logiciels et ce afin de les analyser automatiquement.
Dans [MGB04], les auteurs proposent la traduction des diagrammes de classes UML contraints par des invariants OCL vers Alloy. Cependant, la traduction des pré et postconditions n'est pas considérée. Basé sur une approche MDA, l'outil de transformation présenté dans [BA05,ABGR07] traite quant à lui les pré et postconditions OCL des méthodes. Ces contraintes sont traduites en Alloy, mais encore leur exécution séquentielle ou non déterministe n'est pas abordée. Alloy a aussi été utilisé dans [NT04] pour vérifier si un invariant donné est satisfait avant et après chaque transformation de modèle décrivant le tissage des aspects. Cette approche se limite à la vérification des invariants. [Vaz03] montre comment les spécifications structurelles peuvent être utilisées comme outil pour trouver des bugs dans les systèmes logiciels et présente une technique basée sur le langage de modélisation Alloy.

Comparée aux approches existantes ayant adopté Alloy, notre traduction des modèles Aspect-UML vers Alloy donne un nombre d'idées appréciable quant à la façon de procéder. Elle fournit une approche systématique pour la traduction vers Alloy des modèles orientés objet/aspect. Cette transformation est entièrement formalisée et automatisable. De plus, elle couvre la traduction des propriétés

structurelles et comportementales des objets, tout comme la traduction de leur dynamique.

5.7 Conclusion

Ce chapitre a présenté la formalisation de notre profil Aspect-UML en Alloy. Dans un premier temps, les syntaxes abstraites de Alloy et de Aspect-UML ont été définies. Ensuite, les étapes de traduction ont été décrites et présentées sous forme de règles formelles dans un style dénotationnel. Ce processus de traduction est un élément clé dans l'achèvement de la vérification formelle des interactions dans un modèle Aspect-UML en utilisant Alloy qui sera abordée dans le chapitre suivant.

La traduction présentée est cohérente avec la sémantique en termes de réseaux de Petri présentée au chapitre précédent. Cependant, il est utile et important de remarquer que la résolution des contraintes dans Alloy est bien plus pratique et simple car celle-ci se faisant de façon symbolique. De plus, même n'étant pas orienté objet, Alloy définit des concepts et des mécanismes qui peuvent facilement simuler les concepts du paradigme objet/aspect. Par ailleurs, la formalisation que nous avons présentée nous a dévoilé les nombreux pouvoirs d'abstraction et de modélisation de Alloy.

CHAPITRE 6

VÉRIFICATION DES INTERACTIONS ENTRE ASPECTS : APPLICATION À UNE ÉTUDE DE CAS

6.1 Introduction

En plus de doter Aspect-UML d'une sémantique formelle, la traduction de Aspect-UML vers Alloy, proposée au chapitre précédent, nous permet d'atteindre nos objectifs de vérification en utilisant l'analyseur Alloy [All]. Nous nous intéressons, en particulier, à la vérification des interactions entre aspects, c'est-à-dire à identifier les erreurs liées à la composition et au tissage des aspects dans un modèle Aspect-UML décrit par une spécification Alloy.

L'analyseur Alloy est un outil qui a déjà fait ses preuves. Il a été utilisé dans différents cas de recherche et d'industrie. Les différentes expériences ont montré qu'un nombre important de défauts et de failles de conception très subtils ou même majeurs a été signalé dans la plupart des modèles analysés par Alloy. Entre autres, Alloy a permis de détecter des erreurs d'atteignabilité, de blocages, de conflits, etc. [Jac05] énumère les cas réels analysés par Alloy.

L'objectif de ce chapitre est de démontrer la pertinence de notre approche de vérification en l'illustrant sur une application réelle et concrète. L'étude de cas adoptée consiste en un système de téléphonie conçu à base de services. Comme nous l'avons déjà souligné au chapitre 1, le choix d'une telle application est approprié à plus d'un titre. D'une part, la programmation orientée aspect se prête bien à l'implémentation de ce type de systèmes. Cette approche permet en fait de garantir la séparation effective des fonctionnalités décrivant les services de la description de base du système. De plus, les concepts de *service* et d'*aspect* sont liés dans le sens où ils servent tous les deux à ajouter des fonctionnalités à un système de base. D'autre

part, en proposant une implémentation orientée aspect à ces systèmes de téléphonie, nous ramenons le problème d'interactions de services intrinsèque à ce genre d'application au problème d'interaction entre aspects auquel nous nous intéressons plus particulièrement. Nous allons, en fait, montrer comment l'analyseur Alloy peut révéler des interactions potentielles entre les différents services de l'application de téléphonie.

Ce chapitre est donc consacré à l'application de notre approche de modélisation et de vérification formelle des interactions dues aux aspects à une application de téléphonie conçue à base de services. Nous allons d'abord présenter dans la première section notre étude de cas. Par la suite, nous allons développer un modèle Aspect-UML pour cette application en utilisant notre profil Aspect-UML (décrit au chapitre 2). La section suivante abordera la traduction du modèle Aspect-UML obtenue vers une spécification Alloy équivalente. La traduction est réalisée en appliquant la fonction de transformation décrite au chapitre 5. Ensuite dans la dernière section, avant d'analyser avec l'outil Alloy la spécification obtenue, nous expliciterons notre approche de vérification.

6.2 Étude de cas

Depuis l'apparition du téléphone, les systèmes téléphoniques ont connu un nombre important de transformations et d'évolutions. Afin de répondre aux besoins croissants des consommateurs, les systèmes téléphoniques actuels offrent une multitude de services optionnels à leurs usagers, c'est ce que l'on appelle, dans la terminologie des télécommunication, les *systèmes de téléphonie à base de services.* Dans de telles applications, le noyau du système (appelé aussi système de base) permet uniquement de gérer les connexions téléphoniques entre les abonnés (initiation et rupture des communications). Un service téléphonique, quant à lui, est un composant logiciel qui réalise une fonctionnalité additionnelle (par rapport au

noyau) telle que le transfert d'appel, la messagerie vocale, le blocage d'appel, etc. Les services sont généralement développés et testés à part de façon isolée du reste du système, le plus souvent par des développeurs différents. Ils sont ensuite rajoutés de façon incrémentale au système de téléphonie, durant les différentes phases du cycle de développement. Notons finalement que les services sont parfois optionnels ; ils peuvent être activés ou désactivés dynamiquement.

Cependant, quand plusieurs services sont ajoutés à un système, il peut y avoir des *interactions* entre les différentes fonctionnalités décrivant les services. Si certaines interactions sont bénignes, d'autres peuvent être gravement dommageables et compromettantes pour le développement du système et la satisfaction des attentes de l'utilisateur. L'interaction de services est définie dans [BDC+88] comme suit : *une interaction de services est un comportement où un service inhibe ou déroute l'exécution prévue d'un autre service ou d'une autre fonctionnalité, ou crée un dilemme d'exécution entre les fonctionnalités des services.* Des exemples de telles interactions sont présentés à la section 1.4 du chapitre 1.

Les interactions entre services téléphoniques peuvent être très difficiles à détecter et à résoudre, surtout si plus d'un service doit être rajouté à la description de base. En effet, avec le nombre sans cesse croissant des services, il y a une explosion combinatoire du nombre de scénarios susceptibles de comporter des interactions. En général, ni les inspections manuelles, ni les simples tests n'offrent de solution efficace et flexible pour la détection de ces interactions. Des approches plus précises et rigoureuses s'attaquant à ce problème sont alors nécessaires. Notons que plusieurs recherches industrielles et académiques s'intéressent de plus en plus à ce phénomène appelé dans la terminologie des télécommunications *features interactions* [Nae00, RM02, RMR05].

Dans notre cas, afin de couvrir différents types d'interactions, nous considérons l'application de téléphonie à base de services dont le schéma est déjà donné à la figure 1.6, page 16. Cette application est inspirée des exemples de services présentés

dans [Zav04] et [Asp02]. Les services additionnels que nous ajoutons à la description de base sont les suivants :

- *calcul de la durée d'une communication* (*timing*) : permet de calculer et de sauvegarder le temps de connexion pour chaque appel téléphonique ;
- *facturation* (*billing*) : permet de mettre à jour la facture du client qui a initié la connexion (l'appelant). À la fin de la communication, un total par connexion est calculé, et est rajouté à la facture du client ;
- *statistiques* (*statistics*) : collecte et sauvegarde périodique des informations statistiques sur les connexions, comme par exemple la durée moyenne des connexions ;
- *transfert d'appel* (*call forwarding*) : redirection de tous les appels téléphoniques destinés à un usager donné vers un autre numéro spécifié ;
- *interruption d'appel* (*interrupting callee on busy line*) : service pour la prise en charge des lignes occupées, en interrompant l'appel en cours. Il permet à un abonné A, déjà en conversation téléphonique avec un abonné B, de prendre un nouvel appel provenant d'un autre abonné C, en interrompant la connexion de A vers B ;
- *messagerie vocale* (*voice mail on busy line*) : service pour prendre en charge les lignes occupées en redirigeant l'appel vers la messagerie vocale. Il offre la messagerie vocale à un abonné C qui tente de joindre un abonné A déjà en conversation téléphonique avec un abonné B ;
- *blocage d'appel* (*blocking number*) : permet de restreindre les appels entrants. Les appels vers une destination donnée provenant de certains numéros (listés dans un répertoire) sont refusés et bloqués.

Dans la suite de ce chapitre, nous allons d'abord développer un modèle Aspect-UML pour cette application de téléphonie. Ce modèle sera ensuite traduit vers une spécification Alloy. Finalement la spécification obtenue sera analysée avec l'outil Alloy afin de détecter les interactions qui peuvent être engendrées suite à l'intégration

simultanée des différents services mentionnés.

6.3 Modélisation avec le profil Aspect-UML

Comme nous l'avons déjà vu au chapitre 3, la programmation orientée aspect se prête bien à l'implémentation des systèmes de téléphonie conçus à base de services. Dans ce schéma, la description de base du réseau téléphonique consiste en des appareils téléphoniques (ou *devices*), des clients (usagers du système) et un contrôleur de réseau qui permet de gérer les connexions entre les clients. Les usagers du système peuvent initier et rompre des communications téléphoniques *via* leurs téléphones. Quant aux services téléphoniques additionnels, ils sont implémentés comme des aspects indépendants à intégrer au système au besoin à chaque appel téléphonique.

Suivant notre approche de modélisation avec le profil Aspect-UML, les aspects sont considérés dès la phase de la spécification des besoins et sont pris en charge tout au long du cycle de développement. Dans cette section, nous allons donner le modèle Aspect-UML décrivant cette application de téléphonie dans chacune des vues de notre profil Aspect-UML, soit la vue de cas d'utilisation, la vue structurelle et la vue comportementale.

6.3.1 Vue de cas d'utilisation

Dans l'application de téléphonie, les fonctionnalités de base concernent principalement la gestion des appels téléphoniques, tels que : *initier un appel* et *rompre un appel*. Dans la vue de cas d'utilisation, ces fonctionnalités sont représentées respectivement par les cas d'utilisation de base : *complete* et *drop* (voir la figure 3.1 du chapitre 3). Les aspects sont, quant à eux, modélisés comme des cas d'utilisation d'extension spéciaux. Les points de coupure sont assimilés à des points d'extension particuliers, *c-à-d* un ensemble de locations où les cas d'extension décrivant les aspects peuvent être tissés.

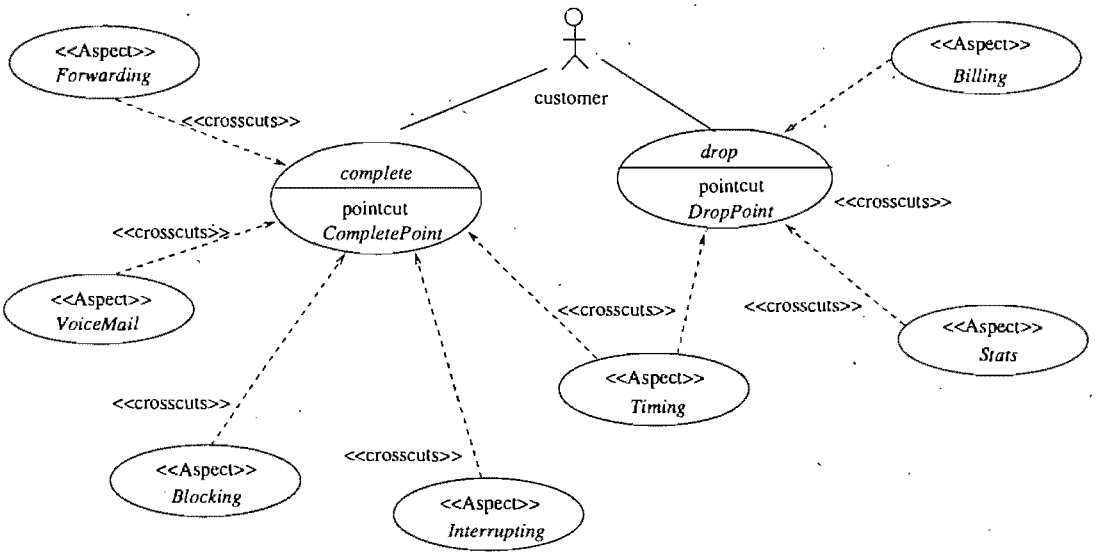


FIG. 6.1 – Vue de cas d'utilisation de l'application de téléphonie.

Dans la section 3.3.1 du chapitre 3, nous avons expliqué et présenté en détail comment les besoins reliés aux fonctionnalités de calcul de la durée de connexion et de facturation sont pris en charge dans la vue de cas d'utilisation. Les fonctionnalités des autres services sont traités de façon similaire comme expliqué sommairement ci-après.

La fonctionnalité de calcul des *statistiques* est réalisée à la fin de chaque connexion afin de collecter la durée de la communication et de mettre à jour la valeur dénotant la durée moyenne des communications. Cette fonctionnalité entrecoupe donc le cas de base *rompre une communication.* Par contre, les fonctionnalités de *transfert d'appel*, *interruption d'appel*, *messagerie vocale* et *blocage d'appel* interviennent toutes au début de chaque connexion. Elles doivent être réalisées avant d'initier une communication. Par conséquent, elles sont transverses et elles entrecoupent la fonctionnalité de base *initier une communication.* Dans la vue de cas d'utilisation de notre système de téléphonie, donné par la figure 6.1, toutes ces nouvelles fonctionnalités transverses sont représentées respectivement par les cas d'utilisation d'extension suivants : *Stats*, *Forwarding*, *Interrupting*, *VoiceMail* et *Blocking*

spécifiant le comportement additionnel à exécuter.

Afin de distinguer les cas d'utilisation de base et les cas d'utilisation transverses, notre profil Aspect-UML propose pour chaque type de cas d'utilisation une documentation qui contient ses informations pertinentes. Dans la section 3.3.1 du chapitre 3, nous avons déjà présenté la documentation du cas d'utilisation de base *complete*, du cas d'utilisation transverse *Timing* et du point de coupure *Complete-Point*.

Dans ce qui suit, nous donnons la documentation du cas d'utilisation de base *drop* (table 6.1), ainsi que celles des cas d'utilisation transverses *Billing*, *Stats*, *Forwarding*, *Interrupting*, *VoiceMail* et *Blocking* (données respectivement par les tables 6.2 à 6.7). Notons, que dans la documentation présentée nous ne présentons que les scénarios principaux décrivant les cas d'utilisation. La documentation décrivant le point de coupure *DropPoint* est donnée à la table 6.8.

Use case :	drop
Level :	But utilisateur (*user goal*)
Actors :	Customer
Preconditions :	S'assurer que la ligne téléphonique est occupée. Une communication est déjà établie sur la ligne de l'appelant. L'appelant et l'appelé sont occupés.
Postconditions :	Assurer que la connexion est bien terminée et rompue, et que l'appelé et l'appelant sont libérés.
Main scenario :	1. L'abonné (appelant ou appelé) accède à son appareil téléphonique pour rompre la communication. 2. Le système se déconnecte de la destination et rompt la communication.
Pointcuts :	DropPoint

TAB. 6.1 – Documentation du scénario décrivant le cas d'utilisation de base *drop*.

Use case :	« aspect» Billing
Crosscut pointcuts :	DropPoint
Level :	Besoin (*requirement*)
Preconditions :	La durée de la communication est calculée.
Postconditions :	La facture du client est mise à jour.
Main scenario :	1. Le système accède au Timer de la connexion qui vient d'être rompue, afin de lire le temps de communication. 2. Les frais de facturation pour cette communication sont calculés. 3. Le système accède à la facture du client qui a initié la communication qui vient d'être rompue. 4. Le système met à jour les frais totaux et les sauvegarde dans la facture du client.
Pointcuts :	none

TAB. 6.2 – Documentation du scénario du cas d'utilisation transverse *Billing*.

Use case :	« aspect» Stats
Crosscut pointcuts :	DropPoint
Level :	Besoin (*requirement*)
Preconditions :	La durée de la communication est calculée.
Postconditions :	La durée de la communication est réinitialisée.
Main scenario :	1. Le système accède au Timer de la connexion qui vient d'être rompue, afin de lire le temps de communication. 2. Ce temps est sauvegardé dans un fichier journalier pour les statistiques. 3. Le système réinitialise le temps de connexion.
Pointcuts :	none

TAB. 6.3 – Documentation du scénario du cas d'utilisation transverse *Stats*.

Use case :	« aspect» Forwarding
Crosscut pointcuts :	CompeltePoint
Level :	Besoin (*requirement*)
Preconditions :	Le numéro de destination de la connexion à établir est dans la liste de transfert des appels.
Postconditions :	La connexion est redirigée vers une autre destination spécifiée.
Main scenario :	1. Le système accède au répertoire des numéros de transfert afin de lire le numéro de la nouvelle destination. 2. Le numéro vers lequel sera transféré la connexion est lu. 3. Le système cherche la destination ayant ce numéro. 4. Le système change la destination initiale de la connexion vers la nouvelle destination de transfert.
Pointcuts :	none

TAB. 6.4 – Documentation du scénario du cas d'utilisation transverse *Forwarding*.

Use case :	« aspect» Interrupting
Crosscut pointcuts :	CompletePoint
Level :	Besoin (*requirement*)
Preconditions :	La destination de la connexion à établir est occupée.
Postconditions :	La destination est libre et la connexion en cours dans laquelle est engagée la destination est interrompue.
Main scenario :	1. Le système accède à la connexion en cours dans laquelle la destination est déjà engagée. 2. Le système rompt cette connexion et met son état à interrompu. 3. Le système change l'état de la destination de occupé à libre.
Pointcuts :	none

TAB. 6.5 – Documentation du scénario du cas d'utilisation transverse *Interrupting*.

Use case :	« aspect» VoiceMail
Crosscut pointcuts :	CompletePoint
Level :	Besoin (*requirement*)
Preconditions :	La destination de la connexion à établir est occupée.
Postconditions :	La destination est toujours occupée et la connexion à établir est à l'état voiceMail.
Main scenario :	1. Le système offre la messagerie vocale à la connexion. 2. Le système met l'état de la connexion à *voiceMail.*
Pointcuts :	none

TAB. 6.6 – Documentation du scénario du cas d'utilisation transverse *VoiceMail.*

Use case :	« aspect» Blocking
Crosscut pointcuts :	CompeltePoint
Level :	Besoin (*requirement*)
Preconditions :	Le numéro de l'origine de la connexion à établir est dans la liste des numéros bloqués par la destination.
Postconditions :	La connexion est refusée et elle est dans l'état bloqué.
Main scenario :	1. Le système accède au répertoire des appels bloqués afin de vérifier si le numéro de l'origine de l'appel fait partie des numéros bloqués par la destination. 2. Le numéro de l'origine de la connexion est retrouvé. 3. Le système bloque la connexion en changeant son état à *blocked.*
Pointcuts :	none

TAB. 6.7 – Documentation du scénario du cas d'utilisation transverse *Blocking.*

Pointcut :	DropPoint
Textual description :	Ce point définit la fin d'une communication.
References :	<drop>[step2]

TAB. 6.8 – Documentation décrivant le point de coupure *DropPoint*.

6.3.2 Vue structurelle

Dans la section 3.3.2 du chapitre 3, nous avons présenté la vue structurelle de notre exemple d'application de téléphonie auquel nous avons seulement intégré les deux fonctionnalités de calcul de la durée de communication et de facturation. La vue structurelle du système de base y est donnée à la figure 3.2, page 68. Cette vue met en évidence les éléments de domaine du système de base, à savoir, la classe *Connection*, la classe *Customer* et la classe *Device*. Ici, nous donnons la vue structurelle de l'application entière, intégrant les sept services déjà mentionnés.

Suivant notre approche de modélisation, les aspects décrits par les cas d'utilisation d'extension *Timing*, *Billing*, *Stats*, *Forwarding*, *Interrupting*, *VoiceMail* et *Blocking* dans la figure 6.1 sont décrits dans la vue structurelle par des classes de même nom portant le stéréotype « `Aspect` ». Quant aux points de coupure *CompletePoint* et *DropPoint*, ils sont modélisés par des interfaces de même nom portant le stéréotype « `Pointcut` ».

La figure 6.2 donne le diagramme de classes de l'application de téléphonie complète. Le diagramme montre l'intégration des aspects décrivant les services additionnels, dans le système de base, à travers les deux interfaces (points de coupure) *CompletePoint* et *DropPoint*.

