N° d'ordre: 2449

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Alain LE GUENNEC

Équipe d'accueil : IRISA/PAMPA
École Doctorale : MATISSE
Composante universitaire : IFSIC

Titre de la thèse :

Génie Logiciel et Méthodes Formelles avec UML
Spécification, Validation et Génération de tests

Soutenue le 29 juin 2001 devant la commission d'examen

M. :	Albert	BENVÉNISTE	Président
MM. :	Jean	BÉZIVIN	Rapporteurs
	Yassine	LAKHNECH	
MM. :	Jean-Marc	JÉZÉQUEL	Examinateurs
	Claude	JARD	

Remerciements

Le travail de recherche que représente une thèse n'est pas quelque chose de linéaire. La rédaction du manuscrit, qui en est une part conséquente, n'échappe pas à cette règle. Aussi cette section qui semble préfigurer l'ensemble de ce travail, est-elle en fait la dernière partie à avoir été rédigée, et vient ainsi parachever plus de trois années de travail. La thèse, activité passionnante s'il en est, n'en reste pas moins une aventure parfois difficile et semée d'embûches. Il m'aurait certainement été impossible de la mener à son terme sans l'aide de plusieurs personnes que je tiens à remercier ici.

Tout d'abord, je voudrais remercier mes directeur et encadrant Claude Jard et Jean-Marc Jézéquel, qui m'ont fait confiance pendant ces quelques années passées à l'IRISA, et qui ont su m'encadrer sans pour autant trop me diriger. Cela n'a sans doute pas toujours été très facile, surtout lorsque la tête chercheuse se transforme en tête de mule, au risque parfois de se disperser au lieu de se focaliser sur les objectifs fixés. J'ai énormément apprécié la liberté qui m'a toujours été accordée afin que je puisse explorer mes propres pistes et reformuler mes objectifs à ma façon.

Je voudrais aussi remercier Albert Benvéniste, qui m'a fait l'honneur de présider le jury, ainsi que Jean Bézivin et Yassine Laknech qui m'ont fait l'honneur d'accepter d'évaluer mon travail de thèse. Leurs remarques furent grandement appréciées.

Le manuscrit ne serait pas ce qu'il est aujourd'hui sans les nombreuses remarques que m'ont également apportées les personnes ayant accepté de relire les versions préliminaires. Merci donc à François, Gerson et Loïc pour leur aide précieuse. Je voudrais aussi décerner une mention spéciale à deux relecteurs particulièrement étonnants : À Solofo tout d'abord, pour l'incroyable énergie qu'il a déployée pour m'apporter nombre de corrections et de suggestions. À ma mère enfin, qui à ma plus grande surprise (et à ma plus grande joie), a lu l'intégralité du manuscrit et corrigé un nombre conséquent de petites erreurs. Merci à tous.

Je souhaiterais aussi remercier tous ceux qui ont contribué à rendre ces trois années à l'IRISA intéressantes tant sur le plan scientifique que sur le plan humain. L'ambiance au sein du projet PAMPA fut fort sympatique, et le travail d'équipe autour d'UML très stimulant. Comme le furent aussi les nombreuses réunions dominicales du club de roller pampaïen, club comportant en outre quelques pâtissiers amateurs qui apportèrent régulièrement de quoi se remettre d'un dur entrainement sur des engins à roulettes à l'équilibre pour le moins précaire. Merci encore à François, Gerson, Loïc, Séverine, Solofo, Wai Ming et les autres pour tout ça. J'espère n'avoir oublié personne.

Enfin, il est de coutume de clore les remerciements par un petit mot tout spécialement dédicacé à une personne chère au cœur du rédacteur. Parmi les découvertes heureuses faites par un chercheur pendant sa quête, il en est cependant qu'il peut parfois préférer garder secrètes. Permettez donc que je déroge à cette règle.

À Toulouse, le 30 octobre 2001.

Table des matières

Chapitre 1

Développer des Logiciels Fiables avec UML

1.1 Introduction

Qui n'a pas déjà eu maille à partir avec un système informatique récalcitrant ? Si certaines défaillances et autres "plantages" peuvent parfois amuser lorsque cela n'a pas de conséquences majeures, d'autres conduisent à des pannes, des pertes de données ou de temps, incidents qui peuvent éventuellement être catastrophiques. L'intérêt d'assurer une fiabilité maximale de ces systèmes est donc évident. Leur omniprésence, leur complexification croissante, et leurs interconnexions toujours plus nombreuses rendent la société moderne chaque jour davantage tributaire de ces systèmes. Leur fiabilité devient donc à la fois un enjeu de plus en plus important et un problème de plus en plus ardu à résoudre.

Bien sûr, la fiabilité des systèmes (informatiques ou non) n'est pas un problème récent. Le domaine des transports, de l'aéronautique, de la banque, pour n'en citer que quelques uns, ont déjà une longue tradition de la recherche de la fiabilité, eu égard à la criticité des systèmes qu'ils déploient. Ces domaines sont ceux où les méthodes de développement sont les plus rigoureuses, faisant parfois appel aux techniques formelles que le reste de l'industrie informatique juge souvent inutilement compliquées et lourdes à mettre en œuvre.

Mais justement, l'importance de ce pan de l'industrie informatique n'a cessé de croître du fait de l'essor des nouvelles technologies de l'informatique et des télécommunications. Leur criticité, et la responsabilité que cela implique, augmente parallèlement. Des défaillances qui n'avaient naguère que des conséquences localisées et sans grande gravité risquent d'entrainer des pannes significatives de par la grande "connexité" des systèmes qui les rend de plus en plus inter-dépendants. On assiste donc à un intérêt grandissant pour les méthodes, langages et techniques permettant d'améliorer la qualité des systèmes logiciels développés en général.

1.2 UML ou la lente émergence d'un consensus

L'Unified Modeling Language (UML) est un langage pour documenter et spécifier graphiquement tous les aspects d'un système logiciel. Comme son nom l'indique, ce langage est le résultat d'une longue maturation. Il est la fusion de plusieurs langages de modélisation, issus notamment de la méthode de Grady Booch [14], de la méthode OOSE [61] de Ivar Jacobson, et de la méthode OMT [90] de James Rumbaugh, pour ne citer que les principales. On trouvera un historique complet de la génèse d'UML sur le site officiel de l'Object Management Group [1] et dans [39]. D'autres méthodes (comme Catalysis[32] ou celle Coad [25, 26]) ont aussi influencé le développement d'UML. L'ambition d'UML est de rassembler en une seule notation les meilleures caractéristiques des différents langages de modélisation à objets. Cette unification a aussi pour effet de donner une masse critique à UML. En tant que standard de l'OMG, et en tant que successeur de notations déjà bien implantées, UML jouit d'une popularité sans précédent à la fois dans l'industrie du logiciel et (par contre coup ?) dans le monde académique. Cette popularité est renforcée par un nombre important d'outils, les Ateliers de Génie Logiciel (AGL), qui permettent de mettre en œuvre la notation.

UML se veut ainsi pour l'architecture des logiciels ce que les plans sont à l'architecture classique : Un médium, sous forme de modèle graphique, servant à coordonner les différents acteurs concourant à la réalisation du système, et à garantir la cohérence et la qualité de sa conception.

1.3 Techniques formelles et UML : Une approche intégrée

La thèse que nous soutenons est qu'il est possible d'adopter une démarche de développement à la fois rigoureuse et pragmatique, utilisant la notation UML comme pivot, et reposant sur des techniques formelles éprouvées. Grâce à sa popularité grandissante, UML devient quasiment incontournable pour les développements logiciels industriels. Nous proposons de compléter cette notation graphique par les bases formelles et les outils complémentaires, afin de rendre enfin accessibles aux développeurs de logiciels des techniques formelles permettant d'améliorer la qualité de leurs réalisations. Plutôt que d'attendre que les développeurs viennent aux techniques formelles, il nous semble plus réaliste et efficace d'amener doucement les techniques formelles aux développeurs, sans brusquer ces derniers et surtout sans révolutionner leurs habitudes et leurs environnements de développement.

1. http://cgi.omg.org/news/pr97/umlprimer.html

1.3.1 UML au delà de simples dessins

Pour atteindre cet objectif ambitieux, il faut donner aux spécifications UML une place nouvelle. Elles doivent dépasser le statut de simples dessins que l'on trace sur un coin de table, et que l'on s'empresse de jeter lorsque l'on passe à la phase de "codage". Elles deviennent le formalisme central d'un environnement de développement intégré.

Il faut donc s'assurer que les différents diagrammes qui composent la notation UML représentent diverses vues cohérentes d'un seul et même système . Pour cela, il faut leur donner un sens, c'est-à-dire une sémantique. À cette condition, l'Atelier de Génie Logiciel (AGL) reposant sur UML deviendrait la pièce maîtresse de l'environnement de développement. Il permettrait aux développeurs de construire un *modèle* du système en se plaçant selon plusieurs points de vue (correspondant aux diagrammes UML) tout en maintenant *par construction* la cohérence entre ces vues.

L'Atelier de Génie Logiciel devient ainsi le dépositaire des artéfacts de modélisation. C'est aussi et surtout le pivot autour duquel gravitent les activités se basant sur ce modèle. C'est donc par son biais que les techniques formelles doivent être mises à la disposition des concepteurs. Ces techniques formelles doivent s'intégrer dans l'environnement de développement. Si le concepteur doit réaliser un autre modèle de son application dans le seul but de pouvoir utiliser un outil de vérification, il y a de fortes chances que ce modèle parallèle ne verra jamais le jour, ou alors ne sera pas maintenu en cohérence avec le modèle UML original dont il est dérivé lorsque celui-ci évoluera. Au contraire, si ce même outil de vérification est capable de manipuler directement (ou via une transformation ne nécessitant pas d'intervention manuelle) le modèle UML, et qu'en plus il est parfaitement intégré à l'Atelier de Génie Logiciel UML que le concepteur a l'habitude d'utiliser, alors le concepteur sera enclin à en tirer profit.

1.3.2 UMLAUT : Une chaîne complète d'outils intégrés

L'Atelier de Génie Logiciel UMLAUT [67] développé à l'IRISA, intègre les travaux présentés dans cette thèse, ainsi que les travaux d'autres personnes de l'équipe PAMPA, dont notamment [52]. UMLAUT est disponible gratuitement en téléchargement depuis le site WWW de l'IRISA [2].

La figure 1.1 illustre la manière dont les techniques formelles sont mises en œuvre au sein du processus de développement. La section 1.3.3 qui suit décrit ces techniques de vérification et de validation plus en détail. L'architecture de l'outil UMLAUT sera décrite plus amplement dans la section 5.1.

2. Pour plus d'informations, voir http://www.irisa.fr/UMLAUT/

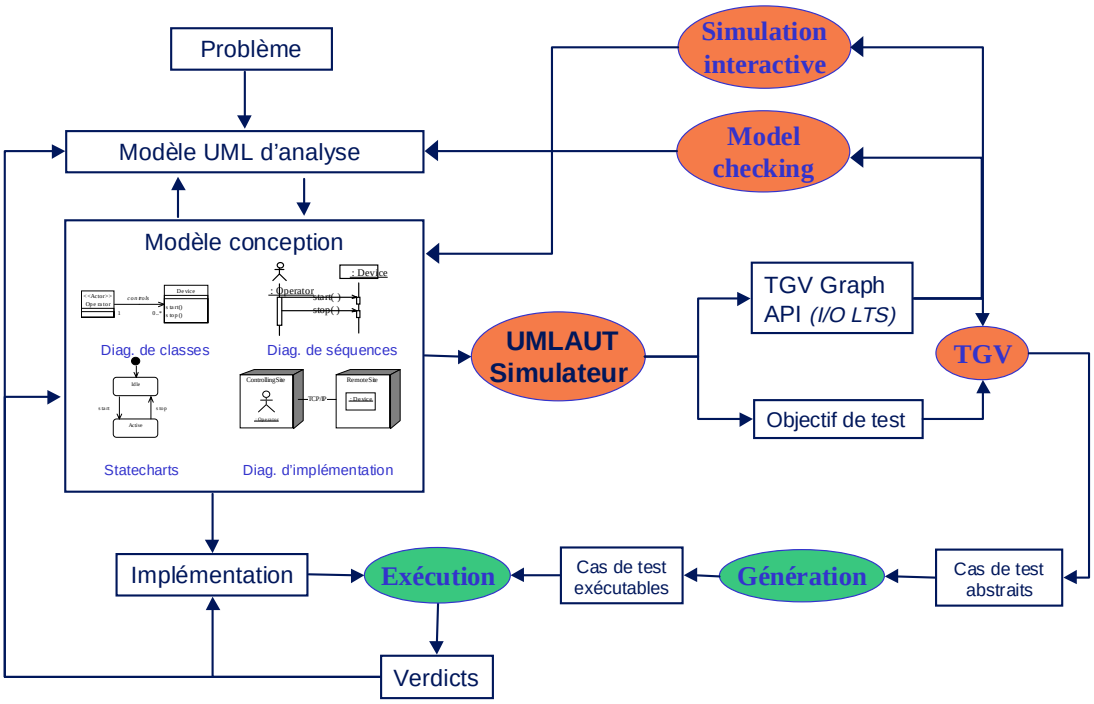


FIG. 1.1 – *Techniques formelles dans le cycle de développement*

1.3.3 Vérification et Validation

Commençons au préalable par quelques définitions [12, 13] :

Vérification La vérification est l'ensemble des actions de revue, inspection, test, preuve automatique, ou autres techniques appropriées permettant d'établir et de documenter la conformité des artéfacts du développement vis-à-vis de critères préétablis. La vérification répond à la question "*is the system being built right?*"[3].

Validation La validation consiste à évaluer l'adéquation du système développé vis-à-vis des besoins exprimés par ses futurs utilisateurs. La validation répond à la question "*is the right system being built?*".

Disposer d'un modèle cohérent, et si possible complet, du système que l'on souhaite développer permet de raisonner sur ce système.

La vérification peut être formelle ou non. Nous nous intéresserons uniquement à la vérification formelle basée sur la technique de dite de "*model-checking*", qui consiste à explorer *un modèle de l'espace des états* (voir sections 3.3 et 4.6) de la spécification afin d'en vérifier certaines propriétés, comme l'absence de blocage, ou encore la possibilité d'atteindre ou non certains états (voir section 3.3.1.1).

3. Voir la FAQ V&V de la NASA http://research.ivv.nasa.gov/docs/ivvfaq/index.html.

La spécification assure aussi la *traçabilité* entre les différentes phases, en liant l'expression informelle des besoins aux artéfacts représentant leur mise en œuvre, à travers les différents niveaux d'abstraction. Il devient ainsi possible de comprendre l'impact d'un besoin sur la structure et sur le comportement du système. C'est aussi un bon moyen pour vérifier qu'aucun des besoins exprimés par les futurs utilisateurs du système n'ont été oubliés.

La validation repose essentiellement sur le test fonctionnel. Une suite de tests adéquats permettra de dire si une implantation est *conforme* à la spécification (voir section 3.3.4). La traçabilité permet notamment d'assurer une certaine couverture de la spécification : Chaque fonctionnalité du système devra être testée, au moins partiellement, en liant au moins un cas de test à l'expression du besoin correspondant.