6.3.3 Vue comportementale

La vue comportementale d'Aspect-UML permet d'enrichir la vue structurelle par des informations décrivant les propriétés caractérisant le contexte et les contraintes

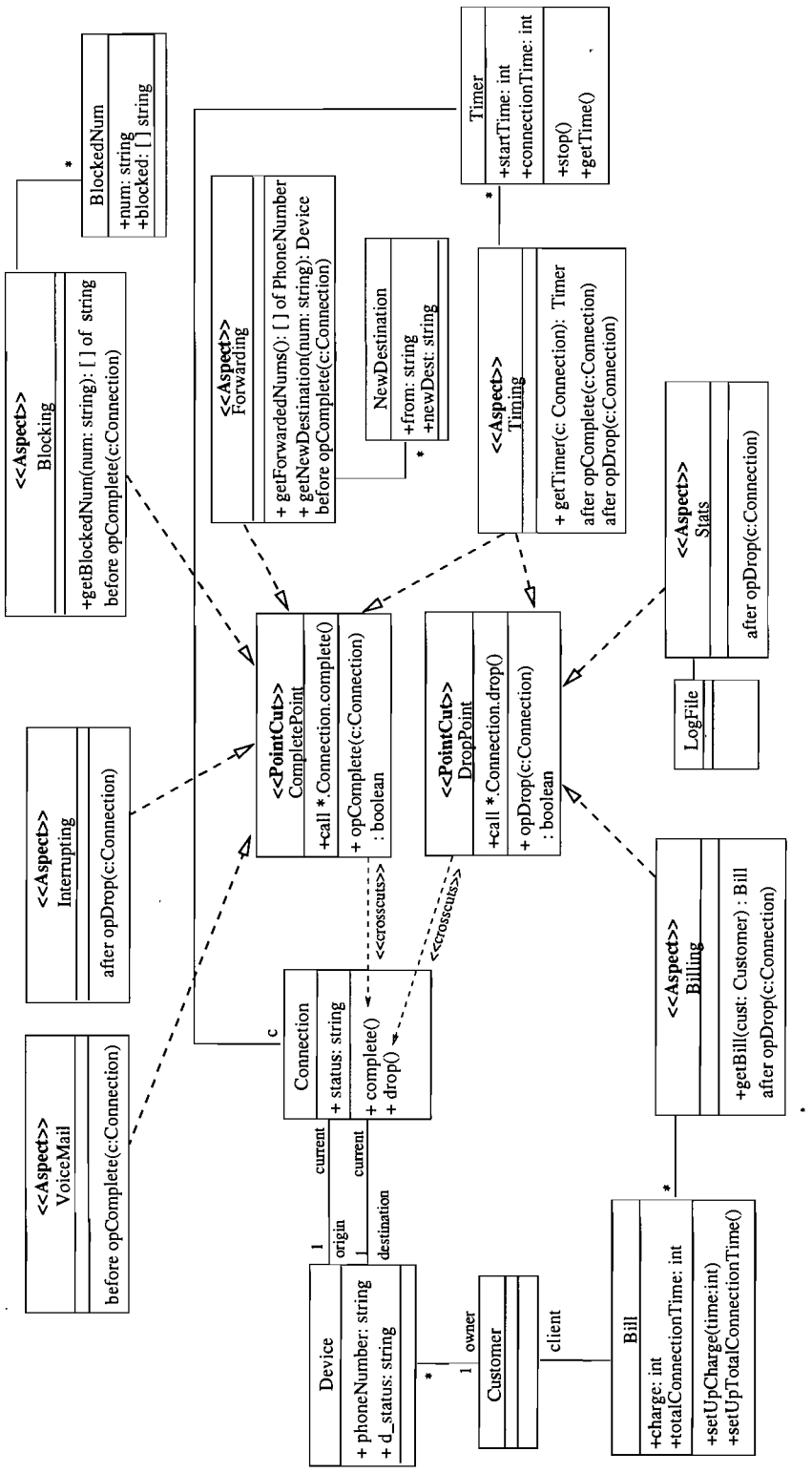


FIG. 6.2 – Vue structurelle de l'application de téléphonie.

de composition des aspects. Ces informations sont décrites ci-après :

1. Contraintes de précédence. Dans l'application de téléphonie, puisqu'il importe de calculer la durée d'un appel pour pouvoir le facturer, on définit la contrainte *Timing* < *Billing*. Cette contrainte indique que l'aspect `Timing` est prioritaire par rapport à l'aspect *Billing*. Par ailleurs, nous supposons qu'aucune relation de précédence n'est définie entre les autres aspects de l'application.

2. Contraintes de passage de contexte. Les contraintes définissant la liaison des informations contextuelles du point de jointure aux paramètres formels des points de coupure *CompletePoint* et *DropPoint* sont respectivement exprimées de la façon suivante :

CompletePoint : :Binding
ToJoinPoint : Connection : :complete()
Binds : c ← target

DropPoint : :Binding
ToJoinPoint : Connection : :drop()
Binds : c ← target

3. Spécification des points de jointure. Les spécifications des points de jointure *Connection : :complete()* et *Connection : :drop()* sont définies respectivement comme suit :

context *Connection : :complete()*
pre : $self.status = disconnected$
pre : $self.origin.d\_status = idle$
pre : $self.origin.current = Null$
post : $self.status = connected$
post : $self.origin.d\_status = busy$
post : $self.destination.d\_status = busy$
post : $self.origin.current = self$
post : $self.destination.current = self$

context *Connection : :drop()*
pre : $self.status = connected$
pre : $self.origin.d\_status = busy$
pre : $self.destination.d\_status = busy$
pre : $self.origin.current = self$
pre : $self.destination.current = self$
post : $self.status = disconnected$

post : $self.origin.d\_status = idle$
post : $self.destination.d\_status = idle$
post : $self.origin.current = Null$
post : $self.destination.current = Null$

4. Spécification des advices. Les advices des différents aspects de l'application de téléphonie sont spécifiés ainsi :

context Timing : :opComplete(c : Connection)
pre : $getTimer(c).startTime < 0$
post : $getTimer(c).startTime \geq 0$

context Timing : :opDrop(c : Connection)
pre : $getTimer(c).startTime \geq 0$
pre : $getimer(c).connectionTime < 0$
post : $getTimer(c).connectionTime \geq 0$

context Billing : :opDrop(c : Connection)
pre : $Timing.getTimer(c).connectionTime \geq 0$
post : $getBill(c.origin.owner).charge > getBill(c.origin.owner).charge_{@pre}$

context Stats : :opDrop(c : Connection)
pre : $Timing.getTimer(c).connectionTime \geq 0$
post : $Timing.getTimer(c).connectionTime < 0$

context Forwarding : :opComplete(c : Connection)
pre : $c.destination.num\ in\ getForwardedNums()$
post : $c.destination = getNewDestination(c.origin.phoneNumber)$

context Interrupting : :opComplete(c : Connection)
pre : $c.destination.d\_status = busy$
pre : $c.destination.current! = Null$
post : $c.destination.d\_status = idle$
post : $c.destination.current.satus = interrupted$
post : $c.status = c.status@pre$

context VoiceMail : :opComplete(c : Connection)
pre : $c.destination.d\_status = busy$
post : $c.status = voicemail$

context Blocking : :opComplete(c : Connection)
pre : $c.origin.phoneNumber\ in\ getBlockedNum(c.destination.phoneNumber)$
post : $c.status = blocked$
post : $c.destination.current.status = c.destination.current.status@pre$

6.4 Traduction vers Alloy

La traduction du modèle Aspect-UML décrivant l'application de téléphonie vers une spécification Alloy se fait suivant l'approche décrite au chapitre 5. Rappelons, qu'au chapitre 5, nous avons présenté les étapes de traduction appliquées à notre étude de cas restreinte, où seuls les aspects `Timing` et `Billing` sont intégrés à l'application de base. La spécification Alloy complète de notre application est présentée à l'annexe II. Dans cette section, nous reviendrons brièvement sur quelques éléments pertinents de la traduction.

6.4.1 Spécification des éléments statiques du modèle Aspect-UML

Rappelons que les types de données, les classes et les aspects du modèle Aspect-UML sont traduits vers des signatures Alloy. Dans le cas de notre application, les types énumérés *status* et *d_status* sont spécifiés respectivement par les signatures abstraites *Status* et *d_Status*. La première est étendue par les sous-signatures *connected*, *disconnected*, *interrupted*, *voicemail* et *blocked* décrivant les littéraux de ce type. De même, *d_Status* est étendue par les sous-signatures *idle* et *busy*. Quant au type entier, il est représenté par le type prédéfini `Int` de Alloy. Nous avons aussi introduit le type *Null* pour la valeur Null utilisée dans les pré et post-conditions.

Les classes *Connection*, *Customer*, *Device*, *Timer*, *Bill*, *NewDestination*, *BlockedNum* sont représentées par des signatures de même nom. De même, les aspects *Timing*, *Billing*, *Stats*, *Forwarding*, *Interrupting*, *VoiceMail* et *Blocking* sont aussi représentés par des signatures de même nom. Les attributs des classes et des aspects sont décrits par des relations dans les signatures correspondantes. Comme nous l'avons bien expliqué à la section 5.5, le type de ces relations est augmenté de `Time`, si les attributs qu'elles décrivent apparaissent dans les pré et post-conditions du modèle Aspect-UML. La dimension `Time` permet aux champs de la signature de garder trace des valeurs des attributs aux différents moments d'une exécution.

6.4.2 Spécification des éléments comportementaux du modèle Aspect-UML

Rappelons que pour être traduits vers Alloy, les points de jointure et les advices sont réifiés en des signatures étendant respectivement les signatures *JP* et *Advice*, elles même étendant la signature *Operation*. Ainsi, les points de jointure et les advices de notre application sont décrits dans Alloy par des signatures. Ces signatures sont contraintes par des faits traduisant les pré et postconditions des opérations respectives. Par ailleurs, les méthodes *getTimer(c :Connection)*, *getBill(cust :Customer)*, *getForwardedNums()*, *getNewDestination(num :string)* et *getBlockedNum(num :string)* apparaissant dans les pré et post-conditions du modèle Aspect-UML sont traduites vers des fonctions Alloy de même nom.

Le problème des frames de l'application de téléphonie est résolu par le fait *unchanged*, donné ci-après. Rappelons que ce fait spécifie de façon succincte quelle opération modifie quel champ de la spécification. Autrement dit, ce fait indique qu'un champ f donné ne doit être modifié que par l'exécution correcte d'une opération spécifique e.

```
fact unchanged { all t : Time - last | let t'= t.next |
        some e :JP+Advice { e.begin = t && e.end= t'
        (status.t=status.t' || e in (Complete + Drop + OpCompleteInterrupting +
                OpCompleteBlocking + OpCompleteVoiceMail) && e.error=False)
        &&
        (d_status.t=d_status.t' || e in (Complete + Drop + OpCompleteInterrupting)
                && e.error=False)
        &&
        (destination.t=destination.t' || (e in OpCompleteForwarding && e.error=False)
        &&
        (current.t= current.t' || (e in Complete + Drop + OpCompleteInterrupting &&
                e.error=False)
        &&
        (startTime.t=startTime.t' || (e in OpCompleteTiming && e.error=False )
        &&
        (connectionTime.t= connectionTime.t' || (e in OpDropTiming + OpDropStats &&
                e.error=False)
        &&
        (charge.t= charge.t' || (e in OpDropBilling && e.error=False) }
```

6.4.3 Spécification de la composition des aspects

La composition des advices au niveau des points de jointure est décrite par des signatures étendant les signatures *SeqComposition* ou *NDComposition*, selon que la composition est séquentielle ou alors non déterministe.

Dans notre application de téléphonie, la contrainte de précédence *Timing* < *Billing* est spécifiée dans le modèle Aspect-UML. Cette contrainte indique que l'aspect *Timing* est prioritaire par rapport à l'aspect *Billing*. Ainsi, puisque ces deux aspects implémentent le même point de coupure *DropPoint*, alors leurs advices *OpDropTiming* et *OpDropBilling*, s'exécutant tous les deux après le point de jointure *Drop*, doivent être composés de façon séquentielle (voir *Composition5* ci-après). Quant à l'advice *OpDropStats* de l'aspect *Stats* s'exécutant aussi après *DropPoint*, il sera composé de façon non déterministe avec la composition séquentielle obtenue puisqu'aucune contrainte de précédence n'est définie pour l'aspect *Stats* (voir *Composition4* donnée ci-après). Rappelons que *NDComposition* impose des contraintes suffisamment flexibles pour prendre en compte l'entrelacement de l'advice *OpDropStats* au milieu de la composition séquentielle formée par les advices *OpDropTiming* et textitOpDropBilling.

```
sig Composition4 extends NDComposition {}
fact composition4 { all s :Composition4 |
        one s.comp1 && one s.comp2 &&
        s.comp1= OpDropStats &&
        s.comp2= Composition5 }

sig Composition5 extends seqComposition {}
fact composition5 { all s :Composition5 |
        (one s.comp1 && one s.comp2) &&
        s.comp1= OpDropTiming &&
        s.comp2= Composition51 }

sig Composition51 extends SeqComposition {}
fact composition51 { all s :Composition51 |
        (one s.comp1 && no s.comp2) &&
        s.comp1= OpDropBilling}
```

D'autre part, aucune relation de précédence n'est définie pour les aspects *Forwarding*, *Interrupting*, *VoiceMail* et *Blocking* implémentant le point de coupure *CompletePoint*. Les advices de ces aspects, s'exécutant tous avant le point de jointure *Connection.complete*, seront donc composés de façon non déterministe (voir *Composition1* donnée ci-après).

```
sig Composition1 extends NDComposition {}
fact composition1 { all s :Composition1 |
        (one s.comp1 && one s.comp2) &&
        s.comp1= OpCompleteInterrupting &&
        s.comp2= Composition11 }

sig Composition11 extends NDComposition {}
fact composition11 { all s :Composition11 |
        (one s.comp1 && one s.comp2) &&
        s.comp1= OpCompleteForwarding &&
        s.comp2= Composition12 }

sig Composition12 extends NDComposition {}
fact composition12 { all s :Composition12 |
        one s.comp1 && one s.comp2 &&
        s.comp1= OpCompleteVoiceMail &&
        s.comp2= Composition13 }

sig Composition13 extends NDComposition {}
fact composition13{ all s :Composition13 |
        (one s.comp1 && no s.comp2) &&
        s.comp1= OpCompleteBlocking }
```

6.4.4 Spécification du tissage des aspects aux points de jointure

Au point de jointure *complete*, les advices *OpCompleteForwarding*, *OpCompleteInterrupting*, *OpCompleteVoiceMail* et *OpCompleteBlocking* (décrits par la composition *Composition1*) doivent être tissés avant le point de jointure et l'advice *OpCompleteTiming* (décrit par la composition *Composition2*) doit être tissé après le point de jointure (voir le code complet à l'annexe II). Par contre, au point de jointure *drop* seuls les advices *OpDropTiming*, *OpDropBilling* et *OpDropStats* (décrits

par la composition *Composition4*) doivent être tissés après le point de jointure. Notons, en effet, qu'aucun aspect ne définit un advice à exécuter avant le point de jointure *drop*.

Ainsi, le tissage des advices aux points de jointure *complete* et *drop* est spécifié respectivement par les deux signatures *WeavingAtComplete* et *WeavingAtDrop*, données ci-après.

```
sig WeavingAtComplete extends Weaving {}
fact weavingAtComplete { all w : WeavingAtComplete |
        contextpassingAtComplete[w.jp] &&
        (one w. jp) && (one w.beforeAdvice) && (one w.afterAdvice) &&
        w.jp = Complete &&
        w.beforeAdvice = Composition1 &&
        w.afterAdvice = Composition2 }


sig WeavingAtDrop extends Weaving {}
fact weavingAtDrop { all w : WeavingAtDrop |
        contextpassingAtDrop[w.jp] &&
        (one w. jp) && (no w.beforeAdvice) && (one w.afterAdvice) &&
        w.jp = Drop &&
        w.afterAdvice = Composition4 }
```

6.5 Vérification avec Alloy

Une fois le modèle Aspect-UML de l'application de téléphonie traduit vers Alloy, l'analyse des interactions dues aux aspects sera accomplie au moyen des assertions. Rappelons que les assertions sont des contraintes présumées valides *c-à-d* vraies pour toutes les instances de modèles. Nous devons ainsi formuler les propriétés désirées du système final (tissé) comme des assertions et vérifier ensuite chacune d'elles par l'analyseur Alloy. Pour chaque cas, Alloy va tenter de trouver un contre-exemple invalidant l'assertion *c-à-d* prouver qu'au moins une instance du modèle ne satisfait pas l'ensemble des contraintes spécifiées par l'assertion.

Avant d'aborder la vérification de l'application de téléphonie, présentons d'abord dans ce qui suit notre approche pour la vérification avec Alloy des interactions dues aux aspects.

6.5.1 Notre approche de vérification

En supposant que le système de base et les aspects sont tous individuellement corrects (*c-à-d* que leur spécification par pré et postconditions est juste), le processus de vérification formelle que nous proposons focalise sur la détection d'erreurs résultant des interactions dues aux aspects. En effet, en retenant seulement les parties critiques d'un système (points de jointure, advices et spécifications du tissage), nous pouvons nous focaliser sur la vérification de l'intégration des aspects dans le système de base, ainsi que sur les interactions qui en découlent. Plus précisément, la vérification, que nous proposons, vise à révéler les problèmes d'interférence dus aux aspects, telles que la violation des propriétés locales ou la violation des propriétés globales tant au niveau du système de base que des aspects eux-mêmes.

Comme nous allons le montrer dans ce qui suit, la vérification des propriétés locales des modèles Aspect-UML n'est pas traitée de la même façon que la vérification des propriétés globales.

1. Verification des propriétés locales

Les opérations d'un système (tel que les points de jointure et les advices) sont *localement* spécifiées au moyen des pré et postconditions (appelées propriétés locales). Le tissage des aspects peut causer la violation de ces propriétés locales. En fait, un aspect tissé à un point de jointure peut interférer avec le système de base ou les autres aspects qui s'y tissent aussi et ainsi violer leurs propriétés. Par ailleurs, les propriétés d'un advice peuvent aussi être violées par le système de base lui-même (point de jointure).

Pour vérifier ces interactions avec Alloy, nous devons formuler pour chaque point de jointure *JP* une assertion appelée *localVerifAtJP* (voir table 6.9). Cette

assertion exprime que pour chaque opération de tissage w au point de jointure JP si les conditions initiales au point de jointure sont vraies avant le tissage (ce qui est exprimé par le prédicat *initialCondAtJP[w]*) alors les conditions finales doivent être vraies après le tissage (ce qui est exprimé par le prédicat *finalCondAtJP[w]*) et le point de jointure (la méthode) tout comme les advices qui s'y tissent doivent avoir été exécutés correctement (ce qui est exprimé par le prédicat *noErrorAtJP[ ]*). Si un contre-exemple est trouvé par l'analyseur Alloy en vérifiant cette assertion, ceci implique que le modèle Aspect-UML spécifié présente un problème d'interférences dues aux aspects au point de jointure *JP*. Le patron utilisé pour décrire en Alloy une telle assertion est donné à la table 6.9.

```
assert localVerifAtJP { all w : WeavingAtJP |
      initialCondAtJP[w] => finalCondAtJP[w] && noErrorAtJP[ ] }

pred initialCondAtJP (w :Weaving) { //spécification des hypothèses initiales du
      // système de base et des aspects, avant le tissage au point de jointure JP
      // (c-à-d à l'état w.begin ). }

pred finalCondATJP(w :Weaving) { //spécification des conditions finales
      // au point JP, après le tissage (c-à-d à l'état w.end). }

pred noErrorAtJP( ) { //indication qu'aucune erreur n'a eu lieu durant
      // le tissage des advices au point JP, c-à-d les opérations
      // ont été exécutées correctement et leur champ error est à faux. }
```

TAB. 6.9 – Patron d'une assertion pour la vérification d'une propriété locale.

2. Vérification des propriétés globales

Cette catégorie inclut la vérification des invariants du système suite à l'introduction de nouveaux aspects. On doit aussi envisager la possible violation des invariants des aspects par le système de base. Un invariant est une propriété de sûreté qui garantit qu'une formule (exprimée ici par un prédicat) est vérifiée par tous les états d'une exécution correcte (quelle qu'elle soit) du système.

Comme montré par le patron donné à la table 6.10, la vérification de chaque

invariant *Invariant* du système (ou d'aspect) se fait au moyen d'une assertion que nous appelons *verifInvariant*. Cette assertion indique que pour chaque opération de tissage w, l'invariant doit être vrai dans tous les états t du système (c'est-à-dire entre l'état *w.begin* et l'état *w.end*, ce qui est exprimé ici à l'aide du prédicat *lte* (less or equal than)).

```
assert verifInvariant { all w : Weaving|
        correctExecution[w] => all t : Time |
                lte[w.begin,t] && lte[t, w.end] && Invariant[t] }

pred Invariant(t :Time) { //spécifie l'invariant à vérifier }

pred correctExecution(w : Weaving ) { //spécifie l'exécution correcte du modèle tissé}
```

TAB. 6.10 – Patron d'une assertion pour la vérification d'une propriété globale.