1.3.4 Pourquoi plutôt UML qu'une autre Technique de Description Formelle ?

Il existe déjà plusieurs formalismes ayant des bases formelles permettant de spécifier des systèmes et d'appliquer des techniques comme la vérification, la simulation, ou la synthèse de tests. L'ingénierie des protocoles a donné naissance à plusieurs techniques de description formelles normalisées que sont Estelle [58], SDL [21], et LOTOS [57]. Ces langages se cantonnent pour l'essentiel à leur domaine d'origine, et ne sont malheureusement pas facilement utilisables pour des développements logiciels plus "classiques", et pour lesquels UML est déjà largement employé.

D'autres langages comme SPIN [53, 54], B [4, 3], Z [92], et VDM [88, 11] permettent aussi de spécifier des systèmes afin de leur appliquer des techniques formelles. Il existe aussi des extensions "objets" de Z [5, 18, 71] et VDM [33]. Des outils associés existent pour tous ces langages.

Nous reparlerons de ces formalismes en comparant certains aspects de leurs sémantiques respectives avec celle que nous proposons pour UML au chapitre 4.

De fait, il est souvent reproché à UML de ne pas avoir de base formelle solide. C'est pourquoi il peut être tentant de traduire UML vers un autre langage de spécification de haut niveau comme ceux évoqués ci-dessus. Cela procurerait ainsi indirectement les fondements formels faisant défaut à UML. Plusieurs articles publiés récemment se font l'écho de cette approche traductionnelle (voir section 3.2.2), et proposent des traductions de sous-ensembles plus ou moins complets de UML vers divers langages de spécification possédant une base formelle plus solide que UML.

Cependant, si l'absence de rigueur formelle était flagrante aux débuts d'UML, c'est de moins en moins le cas. Des travaux complémentaires à ceux cités au paragraphe précédent ont pour but d'étayer les bases formelles d'UML sans pour autant adopter une approche traductionnelle. Plusieurs approches "méta" (voir section 3.2.4) sont actuellement développées : Elles visent toutes à modéliser le domaine sémantique de UML en UML. La plus significative est sans doute l'approche adoptée par le groupe

de travail “Action Semantics” [1] dont le but est de définir *la* sémantique officielle de UML [81].

Surtout, les approches traductionnelles se heurtent à un écueil majeur : Elles introduisent immanquablement un écart important entre deux formalismes. Traduire depuis UML vers un autre formalisme impose généralement des restrictions fortes sur les constructions UML qu'il est possible d'utiliser. Le développeur utilisant UML risque de trouver ces restrictions inacceptables, ce qui remettrait en cause toute la méthode de développement.

Une possibilité que fort peu de développeurs se résoudraient à abandonner est celle d'introduire des fragments de code exécutable au sein d'une spécification, parce qu'une spécification formelle équivalente serait trop “lourde” à mettre en œuvre. Un outil prétendant être utilisé pour des développements “classiques” se doit de posséder cette faculté. Nous expliquons dans la section 4.7 comment l'outil UMLAUT réalisé au cours de notre thèse permet cette inclusion de code natif dans les spécifications UML.

1.4 Plan du document

Afin d'illustrer les notions que nous souhaitons mettre en œuvre, le chapitre 2 qui va suivre est consacré à la spécification en UML d'un système de gestion de réunions virtuelles. Ce chapitre montre les différentes possibilité de la notation UML et montre comment les différents types de diagrammes s'intègrent pour former un tout cohérent.

Le chapitre 3 décrit le bénéfice qu'il est possible de tirer de l'utilisation de techniques formelles appliquées à une spécification UML telle que celle présentée au chapitre 2.

Pour pouvoir appliquer ces techniques formelles, nous avons besoin d'une sémantique, qui sera donnée sous forme de système de transitions. Le chapitre 4 explique ainsi le modèle sémantique sous-jacent aux techniques et outils présentés dans le chapitre 3 et la manière dont le système de transitions est construit.

Enfin, l'outil UMLAUT est présenté au chapitre 5, section 5.1. Nous décrivons ensuite succintement deux cas d'étude industriels ayant utilisé notre outil. Le premier est une modélisation d'un système de contrôle de trafic aérien, réalisée dans le cadre du projet RNRT OURAL, et décrite en section 5.3. Le second est une modélisation d'un porte-monnaie électronique, réalisée par la société Gemplus et décrite en section 5.4.

Nous concluons en faisant un bilan de nos contributions et en proposant quelques perspectives, au chapitre 6.

Les travaux présentés dans cette thèse ont fait l'objet de plusieurs publications : [63, 66, 45, 62, 67, 95, 49, 96].

Chapitre 2

Spécification avec UML

2.1 UML par l'exemple

La notation UML est d'une très grande richesse. Elle permet de couvrir à peu près toutes les phases du développement :

- les besoins des utilisateurs du futur système, exprimés à l'aide de cas d'utilisation (section 2.2)
- la spécification complète du système, sous forme de diagrammes couvrant les parties statiques (section 2.3) et dynamiques (section 2.4) du système.
- la conception détaillée, jusqu'à un niveau très proche du code en langage object de l'implantation.
- les suites de tests permettant de s'assurer qu'une implémentation candidate est effectivement conforme à la spécification élaborée en phase amont, sous forme de diagrammes de séquences.

UML souffre cependant de quelques limitations sur ce dernier point. Le projet RNTL COTE vise à en lever certaines. Nous proposons pour notre part quelques solutions à la section 6.2.2.

Il ne s'agit pas pour nous de constituer un nouveau manuel de référence d'UML. Il en existe en effet déjà de forts complets, en commençant par les documents officiels de normalisation d'UML [89]. Cependant, les documents de normalisation se révélant souvent assez abscons, le lecteur intéressé pourra se référer avec profit à d'autres ouvrages complémentaires davantage destinés aux utilisateurs d'UML qu'aux concepteurs d'outils et de méthodologie. Citons notamment les livres des « trois amigos » [15, 91, 60], ainsi que l'ouvrage de Pierre-Alain Muller [79] (auquel nous avons contribué en relisant et commentant le très complet chapitre de référence sur UML).

Nous avons conçu ce chapitre comme un exposé sur UML à la fois didactique mais d'un niveau assez avancé. Plutôt que de faire l'inventaire exhaustif des nombreuses facettes offertes par la syntaxe d'UML, nous illustrerons les principaux points d'UML

à l'aide d'un cas d'étude. Ce chapitre suppose une connaissance minimale des concepts de base de la modélisation objet.

Certaines parties de UML sont ardues, et parfois mal définies et peu abordées dans la littérature sur UML, pourtant abondante. Les correspondances inter-vues et les contraintes sous-jacentes que cela implique font partie de ces aspects difficiles de UML, et ne sont pas ou peu abordées par les règles de la sémantique statique de UML. Ces points seront traités plus en détails dans le chapitre 4.3, dans lequel nous apportons quelques contributions visant à compléter ou à éclaircir les documents officiels décrivant la notation UML et sa sémantique.

2.2 Cahier des charges d'un système de réunions virtuelles

2.2.1 Description informelle

Dans les sections qui suivent, nous allons donc présenter la spécification complète d'un système logiciel classique, mais néanmoins assez représentatif et permettant d'illustrer les principales phases du cycle de vie d'un système : Il s'agit d'un système serveur permettant d'organiser des réunions virtuelles, dont le fonctionnement est similaire dans le principe aux serveurs de "chat" de type IRC [1]. Cet exemple a été utilisé dans le cadre de l'enseignement d'UML dispensé aux élèves de troisième année de l'ENST de Bretagne. Le cahier des charges de ce système est le suivant :

Il s'agit de réaliser la partie serveur d'une application client-serveur permettant de faire des réunions virtuelles multimédia sur Internet. L'objectif de cette application est de permettre d'imiter le plus possible le déroulement de réunions de travail classiques. Cependant, dans la première version de ce projet, les interventions des participants se feront en mode mono-média seulement (i.e. échanges en forme textuelle).