Les sous-sections (6.5.2 et 6.5.3) qui suivent sont consacrées respectivement à la vérification des propriétés locales et à la vérification des propriétés globales de notre application de téléphonie. Les résultats de la vérification sont ensuite résumés à la sous-section 6.5.4 où toutes les interactions détectées sont répertoriées dans le tableau 6.23 récapitulatif, page 215.

6.5.2 Vérification des propriétés locales de l'application de téléphonie

Dans l'application de téléphonie, nous souhaitons vérifier que les aspects `Forwarding`, `Interrupting`, `VoiceMail`, `Blocking` et `Timing` interagissent correctement au point de jointure `Complete` (c'est-à-dire qu'ils ne violent pas la spécification du système de base et n'interférent pas avec la spécification des autres aspects). De même nous souhaitons vérifier que les aspects `Timing`, `Billing` et `Stats` interagissent correctement au point de jointure `Drop`. Pour cela, nous introduisons les assertions `localVerifAtComplete` et `localVerifAtDrop` que nous tenterons de vérifier avec Alloy. Ces deux assertions sont détaillées dans ce qui suit.

6.5.2.1 Vérification locale au point de jointure `Complete`

Pour vérifier les propriétés locales au point de jointure `Complete`, nous formulons l'assertion `localVerifAtComplete` décrite à la table 6.11, conformément au patron 6.9.

1. Description de l'assertion `localVerifAtComplete`

Cette assertion suppose que les hypothèses initiales sont correctes, c'est-à-dire que le système de base et les advices des aspects (`Forwarding`, `Interrupting`, `VoiceMail`, `Blocking` et `Timing`) sont individuellement corrects. Ces hypothèses sont celles imposées par les préconditions définies dans le modèle Aspect-UML de l'application de téléphonie. Sous ces conditions, l'analyseur Alloy doit vérifier que les conditions finales du point de jointure `Complete` (qui sont les postconditions du point de jointure) ne sont pas violées et que les opérations (le point de jointure et les advices) ont été exécutées sans erreurs. Autrement dit, on doit s'assurer que le modèle demeure cohérent malgré l'ajout des aspects.

Rappelons que la commande `check` d'Alloy utilisée pour la vérification des assertions nécessite la définition d'une portée limitant le nombre d'éléments dans chaque signature. La portée des signatures est parfois facile à déterminer. Par exemple, pour l'assertion `localVerifAtComplete`, il est clair qu'au moins 12 éléments `Operation` (définissant le point de jointure, les cinq advices, les cinq compositions et le tissage) sont nécessaires ; sept atomes de la signature `Time` sont requis pour coordonner l'exécution du point de jointure `Complete` tissé avec les advices `OpCompleteForwarding`, `OpCompleteInterrupting`, `OpCompleteVoiceMail`, `OpCompleteTiming` et `OpCompleteBlocking`. Bien sûr, un seul élément `Weaving` (qui est `WeavingAtComplete`) et un seul élément `Complete` sont nécessaires pour spécifier respectivement l'opération de tissage et le point de jointure. Réciproquement, seul un élément de chaque signature advice est requis.

```
assert localVerifAtComplete {all w : WeavingAtComplete |
        initialCondAtComplete[w]=> finalCondAtComplete[w] && noErrorAtComplete[ ] }

pred initialCondAtComplete(w :Weaving) {
        Complete.self.status.(w.begin)= disconnected &&
        Complete.self.origin.d_status.(w.begin)= idle &&
        Complete.self.origin.current.(w.begin) = Null &&
        OpCompleteForwarding.c.destination.(w.begin).num in
                getForwardedNum[Forwarding] &&
        OpCompleteInterrupting.c.destination.(w.begin).d_status.(w.begin)= busy &&
        OpCompleteInterrupting.c.destination.(w.begin).current.(w.begin) != Null &&
        OpCompleteVoiceMail.c.destination.(w.begin).d_status.(w.begin)= busy &&
        OpCompleteBlocking.c.destination.(w.begin).num in
                getBlockedNum[Blocking,OpCompleteBlocking.c.origin.num] &&
        neg[getTimer[Timing, OpCompleteTiming.c].startTime.(w.begin)] }

pred finalCondAtComplete(w :Weaving) {
        (Complete.self.status.(w.end)= connected) &&
        (Complete.self.origin.d_status.(w.end)=busy) &&
        (Complete.self.destination.(w.end).d_status.(w.end)=busy) &&
        (Complete.self.origin.current.(w.end)= Complete.self) &&
        (Complete.self.destination.(w.end).current.(w.end)= Complete.self) }

pred noErrorAtComplete() {
        Complete.error = False &&
        OpCompleteForwarding.error= False &&
        OpCompleteInterrupting.error=False &&
        OpCompleteVoiceMail.error = False &&
        OpCompleteBlocking.error= False &&
        OpCompleteTiming.error=False }

check localVerifAtComplete for 3 but 12 Operation, 1 Weaving, 1 WeavingAtComplete,
        1 JP, 1 OpCompleteInterrupting, 1 OpCompleteForwarding, 1 OpCompleteTiming,
        1 OpCompleteVoiceMail, 1 OpCompleteBlocking, 7 Time
```

TAB. 6.11 – Assertion `localVerifAtComplete`.

2. Analyse de l'assertion `localVerifAtComplete`

En analysant `localVerifAtComplete`, l'analyseur Alloy trouve un scénario qui viole l'assertion. Alloy exhibe le contre-exemple décrit par les deux figures 6.3 et 6.4.

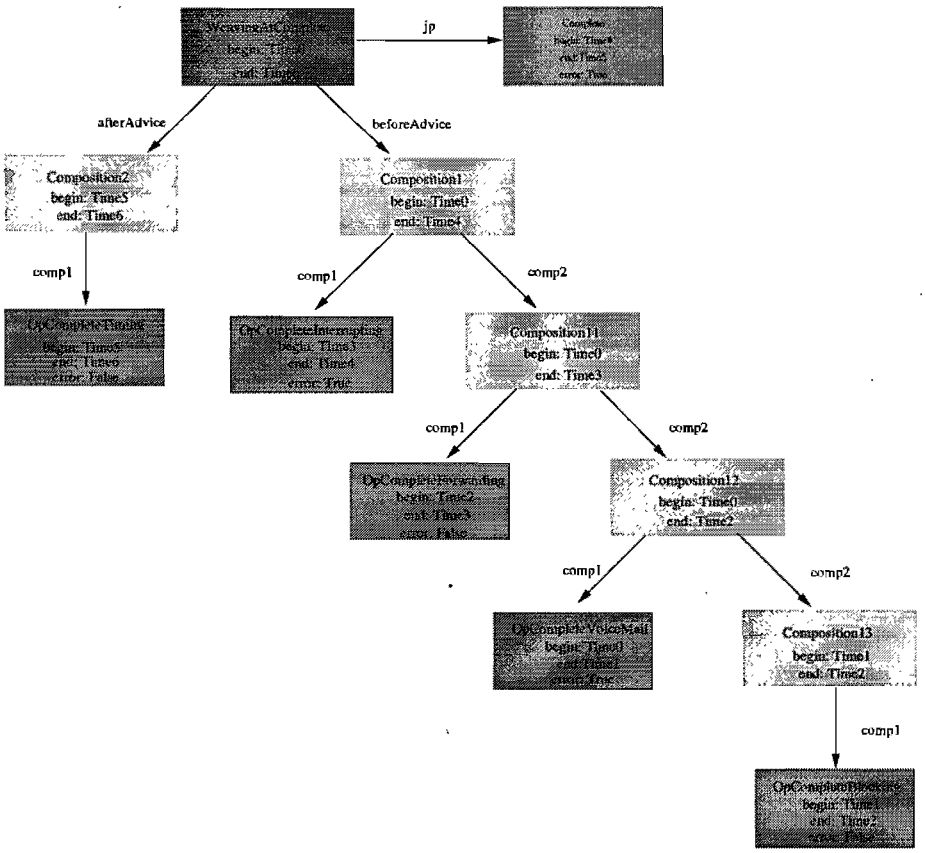


FIG. 6.3 – Contre-exemple généré pour l'assertion `localVerifAtComplete` montrant les opérations exécutées ainsi que leur ordonnancement.

La première figure (6.3) montre les différentes opérations exécutées ainsi que leur coordination (composition et tissage) à travers le temps *Time*. Dans la deuxième figure (6.4), les éléments pertinents pour la vérification y sont schématisés. Alloy permet en effet de personnaliser l'interface de sortie en sélectionnant les signatures et les relations à afficher en sortie. Ceci permet une meilleur visualisation et analyse des contre-exemples. Il est clairement montré dans les figures qu'une erreur a eu lieu durant l'exécution des advices `OpCompleteInterrupting`, `OpCompleteVoiceMail`

et `OpCompleteBlocking` ($error = True$), alors que les autres opérations `Complete`, `OpCompleteForwarding` et `OpCompleteTiming` ($error = False$) ont été exécutées correctement.

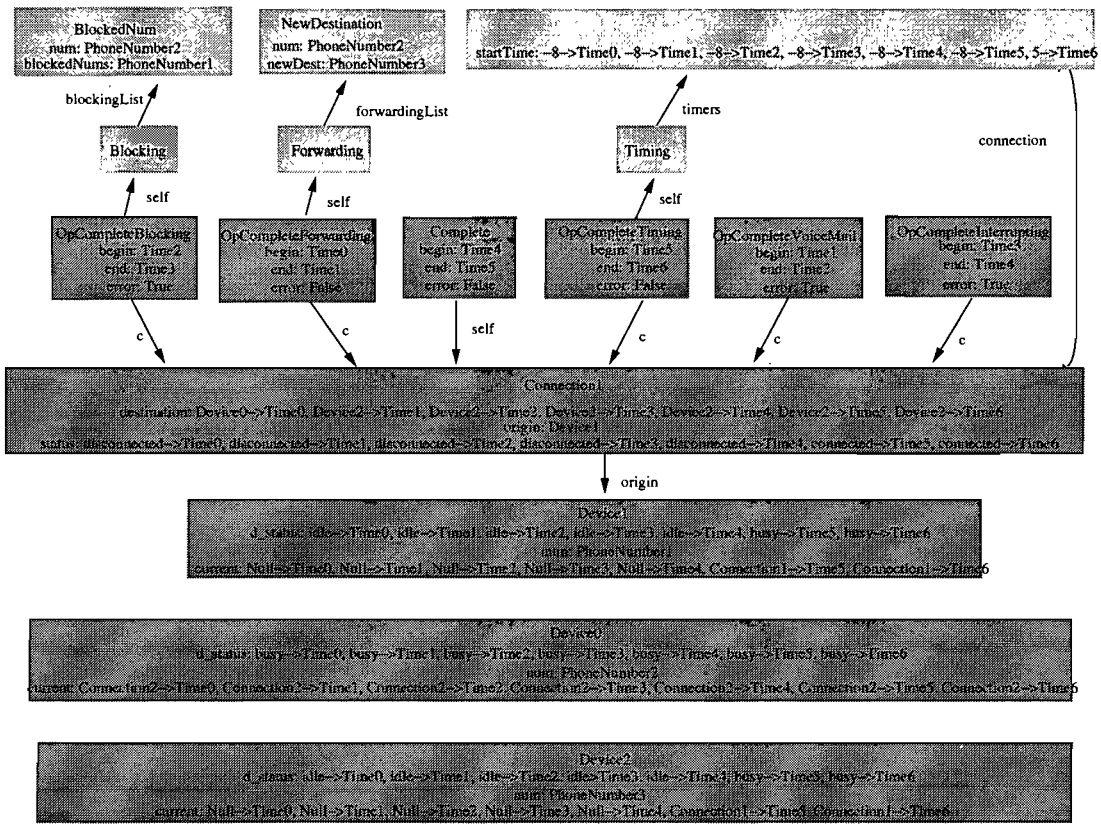


FIG. 6.4 – Contre-exemple généré pour l'assertion `localVerifAtComplete`.

Le contre-exemple peut correspondre concrètement au scénario suivant :

- Pour sa ligne résidentielle, Mme Leblanc souscrit aux services de blocages des appels interurbains et au service d'interruption d'appels si la ligne est occupée.
- D'autre part, M. Leblanc souscrit au service de boite vocale si la ligne est occupée.
- Étant à l'extérieur pour la journée, Mme Leblanc transfère ses appels de la maison vers son cellulaire.
- À la maison, la gardienne des enfants de Mme Leblanc est en communication avec son ami.
- Au même moment, l'appel interurbain entrant vers le numéro de résidence de Mme Leblanc est transféré vers son cellulaire. Ainsi, cet appel interurbain n'a pas été transféré vers la boite vocale et n'a pas été bloqué. De même, la communication en cours établie par la gardienne n'a pas été interrompue.

Dans la figure 6.4, il est montré qu'initialement, à l'état *Time*0, la destination (*Device*0) de la connexion (*Connection*1) à compléter est occupée (état *busy*), son numéro de téléphone (*PhoneNumber*2) est dans la liste de transfert d'appels *forwardingList*. Exécuté au temps *Time*0, l'advice `OpCompleteForwarding` a transféré l'appel vers le numéro *PhoneNumber*3. En effet, au temps *Time*1, la destination est devenue *Device*2 ayant le numéro *PhoneNumber*3. À présent, cette nouvelle destination est à l'état `idle`, ce qui viole les préconditions de l'advice `OpCompleteVoicemail`, l'empêchant ainsi de s'exécuter au temps *Time*1. Quant à l'advice `OpCompleteBlocking`, qui doit bloquer les appels venant du numéro *PhoneNumber*1 du *Device*1 (origine de l'appel) vers *PhoneNumber*2 du *Device*0 (destination initiale de l'appel), il ne s'est pas exécuté. La raison est que, exécuté en premier, l'advice `OpCompleteForwarding` a changé la destination de l'appel violant ainsi les préconditions de l'advice `OpCompleteBlocking`. De même, l'advice `OpCompleteInterrupting`, dont l'exécution devait interrompre la connexion courante du device *device*0 (puisqu'initialement son état était `busy`), ne s'est pas exécuté au temps *Time*3. Dans ce cas aussi, l'exécution de `OpCompleteForwarding` a violé les préconditions de l'advice `OpCompleteInterrupting` en transférant l'appel vers la destination *Device*2 dont l'état au temps *Time*3 est *idle*.

En revanche, le point de jointure `Complete` s'est correctement exécuté au temps *Time*4. On voit bien que l'état de la connexion passe de *disconnected* (*Time*4) à *connected* (*Time*5), et son origine (*Device*1) et sa destination (*Device*2) passent de l'état *idle* à l'état *busy*. L'advice `OpCompleteTiming` s'est aussi exécuté correctement au temps *Time*5. Le Timer de la connexion affiche bien un temps de début de connexion positif à la fin de l'exécution de cet advice (*Time*6), alors que ce temps était négatif avant l'exécution de `OpCompleteTiming`. En outre, les postconditions du point de jointure sont assurées à la fin de l'exécution de cet advice (au temps *Time*6).

Le contre-exemple généré par Alloy pour l'assertion `localVerifAtComplete`

indique donc que l'aspect `Forwarding` interfère avec les aspects `Interrupting`, `VoiceMail` et `Blocking` au point de jointure `Complete`, s'il est exécuté en premier.

3. Suite de l'analyse : énumération des autres contre-exemples

L'outil Alloy permet d'énumérer tous les contre-exemples générés par l'analyse d'une assertion donnée. En effet, l'outil dispose d'un bouton `Next` qui permet de visualiser le prochain contre-exemple s'il y en a.

Pour l'assertion `localVerifAtComplete`, les contre-exemples énumérés par Alloy ont révélé les cas d'interactions suivants dus aux aspects `Forwarding`, `Interrupting`, `VoiceMail` et `Blocking`.

- L'aspect `Forwarding` interfère avec les aspects `Interrupting`, `VoiceMail` et/ou `Blocking` à chaque fois qu'il est exécuté avant au moins un de ces aspects (peu importe l'ordre d'exécution des autres aspects).
- Les aspects `Blocking` et `VoiceMail` interfèrent avec le point de jointure `Complete`. Ce cas d'interférence est montré par le contre-exemple trouvé par Alloy et donné à la figure 6.5. Dans ce contre-exemple, il est montré que tous les advices se sont exécutés sans erreur ($error = False$), par contre une erreur a eu lieu dans l'exécution du point de jointure `Complete` ($error = True$). L'exécution correcte des advices `OpCompleteBlocking` et `OpCompleteVoiceMail` a violé une des postconditions du point de jointure `Complete` (qui est $status = disconnected$) l'empêchant ainsi de s'exécuter.
- L'aspect `Interrupting` interfère avec l'aspect `VoiceMail` s'il est exécuté avant. En effet, l'exécution correcte de l'advice `OpCompleteInterrupting` viole la précondition de l'advice `OpCompleteVoiceMail`. Cette précondition indique que pour offrir la boite vocale à une connexion il faut que sa destination soit occupée.

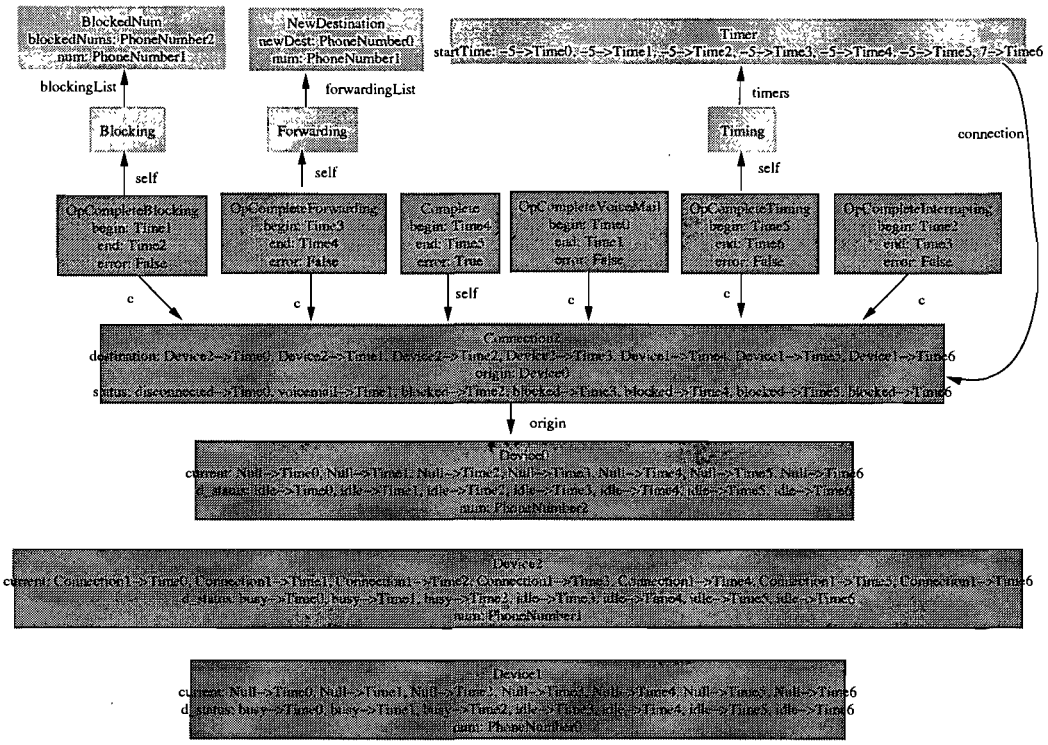


FIG. 6.5 – Un autre contre-exemple pour l'assertion `localVerifAtComplete`.

6.5.2.2 Vérification locale au point de jointure `Drop`

Pour vérifier les propriétés locales au point de jointure `Drop`, nous formulons l'assertion `localVerifAtDrop` décrite à la table 6.12.

1. Description de l'assertion `localVerifAtDrop`

Cette assertion suppose que les hypothèses initiales sont correctes, c'est-à-dire que le système de base et les aspects (`Timing`, `Billing` et `Stats`) sont individuellement corrects au point de jointure `Drop`. Ces hypothèses sont celles imposées par les préconditions définies dans le modèle Aspect-UML de l'application de téléphonie. Sous ces conditions, l'analyseur Alloy doit vérifier que les conditions finales du point de jointure `Drop` (qui sont les postconditions du point de jointure) ne sont pas violées et que la composition des opérations (le point de jointure et les advices) n'est pas incohérente.

```
assert localVerifAtDrop {all w : WeavingAtDrop |
        initialCondAtDrop[w]=> finalCondAtDrop[w] && noErrorAtDrop[ ] }

pred initialCondAtDrop(w :Weaving) {
        Drop.self.status.(w.begin)= connected &&
        Drop.self.origin.d_status.(w.begin)= busy &&
        Drop.self.destination.(w.begin).d_status.(w.begin)= busy &&
        Drop.self.origin.current.(w.begin) = Drop.self &&
        Drop.self.destination.(w.begin).current.(w.begin) = Drop.self &&
        pos[getTimer[Timing,Drop.self].startTime.(w.begin)] &&
        neg[getTimer[Timing,Drop.self].connectionTime.(w.begin)] }

pred finalCondAtDrop(w :Weaving) {
        (Drop.self.status.(w.end)= disconnected) &&
        (Drop.self.origin.d_status.(w.end)=idle) &&
        (Drop.self.destination.(w.end).d_status.(w.end)=idle) &&
        (Drop.self.origin.current.(w.end)= Null) &&
        (Drop.self.destination.(w.end).current.(w.end)= Null) }

pred noErrorAtDrop() {
        Drop.error = False &&
        OpDropTiming.error= False &&
        OpDropBilliing.error=False &&
        OpStats.error=False }

check localVerifAtDrop for 3 but 9 Operation, 1 Weaving, 1 WeavingAtDrop,
        1 JP, 1 OpCompleteTiming, 1 OpCompleteBilling, 1 OpCompleteStats, 5 Time
```

TAB. 6.12 – Assertion `localVerifAtDrop`.