Le serveur devra permettre de planifier et de gérer le déroulement de plusieurs réunions simultanées. Nous supposerons l'existence de programmes clients permettant à des personnes (identifiées par leur adresse électronique) désirant organiser des réunions virtuelles ou y participer de dialoguer avec le serveur en utilisant un protocole ad hoc basé sur les sockets. Ce protocole devra être explicité afin de permettre à des équipes indépendantes de réaliser des applications clientes.

Le serveur devra permettre aux clients de :

- planifier des réunions virtuelles (choix d'un nom, définition du sujet, date de début et durée prévue, ordre du jour),
- consulter les détails d'organisation d'une réunion,
- modifier les détails d'organisation (seulement l'organisateur),

1. Internet Relay Chat

- ouvrir et clôturer une réunion (seulement l'organisateur),
- entrer (virtuellement) dans une réunion précédemment ouverte,
- en sortir.

En cours de réunion, un participant peut demander à prendre la parole. Quand elle lui est accordée, il peut entrer le texte d'une intervention qui sera transmise en "temps réel" par le serveur à tous les participants de la réunion.

Plusieurs réunions doivent pouvoir être organisables :

Réunions standards , avec un organisateur qui se charge de la planification de la réunion et désigne un animateur chargé de choisir les intervenants successifs parmi ceux qui demandent la parole.

Réunions privées , qui sont des réunions standards dont l'accès est réservé à un groupe de personnes défini par l'organisateur.

Réunions démocratiques , qui sont planifiées comme des réunions standards, mais où les intervenants successifs sont choisis automatiquement par le serveur sur la base d'une politique premier demandeur, premier servi.

2.2.2 Les cas d'utilisation

La première étape de la modélisation consiste à déterminer précisément les besoins des utilisateurs du système. Ces besoins sont généralement stipulés de manière assez informelle, en langage naturel, dans le cahier des charges.

Les différentes fonctionnalités offertes par ce système forment ainsi un ensemble de *cas d'utilisation* (*"UseCase"*), exprimés informellement et volontairement très ambigue dans le cas présent, dans un but pédagogique, afin de montrer l'intérêt qu'il peut y avoir à les formaliser.

UML propose au travers des cas d'utilisation de répertorier et de structurer les fonctionalités que le futur système offrira à ses utilisateurs. Cette approche fonctionnelle est héritée de la méthode OOSE (Object Oriented Software Engineering [61]) avant d'être intégrée dans UML.

Graphiquement, le système est représenté par un rectangle (le système est ainsi vu comme une boîte noire), autour duquel sont représentés ses utilisateurs potentiels, sous la forme symbolique de "bonhommes" en fil de fer appelés acteurs, qui sont extérieurs au système. Chaque utilisation possible du système par les acteurs se traduit par une ellipse étiquetée avec le nom du cas d'utilisation et connectée aux acteurs concernés par cette interaction avec le système.

UML offre plusieurs mécanismes permettant de structurer les cas d'utilisation. Plusieurs types de relations existent entre cas d'utilisation :

- la spécialisation (un *UseCase* est un *Classifier*)
- la réutilisation de comportement par inclusion

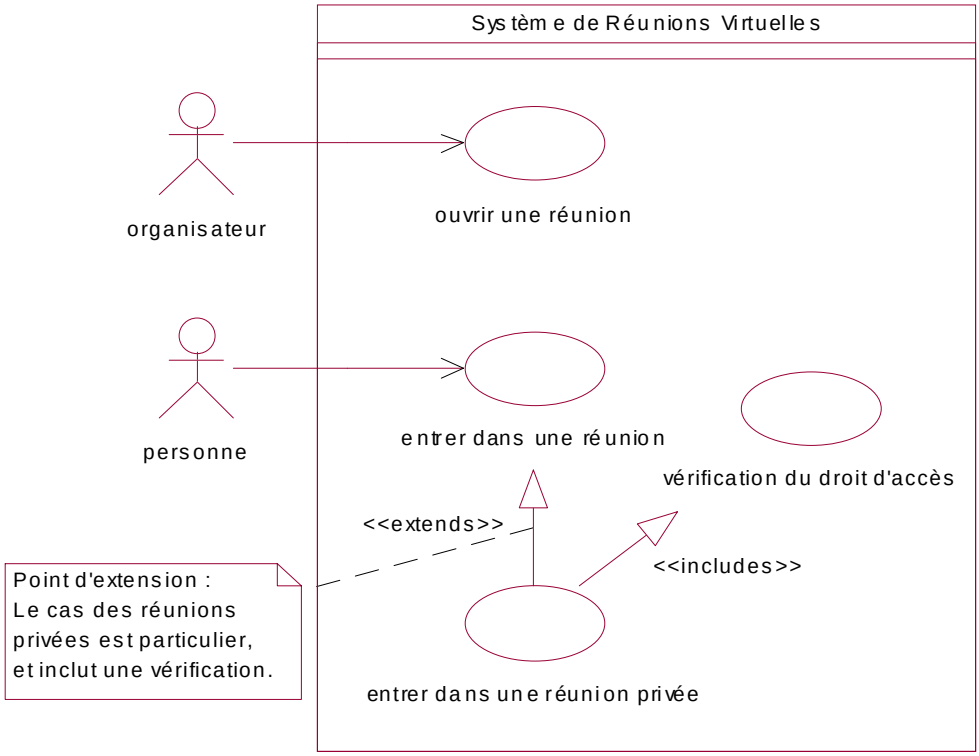


FIG. 2.1 – *Cas d'utilisation pour les réunions virtuelles*

– l'extension par insertion de comportements alternatifs ou exceptionnels au sein d'un cas "nominal", lorsque certaines conditions sont remplies au point dit *point d'extension*.

Ces relations sont représentées par des flèches reliant les ellipses et, pour les deux dernières, étiquetées par les mots clés «`include`» ou «`extend`», respectivement. On pourra consulter avec profit l'article [94] qui décrit toutes ces relations en détails.

2.2.3 Illustrer les cas d'utilisation

UML permet de décrire un cas d'utilisation en montrant les interactions entre les différents acteurs et le système. Ces descriptions reposent sur une partie structurelle appelée *collaboration* définissant les rôles des participants, et une partie *interaction* proprement dite définissant les messages que ces participants s'échangent.

Dans le cas des réunions virtuelles, une annexe du cahier des charges spécifie aussi le protocole exact entre les clients et le serveur. Par exemple, pour entrer dans une réunion, un client (préalablement authentifié) doit envoyer le message `ENTER réunion` (où `réunion` représente le nom de la réunion). En retour, chacune des personnes qui participent déjà à la réunion en question reçoit le message `ENTERING réunion nouveau_participant`, et le nouveau participant reçoit quant à lui la liste des autres participants grâce au message `PRESENT réunion participant1 ... participantN`.

UML permet de présenter cette même interaction de deux manières différentes :

– Par un diagramme de collaboration, donné par la figure 2.2.

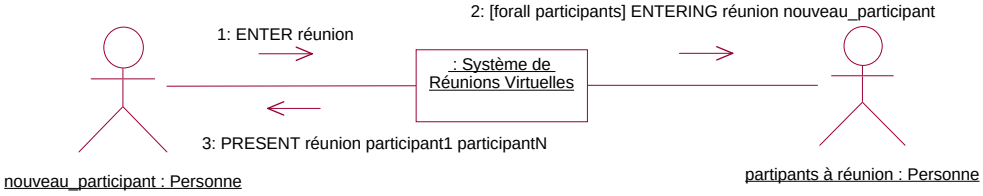


FIG. 2.2 – *Entrée : Diagramme de collaboration*

– Par un diagramme de séquences, donné par la figure 2.3.