La commande `check`, utilisée pour la vérification de cette assertion, définit une portée limitant le nombre d'éléments dans la signature `Operation` à neuf (définissant le point de jointure, les trois advices, les quatre compositions et le tissage) et la limite dans la signature `Time` à cinq pour coordonner l'exécution du point de jointure `Drop` tissé avec les advices `OpCompleteTiming`, `OpCompleteBilling` et `OpCompleteStats`. Bien sûr, un seul élément `Weaving` et un seul élément `JP` sont nécessaires pour spécifier le tissage et le point de jointure. Réciproquement, seul un élément de chaque signature advice est requis.

2. Analyse de l'assertion `localVerifAtDrop`

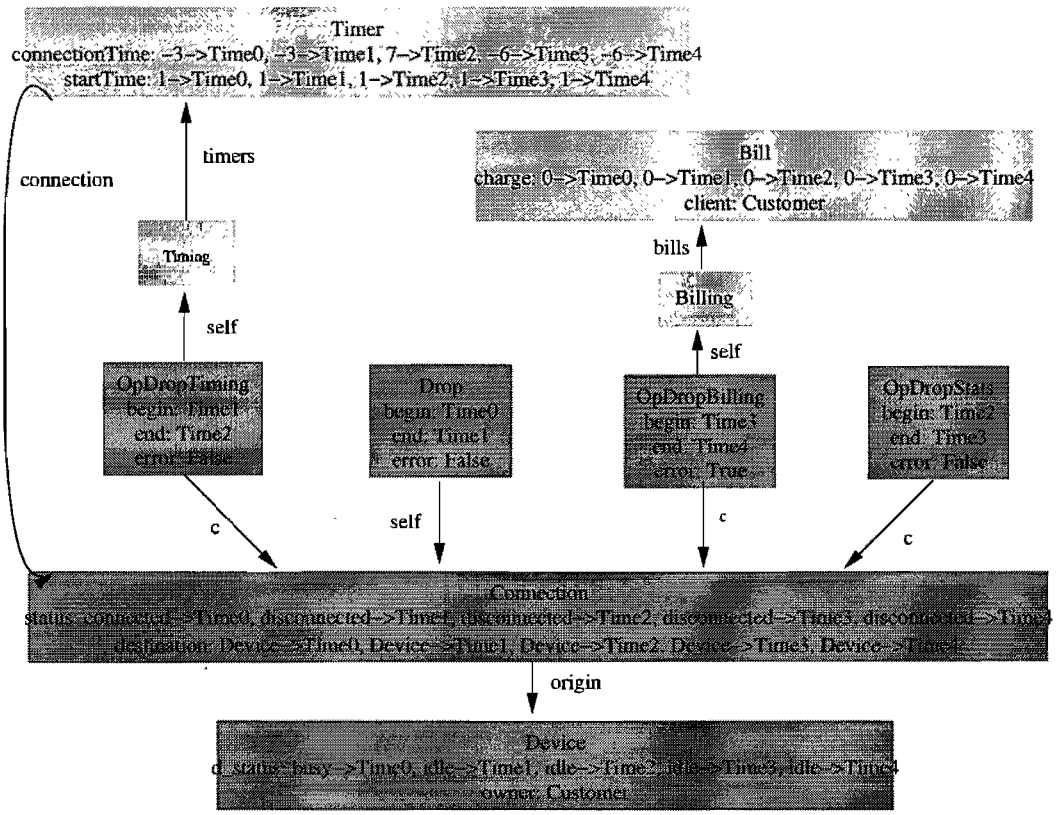


FIG. 6.6 – Contre-exemple généré pour l'assertion `localVerifAtDrop`.

L'analyse de `localVerifAtDrop` a généré le contre-exemple décrit par la figure 6.6. La figure ne montre que les éléments pertinents à la vérification. Il est clairement montré qu'une erreur a eu lieu durant l'exécution de l'advice : `OpDropBilling` ($error = True$), alors que les autres opérations `Drop`, `OpDropTiming` et `OpDropStats` ($error = False$) ont été exécutées correctement.

Notons que l'ordre d'exécution des trois advices est tel que `OpDropBilling` est exécuté après `OpDropTiming` et `OpDropStats` est exécuté de façon non déterministe. Dans le contre-exemple généré par Alloy, on voit bien que l'exécution de `OpDropTiming` est suivie par celle de `OpDropStats`, qui elle est suivie par celle de `OpDropBilling`.

Initialement, à l'état $Time0$, le point de jointure `Drop` s'est exécuté correctement

assurant ainsi toutes ses postconditions (au temps $Time1$). À l'état $Time1$, l'advice `OpDropTiming` s'exécute à son tour et change le temps de connexion *connectionTime* de négatif à positif (à $Time2$). Ceci permet à `OpDropStats` de s'exécuter au temps $Time2$. Cette exécution réinitialise *connectionTime* à une valeur négative (post-condition de `OpDropStats`) et empêche ainsi l'advice `OpDropBilling` de s'exécuter (en violant sa précondition qui stipule que le temps de connexion doit être non négatif pour pouvoir facturer la communication). Le contre-exemple généré par Alloy pour l'assertion `localVerifAtDrop` indique donc que l'aspect `Stats` interfère avec l'aspect `Billing` s'il est exécuté en premier.

3. Suite de l'analyse : énumération des autres contre-exemples

Pour l'assertion `localVerifAtDrop`, d'autres contre-exemples ont été trouvés par Alloy. Ces contre-exemples ont révélé les cas d'interactions suivants entre les aspects `Timing`, `Billing` et `Stats`.

- Un contre-exemple a révélé une erreur dans `OpDropStats` s'il est exécuté avant `OpDropTiming`.
- Aucun contre-exemple ne correspond au cas où les aspects `Timing`, `Billing` et `Stats` sont exécutés respectivement dans cet ordre. Ceci assure que (dans la portée choisie) les aspects exécutés, dans l'ordre imposé, n'interfèrent pas mutuellement sur leurs pré et postconditions et sur les pré et postconditions du point de jointure `Drop`.

Par ailleurs, la vérification des propriétés locales au point de jointure `Drop`, en supposant qu'aucun ordre n'est défini entre les aspects `Timing` et `Billing` (*c-à-d* en considérant un modèle où les trois aspects sont exécutés dans un ordre non-déterministe) a généré un contre-exemple. Ce contre-exemple révèle qu'une erreur survient dans `OpDropBilling` s'il est exécuté avant `OpDropTiming`.

6.5.3 Vérification des propriétés globales

L'introduction d'un aspect dans un système de base, peut violer les invariants de ce système, les invariants d'autres aspects ou encore ses propres invariants. En utilisant notre approche, la vérification des invariants se fait en décrivant une assertion `verifInvariant`, suivant le patron présenté à la table 6.10, page 191.

Dans notre application de téléphonie, nous nous intéressons à vérifier les invariants suivants :

- **Inv1** : les appels d'urgence ne sont jamais interrompus ;
- **Inv2** : pour chaque connexion donnée, l'origine et la destination sont différentes ;
- **Inv3** : si la boite vocale est offerte à une connexion, alors l'état de sa destinataire doit rester occupé (`busy`) ;
- **Inv4** : le temps de début de communication n'est calculé que pour les connexions initiées (complétées) ;
- **Inv5** : la durée de communication n'est positive que si le temps de début de connexion est positif.
- **Inv6** : toute connexion terminée est facturée ;
- **Inv7** : seules les connexions ayant un temps de communication positif sont facturées ;
- **Inv8** : la boite vocale ne doit pas être offerte aux appels dont la destination bloque les appels venant de l'origine ;
- **Inv9** : si un usager (numéro de téléphone $N1$) demande de bloquer tous les appels émanant d'un autre numéro $N2$, alors aucune connexion ne doit être établie (complétée) entre $N2$ et $N1$.
- **Inv10** : si un usager (numéro de téléphone $N1$) demande de transférer tous ses appels vers une autre destination $N2$, alors aucune connexion initiée (complétée) ne doit avoir comme destination le numéro $N1$.

6.5.3.1 Vérification de Inv1

L'invariant `Inv1` est un invariant du système de base. Il indique que les appels d'urgence ne sont jamais interrompus. Il est spécifié par le prédicat `Inv1` et sa vérification est réalisée en analysant l'assertion `verifInv1` donnés par la table 6.13.

```
pred Inv1 (t :Time) { all c : Connection |
        ( c.destination.t.num = emergencyNum || c.origin.num=emergencyNum )
                => ( c.status.t != interrupted ) }

assert verifInv1 { all w : WeavingAtComplete + WeavingAtDrop |
        correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv1[t] }

check verifInv1 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.13 – Assertion pour la vérification de l'invariant `Inv1`.

L'analyse de cette assertion avec Alloy a généré le contre-exemple donné par la figure 6.7. Le contre-exemple décrit un scénario qu'on peut imaginer comme suit :

- Alice est en communication urgente vers le 911.
- L'appel de Bob à Alice interrompt soudainement la communication courante déjà établie par Alice.

Dans le scénario exhibé par Alloy (figure 6.7), la connexion (*Connection*0) à établir a comme destination (*Device*0) au temps *Time*0. L'exécution de `OpComplete-Forwarding` a causé le transfert (au temps *Time*3) de cette connexion vers la destination (*Device*1). Or, cette nouvelle destination est à l'état `busy` et est déjà engagée dans la connexion *Connection*1. L'exécution de l'advice `OpCompleteInterruting` à l'état *Time*3 libère la destination *Device*1 en mettant son état à *idle* ce qui entraîne l'interruption de la *Connection*1. Or, celle-ci est une connexion urgente qui a comme destination *Device*0, dont le numéro est un numéro d'urgence *emergency-Num*. L'exécution de `OpCompleteInterrupting` a ainsi causé la violation de l'invariant `Inv1`.

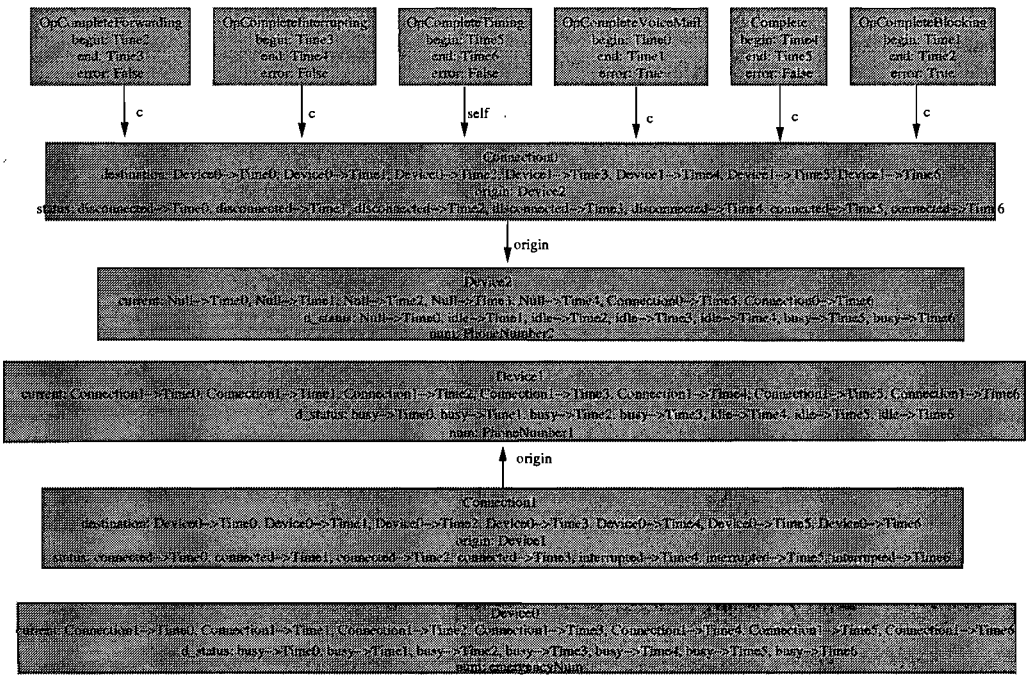


FIG. 6.7 – Contre-exemple généré pour la vérification de l'invariant `Inv1`.

6.5.3.2 Vérification de Inv2

L'invariant `Inv2` est un invariant du système de base. Il indique que pour toute connexion, l'origine et la destination sont différentes. Cet invariant est spécifié par le prédicat `Inv2` et est vérifié par l'assertion `verifInv2`, donnés par la table 6.14.

```
pred Inv2 (t :Time) { all c : Connection |
      (c.origin != c.destination.t) }

assert verifInv2 { all w : WeavingAtComplete + WeavingAtDrop |
      correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv2[t] }

check verifInv2 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.14 – Assertion pour la vérification de l'invariant `Inv2`.

L'analyse de cette assertion avec Alloy a généré le contre-exemple de la figure 6.8. Le contre-exemple décrit un scénario qu'on peut imaginer comme suit :

- Bob a activé le transfert de tous ses appels vers ses parents pour le week-end.
- Une fois de retour, Bob a oublié de désactiver la fonction de transfert.
- Ses parents n'arrivent pas à le joindre (puisque le transfert d'appels est toujours activé). Ainsi, leur appel leur est retourné.

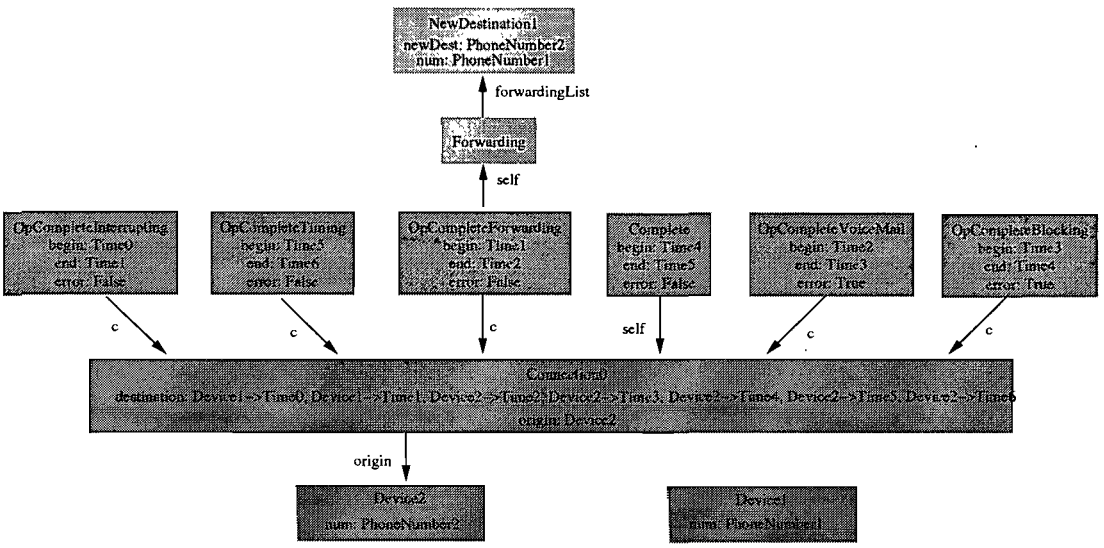


FIG. 6.8 – Contre-exemple généré pour la vérification de l'invariant `Inv2`.

Dans ce contre-exemple (figure 6.8), l'invariant `Inv2` est violé suite à l'exécution de l'advice `OpCompleteForwarding`. Puisque la destination *Device*1 (*PhoneNumber*1) a demandé le transfert de ses appels vers *Device*2 (*PhoneNumber*2), l'exécution de `opCompleteForwarding` a causé le transfert de la connexion *Connection*0 vers la destination *Device*2 qui n'est autre que l'origine de la connexion, violant ainsi l'invariant `Inv2`.

6.5.3.3 Vérification de Inv3

L'invariant `Inv3` est un invariant de l'aspect `VoiceMail`. Il indique que si la boite vocale est offerte à une connexion, alors sa destination doit être occupée. Il

est spécifié par le prédicat `Inv3` et vérifié par l'assertion `verifInv3`, donnés par la table 6.15.

```
pred Inv3 (t :Time) { all c : Connection |
        c.status.t = voicemail => c.destination.t.d_status.t = busy }

assert verifInv3 { all w : WeavingAtComplete + WeavingAtDrop |
        correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv3[t] }

check verifInv3 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.15 – Assertion pour la vérification de l'invariant `Inv3`.

L'analyse de cette assertion avec Alloy a généré le contre-exemple de la figure 6.9. L'erreur peut correspondre concrètement au scénario suivant :

- Mme Leblanc transfère ses appels professionnels du bureau vers la maison.
- L'appel ainsi destiné au bureau de Mme Leblanc est transféré vers son domicile.
- En même temps, puisque Mme Leblanc est en communication avec la gardienne de ses enfants, la boite vocale est offerte à cet appel.
- Or, ayant souscrit aussi au service d'interruption d'appels, la communication de Mme Leblanc vers sa gardienne est quand même interrompue, libérant ainsi la destination de l'appel transféré déjà à la boite vocale.

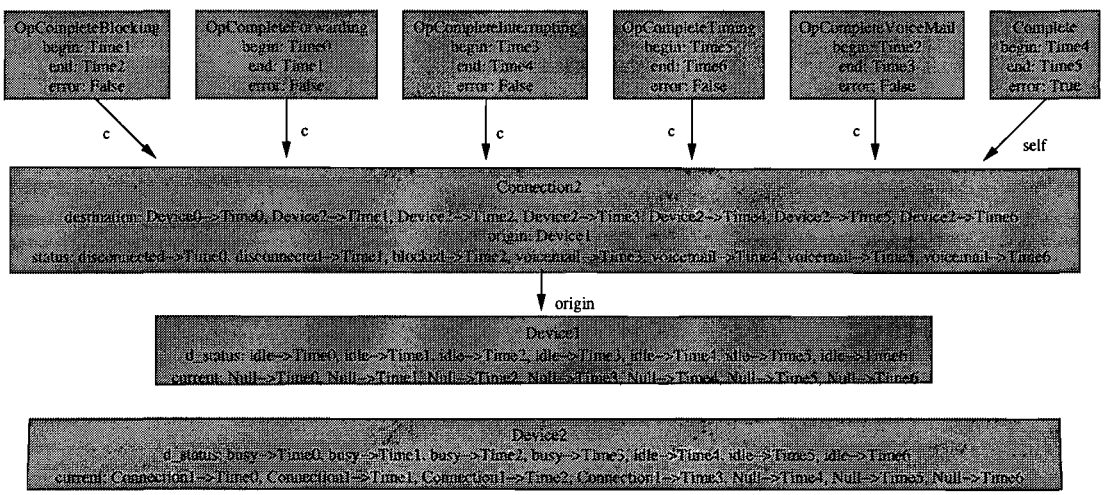


FIG. 6.9 – Contre-exemple généré pour la vérification de l'invariant `Inv3`.

Dans le scénario exhibé par Alloy (figure 6.9) la connexion (*Connection*0), à

établir, a comme destination (*Device*0) au temps *Time*0. L'exécution de `OpCompleteForwarding` a causé le transfert (au temps *Time*1) de cette connexion vers la destination (*Device*2). Comme cette nouvelle destination est à l'état *busy*, l'exécution de l'advice `OpCompleteVoiceMail` au temps *Time*2 offre la boite vocale à cette connexion sans libérer la destination *Device*2 (*c-à-d* en gardant son état *busy*). Par ailleurs, l'exécution de `OpCompleteInterrupting`, au temps *Time*3, libère la destination *Device*2 et interrompt la connexion dans laquelle elle était engagée causant ainsi la violation de l'invariant `Inv3` (notons que l'advice `OpCompleteBlocking` s'est aussi exécuté entre les temps *Time*1 et *Time*2, mais ce n'est pas son exécution qui a entraîné la violation l'invariant `Inv3`).

6.5.3.4 Vérification de Inv4

L'invariant `Inv4` est un invariant de l'aspect `Timing`. Il indique que le temps de début de communication ne doit être calculé que pour les connexions initiées (complétées). Il est spécifié par le prédicat `Inv4`, et vérifié par l'assertion `verifInv4`, donnés par la table 6.16.

```
pred Inv4 (t :Time) { all c : Connection |
      pos[getTimer[Timing, c].startTime.(t)] => c.status.t= connected }

assert verifInv4 { all w : WeavingAtComplete + WeavingAtDrop |
      correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv4[t] }

check verifInv4 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.16 – Assertion pour la vérification de l'invariant `Inv4`.

L'analyse de cette assertion avec Alloy a généré le contre-exemple de la figure 6.10.