2.2.4 Limites des cas d'utilisation

Les cas d'utilisation de UML ont certes l'avantage d'être graphiquement très simples et donc faciles à appréhender. Malheureusement, cette simplicité ne va pas sans une certaine pauvreté sémantique. En effet, s'il est facile de voir que "entrer dans une réunion" est une des fonctionalités requises du système, une simple ellipse connectée

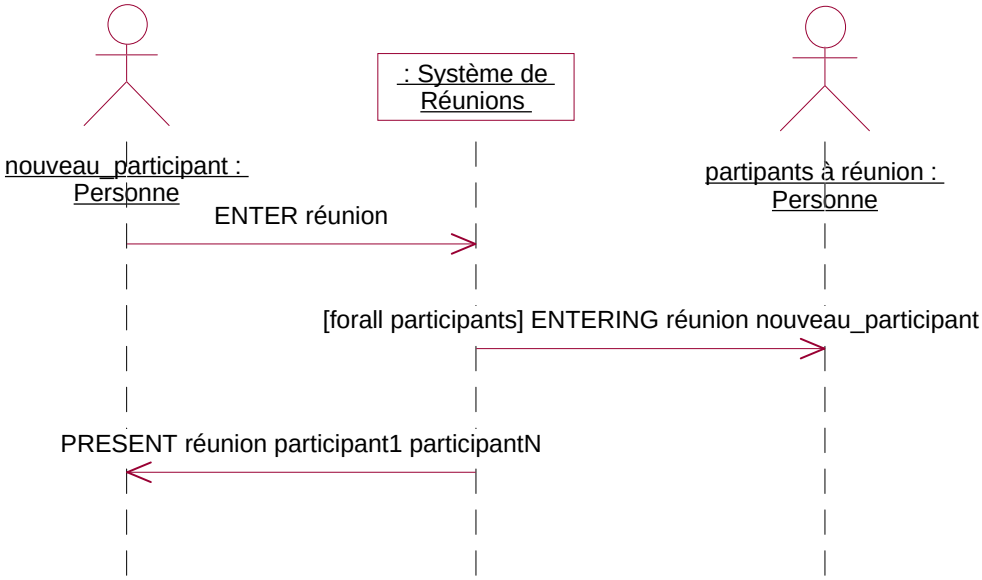


FIG. 2.3 – *Entrée : Diagramme de séquences*

à un acteur ne permet pas d'exprimer clairement quel sera l'effet de cette fonctionalité sur le système lorsqu'un utilisateur la mettra en œuvre.

Aller plus loin dans la formalisation nécessite d'entrer *à l'intérieur du système* et d'expliciter sa connaissance du monde extérieur. Nous devons donc disposer du modèle précis que le système a du monde extérieur, en termes de réunions et de personnes. C'est en effet ce modèle qui conditionne les réactions du système aux stimuli qu'il reçoit. Définir un tel modèle en UML est l'objet de la section qui va suivre. La section 2.4.1 expliquera quant à elle comment les cas d'utilisation peuvent être formalisés en se basant sur ce modèle.

Notons aussi qu'une bonne modélisation par objets doit pouvoir s'accomoder facilement de nouveaux cas d'utilisation, puisqu'il n'est pas rare que les utilisateurs souhaitent l'ajout de nouvelles fonctionalités, et ce même bien au delà des phases préliminaires de la spécification. Il faut donc prendre garde de ne pas adopter une approche trop fonctionnelle risquant de rigidifier la structure du système.

2.3 Spécifier la structure du système

2.3.1 Le cœur d'UML

Le cahier des charges élaboré précédemment permet déjà d'établir une ébauche assez précise de la structure du système serveur, représentée par le diagramme statique de la figure 2.4. Notons qu'il s'agit seulement du modèle de données du serveur. Les

interactions avec l'extérieur, notamment via une couche réseau basée sur les sockets, n'y sont pas représentées. Ce diagramme illustre la quasi-totalité des concepts centraux de UML.

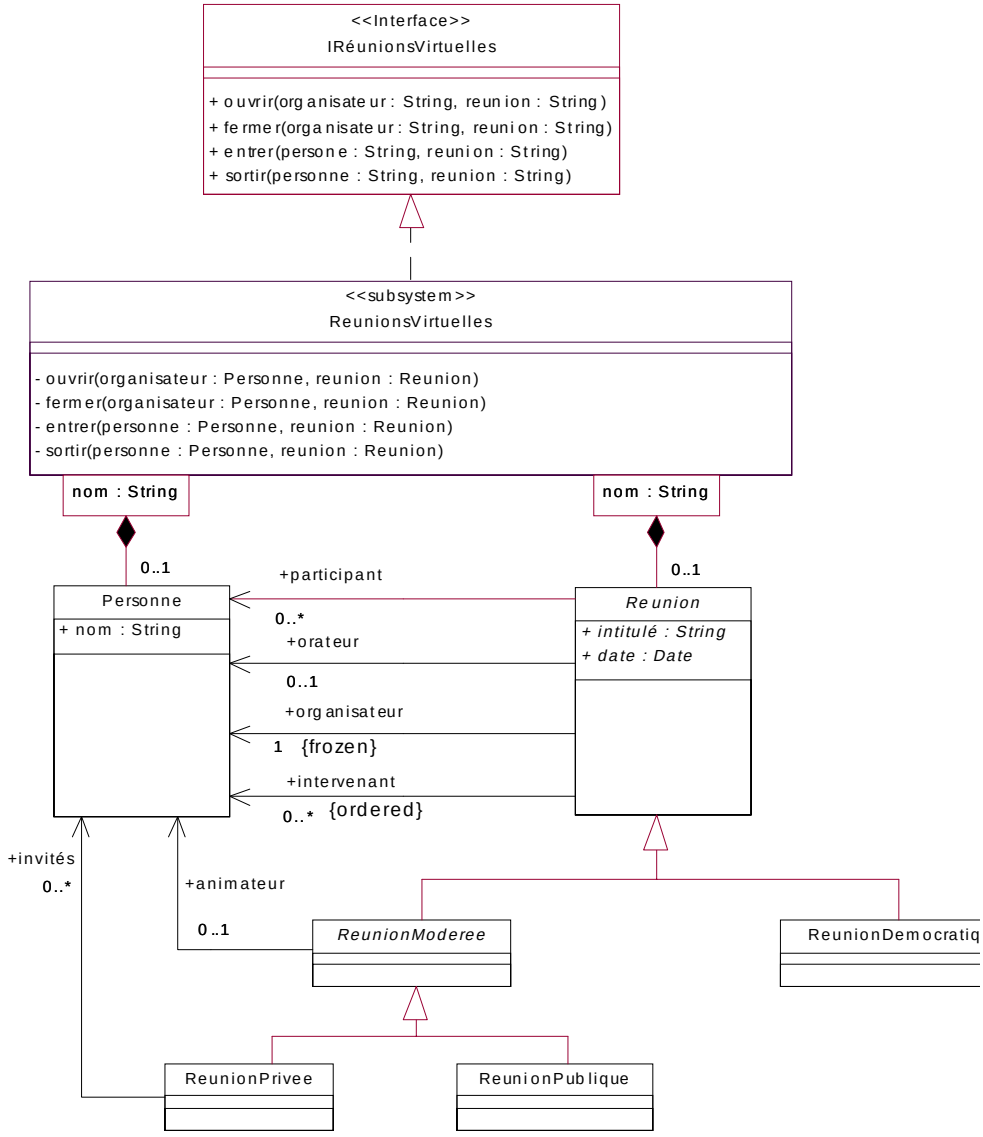


FIG. 2.4 – *Diagramme de classe pour les réunions virtuelles*

2.3.2 Modèle de classes

On y distingue en premier lieu le *sous-système* ("Subsystem") central du serveur. Un sous-système UML est à la fois un classificateur (avec des opérations et des at-

tributs) et un paquetage (espace de nommage pouvant contenir d'autres éléments de modélisations définis localement). Les opérations du sous-système sont donc les opérations qui peuvent être appliquées pour modifier l'état du cœur du serveur. Ces opérations seront appelées indirectement lorsqu'un utilisateur du système (via un programme client) enverra une commande au serveur.