Dans le scénario exhibé par Alloy (figure 6.10) la connexion (*Connection*2) à établir ne peut pas être initiée (complétée) au temps *Time*4 car l'exécution de

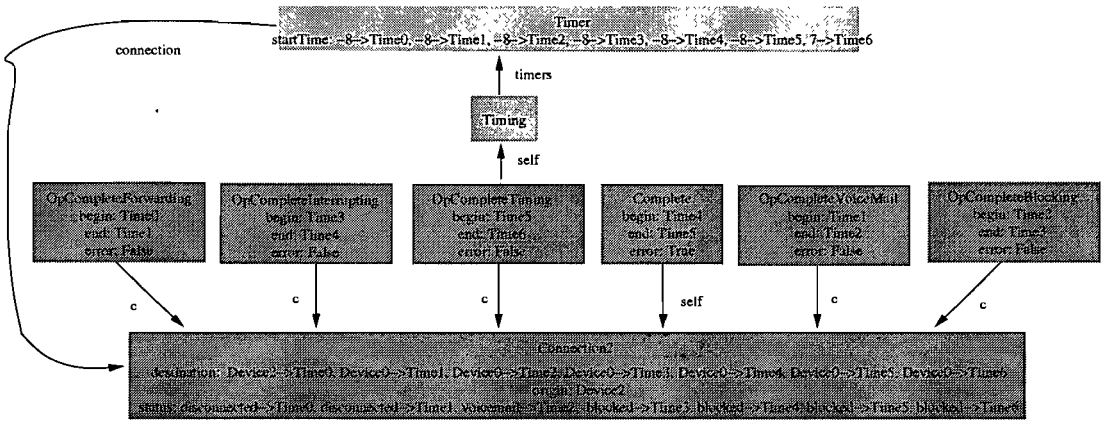


FIG. 6.10 – Contre-exemple généré pour la vérification de l'invariant `Inv4`.

l'advice `OpCompleteBlocking` bloque l'exécution du point de jointure `Complete`, en mettant l'état de la connexion à *blocked*. Or, ceci n'a pas empêché l'advice `opCompleteTiming` de s'exécuter au temps $Time5$, calculant ainsi le temps de début de communication (qui passe de négatif à positif au temps $Time6$). Ceci viole l'invariant `Inv4`.

6.5.3.5 Vérification de Inv5

L'invariant `Inv5` indique que la durée de communication n'est positive que si le temps de début de connexion est positif. Ceci est spécifié par le prédicat `Inv5` et vérifié par l'assertion `verifInv5`, donnés par la table 6.17.

```
pred Inv5 (t :Time) { all c : Connection |
      pos[getTimer[Timing, c].startTime.(t)] =>
            pos[getTimer[Timing, c].connectionTime.(t)] }

assert verifInv5 { all w : WeavingAtDrop |
      correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv5[t] }

check verifInv5 for 3 but 9 Operation, 1 JP, 1 Weaving, 5 Time
```

TAB. 6.17 – Assertion pour la vérification de l'invariant `Inv5`.

Notons que cette assertion ne considère que les instances de modèles qui spécifient le tissage au point de jointure `Drop`, *c-à-d* les signatures de type `WeavingAtDrop` puisque la durée de communication n'est calculée et n'est connue qu'à ce niveau (à la fin d'une connexion). L'analyse de cette assertion avec Alloy n'a généré aucun contre-exemple. Ceci nous assure que dans la portée choisie (*c-à-d* dans n'importe quel modèle décrivant un tissage correct au point de jointure `Drop`), l'invariant `Inv5` est vérifié.

6.5.3.6 Vérification de Inv6

L'invariant `Inv6` est un invariant de l'aspect `Billing`. Il indique que toute connexion terminée doit être facturée. Il est spécifié par le prédicat `Inv6` et vérifié par l'assertion `verifInv6`, donnés dans la table 6.18.

```
pred Inv6 (t :Time) { all c : Connection |
      c.status.t= disconnected => pos[getBill[Billing, c].charge.(t)] }

assert verifInv6 { all w : WeavingAtDrop |
      correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv6[t] }

check verifInv6 for 3 but 9 Operation, 1 JP, 1 Weaving, 5 Time
```

TAB. 6.18 – Assertion pour la vérification de l'invariant `Inv6`.

Comme pour l'assertion précédente, `verifInv6` ne considère que les instances de modèles qui spécifient le tissage au point de jointure `Drop`, puisque la facturation n'est réalisée qu'à la fin de chaque connexion. L'analyse de cette assertion avec Alloy n'a généré aucun contre-exemple. Ceci nous assure que pour les instances choisies (tout modèle décrivant un tissage correct au point de jointure `Drop`), l'invariant `Inv6` est vérifié.

6.5.3.7 Vérification de Inv7

L'invariant `Inv7` est un invariant de l'aspect `Billing`. Il indique que seules les connexions ayant une durée de communication positive sont facturées aux clients. Ceci est spécifié par le prédicat `Inv7` et vérifié par l'assertion `verifInv7`, donnés ci-après :

```
pred Inv7 (t :Time) { all c : Connection |
      pos[getTimer[Timing, c].connectionTime.(t)] => pos[getBill[Billing, c].charge.(t)] }

assert verifInv7 { all w : WeavingAtDrop |
      correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv7[t] }

check verifInv7 for 3 but 9 Operation, 1 JP, 1 Weaving, 5 Time
```

TAB. 6.19 – Assertion pour la vérification de l'invariant `Inv7`.

Comme pour les deux précédentes, l'analyse de cette assertion avec Alloy n'a généré aucun contre-exemple. Ceci nous assure que dans la portée choisie, l'invariant `Inv7` est vérifié.

6.5.3.8 Vérification de Inv8

L'invariant `Inv8` est un invariant de l'aspect `Blocking`. Il indique que la boite vocale ne doit pas être offerte aux appels dont la destination bloque les appels venant de l'origine. Il est spécifié par le prédicat `Inv8` et est vérifié par l'assertion `verifInv8`, donnés par la table 6.20.

L'analyse de cette assertion avec Alloy a généré le contre-exemple de la figure 6.11. L'erreur peut correspondre concrètement au scénario suivant :

- L'appel de longue distance destiné au domicile des Leblanc est bloqué (refusé) puisque M. Leblanc a activé le service de blocage des appels émanant de l'étranger.
- En même temps, alors que Mme Leblanc est en communication, la boite vocale est offerte à ce même appel indésirable.

```
pred Inv8 (t :Time) {all b :BlockedNum | all c : Connection |
      (b in Blocking.blockingList) && (c=Complete.self) &&
      (c.destination.t.num= b.num) && (c.origin.num= b.blockedNums)
            => (c.status.t !=voicemail)

assert verifInv8 { all w : WeavingAtComplete + WeavingAtDrop |
      correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv8[t] }

check verifInv8 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.20 – Assertion pour la vérification de l'invariant `Inv8`.

Dans le contre-exemple exhibé par Alloy (figure 6.11), la connexion (*Connection*1), à établir, a comme destination (*Device*2) dont l'état est *busy* au temps *Time*0. Comme cette destination (*PhoneNumber*2) bloque tous les appels émanant de l'origine *Device*1 (*PhoneNumber*1), l'exécution de `OpCompleteBlocking` au temps *Time*0, cause le blocage de *Connection*1, en changeant son état de *disconnected* à *blocked*. Par ailleurs, l'exécution de `OpCompleteVoiceMail`, au temps *Time*1, a offert la boite vocale à cette même connexion (*Connection*1) causant ainsi la violation de l'invariant `Inv8`.

6.5.3.9 Vérification de Inv9

L'invariant `Inv9` est un invariant de l'aspect `Blocking`. Il indique que si un usager (numéro de téléphone $N1$) demande de bloquer tous les appels émanant d'un autre numéro $N2$, alors aucune connexion ne doit être établie entre $N1$ et $N2$. Cet invariant est spécifié par le prédicat Inv9 et est vérifié par l'assertion `verifInv9` donnés ci-après :

L'analyse de cette assertion avec Alloy a généré le contre-exemple donné à la figure 6.12. L'erreur peut correspondre concrètement au scénario suivant :

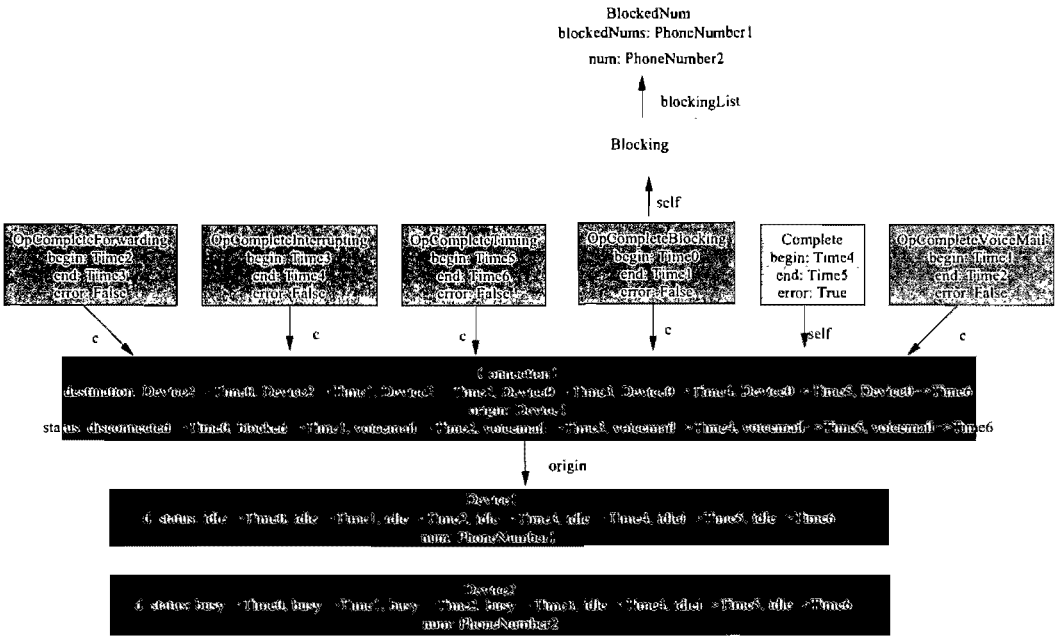


FIG. 6.11 – Contre-exemple généré pour la vérification de l'invariant `Inv8`.

- Bob a activé le transfert de tous ses appels vers Alice pour la durée de son voyage.
- Voulant joindre Bob, Marie tombe sur Alice au bout du fil. Ce qui déplait à Mary puisque celle-ci a activé le blocage de tous les appels venant de Marie.

Dans le scénario exhibé par Alloy (figure 6.12), la connexion $Connection1$, à établir, a comme origine $Device2$ ($PhoneNumber2$) et comme destination $Device1$ ($PhoneNumber1$) au temps $Time0$. L'exécution de `OpCompleteForwarding`, au temps $Time1$, transfert cette connexion à la destination $Device0$ ($PhoneNumber0$). Ainsi, au temps $Time5$, la connexion sera complétée ($status = connected$) suite à l'exécution correcte de `Complete`. Or, la nouvelle destination $Device0$ ($PhoneNumber0$) bloque tous les appels venant du numéro $PhoneNumber2$ ($Device2$). Ceci viole l'invariant `Inv9`.

```
pred Inv9 (t :Time) { all b :BlockedNum | all c : Connection |
        (b in Blocking.blockingList)&& (c=Complete.self)&&
        (c.destination.t.num= b.num) && (c.origin.num= b.blockedNums)
              => (c.status.t != connected) }

assert verifInv9 { all w : WeavingAtComplete + WeavingAtDrop |
        correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv9[t] }

check verifInv9 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.21 – Assertion pour la vérification de l'invariant `Inv9`.

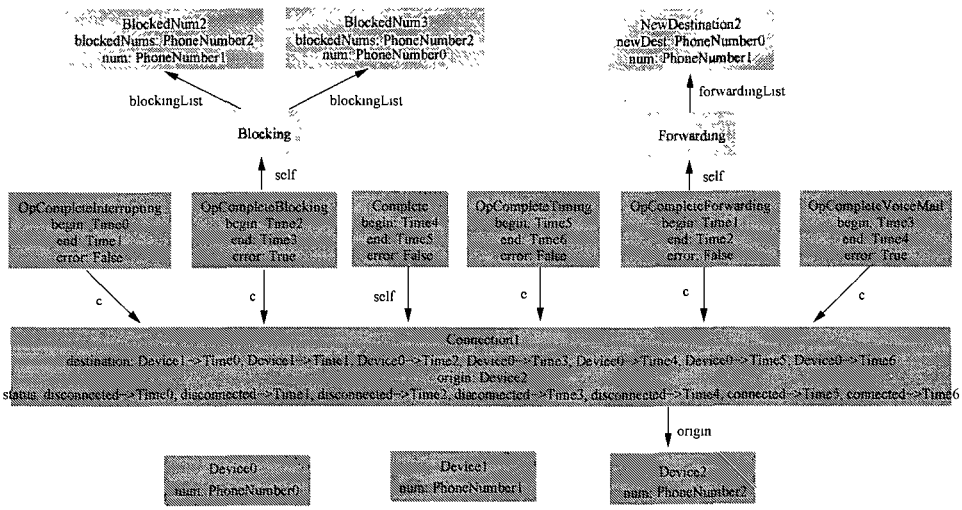


FIG. 6.12 – Contre-exemple généré pour la vérification de l'invariant `Inv9`.

6.5.3.10 Vérification de Inv10

L'invariant `Inv10` est un invariant de l'aspect `Forwarding`. Il indique que si un usager (numéro de téléphone N) demande de transférer ses appels alors aucune connexion initiée ne doit avoir comme destination le numéro N. Cet invariant est spécifié par le prédicat Inv10 et est vérifié par l'assertion `verifInv10` donnés à la table 6.22.

L'analyse de cette assertion avec Alloy n'a généré aucun contre-exemple. Ceci

```
pred Inv10 (t :Time) { all d : NewDestination | all c : Connection |
        (d in Forwarding.forwardingList) && (c=Complete.self)
        && (c.destination.t.num= d.num)
                => (c.status.t !=connected) }

assert verifInv10 { all w : WeavingAtComplete + WeavingAtDrop |
        correctExecution[w] => all t :Time | lte[w.begin,t] && lte[t, w.end] && Inv10[t] }

check verifInv10 for 3 but 12 Operation, 1 JP, 1 Weaving, 7 Time
```

TAB. 6.22 – Assertion pour la vérification de l'invariant `Inv10`.

nous assure que dans la portée choisie (*c-à-d* pour n'importe quel modèle décrivant un seul tissage correct au point de jointure `Complete` ou au point de jointure `Drop`) aucun aspect ne viole l'invariant `Inv10`.

6.5.4 Résumé de la vérification

La vérification avec Alloy de l'application de téléphonie à base de services a révélé des cas d'interactions intéressants entre les fonctionnalités des différents services. Ces interactions sont résumées dans la table 6.23. Les cas d'interactions causant la violation des propriétés globales sont représentées par la lettre `G`, et celles causant la violation des propriétés locales y sont référées par la lettre `L`.

La vue globale de ces interactions dans une même table permet à l'utilisateur de mieux comprendre les interférences entre les différents services du système, de définir des priorités entre les services et de faire des choix de conception en conséquence. Tout d'abord, notons que les interactions ne sont pas toutes indésirables. En effet, on peut vouloir concevoir un aspect dont le rôle justement est d'inhiber le comportement du système de base ou d'autres aspects. Comme par exemple les aspects `VoiceMail` et `Blocking` qui empêchent l'exécution du point de jointure `Complete` en violant ses pré-conditions : ce cas-ci est un cas d'interaction voulu. Ou encore, l'aspect `Interrupting` qui viole les propriétés locales de l'aspect `voi-`

	Base system	Timing	Billing	Stats	Forwarding	Interrupting	VoiceMail	Blocking
Base System	-	-	-	-	-	-	-	-
Timing	-	-	L^1	L^1	-	-	-	-
Billing	-	-	-	L^1	-	-	-	-
Stats	-	-	L^1	-	-	-	-	-
Forwarding	G (Inv2)	-	-	-	-	L^1	L^1	L^1, G (Inv9)
Interrupting	G (Inv1)	-	-	-	-	-	L^1,G (Inv3)	L^1
VoiceMail	L	G (Inv4)	-	-	-	-	-	G (Inv8)
Blocking	L	G (Inv4)	-	-	-	-	-	-

G : Violation des propriétés globales.

L : Violation des propriétés locales.

[1] Si aucun ordre n'est spécifié pour les apsects.

TAB. 6.23 – Résumé des cas d'interactions entre les services de l'application de téléphonie

ceMail l'empêchant ainsi de s'exécuter. Dans ce cas aussi, le comportement de l'aspect voiceMail est inhibé. En effet, si une destination, occupée initialement, est interrompue pour établir un nouvel appel, il est inutile d'offrir la boite vocale à cet appel.

Aussi, certaines interactions n'ont lieu que si aucun ordre n'est défini entre les aspects. C'est le cas des aspects Timing, Billing et Stats qui n'interagissent correctement que s'ils sont exécutés dans un ordre séquentiel. Ou encore l'aspect Forwarding qui viole les propriétés locales des aspects Interrupting, VoiceMail et Blocking, s'il est exécuté avant.

Par ailleurs, dans certains cas d'interactions, l'utilisateur doit faire des choix exclusifs. C'est le cas où les aspects qui interagissent ont des besoins mutuellement exclusifs. C'est le cas des aspects Interrupting et VoiceMail. En effet, l'aspect Interrupting viole les propriétés locales de VoiceMail et empêche son exécution, s'il est exécuté avant. D'autre part, s'il est exécuté après, il viole une propriété globale (Inv3) de VoiceMail.

6.6 Conclusion

Dans ce chapitre, nous avons prêté notre cadre formel de développement orienté aspect à la modélisation et la vérification d'une application réelle de téléphonie conçue à base de services. En mettant notamment l'accent sur l'aspect vérification, ce chapitre a illustré en détails l'application de notre méthodologie de développement à cette étude de cas. Dans un premier temps, nous avons développé un modèle Aspect-UML pour l'application de téléphonie en utilisant le profil Aspect-UML. Ensuite, nous avons traduit le modèle obtenu vers une spécification formelle Alloy. Par ailleurs, pour parvenir à la vérification de la spécification Alloy obtenue, nous avons proposé une approche pour la vérification des interactions dues aux aspects en utilisant l'analyseur Alloy. Pour cela, nous avons défini deux patrons décrivant

les assertions à utiliser respectivement pour la vérification des propriétés locales et des propriétés globales. Notons finalement que l'application de notre approche de vérification à l'application de téléphonie a révélé plusieurs cas d'interactions intéressants entre les différents services de l'application considérée.

CHAPITRE 7

CONCLUSION

7.1 Résumé et contributions

Le paradigme Aspect a émergé dans la communauté du génie logiciel afin de favoriser et de faciliter l'ingénierie des systèmes logiciels de bonne qualité. Par rapport aux approches de programmation traditionnelles, ce paradigme préconise une programmation modulaire qui permet une meilleure séparation des préoccupations d'un système non seulement principales (services et fonctions) mais également secondaires et transverses. Il propose, en effet, de décomposer les programmes non seulement en unités modulaires propres aux préoccupations principales, mais aussi en unités modulaires spécifiques aux préoccupations transverses, appelées communément *aspect*, très souvent gérées de façon inadéquate dans la programmation traditionnelle. Cette modularité permet ainsi d'offrir de nouvelles perspectives quant à lisibilité, la compréhension, la traçabilité, l'évolution et la réutilisation des systèmes logiciels produits. À l'origine [KLM+97], le paradigme aspect est apparu pour prendre en charge la séparation des préoccupations dans les implémentations des applications. Par la suite, beaucoup de travaux de recherche se sont consacrés à offrir des supports et des moyens à ce paradigme pour la prise en charge de la séparation des préoccupations et de la modularité durant les phases de cycle de développement en amont de la phase d'implémentation. Cependant, malgré la multitude des propositions concernant la modélisation par aspects, il n'existe pas de nos jours un standard pour une méthodologie de développement ni pour un langage de modélisation pour ce nouveau paradigme. À notre sens, un processus de développement orienté aspect rigoureux devra s'intéresser d'une part à la représentation et à la description des préoccupations au niveau des modèles et d'autre part

à la vérification de la composition de ces préoccupations et ce afin de former un système final cohérent.

Partant de ces constats, notre travail de recherche a eu pour objectif de doter le paradigme Aspect d'une approche de développement rigoureuse. Plus précisément, nous avons proposé, dans cette thèse, un cadre formel pour la modélisation et la vérification des systèmes par aspects, en nous intéressant plus particulièrement aux interactions dues aux aspects. Dans cette perspective nos contributions majeures ont été les suivantes.

Définition d'un profil Aspect-UML pour la modélisation orientée aspect [VM04a, MV06a]

Notre premier objectif a porté sur la définition d'un langage de modélisation pour le paradigme aspect afin de pouvoir prendre en charge les aspects dès les phases d'analyse et de conception et d'assurer ainsi leur traçabilité jusqu'à la phase d'implémentation. D'autre part, ce langage permet d'exprimer clairement la composition des aspects au niveau des modèles. Pour cette première contribution, nous avons proposé d'étendre le langage de modélisation orienté objet UML par un nouveau profil (Aspect-UML) que nous avons mis au point pour la modélisation orientée aspect. Ce profil définit des stéréotypes pour décrire les principaux concepts du paradigme Aspect et introduit aussi un formalisme de contraintes pour décrire la sémantique relative à la composition des aspects. Par ailleurs, ce profil permet de développer des modèles pour chacune des vues de cas d'utilisation, structurelle et comportementale.

Définition d'une sémantique du profil Aspect-UML en termes de réseaux de Petri [MV05, MV06a, MV06b]

L'approche de modélisation par aspects que nous avons proposée, à travers le profil Aspect-UML, permet de bien représenter et de décrire les aspects au niveau

des modèles d'analyse et de conception et offre des informations sémantiques précises quant à la composition des aspects. En particulier, ces informations sémantiques servent à la vérification formelle de la composition des aspects.

Bien que peu de travaux proposent d'intégrer la vérification formelle de la composition des aspects dans les modèles de conception, nous pensons que pour une méthodologie de développement rigoureuse la vérification de la composition des aspects est tout aussi importante que leurs représentation et description au niveau de la phase de conception.

Ainsi afin d'atteindre nos objectifs de vérification, la deuxième contribution de notre thèse était de doter notre profil Aspect-UML d'une sémantique formelle où le tissage des aspects est explicité formellement.

Notre but étant la vérification, nous avons choisi pour ce faire un formalisme disposant de mécanisme de vérification automatique. C'est ainsi que nous avons proposé d'abord une traduction des modèles Aspect-UML vers le formalisme des réseaux de Petri colorés (rdPc) [MV05, MV06a]. Notre choix de ce type de réseau était principalement motivé par le fait que les réseaux de Petri colorés disposent d'un outil de simulation et d'analyse automatique qui a fait ses preuves, soit CPN Tools [KCJ98]. Dans cette traduction nous avons décrit, en particulier, les aspects et les fonctionnalités de base d'un modèle Aspect-UML comme des réseaux de Petri colorés. Ensuite pour former le système global tissé, les réseaux de Petri ainsi obtenus sont composés et tissés à l'aide d'opérateurs de composition que nous avons définis.

Par ailleurs, ce type de réseau de Petri que nous avons choisi, dit coloré, fait appel au langage fonctionnel ML pour la représentation de données. Notre traduction d'Aspect-UML vers le formalisme des réseaux colorés a donc dû tenir compte d'un passage d'un monde objet/aspect vers un monde fonctionnel. Ce passage tel que décrit au chapitre 4 est assez complexe et délicat, vu justement la disparité des paradigmes objet et fonctionnel.

C'est ainsi que nous nous sommes tournés dans un second temps vers un autre type de réseaux de Petri dit orienté objet, incarné par le formalisme COOPN/2. Nous avons défini et proposé une traduction d'un modèle Aspect-UML vers une spécification COOPN/2 [MV06b]. En plus d'offrir une sémantique naturelle et intuitive au paradigme aspect, cette formalisation démontre une application intéressante pour les réseaux de Petri orientés objet et en particulier pour COOPN/2. En revanche, l'inconvénient majeur de ce type de réseaux est qu'ils ne sont pas soutenus par des outils de vérification automatique. À notre connaissance, il ne semble pas y avoir d'outils de vérification satisfaisants et performants pour ce type de réseaux.

Traduction de Aspect-UML vers Alloy [MV07a]

Si la traduction du profil Aspect-UML vers le formalisme des réseaux de Petri nous a permis de définir une sémantique de Aspect-UML à la fois précise, claire et naturelle, les réseaux de Petri présentent cependant des inconvénients, hélas, limitatifs quant à notre objectif de vérification. D'une part, comme nous l'avons souligné précédemment, les réseaux de Petri orientés objet ne sont pas soutenus par un outil de vérification automatique. D'autre part, tout comme la majorité des outils de vérification utilisant la méthode de vérification basée sur les modèles, l'outil CPN Tools supportant les réseaux de Petri colorés se voit rapidement confronté au problème d'explosion d'états. Rappelons, en effet, que pour procéder à la vérification les réseaux de Petri privilégient souvent l'approche par énumération d'états. Dans notre cas, par exemple, lors de la traduction de Aspect-UML vers les réseaux de Petri nous avons adopté des représentations symboliques des données pour réduire le nombre d'états. Notre solution a consisté à décrire les états du système au moyen de valeurs symboliques (plus abstraites) au lieu d'étayer l'infinité de combinaisons des valeurs concrètes correspondantes, où chaque valeur symbolique est une approximation d'un ensemble de valeurs concrètes.

C'est ainsi que notre troisième objectif dans le cadre de cette thèse a porté sur la formalisation du profil Aspect-UML en Alloy. Alloy est un langage de style déclaratif tout comme Aspect-UML qui nous offre des possibilités de vérification intéressantes. Il dispose, en effet, d'un outil de vérification qui permet de contourner le problème d'explosion d'états. D'une part, il procède pour la vérification par résolution de contraintes plutôt que par énumération des états. D'autre part, il permet à l'utilisateur de limiter la taille des modèles à analyser.

Pour la traduction de Aspect-UML vers Alloy, nous avons proposé une fonction de transformation qui permet de traduire un modèle Aspect-UML vers une spécification Alloy. La traduction inclut notamment la spécification en Alloy des éléments statiques d'un modèle Aspect-UML, de ses éléments comportementaux et des processus de composition et de tissage des aspects. Notons que cette traduction vers Alloy est inspirée de la sémantique en termes de réseaux de Petri que nous avons présentée.

Vérification des interactions dues aux aspects [MV07b, MV07a]

La traduction de Aspect-UML vers Alloy est un élément clé dans l'achèvement de la vérification formelle des interactions dans un modèle Aspect-UML à laquelle nous nous sommes intéressés plus particulièrement dans le cadre de cette thèse. L'approche de vérification que nous avons proposée suppose que le système de base et les aspects sont tous individuellement corrects (c'est-à-dire qu'on suppose que les spécifications par pré et postconditions sont correctes) et focalise sur la détection d'erreurs résultant des interactions dues aux aspects. Plus précisément, la vérification que nous avons proposée vise à révéler des problèmes d'interférence importants, telles que la violation des propriétés locales et la violation des propriétés globales dans un modèle Aspect-UML.

Pour atteindre nos objectifs de vérification avec Alloy, nous avons présenté une approche pour la vérification des interactions dues aux aspects qui a consisté à

définir dans le langage Alloy un patron générique pour chaque type de propriétés à vérifier (locale et globale). Pour la vérification d'une propriété donnée, il suffit alors d'instancier le patron correspondant et de l'exécuter avec l'analyseur Alloy.

Détection et résolution du problème d'interactions entre services téléphoniques [MV07a]

Pour appuyer nos contributions, nous avons présenté une étude de cas réelle qui a consisté en un système de téléphonie à base de services. À travers cet exemple d'application, nous avons défini et détaillé les différentes étapes liées à l'utilisation et à l'application de notre cadre formel de développement par aspects notamment en ce qui a trait à la modélisation et à la vérification des interactions dues aux aspects.

En particulier, l'utilisation de cette application de téléphonie a révélé des cas d'interactions dues aux aspects intéressants et subtils à la fois. Ceci nous a confirmé que le problème des interactions dues aux aspects peut être parfois très difficile à détecter et à résoudre par des inspections manuelles ou de simples tests, surtout si plus d'un aspect est rajouté à un même point de jointure.

Par ailleurs, l'étude de cas présentée a contribué largement à la résolution du problème crucial des interactions entre services connu dans les systèmes de téléphonie. Nous avons montré, en effet, que l'approche aspect peut être une solution intéressante pour ce genre de systèmes, en implémentant les services comme des aspects.

7.2 Perspectives et travaux futurs

Ce travail de recherche a défini un cadre formel pour le développement orienté aspect. En particulier, il a largement contribué dans les domaines de la modélisation et de la vérification des interactions dues aux aspects.

L'utilisation de notre cadre formel dans un environnement de développement par aspects permet de mieux comprendre les dépendances et les interactions entre aspects dans un système et surtout de révéler les interactions indésirables dues aux aspects. En particulier, ce cadre formel peut être appliqué à des systèmes comportant plusieurs aspects dont les dépendances ne sont pas forcément directes. Nous sommes convaincus que la détection des interactions dues aux aspects à un niveau précoce dans le processus de développement permet d'aider les développeurs à corriger rapidement les erreurs dans les systèmes bien avant l'implémentation. Ce qui réduit considérablement le temps de développement et de maintenance. D'autre part, nous aimerions évaluer nos travaux expérimentalement en appliquant notre cadre formel à des projets réels académiques et industriels. Cette évaluation doit nous permettre de qualifier l'intérêt pratique de notre approche.

Par ailleurs, ce projet de recherche ouvre la voie à un certain nombre de travaux futurs permettant d'enrichir et d'étendre notre cadre formel de développement par aspects. Les grandes lignes de ces travaux sont les suivantes.

Une perspective immédiate de notre contribution est la mise en oeuvre d'un outil supportant notre cadre formel de développement par aspects. En effet, il serait fort intéressant d'implémenter le profil Aspect-UML proposé et ainsi d'automatiser la transformation d'un modèle Aspect-UML vers Alloy, offrant ainsi à notre cadre formel un premier prototype opérationnel.

En ce qui concerne la modélisation par aspects, une perspective intéressante serait la généralisation de notre profil Aspect-UML à d'autres modèles du paradigme aspect. En effet, nous n'avons pour l'instant considéré que le modèle asymétrique dit modèle du point de jointure, mais il est intéressant d'étudier dans quelles mesures nos propositions sont applicables pour d'autres modèles de ce paradigme (telles que le modèle symétrique et les filtres de composition). En particulier, il est intéressant de voir quelles extensions devraient être apportées à notre profil Aspect-UML (ajout de nouveaux éléments et relations) afin de prendre en charge d'autres

concepts du paradigme aspect. Par ailleurs, il serait intéressant d'élargir les vues de notre profil Aspect-UML à d'autres types de diagrammes UML. Cette extension permettrait, d'une part, de montrer comment adapter les diagrammes UML tels que les diagrammes de séquences, les diagrammes de communication ou encore les diagrammes d'états afin de prendre en charge les concepts aspect. D'autre part, l'élargissement de notre profil à ces diagrammes permettrait une description plus riche et mieux élaborée des aspects au niveau des phases d'analyse et de conception. Déjà, un exemple de diagramme de séquence élaboré avec notre profil Aspect-UML est décrit dans [VM04a].

En ce qui concerne les annotations sémantiques qui enrichissent notre profil Aspect-UML, il convient certainement d'étudier les possibilités d'exprimer ces annotations dans un langage de spécification bien établi tel que OCL [Obj04] ou JML [LPC+07]. Il est surtout avantageux de voir dans quelles mesures un langage doté d'outils de vérification automatique tel que JML par exemple, pourrait servir et répondre à nos intérêts de vérification.

En ce qui concerne la vérification des systèmes par aspects, une perspective immédiate serait d'étendre notre approche de vérification en permettant de plus la modélisation de comportements. Actuellement, les annotations sémantiques utilisées dans le profil Aspect-UML spécifient le comportement des opérations sous forme de pré et post-conditions. Lors de la vérification de la composition des aspects, nous supposons que ces spécifications sémantiques sont correctes. Or, au lieu de demander à l'utilisateur de déclarer ce comportement par des pré et post-conditions, nous pensons qu'il serait plus intéressant de le décrire par des diagrammes dynamiques tels que les diagrammes d'états par exemple, et ensuite de déduire les spécifications (c'est-à-dire les pré et les post-conditions) minimales nécessaires à la vérification.

Enfin, comme nous l'avons déjà indiqué, nous avons axé notre approche de vérification sur la vérification modulaire des propriétés locales (spécifications des opérations) et des propriétés globales (invariants de système). Une perspective in-

téressante de notre proposition serait l'extension de notre approche de vérification modulaire à un ensemble plus large de propriétés des systèmes. Nous pensons en particulier à la vérification modulaire des propriétés de vivacité et de sûreté (exprimées sous forme de propriétés temporelles).

Finalement, nous espérons que le travail exposé dans ce document inspirera d'autres idées de recherche que ce soit sur le plan d'approfondissement du travail proposé ou sur le plan d'élargissement de ce domaine de recherche.

BIBLIOGRAPHIE

[ABC+05] J. Araujo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdoan. Early aspects : The current landscape. Technical report, Lancaster University, Lancaster, February 2005.

[ABGR07] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy : A challenging model transformation. In 10^{th} *International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, 2007.

[AEB03] O. Aldawud, A. Elrad, and A. Bader. A UML profile for aspect oriented software development. In *Int. Workshop on AOP at Int. Conf. on AOSD'2003*, 2003.

[All] Alloy Homepage. http ://alloy.mit.edu.

[And01] J. H. Andrews. Process algebra foundation of aspect oriented programming. In *3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209, 2001.

[Asp02] AspectJ Team. The AspectJ programming guide. Xerox Corporation, 2002.

[AT98] M. Aksit and B. Tekinerdogan. Aspect-oriented programming using composition filters. In *ECOOP'98 : Workshop on Object-Oriented Technology*, page 435, London, UK, 1998. Springer-Verlag.

[BA05] B. Bordbar and K. Anastasakis. UML2ALLOY : A tool for lightweight modelling of discrete event systems. In *IADIS Applied Computing*, 2005.

[BBF+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and software verification.* Springer-Verlag, 2001.

[BCM05] D. Balzarotti, A. Castaldo, and M. Monga. Slicing AspectJ woven code. In *The 4th Workshop on Fondations of Aspect-Oriented languages, FOAL'05 associated with AOSD2005*, March 2005.

[BDC+88] T.F. Bowen, F.S. Dworak, C.-H. Chow, N.D. Griffeth, G.E. Herman, and Y.-J. Lin. Views on the feature interaction problem. Technical report, Bellcore, 1988.

[BdC92] L. Bernardinello and F. de Cindio. A survey of basic net models and modular net classes. In *Advances in Petri Nets 1992, The DEMON Project*, pages 304–351, London, UK, 1992. Springer-Verlag.

[BGL04] E. Bara, G. Génova, and J. Llorens. An approach to aspect modelling with UML 2.0. In *The 5th Aspect-Oriented Modeling Workshop, In Conjunction with UML 2004*, 2004.

[Bib97] O. Biberstein. *COOPN/2 : An object-oriented formalism for the specification of concurrent systems.* PhD thesis, Department of Computer Science EPFL, University of Geneva, Switzerland, 1997.

[BM04] I. Brito and A. Moreira. Integrating the NFR framework in a RE model. In *Workshop on Early Aspects in conjunction with 3rd International Conference on Aspect-Oriented Software Development*, 2004.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering*, 21(10) :785–798, 1995.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin Cumming Series in Object-Oriented Software Engineering, 1991.

[Bra82] G.W. Brams. *Réseaux de Petri : Théorie et pratique, tomes 1 et 2.* Masson Editions, 1982.

[BS03] M. Bash and A. Sanchez. Incorporating aspects into the UML. In *AOSD 2003, 3rd Workshop on Aspect-Oriented Modeling with UML 2003*, 2003.

[CB05] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design : The Theme Approach.* Addison-Wesley Professional, March 2005.

[CL02a] C. Chavez and C. Lucena. A metamodel for aspect-oriented modeling. In *AOSD 2002, at Workshop on Aspect Oriented Modeling with UML 2002*, 2002.

[CL02b] C. Clifton and G. T. Leavens. Spectators and assistants : Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, 2002.

[CL02c] C. Clifton and G.T. Leavens. Observers and assistants : a proposal for modular aspectoriented reasoning. In *Foundations of Aspect Languages (FOAL) Workshop associated with AOSD 2002*, pages 308–309, 2002.

[Cla02] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer programming*, 44(1) :71–100, 2002.

[Cli05] C. Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs.* PhD thesis, Iowa State University, Ames, IA, USA, 2005.

[Coc00] A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, January 2000.

[CRS+05] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design approaches. Technical report, Lancaster University, Lancaster, May 2005. AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9.

[CW01] S. Clarke and R. J. Walker. Composition patterns : An approach to designing reusable aspects. In *ICSE '01, ACM, 5-14*, May 2001.

[Dev03] B. Devereux. Compositional reasoning about aspects using alternating-time logic. In *Foundations of Aspect Languages (FOAL) Workshop associated with AOSD2003*, 2003.

[DFS04] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In 3^{rd} *International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150, 2004.

[Dia01] M. Diaz. *Les réseaux de Petri- Modèles fondamentaux, Volume 1.* Hermès, Sciences Publications, 2001.

[DM01] G. Denaro and M. Monga. An experience on verification of aspect properties. In *Int. Workshop on Principles of Software Evolution*, 2001.

[Emm88] W. Emmerich. Timed and stochastic Petri nets, October 1988.

[FGM98] M. Felder, A. Gargantini, and A. Morzenti. A theory of implementation and refinement in timed Petri nets. *Theoretical Computer Science*, 202(1-2) :127–16, 1998.

[FS06] L. Fuentes and P. Sanchez. Elaborating UML2.0 profiles for AO design. In *8th Workshop on Aspect-Oriented Modelling (AOM), 5th Int. Conf. on Aspect-Oriented Software Development (AOSD)*, 2006.

[FS07] L. Fuentes and P. Sànchez. Designing and weaving aspect-oriented executable UML models. *Journal of Object Technology, Special Issue : Aspect-Oriented Modeling*, 6(7) :109–136, August 2007.

[GCDH+01] D. Gluch, S. Comella-Dorda, J. Hudak, G. Lewis, and C. Weinstock. Model-based verification : Scope, formalism, and perspective guidelines. Technical Report CMU/SEI-2001-TN-024 ADA396628, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[GJ08] M. W. Gong and H. A. Jacobsen. AspeCt-oriented C (ACC) langage specification, version 0.8. Technical report, Middlewar systems research group, Department of Electrical and Computer Engineering & Department of Computer Science, University of Toronto, Canada, 2008.

[GK06] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Fondation of Aspect Oriented Languages Workshop, FOAL'06, held with AOSD Conference*, 2006.

[GK07] M. Goldman and S. Katz. Maven : Modular aspect verification. In *TACAS*, pages 308–322, 2007.

[GM92] J.A. Goguen and J. Meseguer. Order-sorted algebra I : Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report 2, 1992.

[Hol97] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5) :279–295, 1997.

[Jac99] D. Jackson. A comparison of object modelling notations : Alloy, UML and Z. MIT Computer Science and Artificial Intelligence Laboratory, August 1999.

[Jac00] D. Jackson. Automating first-order relational logic. In *Foundations of Software Engineering*, pages 130–139', November 2000.

[Jac02] D. Jackson. Alloy : a lightweight object modelling notation. *Transactions on Software Engineering and Methodology (TOSEM'02)*, 11(2) :256–290, 2002.

[Jac03] I. Jacobson. Use cases and aspects - working seamlessly together. *Journal of Object Technology*, 2(4) :7–28, 2003.

[Jac05] D. Jackson. Alloy in 90 minutes. *Requirements Engineering Conference(RE'05)*, 2005.

[Jac06] D. Jackson. *Software Abstractions Logic, Language and Analysis.* MIT Press, 2006.

[JCJd92] I. Jacobson, M. Christerson, P. Jonsson, and G. övergaard. *Object-Oriented Softaware Engineering : A Use Case Driven Approach.* Addison-Wesley, 1992.

[Jen97] K. Jensen. *Colored Petri nets. Basic concepts, analysis methods and practical use.* Springer, 1997.

[JN05] I. Jacobson and P. W. Ng. *Aspect-oriented software development with use cases.* Adisson wesley Professional, 2005.

[JPWG02] J.M. Jézéquel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in UML. In *Aspect-Oriented Modeling with UML Workshop at AOSD'02*, 2002.

[JRB99] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Modeling Language Reference Manuel.* Addison-Wesley Object Technology Series, 1999.

[Kat04] S. Katz. Diagnostic of harmful aspects using regression verification. In *Foundations of Aspect Languages (FOAL) Workshop associated with AOSD2004*, 2004.

[KCJ98] L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured Petri nets. *Int. Journal on Software Tools for Technology Transfer*, 2(2) :98–132, 1998.

[KFG04] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT'04/FSE-12*, 2004.

[KG99] S. Katz and J. Gil. Aspects and superimpositions. In Springer Lecture Notes in Computer Science, editor, *Aspect-Oriented Programming Workshop with ECOOP99*, volume 1743, pages 308–309, 1999.

[KG02] J. Kienzle and R. Guerraoui. AOP : Does it make sense? the case of concurrency and failures. In Springer Lecture Notes in Computer Science, editor, *16th European Conference on Object Oriented Programming ECOOP*, volume 2374, pages 37–61, 2002.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In Springer Lecture Notes in Computer Science, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, pages 327–353, June 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, Ch. Maeda, Ch. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Springer Lecture Notes in Computer Science, editor, *European Conference on Object Oriented Programming (ECOOP)*, pages 220–242, 1997.

[KR06] S. Khan and A. Rashid. Analysis requirements dependencies and change impact using concern slicing. In *Aspects, Dependencies and Interactions Workshop at ECOOP*, July 2006.

[KYX03] J. Kienzle, Y.Yu, and J. Xiongn. On composition and reuse of aspects. In 2^{nd} *Foundations of Aspect-oriented Languages Workshop at AOSD 2003*, 2003.

[Lad02a] R. Laddad. Learn AspectJ to better understand aspect oriented programming. In *Javaworld magazine, http ://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html*, March 2002.

[Lad02b] R. Laddad. Separate software concerns with aspect oriented programming. In *Javaworld magazine,*

http ://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html, April 2002.