Le sous-système contient bien sûr des réunions, ce qui se représente par une association entre le classificateur représentant le serveur et celui représentant les réunions (un sous-système est rappelons-le aussi un classificateur, et peut donc prendre part à des associations). La relation de contenance (forte), appelée composition, est représentée par un petit losange noir placé à l'extrémité de l'élément contenant.

Même si cela peut surprendre, le sous-système contient aussi des personnes. Par personne, il faut entendre "modèle d'une personne", c'est-à-dire un objet informatique représentant l'ensemble des connaissances que le sous-système possède d'une personne du monde réel. L'entrée dans une réunion n'est ici que virtuelle.

Notons enfin qu'il existe deux notions distinctes de contenances entre le sous-système serveur d'une part et réunion(s) et personne(s) d'autre part : L'objet serveur contiendra des objets personnes et réunions, au sens ci-dessus. Mais les classes Personne et Réunion peuvent aussi être définies à l'intérieur de la définition du sous-système. En ce sens, le sous-système Serveur est vu comme un *espace de nommage* ("Namespace"). Pour bien marquer cette imbrication des définitions, il est possible de représenter les définitions imbriquées à l'intérieur de leur espace de nommage, comme l'illustre la figure 2.5. Cependant, l'imbrication graphique des rectangles de classes est ambiguë en UML : Cela peut signifier soit une relation de composition, soit une imbrication des définitions. Dans le cas présenté ici, les deux significations sont valides.

2.3.3 Spécification et réalisation d'un sous-système

Un sous-système se divise en deux parties :

- Une partie dite de spécification, qui spécifie la vue extérieure du système, et notamment ses *interfaces* avec l'extérieur.
- Une partie dite de réalisation, qui spécifie la vue interne du système.

Le sous-système des réunions virtuelles est accessible via une interface qui contient des opérations publiques (comme en témoigne le signe '+' devant leurs signatures). Notons que ces routines ne manipulent pas directement des instances des classes Personnes et Réunions, dont seul le sous-système a connaissance. Les opérations de l'interface ne manipulent que des noms de personnes et de réunions, représentés par des chaînes de caractères.

Le sous-système est implémenté par des opérations privées (comme en témoigne le signe '-' devant leurs signatures), qui elles, peuvent manipuler des instances de Personne et de Réunion.

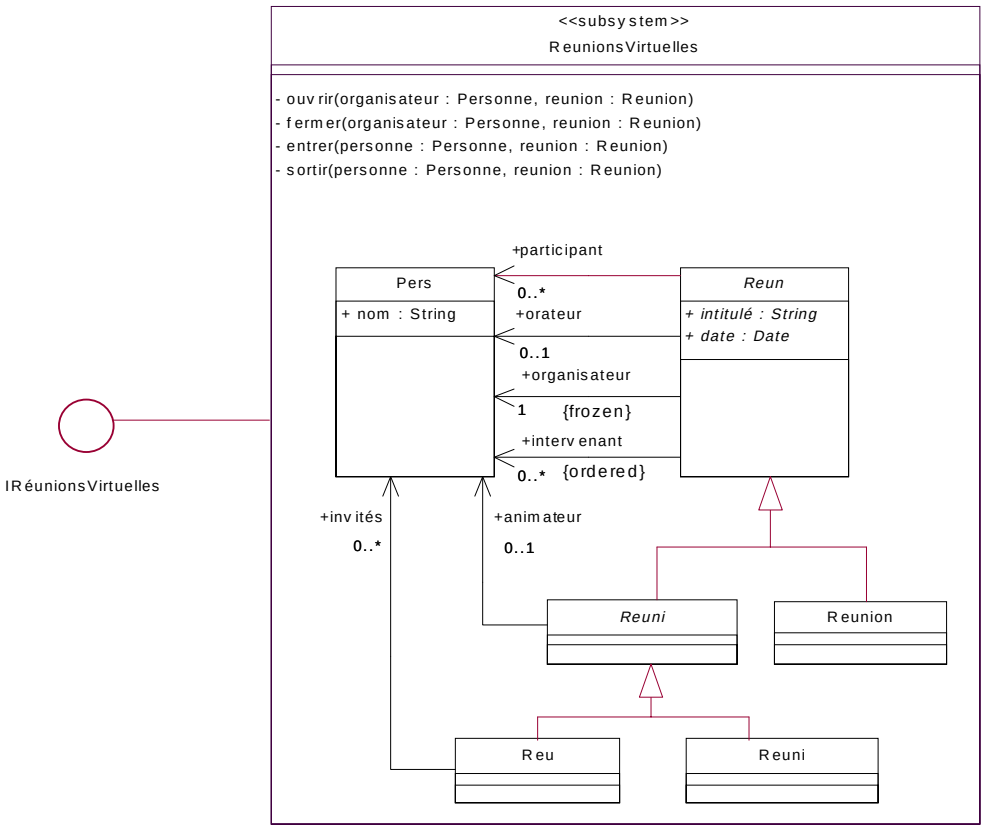


FIG. 2.5 – *Système de réunions virtuelles*

2.3.4 Associations qualifiées

Les opérations réalisant l'interface du sous-système peuvent retrouver facilement les instances de Personnes et de Réunions correspondant aux noms, qui vont ensuite être simplement *déléguées* aux opérations d'implémentation pouvant les manipuler.

En effet, les relations de compositions vues précedemment sont *qualifiées* par les noms des personnes ou réunions, respectivement. Le qualificateur "nom" est une clé, qui, à une chaîne de caractères, associe au plus une instance de Personne ou de Réunion, respectivement. Ce nombre d'instances associées à une clé est donné par la multiplicité du côté opposé. Pour Personne et Réunion, elle vaut bien 0..1.

2.3.5 L'héritage

Le cahier des charges établit clairement une classification des différentes sortes de réunions qu'il est possible d'organiser. Cette classification peut même être raffinée, en introduisant une classe de réunions intermédiaires, appelées réunions modérées (par opposition aux réunions démocratiques qui n'ont pas de modérateur . Cette classe intermédiaire est une abstraction, plus précisément une généralisation des deux concepts plus spécialisés que sont les réunions publiques ou privées (l'accès est cette fois le critère discriminant). La classe des réunions modérées *n'a donc pas d'instance* qui ne soient pas ou publiques, ou privées. C'est donc une classe *abstraite*, raison pour laquelle son nom apparaît en italique sur le diagramme. Cette propriété peut aussi se représenter en attachant $\{abstract[= true]\}$ au classificateur, dans le cas où le média utilisé pour afficher le diagramme UML ne permet pas d'avoir du texte en italique.

Les différentes classes de réunions forment donc une *hiérarchie* dont la racine est la classe la plus générale et les feuilles, des classes spécialisées. La relation généralisation-spécialisation est représentée par une flêche à extrémité triangulaire.

La notion d'héritage présente dans les langages à objets est un mécanisme de réutilisation important : En spécialisant une classe ancêtre, une classe fille réutilise l'ensemble des propriétés définies dans le contexte de la classe ancêtre, tout en définissant de nouvelles propriétés qui lui sont propres.

Combiné à la notion de liaison dynamique, l'héritage est avant tout un mécanisme fondamental d'abstraction : En factorisant les propriétés communes à plusieurs classes dans une classe ancêtre, il devient possible de cacher aux clients (c'est-à-dire les classes n'appartenant pas à cette hiérarchie et qui utilisent leurs services) la classe exacte des objets qu'ils manipulent, dans la mesure où la connaissance des services "génériques" offerts par la classe abstraite est suffisante.

2.3.6 Les associations

Des relations existent entre réunions et personnes. Certaines de ces relations n'ont de sens que pour certains types de réunions, et apparaissent par conséquent plus ou moins haut dans la hiérarchie des classes de réunions. Les personnes impliquées dans une *association* avec une réunion jouent un *rôle* particulier vis-à-vis de celle-ci. Pour une réunion donnée, seul un certain nombre de personnes peuvent jouer un rôle donné, et vice-versa. Le rôle est indiqué près de l'extrémité de l'association, ainsi que la multiplicité des participants jouant ce rôle.

Ainsi, toute réunion est organisée par une et une seule personne, et cette personne reste l'organisateur tout le long de l'existence de la réunion. Ce rôle d'organisateur est incessible (ou figé, ce qui se traduit par $\{frozen[= true]\}$ en UML). Dans toute réunion, il ne peut y avoir à un moment donné qu'au plus un orateur (ou aucun si personne n'a la parole à ce moment-là). Les personnes qui ont demandé la parole sont en attente d'intervenir. L'ordre est important dans cette relation, car c'est cet ordre qui indiquera le futur orateur (politique du premier arrivé, premier servi). L'annotation $\{ordered[= true]\}$ est donc apposée sur l'extrémité "intervenant" de l'association.

La notion de rôle pour les associations joue ... un rôle (sic) très important dans la modélisation. On retrouvera ce même concept de rôle plus loin en section 2.4.2 lorsque nous étudierons les collaborations d'UML, qui permettent de montrer la dynamique du système. UML possède donc deux concepts de rôles, très similaires, qui gagneraient à être syntaxiquement unifiés. Une telle unification est d'ailleurs proposée dans [93] pour supprimer cette redondance avec élégance.

2.4 Spécifier la dynamique du système

2.4.1 Cas d'utilisation et opérations du sous-systèmes

Comme nous l'avons vu en section 2.2.4, il est difficile de spécifier précisément les cas d'utilisation sans disposer d'information sur le fonctionnement du système. Nous disposons depuis la section précédente d'un modèle précis de l'intérieur du système, qui va nous permettre d'aller plus loin dans la formalisation.

Ainsi, chaque opération définie dans l'interface du sous-système central correspond à un cas d'utilisation du système. Formaliser ces opérations permet donc indirectement de formaliser les cas d'utilisation.

2.4.2 Collaborations

De même que collaborations et interactions pouvaient compléter la définition des cas d'utilisation, il est aussi possible en UML de les utiliser pour décrire des opérations,

et donc notamment les opérations du sous-système. Une collaboration représente alors une assemblée d'objets participant tous ensemble à la réalisation d'une opération.

Mais contrairement aux diagrammes 2.2 et 2.3 de la section 2.2.3 pour lesquels le système était vu comme une boîte noire et seules les interactions à la frontière étaient pertinentes, nous allons pouvoir à présent rentrer à l'intérieur du système. Nous disposons en effet du modèle du système défini dans la section 2.3.1.

Les rôles apparaissant au sein d'une collaboration sont étroitement liés aux rôles dans les associations que nous avons déjà rencontrés dans la section 2.3.1, lors de l'élaboration de la vue statique du système représentée par le diagramme de la figure2.5.

En nommant les objets, les rôles qu'ils jouent ainsi que les associations qui les lient, la partie structurelle fournit un *vocabulaire* formel qui nous permet de *formuler* des propriétés précises concernant le système. Afin d'écrire ces formules, UML a été doté d'un langage (textuel) dédié appelé OCL [100] (pour *Object Constraint Language*). Initialement dévolu à la spécification de la sémantique statique de UML (les "well-formedness rules" du méta-modèle d'UML), le langage OCL peut être utilisé pour compléter tout modèle UML, et non plus uniquement le méta-modèle d'UML (qui n'est qu'un modèle UML particulier).

2.4.3 Conception par contrats avec UML et OCL

Une application intéressante d'OCL dans ce contexte, proposée par la méthodologie Catalysis [32], consiste à décrire l'effet d'une opération à l'aide de préconditions et de postconditions, qui sont des formules OCL reposant sur la partie structurelle et décrivant respectivement les conditions requises du système pour que la réalisation de la fonctionalité puisse avoir lieu, ainsi que les propriétés vérifiées par le système lorsque cette réalisation sera achevée. On retrouve aussi cette approche de la modélisation par contrat dans le langage Eiffel [78, 65].

La collaboration fournit plusieurs informations structurelles particulièrement utiles :

- La topologie du réseau d'objets, par l'intermédiaire des rôles d'associations et des stéréotypes figurant éventuellement sur leurs extrémités :
  - «self» représente l'objet auquel s'applique l'opération
  - «global» représente un objet globalement accessible
  - «local» représente une objet défini localement au sein de l'opération
  - «parameter» représente un des objets passés en paramètre lors de l'appel de l'opération.
- Les messages échangés entre les objets afin de réaliser la tâche. Ces messages empruntent les connexions entre objets.

Il est aussi possible d'accompagner les pré et post conditions par deux diagrammes d'objets (encore appelés "snapshots", correspondant au niveau M0 de l'architecture à quatre niveaux étudiée à la section 4.2.1) pour représenter le système avant et après la

réalisation de la fonctionalité. C'est d'ailleurs en imaginant ces snapshots et en cherchant le vocabulaire et les liaisons nécessaires pour exprimer les formules en OCL que l'on peut progressivement établir la structure du système en terme de classes, d'associations, de rôles et d'états.

La figure 2.6 représente ainsi une définition complète de l'opération `ouvrir` du système qui permet à l'organisateur d'une réunion d'ouvrir celle-ci. La signature de l'opération est à présent accompagnée d'une collaboration fournissant le contexte permettant d'écrire les préconditions et les postconditions en OCL.