[Lad02c] R. Laddad. Use AspectJ to modularize crosscutting concerns in real world problems. In *Javaworld magazine, http ://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html*, January 2002.

[Lad03] R. Laddad. *AspectJ in action.* Manning, 2003.

[Lam02] R. Lammel. A semantics for method-call interception. In *1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, 2002.

[LPC+07] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and Joseph Kiniry. Jml reference manual, October 2007.

[MB01] R. Malan and M. Bredemeyer. Defining non functional requirements, 2001.

[McC68] J. McCarthy. Situations, actions and causal laws. *Semantic Information Processing*, pages 410–417, 1968.

[MGB04] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *In Third International Workshop on Critical Systems Development with UML (CSDUML), affiliated with UML Conference*, 2004.

[MH69] J. McCarthy and P. Hayes. Philosophical problems from the standpoint of aritificial intelligence. *Machine Intelligence 4, B. Melzter and D. Michie Editions*, pages 463–502, 1969.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency : state models & Java programs.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Moo77] R. Moore. *Interval analysis.* Prentice-Hall, 1977.

[MTHM96] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, Revised Edition.* MIT Press, 1996.

[Mur89] T. Murata. Petri nets : properties, analysis and applications. In *Proc. of the IEEE*, volume 77(4), 1989.

[MV05] F. Mostefaoui and J. Vachon. Modélisation et vérification formelle de la composition des aspects. In *Actes de la Journée Francophone sur le développement de logiciels par aspects JFDLPA'05*, 2005.

[MV06a] F. Mostefaoui and J. Vachon. Approche basée sur les réseaux de Petri pour la vérification de la composition dans les systèmes par aspects. *RSTI - L'Objet*, 12(2-3) :157–182, September 2006.

[MV06b] F. Mostefaoui and J. Vachon. Formalization of an aspect-oriented modeling approach. In *Formal Methods 2006*, August 2006.

[MV07a] F. Mostefaoui and J. Vachon. Design-level detection of interactions in Aspect-UML models using Alloy. *Journal of Object Technology*, 6(7) :137–165, August 2007.

[MV07b] F. Mostefaoui and J. Vachon. Verification of Aspect-UML models using Alloy. In *AOM '07 : Proceedings of the 10th international workshop on Aspect-oriented Modeling*, pages 41–48, New York, NY, USA, March 2007. ACM Press.

[Nae00] G. Naeser. *A Flexible Framework for Detection of Feature Interactions in Telecommunications Systems.* PhD thesis, Uppsala University, Sweden, 2000.

[NBA04] I. Nagy, L. Bergmans, and M. Aksit. Declarative aspect composition. In *Software Engineering Properties of Languages and Aspect Technologies (SPLAT)Workshop*, 2004.

[NCA01] T. Nelson, D. Cowan, and P. Alencar. Supporting formal verification of crosscutting concerns. In *REFLECTION 2001, Lecture Notes in Computer Science, Springer, vol.2192*, pages 153–169, 2001.

[NT04] S. Nakajima and T. Tamai. Lightweight formal analysis of aspect-oriented models. In *Workshop on Aspect-Oriented Modeling at UML'04*, 2004.

[Obj02a] Object Management Group. UML profile for CORBA specification. Technical Report OMG Adopted Specification ptc/02-054-01 , April 2002.

[Obj02b] Object Management Group. UML profile for enterprise distributed object computing specification. Technical Report OMG Adopted Specification ptc/02-02-05 , February 2002.

[Obj02c] Object Management Group. UML profile for schedulability, performance and time specification. Technical Report OMG Adopted Specification ptc/02-03-02 , March 2002.

[Obj04] Object Management Group. UML 2.0 : OCL Specification , 2004.

[Obj05] Object Management Group. Unified Modeling Language : Superstructure. version 2.0, August 2005.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS : A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[OT00] H. Ossher and P.L. Tarr. Hyper/j^{TM} : Multi-dimensional separation of concerns for Java. In *ICSE 2000, International Conference on Software Engineering*, June 2000.

[Pet62] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, Schriften des IIM, Bonn, Germany, 1962.

[Pet81] J.L. Peterson. *Petri net theory and the modelling of system.* Prentice Hall, 1981.

[PSD+04] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. Jac : an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 3(12) :1119–1148, 2004.

[Rat01] Rational Software Corporation. UML profile for EJB. Public draft , 2001.

[RBE+91] J. Rumbaugh, M. Blaha, F. Eddy, P. W., and W. Lorensen. *Object Oriented Modeling and Designs.* Prentice-Hall Inc., Englewood Cliffs, New Jersy, USA, 1991.

[Rei91] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80 :1–34, 1991.

[RGF+06] Y.R. Reddy, S. Ghosh, R.B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect Oriented Software Development I*, 3880 :75–105, 2006.

[RJ03] J. Riely R. Jagadeesan, A. Jeffrey. An untyped calculus for aspect oriented programs. In *ECOOP'03*, 2003.

[RM02] S. Reiff-Marganiec. *Runtime Resolution of Features Interactions in Evolving Telecommunications Systems.* PhD thesis, Department of Computing Science, University of Glasgow, UK, 2002.

[RMR05] S. Reiff-Marganiec and M. D. Ryan, editors. *Feature Interactions in Telecommunications and Software Systems VIII.* IOS Press, Amsterdam, The Netherlands, June 2005.

[RSB04] M Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *International Conference on Foundations of Software Engineering (FSE)*, 2004.

[Sco02] K. Scott. *The Unified Process Explained.* Addison-Wesley, 2002.

[SdM03] D. Sereni and O. de Moor. Static analysis of aspects. In *Aspect-Oriented Software Development AOSD'03*, pages 30–39, 2003.

[SGSP02] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. AspectC++ : An aspect-oriented extension to C++. In *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, February 2002.

[SHU02] D. Stein, S. Hanenberg, and R. Unland. A UML-based aspect-oriented design notation for AspectJ. In *International Conference on Aspect-Oriented Software Development*, pages 106–112. ACM, 2002.

[Sip03] H.B. Sipma. A formal model for cross-cutting modular transition systems. In *FOAL Workshop associated with AOSD2003*, 2003.

[SK] M. Storzer and J. Krinke. Interference analysis for AspectJ. In *FOAL-Workshop associated with AOSD2003.*

[SKB03] M. Storzer, J. Krinke, and S. Breu. Trace analysis for aspect application. In *AAOS worshop with ECOOP 2003*, 2003.

[SLR+06] F. Sanen, N. Loughran, A. Rashid, A. Nedos, A. Jackson, S. Clarke, E. Truyen, and W. Joosen. Classifying and documenting aspect interactions. In 5^{th} *AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software (ACP4IS) at AOSD'2006*, March 2006.

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

[STW+06] F. Sanen, E. Truyen, B. De Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, and S. Clarke. Study on interaction issues. Technical report, Lancaster University, Lancaster, February 2006. AOSD-Europe-KUL-7, Deliverable D44.

[SY99] J. Suzuki and Y. Yamamoto. Extending UML with aspects : aspect support in the design phase. In *AOP Workshop at ECOOP'99*, 1999.

[TMY] H. Tatsuzawa, H. Masuhara, and A. Yonezawa. Aspectual caml : an aspect-oriented functional language. In *Workshop on Foundations of Aspect Oriented Languages FOAL'05 associated with AOSD2005.*

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and JR. S. M. Sutton. N degrees of separation : Multi-dimensional separation of concerns. In *the 22th International Conference of Software Engineering, IEEE Computer Society Press*, pages 107–119, 1999.

[Vaz03] M. Vaziri. *Finding Bugs in Software with a Constraint Solver.* PhD thesis, MIT Computer Science and Artificial Intelligence Laboratory, Massachussets, USA, 2003.

[VM04a] J. Vachon and F. Mostefaoui. Achieving supplementary requirements using aspect-oriented development. In *ICEIS*, pages 584–587, 2004.

[VM04b] J. Vachon and F. Mostefaoui. An aspect-oriented development method based on MDA. In *Workshop on Software Architecture Description and UML in the Seventh International Conference on UML Modeling Languages and Applications*, Oct 2004.

[WKD04] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect oriented programming. *Transactions On Programming Languages and Systems*, 5 :890–910, 2004.

[WZL03] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Int. Conf. on Functional Programming*, 2003.

[Zav04] P. Zave. FAQ sheet on feature interaction, copyright AT&T, 2004.

[ZCdBG07] J. Zhang, T. Cottenier, A. Van den Berg, and J. Gray. Aspect composition in the motorola aspect-oriented modeling weaver. *Journal of Object Technology, Special Issue : Aspect-Oriented Modeling*, 6(7) :89–108, August 2007.

Annexe I

Spécification COOPN/2 de l'application de téléphonie

Cette annexe présente la spécification COOPN/2 de l'application de téléphonie (restreinte) présentée au chapitre 3 et dont le modèle Aspect-UML est donné par la figure 3.4, page 72.

```
;; - - - Les modules des types de données abstraits

;; En plus des modules décrivant les types de données abstraits(ADT) de base tels
que les booleens et les entiers,
la spécification COOPN/2 de l'application de téléphonie contient les modules ADT
suivants :

ADT Status;
Interface
   Sort status;
   Generators
      connected, disconnected : -> status;
End Status

ADT D_Status;
Interface
   Sort d_Status;
   Generators
      idle, busy : -> d_Status;
End D_Status

;; - - - Les modules de classes

Class Customer;
Interface
   Type customer;
   Generators
```

```
        Cust1, Cust2,... , Custn : -> customer;
End Customer;

Class Device;
Interface
    Use Customer, AbstractConnection;
    Type device;
    Methods
        getD_status _, setD_status _ : d_Status;
        getOwner _ : customer;
        getCurrent _ , setCurrent _ : abstractConnection;
Body
    Use D_Status, Customer, AbstractConnection;
    Place
        deviceStatus _ : d_Status;
        owner _ : customer;
        current _ : abstractConnection;
    Axioms
        getD_status : : deviceStatus st -> ;
        setD_status : : -> deviceStatus st;
        getOwner : : owner o -> owner o;
        getCurrent : : current c -> ;
        setCurrent : : -> current c;
        where
        st : d_Status, o : customer, c : abstractConnection;
End Device;

;; Cette classe permet d'introduire un objet Null pour dénoter une valeur non
;; référencée de type AbstractConnection.
Class NullConnection;
Interface
    Type nullConnection;
    Object Null;
End NullConnection

;; Cette classe réalise la couche abstraction d'une connexion. C'est avec cette
couche
;;que les clients interagissent.
;; Cette classe est entrecoupée par les aspects Timing et Billing.
Class AbstractConnection;
```

```
Interface
   Use NullConnection;
   Type abstractConnection;
   Subtype abstractConnection < nullConnection;
   Methods
      complete;
      drop;
   Creation
      create-connection;
Body
   Use ConnectionState, Timing_Aspect, Billing_Aspect;
   Place
      state _ : connectionState;
   Axioms
     ;; tissage de l'aspect Timing au point de jointure complete()
      complete with s.complete .. Timing.opComplete self
            : : state s -> state s;
     ;; tissage des aspects Timing et Billing au point de jointure drop()
      drop with s.drop .. (Timing.opDrop self .. Billing.opDrop self)
            : : state s -> state s;
      create-connection with s.create-connectionState self
            : : -> state s, state s, state s;
      where
      s : connectionState;
End AbstractConnection;

;; Cette classe réalise la couche implémentation d'une connexion
;; Elle n'est accessible que depuis la couche abstraite.
Class ConnectionState;
Interface
   Use AbstractConnection;
   Type connectionState;
   Methods
      getOrigin _ : device;
      complete;
      drop;
   Creation
      create-connectionState _ : abstractConnection;
Body
   Use Device, Status, D_Status;
```

```
    Place
      connect _ : abstractConnection ;
      status _ : status ;
      origin _ : device
      destination _ : device ;
    Axioms
      complete with (o.getD_status idle || o.getCurrent Null ||
            (o.setD_status busy || o.setCurrent c ||
            d.setD_status busy || d.setCurrent c )
        : : => origin o, destination d, status disconnected, connect c
            -> origin o, destination d, status connected, connect c ;
      drop with (o.getD_status busy || o.getCurrent c ||
            d.getD_status busy || d.getCurrent c) ..
            (o.setD_status idle || o.setCurrent Null ||
            d.setD_status idle || d.setCurrent Null )
        : : => origin o, destination d, status connected, connect c
            -> origin o, destination d, status disconnected, connect c ;
      create-connectionState c : : -> status disconnected, connect c ;
      where
      o, d : device ;
      c : abstractConnection ;
End ConnectionState ;


;; Point de coupure OpComplete
 abstract Class OpCompletePointCut
Interface
    Use AbstractConnection
    Type opCompletePointCut
    Methods ;; Advices
      opComplete _ : abstractConnection ;
End OpCompletePointCut ;


;; Point de coupure OpDrop
 abstract Class OpDropPointCut
Interface
    Use AbstractConnection
    Type opDropPointCut
    Methods ;; Advices
      opDrop _ : abstractConnection ;
```

```
End OpDropPointCut;

;; Aspect Timing
Class Timing_Aspect;
Inherit OpCompletePointCut;
   Rename opCompletePointCut -> timing;
   Rename opDropPointCut -> timing;
Interface
   Use Timer, AbstractConnection;
   Object Timing;
   Methods
      getTimer _, _ : abstractConnection, timer;
Body
   Use Timer, AbstractConnection;
   Place
      timers _ : timer;
   Axioms
      getTimer c, t with t.getOwnConnection c'
            : : (c =c') => timers t -> timers t
      opComplete c with t.create-timer c i : : (i>=0) => -> timers t;
      opDrop c with (getTimer c t ..
            (t.getStartTime i .. t.setConnectionTime j))
            : : (i>=0) and (j>=0) => -> ;
      where
      i,j : integer
      o, d : device;
      c : abstractConnection;
End Timing_Aspect;

;; Classe Timer introduite par l'aspect Timing
Class Timer;
Interface
   Use Integer, AbstractConnection;
   Type timer;
   Methods
      getStartTime _, getConenctionTime _ : integer;
      getConnection _ : abstractConnection;
   Creation
      create-timer _ _ : abstractConnection, integer;
```

```
Body
   Use Integer, AbstractConnection;
   Place
      startTime _ , connectionTime _ : integer;
      ownConnection _ : abstractConnection;
   Axioms
      getStartTime i : : startTime i -> startTime i;
      getConnectionTime i : : connectionTime i -> connectionTime i;
      getConenction c : : ownConnection c -> ownConnection c;
      create-timer c i : : -> startTime i, ownConnection c;
      where
      i : integer, c : abstractConnection;
End Timer;


;; Aspect Billing
Class Billing_Aspect;
Inherit OpDropPointCut;
   Rename opDropPointCut -> billing;
Interface
   Use Customer, Bill;
   Object Billing;
   Methods
      getBill _, _ : customer, bill;
Body
   Use integer, Device, Customer, AbstractConnection, Timer, Timing_Aspect;
   Place
      bills _ : bill;
   Axioms
      getBill cl, b with b.getClient cl' : : (cl=cl') => bills b -> bills b
      opdrop c with (Timing.getTimer c t .. t.getConnectionTime i)..
               ((c.getOrigin o .. o.getOwner cl).. getBill cl, b) ..
               (b.getCharge a .. b.setCharge a+ i*2)
            : : (i>=0) => -> ;
      where
      i,j : integer
      o : device;
      c,c' : abstractConnection;
      cl,cl' : customer;
      t : timer;
```

```
End Billing_Aspect;


;; Classe Bill introduite par l'aspect Billing
Class Bill;
Interface
    Use Integer, Customer;
    Type bill;
    Methods
      getCharge _, setCharge _ , getTotalTime _, setTotalTime _ : integer;
      getClient _, : customer;
Body
    Use Integer, Customer;
    Place
      charge _ , TotalTime _ : integer;
      client _ : customer;
    Axioms
      getCharge i : : charge i -> ;
      setCharge i : : -> charge i;
      getClient c : : client c -> client c;
      where
      i : integer, c : customer;
End Bill;
```

Annexe II

Spécification Alloy de l'application de téléphonie

```
module telephony
open util/integer as I open operations

-----------------------------------------------------------------
-- Specification of the classes
-----------------------------------------------------------------
sig Connection {
     status: Status one -> Time,
     origin: Device,
     destination: Device one -> Time }

one sig Null {}

abstract sig Status {}

 one sig  connected extends Status {}
 one sig disconnected extends Status {}
 one sig interrupted extends Status {}
 one sig blocked extends Status {}
 one sig voicemail extends Status {}

sig Device {
     d_status: d_Status one -> Time,
     current: (Connection + Null ) one -> Time,
     num: PhoneNumber,
     owner: Customer }

sig PhoneNumber {}
one sig emergencyNum extends PhoneNumber {}

abstract sig d_Status {}

one sig idle extends d_Status {}
one sig busy extends d_Status{}
```

```
sig Customer {
     devices: set Device }

-----------------------------------------------------------------
--            Specification of the Aspects
-----------------------------------------------------------------
one sig Timing {
     timers: some Timer }

sig Timer {
     startTime, connectionTime: Int one -> Time,
     connection: Connection }

one sig Billing {
     bills: some Bill }

sig Bill {
     charge: Int one -> Time,
     client: Customer }

sig Stats {}

one sig Forwarding {
     forwardingList: set NewDestination }

sig NewDestination {
     num: PhoneNumber,
     newDest: PhoneNumber }

one sig VoiceMail {}

one sig Blocking {
     blockingList: set BlockedNum }

sig BlockedNum {
     num: PhoneNumber,
     blockedNums: PhoneNumber }

one sig Interrupting {}
```

```
-------------------------------------------------------------------
--      Specification of methods to be used as functions
-------------------------------------------------------------------
fun getTimer(t: Timing, c: Connection): Timer {
    { result:Timer | (result in (t.timers)
                     && result.connection=c)} }

fun getBill(b: Billing, c: Connection): Bill {
    { result: Bill | (result in (b.bills)
                     && result.client=c.origin.owner)} }

fun deviceOf(n: set PhoneNumber): Device{
     { d:Device | d.num in n } }

fun getForwardedNums(f: Forwarding): set PhoneNumber {
    { result: PhoneNumber | result in f.forwardingList.num} }

fun getNewDestination (f: Forwarding, n: PhoneNumber):  Device {
    { result: Device | result in deviceOf[f.forwardingList.newDest]
                     && (n = f.forwardingList.num)
                     && not (result.num =  n)} }

fun getBlockedList(b:Blocking, number:PhoneNumber): set BlockedNum {
   { result: BlockedNum | result in b.blockingList
                          && number in result.num} }

fun getBlockedNum(b:Blocking, number:PhoneNumber): set PhoneNumber {
   { result: PhoneNumber |
         result in getBlockedList[b,number].blockedNums
         && not (result = number)} }

-------------------------------------------------------------------
--        Specification of join points and advices
-------------------------------------------------------------------
sig Complete extends JP {
     self: Connection }
fact complete { all op:Complete |
     (op.self.status.(op.begin) = disconnected) &&
     (op.self.origin.d_status.(op.begin) = idle) &&
```

```
      (op.self.origin.current.(op.begin) = Null)
       =>
        ((op.self.status.(op.end) = connected) &&
        (op.self.origin.d_status.(op.end) = busy) &&
        (op.self.destination.(op.end).d_status.(op.end) = busy) &&
        (op.self.origin.current.(op.end) = op.self) &&
        (op.self.destination.(op.end).current.(op.end) = op.self)&&
        (op.error = False))
      else op.error = True
}

sig Drop extends JP {
     self: Connection }
fact drop { all op: Drop |
     (op.self.status.(op.begin) = connected) &&
     (op.self.origin.d_status.(op.begin) = busy) &&
     (op.self.destination.(op.begin).d_status.(op.begin) = busy) &&
     (op.self.origin.current.(op.begin) = op.self) &&
     (op.self.destination.(op.begin).current.(op.begin) = op.self)
     =>
       (op.self.status.(op.end) = disconnected) &&
       (op.self.origin.d_status.(op.end) = idle) &&
       (op.self.destination.(op.end).d_status.(op.end) = idle) &&
       (op.self.origin.current.(op.end) = Null) &&
       (op.self.destination.(op.end).current.(op.end) = Null) &&
       (op.error = False)
    else op.error = True }

sig OpCompleteTiming extends Advice {
     self: Timing,
     c: Connection }
fact opCompleteTiming { all op: OpCompleteTiming |
      neg[getTimer[op.self,op.c].startTime.(op.begin)]
        => pos[getTimer[op.self,op.c].startTime.(op.end)] &&
           op.error = False
       else op.error = True }

sig OpCompleteInterrupting extends Advice {
     self: Interrupting,
     c: Connection }
```

```
fact opCompleteInterrupting { all op: OpCompleteInterrupting |
    (op.c.destination.(op.begin).d_status.(op.begin) = busy) &&
    (op.c.destination.(op.begin).current.(op.begin) != Null)
     =>
      (op.c.destination.(op.end).d_status.(op.end) = idle) &&
      (op.c.origin.d_status.(op.end) =
       op.c.origin.d_status.(op.begin)) &&
      (op.c.status.(op.end) = op.c.status.(op.begin)) &&
      (op.c.destination.(op.end).current.(op.end).status.(op.end)=
          interrupted) &&
      (op.error = False)
     else op.error = True }

sig OpCompleteForwarding  extends Advice {
     self: Forwarding,
     c: Connection }
fact opCompleteForwarding { all op: OpCompleteForwarding |
     (op.c.destination.(op.begin).num in getForwardedNums[op.self])
      =>
      (op.c.destination.(op.end) =
       getNewDestination[op.self,op.c.destination.(op.begin).num])&&
      (op.error = False)
     else op.error = True }

sig OpCompleteVoiceMail extends Advice {
    self: VoiceMail,
    c: Connection }
fact opCompleteVoiceMail { all op: OpCompleteVoiceMail |
     (op.c.destination.(op.begin).d_status.(op.begin) = busy)
        => (op.c.status.(op.end) = voicemail) &&
           (op.error = False)
        else op.error = True }

sig OpCompleteBlocking extends Advice {
      self: Blocking,
      c: Connection }
fact opCompleteBlocking  { all op: OpCompleteBlocking |
     (op.c.origin.num in
         getBlockedNum[op.self,op.c.destination.(op.begin).num])
     =>
```

```
     (op.c.status.(op.end) = blocked) &&
     (op.c.destination.(op.end).current.(op.end).status.(op.end) =
      op.c.destination.(op.end).current.(op.end).status.(op.begin))&&
     (op.error = False)
     else op.error = True }

sig OpDropTiming extends Advice {
     self: Timing,
     c: Connection }
fact opDropTiming { all op: OpDropTiming |
      (pos[getTimer[op.self,op.c].startTime.(op.begin)] &&
       neg[getTimer[op.self,op.c].connectionTime.(op.begin)])
         =>  pos[getTimer[op.self,op.c].connectionTime.(op.end)] &&
             op.error = False
         else op.error = True }

sig OpDropBilling extends Advice {
     self: Billing,
     c:Connection }
fact opDropBilling { all op: OpDropBilling |
      pos[getTimer[Timing,op.c].connectionTime.(op.begin)]
        => ((int getBill[op.self,op.c].charge.(op.begin)) <
           (int getBill[op.self,op.c].charge.(op.end)) &&
            op.error = False)
        else op.error = True }

sig OpDropStats extends Advice {
     self: Stats,
     c: Connection }
fact opDropstats { all op: OpDropStats |
      pos[getTimer[Timing,op.c].connectionTime.(op.begin)]
        => neg[getTimer[Timing,op.c].connectionTime.(op.end)] &&
           op.error = False
        else op.error = True }

-------------------------------------------------------------------
--    Fact solving the frame problem
-------------------------------------------------------------------
fact unchanged { all t: Time - last | let t' = t.next |
      some e: JP + Advice { e.begin = t && e.end = t'
```

```
    (!status.t = status.t' =>
            (e in (Complete +  Drop + OpCompleteInterrupting +
             OpCompleteBlocking + OpCompleteVoiceMail) &&
             e.error = False))
    &&
    (!d_status.t = d_status.t' =>
            (e in (Complete + Drop + OpCompleteInterrupting) &&
             e.error = False))
    &&
    (!destination.t = destination.t' =>
            (e in OpCompleteForwarding && e.error = False))
    &&
    (!current.t = current.t' =>
            (e in Complete + Drop + OpCompleteInterrupting &&
             e.error = False))
    &&
    (!startTime.t = startTime.t' =>
            (e in OpCompleteTiming && e.error=False))
    &&
    (!connectionTime.t = connectionTime.t' =>
            (e in OpDropTiming + OpDropStats && e.error = False))
    &&
    (!charge.t = charge.t' =>
            (e in OpDropBilling && e.error = False)) } }

-------------------------------------------------------------
--        context passing at join points complete and drop
-------------------------------------------------------------
pred contextpassingAtComplete (jp:Complete) {
    jp.self = OpCompleteTiming.c &&
    jp.self = OpCompleteInterrupting.c &&
    jp.self = OpCompleteForwarding.c  &&
    jp.self = OpCompleteVoiceMail.c &&
    jp.self = OpCompleteBlocking.c }

pred contextpassingAtDrop (jp:Drop) {
    jp.self = OpDropTiming.c &&
    jp.self = OpDropBilling.c &&
    jp.self = OpDropStats.c }
```

```
-------------------------------------------------------------------
--     Composition of advices at join point complete
-------------------------------------------------------------------
sig  Composition1 extends NDComposition {}

fact composition1 { all s: Composition1 |
       one s.comp1 && one s.comp2 &&
       s.comp1 = OpCompleteInterrupting &&
       s.comp2 = Composition11 }

sig  Composition11 extends NDComposition {}

fact composition11 { all s: Composition11 |
       one s.comp1 && one s.comp2 &&
       s.comp1 = OpCompleteForwarding &&
       s.comp2 = Composition12 }

sig  Composition12 extends NDComposition {}

fact composition12{ all s: Composition12 |
       one s.comp1 && one s.comp2 &&
       s.comp1 = OpCompleteVoiceMail &&
       s.comp2 = Composition13 }

sig  Composition13 extends NDComposition {}

fact composition13 {all s:Composition13 |
       one s.comp1 && no s.comp2 &&
       s.comp1 = OpCompleteBlocking }

sig Composition2 extends NDComposition{}

fact composition2 { all s:Composition2 |
      (one s.comp1 && no s.comp2) &&
       s.comp1 = OpCompleteTiming }

-------------------------------------------------------------------
--   Weaving at join point complete
-------------------------------------------------------------------
sig WeavingAtComplete extends Weaving{}
```

```
fact { all w: WeavingAtComplete |
       contextpassingAtComplete[w.jp] &&
       (one w.jp) &&
       (one w.beforeAdvice) &&
       (one w.afterAdvice) &&
       (w.jp = Complete) &&
       (w.beforeAdvice = Composition1) &&
       (w.afterAdvice = Composition2) }

-----------------------------------------------------------------
--   Composition of advices at join point drop
-----------------------------------------------------------------
sig Composition4 extends NDComposition {}

fact composition4 {all s: Composition4 |
      (one s.comp1 && one s.comp2) &&
       s.comp1 = OpDropStats &&
       s.comp2 = Composition5 }

sig Composition5 extends SeqComposition {}

fact composition5 { all s: Composition5 |
    (one s.comp1 && one s.comp2) &&
     s.comp1 = OpDropTiming &&
     s.comp2 = Composition51 }

sig Composition51 extends SeqComposition {}

fact composition51 { all s: Composition51 |
     (one s.comp1 && no s.comp2) &&
      s.comp1 = OpDropBilling }

-----------------------------------------------------------------
--   Weaving at join point drop .
-----------------------------------------------------------------
sig WeavingAtDrop extends Weaving{}

fact { all w: WeavingAtDrop |
         contextpassingAtDrop[w.jp]&&
```

```
        (one w.jp) &&
        (one w.afterAdvice) &&
        (no w.beforeAdvice) &&
        (w.jp = Drop) &&
        (w.afterAdvice = Composition4) }

-----------------------------------------------------------------
--      Local verification at complete
-----------------------------------------------------------------
pred initialCondAtComplete(w: Weaving) {
  Complete.self.status.(w.begin) = disconnected &&
  Complete.self.origin.d_status.(w.begin) = idle &&
  Complete.self.origin.current.(w.begin) = Null &&
  OpCompleteForwarding.c.destination.(w.begin).num in
  getForwardedNums[Forwarding] &&
  OpCompleteInterrupting.c.destination.(w.begin).d_status.(w.begin)=
  busy &&
  OpCompleteInterrupting.c.destination.(w.begin).current.(w.begin)!=
  Null &&
  OpCompleteVoiceMail.c.destination.(w.begin).d_status.(w.begin)=
  busy &&
  OpCompleteBlocking.c.origin.num =
  getBlockedNum[Blocking,OpCompleteBlocking.c.destination.(w.begin).num]
  &&
  neg[getTimer[Timing, OpCompleteTiming.c].startTime.(w.begin)] }

pred finalCondAtComplete(w: Weaving) {
  Complete.self.status.(w.end) = connected &&
  Complete.self.origin.d_status.(w.end) = busy &&
  Complete.self.destination.(w.end).d_status.(w.end) = busy &&
  Complete.self.origin.current.(w.end) = Complete.self &&
  Complete.self.destination.(w.end).current.(w.end) = Complete.self }

pred noErrorAtComplete {
  Complete.error = False &&
  OpCompleteInterrupting.error = False &&
  OpCompleteForwarding.error = False &&
  OpCompleteVoiceMail.error = False &&
  OpCompleteBlocking.error = False &&
  OpCompleteTiming.error = False }
```

```
assert localVerifAtComplete {
        all w:  WeavingAtComplete | initialCondAtComplete[w]
            => finalCondAtComplete[w] && (noErrorAtComplete[]) }

check localVerifAtComplete for 3 but 12 Operation, 7 Time, 1
Weaving, 1 JP, 1 NewDestination, 1 BlockedNum, 1 Customer, 1
BlockedNum, 1 NewDestination, 2 Connection, 4 Status

-----------------------------------------------------------------
--      Local verification at drop
-----------------------------------------------------------------
pred initialCondAtDrop(w: Weaving){
    Drop.self.status.(w.begin)= connected &&
    Drop.self.origin.d_status.(w.begin) = busy &&
    Drop.self.destination.(w.begin).d_status.(w.begin) = busy &&
    Drop.self.origin.current.(w.begin) = Drop.self &&
    Drop.self.destination.(w.begin).current.(w.begin) = Drop.self &&
    pos[getTimer[Timing,Drop.self].startTime.(w.begin)]) &&
    neg[getTimer[Timing,Drop.self].connectionTime.(w.begin)] }

pred finalCondAtDrop(w: Weaving){
    Drop.self.status.(w.end) = disconnected &&
    Drop.self.origin.d_status.(w.end) = idle &&
    Drop.self.destination.(w.end).d_status.(w.end) = idle &&
    Drop.self.origin.current.(w.end) = Null &&
    Drop.self.destination.(w.end).current.(w.end) = Null }

pred noErrorAtDrop(){
    Drop.error = False &&
    OpDropTiming.error = False &&
    OpDropBilling.error = False &&
    OpDropStats.error = False }

assert localVerifAtDrop {
       all w:  WeavingAtDrop | initialCondAtDrop[w]
           => finalCondAtDrop[w] && noErrorAtDrop[] }

check localVerifAtDrop for 3 but 9 Operation, 5 Time, 1 Weaving, 1
JP, 1 OpDropTiming, 1 OpDropBilling, 1 OpDropStats
```

```
-------------------------------------------------------------------
--      Verification of invariants
-------------------------------------------------------------------
pred correctExecution(w: Weaving){
     (w in WeavingAtComplete => initialCondAtComplete[w]) &&
     (w in WeavingAtDrop =>  initialCondAtDrop[w]) &&
     (Complete.error = False ||
        (OpCompleteBlocking.error = False &&
         OpCompleteVoiceMail.error = False)) &&
     OpCompleteInterrupting.error = False &&
     OpCompleteForwarding.error = False &&
     OpCompleteTiming.error = False }

-------------------------------------------------------------------
pred Inv1(t: Time){
      all c: Connection | c.destination.t.num = emergencyNum
          => c.status.t != interrupted }

assert verifInv1 {
    all w: WeavingAtComplete + WeavingAtDrop | correctExecution[w]
      => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv1[t] }

check verifInv1 for 3 but 12 Operation, 7 Time, 1 Weaving, 1 JP

-------------------------------------------------------------------
pred Inv2(t: Time){
          all c: Connection | (c.origin != c.destination.t) }

assert verifInv2 {
    all w: WeavingAtComplete + WeavingAtDrop | correctExecution[w]
      => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv2[t] }

check verifInv2 for 3 but 12 Operation, 7 Time, 1 Weaving, 1 JP, 2
Device

-------------------------------------------------------------------
pred Inv3(t: Time) {
    all c: Connection | (c.status.t= voicemail && c= Complete.self)
         => c.destination.t.d_status.t = busy }
```

```
assert verifInv3 {
    all w: WeavingAtComplete + WeavingAtDrop | correctExecution[w]
      => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv3[t] }

check  verifInv3 for 3 but 12 Operation, 7 Time, 1 Weaving, 1 JP

-----------------------------------------------------------------
pred Inv4(t: Time) {
      all c:Connection | pos[getTimer[Timing, c].startTime.(t)]
        => c.status.t = connected }

assert verifInv4 {
    all w:  WeavingAtComplete + WeavingAtDrop| correctExecution[w]
      => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv4[t] }

check  verifInv4 for 3 but 12 Operation, 7 Time, 1 Weaving, 1 JP
-----------------------------------------------------------------
pred Inv5(t: Time) {
   all c: Connection | pos[getTimer[Timing, c].startTime.(t)]
        => pos[getTimer[Timing, c].connectionTime.(t)] }

assert verifInv5 {
    all w:  WeavingAtDrop | correctExecution[w]
     => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv5[t] }

check verifInv5 for 3 but 9 Operation, 5 Time, 1 Weaving, 1 JP

-----------------------------------------------------------------
pred Inv6(t: Time) {
   all c: Connection | c.status.t = disconnected
     => pos[getBill[Billing, c].charge.(t)] }

assert verifInv6 {
   all w: WeavingAtDrop | correctExecution[w]
    => all t:Time | lte[w.begin,t] && lte[t,w.end] && Inv6[t] }

check  verifInv6 for 3 but 9 Operation, 5 Time, 1 Weaving, 1 JP

-----------------------------------------------------------------
```

```
pred Inv7(t: Time) {
    all c: Connection | pos[getTimer[Timing, c].connectionTime.(t)]
       => pos[getBill[Billing, c].charge.(t)] }

assert verifInv7 {
    all w: WeavingAtDrop | correctExecution[w]
      => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv7[t] }

check verifInv7 for 3 but 9 Operation, 5 Time, 1 Weaving, 1 JP

-----------------------------------------------------------------
pred Inv8(t: Time) {
     all  b: BlockedNum |  all c: Connection |
      (b in Blocking.blockingList) && (c= Complete.self) &&
      (c.destination.t.num= b.num) && (c.origin.num= b.blockedNums)
        => (c.status.t != voicemail) }

assert verifInv8 {
     all w: WeavingAtComplete + WeavingAtDrop | correctExecution[w]
       => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv8[t] }

check  verifInv8 for 3 but 12 Operation, 7 Time, 1 Weaving, 1 JP

-----------------------------------------------------------------
pred Inv9(t: Time) {
     all  b:BlockedNum |  all c: Connection |
       (b in Blocking.blockingList) && (c= Complete.self)&&
       (c.destination.t.num= b.num) && (c.origin.num= b.blockedNums)
         =>   (c.status.t != connected) }

assert  verifInv9 {
     all w:  WeavingAtComplete + WeavingAtDrop | correctExecution[w]
        => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv9[t] }

check verifInv9 for 4 but 12 Operation, 7 Time, 1 Weaving, 1 JP, 2
Connection

-----------------------------------------------------------------
pred Inv10(t: Time) {
     all  d: NewDestination |  all c: Connection |
```

```
      (d in Forwarding.forwardingList) && (c= Complete.self) &&
      (c.destination.t.num = d.num) => (c.status.t != connected)) }

assert  verifInv10 {
     all w:  WeavingAtComplete + WeavingAtDrop | correctExecution[w]
       => all t: Time | lte[w.begin,t] && lte[t,w.end] && Inv10[t] }

check  verifInv10 for 4 but 12 Operation, 7 Time, 1 Weaving, 1 JP
```

```
-------------------------------------------------------------------
Abstract signatures describing composition and weaving operations
-------------------------------------------------------------------
module operations

open util/ordering[Time] as timeorder

sig Time {}

abstract sig Error{}

one sig True extends Error {}
one sig False extends Error {}

-------------------------------------------------------------------
abstract sig Operation {
          begin, end: Time,
          error: Error }

abstract sig JP, Advice extends Operation {}

fact indivisible { all op: JP + Advice | op.end = op.begin.next }

abstract sig Composition extends Operation {
               comp1:   Composition + Advice,
               comp2:  lone Composition }

-------------------------------------------------------------------
// describes sequential composition
abstract sig SeqComposition extends Composition{}

fact seqComposition { all s: SeqComposition |
          (s.begin = s.comp1.begin) &&
          (no s.comp2 => s.end = s.comp1.end
                      else (s.end = s.comp2.end &&
                            lte[s.comp1.end,s.comp2.begin]) ) }

// describes non deterministic  composition
abstract sig NDComposition extends Composition {}
```

```
fact ndComposition { all nd: NDComposition |
          no nd.comp2 => (nd.begin = nd.comp1.begin &&
                          nd.end = nd.comp1.end)
          else ((nd.begin = nd.comp1.begin &&
                 nd.end = nd.comp2.end &&
                 lte[nd.comp1.end,nd.comp2.begin])
                ||
                (nd.begin = nd.comp2.begin &&
                 nd.end = nd.comp1.end &&
                 lte[nd.comp2.end,nd.comp1.begin]) ) }

-----------------------------------------------------------------
abstract sig Weaving extends Operation{
                 jp:  JP,
                 beforeAdvice: lone Composition,
                 afterAdvice: lone Composition }


fact mustWeave { all w: Weaving |
                 w.beforeAdvice != none || w.afterAdvice != none }

fact weavingBefore { all w: Weaving |
          w.afterAdvice=none => (w.begin = w.beforeAdvice.begin &&
                                 w.end = w.jp.end  &&
                                 w.beforeAdvice.end = w.jp.begin) }

fact weavingAfter { all w: Weaving |
          w.beforeAdvice = none => (w.begin = w.jp.begin &&
                                    w.end = w.afterAdvice.end &&
                                    w.jp.end = w.afterAdvice.begin) }

fact weavingBeforeAfter { all w: Weaving |
          w.beforeAdvice != none && w.afterAdvice !=none
                       => (w.begin = w.beforeAdvice.begin &&
                           w.end = w.afterAdvice.end &&
                           w.beforeAdvice.end = w.jp.begin &&
                           w.jp.end = w.afterAdvice.begin) }
```

```
-------------------------------------------------------------------
--              module ordering
-------------------------------------------------------------------
module util/ordering[elem] /* Creates a single linear ordering over
the atoms in elem. It also
   constrains all the atoms to exist that are permitted by the
   scope on elem. That is, if the scope on a signature S is 5,
   opening util/ordering[S] will force S to have 5 elements and
   create a linear ordering over those five elements.
   The predicates and functions below provide access to properties
   of the linear ordering, such as which element is first in the
   ordering, or whether a given element precedes another.
   You cannotcreate multiple linear orderings over the same
   signature with this model. If you that functionality, try
   using the util/sequence module instead.
 * Technical comment:
   An important constraint: elem must contain all atoms permitted
   by the scope. This is to let the analyzer optimize the analysis
   by setting all fields of each instantiation of Ord to predefined
   values: e.g. by setting 'last' to the highest atom of elem and by
   setting 'next' to {<T0,T1>,<T1,T2>,...<Tn-1,Tn>}, where n is the
   scope of elem. Without this constraint, it might not be true that
   Ord.last is a subset of elem, and that the domain and range of
   Ord.next lie inside elem.
 * author: Ilya Shlyakhter
 * revisions: Daniel jackson
 */
one sig Ord {
```

```
   First, Last: elem,
   Next, Prev: elem -> lone elem}
{
  // constraints that actually define the total order
  Prev = ~Next
  one First
  one Last
  no First.Prev
  no Last.Next
  (
   // either elem has exactly one atom,
   // which has no predecessor or successor...
   (one elem && no elem.Prev && no elem.Next) ||
   // or...
    (all e: elem | {
     //...each element (except the first) has one predecessor, and...
     (e = First || one e.Prev)
     //...each element (except the last) has one successor, and...
     (e = Last || one e.Next)
     //...there are no cycles
     (e !in e.^Next)
    })
  )
  // all elements of elem are totally ordered
  elem in First.*Next
}
// first
fun first: elem { Ord.First }
```

```
// last
fun last: elem { Ord.Last }
// return the predecessor of e, or empty set if e is the first element
fun prev [e: elem]: lone elem { e.(Ord.Prev) }
// return the successor of e, or empty set of e is the last element
fun next [e: elem]: lone elem { e.(Ord.Next) }
// return elements prior to e in the ordering
fun prevs [e: elem]: set elem { e.^(Ord.Prev) }
// return elements following e in the ordering
fun nexts [e: elem]: set elem { e.^(Ord.Next) }
// e1 is less than e2 in the ordering
pred lt [e1, e2: elem] { e1 in prevs[e2] }
// e1 is greater than e2 in the ordering
pred gt [e1, e2: elem] { e1 in nexts[e2] }
// e1 is less than or equal to e2 in the ordering
pred lte [e1, e2: elem] { e1=e2 || lt [e1,e2] }
// e1 is greater than or equal to e2 in the ordering
pred gte [e1, e2: elem] { e1=e2 || gt [e1,e2] }
// returns the larger of the two elements in the ordering
fun larger [e1, e2: elem]: elem { lt[e1,e2] => e2 else e1 }
// returns the smaller of the two elements in the ordering
fun smaller [e1, e2: elem]: elem { lt[e1,e2] => e1 else e2 }
// returns the largest element in es  or the empty set if es is empty
fun max [es: set elem]: lone elem { es - es.^(Ord.Prev) }
// returns the smallest element in es or the empty set if es is empty
fun min [es: set elem]: lone elem { es - es.^(Ord.Next) }
```