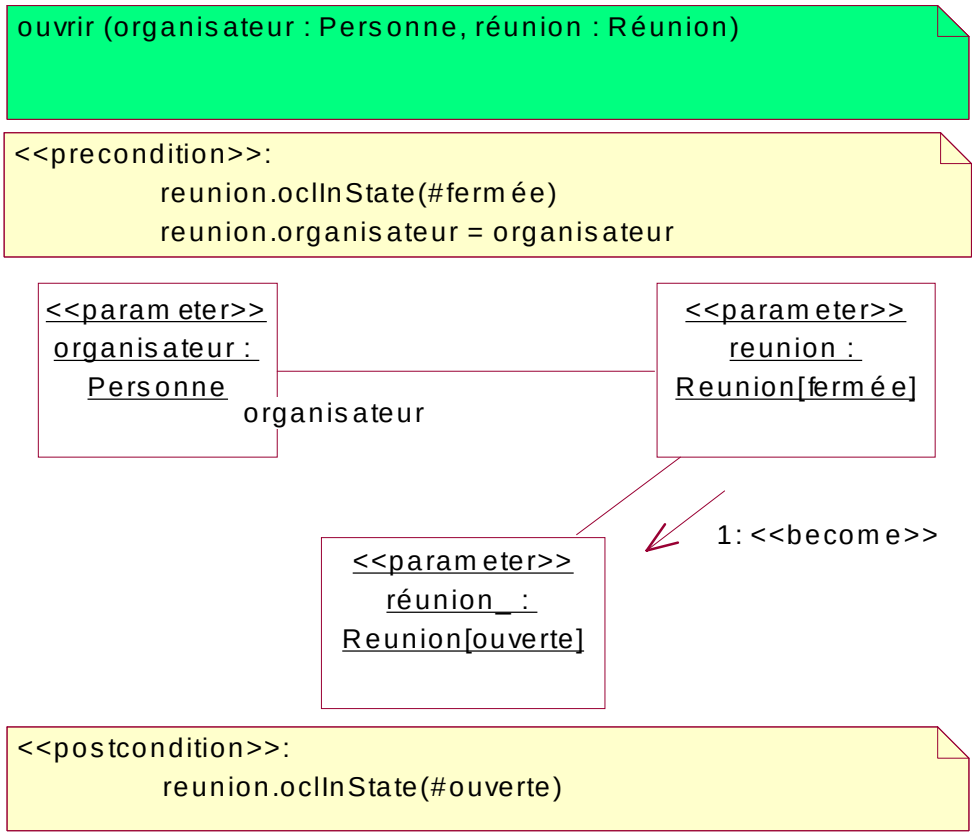


FIG. 2.6 – *Ouverture d'une réunion*

On peut d'ores et déjà constater que certaines assertions portent sur les rôles respectifs des paramètres, alors que d'autres portent sur leur état. La *transition* d'état que subit l'instance de réunion se représente à l'aide d'un *flot* stéréotypé «`become`». Nous reviendrons plus en détail sur cet aspect dans la section 4.3.

La figure 2.7 représente quant à elle une définition complète de l'opération `entrer` du système qui permet à une personne d'entrer dans une réunion dans la mesure où celle-ci est ouverte.

Une question importante est de savoir quelle partie du système doit s'assurer que

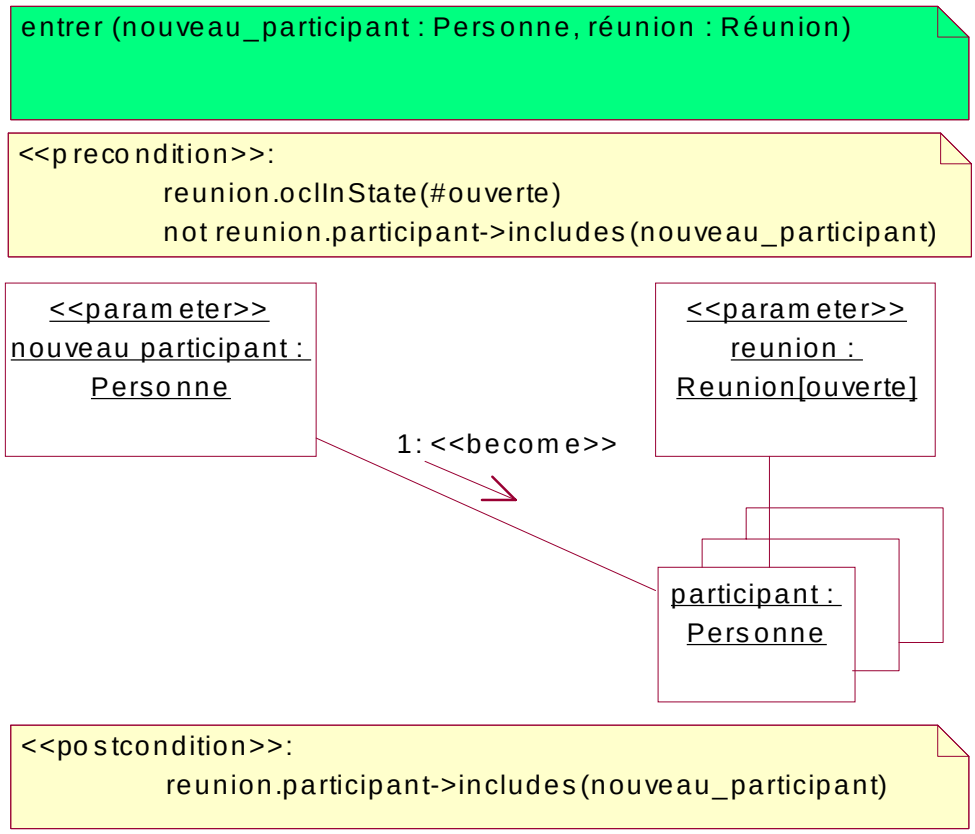


FIG. 2.7 – *Entrée dans une réunion*

les préconditions ne seront jamais violées lors de l'appel de ces opérations (sinon on sort du cadre de cette spécification et rien ne pourra être garanti quant à l'effet de l'opération). L'idéal est de faire les vérifications adéquates à la frontière du système, c'est-à-dire entre son interface et sa réalisation. Les opérations qui convertissent les chaînes de caractères en instances de Personne et de Réunion avant de déléguer aux opérations de réalisation sont donc les meilleurs candidats pour le traitement des erreurs éventuelles.

2.4.4 Les actions conjointes : Une vision synchrone

L'utilité de la partie interaction en phase préliminaire de la modélisation est souvent plus discutable. En effet, cette phase se concentre sur ce que fait le système (le "quoi") et non sur une manière précise dont peut être réalisée telle ou telle fonctionalité (le ou les "comment"). Or hormis les interactions ayant lieu à la frontière entre le système et son environnement, qui sont très importantes et font partie de la spécification (i.e. du "quoi"), on ne peut pas (ou surtout, on ne souhaite pas) expliciter les envois de message entre objets particuliers *internes au système*.

Une telle répartition des responsabilités est souvent prématurée. On risque fort ainsi de sur-spécifier, empêchant peut-être par la suite les développeurs implantant la spécification de faire certains choix de conception qui auraient pourtant été judicieux. Afin justement de ne pas sur-spécifier prématurément, la méthodologie Catalysis propose une petite extension aux interactions d'UML qui évite d'avoir à distinguer un objet émetteur, un ou plusieurs destinataire(s), et les objets qui sont relégués au rôle de simples paramètres.

Cette extension est nommée *action conjointe*. Une action conjointe met en œuvre un ensemble d'objets, indistinctement. On notera avec intérêt la très forte similarité entre les notions d'actions conjointes de Catalysis et les multi rendez-vous du langage de description formel LOTOS [57]. À ce niveau d'abstraction, tout ce passe donc comme si le sous-système formait un tout indivisible, ses composants semblant réagir de manière *synchrone* aux stimuli de l'environnement du sous-système.

Un cas d'utilisation peut ainsi être vu comme une action conjointe de forte granularité (une transaction en quelque sorte), éventuellement décomposée en un ordre partiel d'actions conjointes de granularité plus fine. Cet ordre partiel peut être représenté d'une manière assez similaire à une interaction UML classique, en remplaçant les flêches représentant les messages par des barres horizontales représentant les actions conjointes. S'il l'on parvient ultérieurement à raffiner successivement ces actions conjointes jusqu'à ne plus avoir que des actions dites "localisées", alors on obtiendra à nouveau des interactions UML classiques, donnant une vision asynchrone du comportement lorsque l'on rentre dans les détails.

2.4.5 États des objets

La collaboration de la figure 2.6 a fait apparaître des évolutions dans l'état de certains objets au cours de l'exécution d'une opération. Alors que les collaborations sont par définition focalisées sur les liens entre les objets, leurs rôles respectifs, et les interactions qui les unissent, elles sont mal adaptées à la description de l'état des objets. La seul manière de faire apparaître la notion d'état est de suffixer la classe de base d'un rôle par l'état dans lequel se trouve l'objet jouant ce rôle.

La section que nous abordons à présent a pour but de définir précisément les notions d'*états* et de *transitions* que l'on a pu entre-apercevoir dans le cadre des collaborations. À cette fin, UML propose les diagrammes d'états transitions, directement inspirés des statecharts originalement introduits par David Harel [51].

La figure 2.8 représente les évolutions possibles d'une réunion. Nous avons vu grâce à la collaboration de la figure 2.6 que l'appel de l'opération `ouvrir` pouvait changer l'état de la réunion passée en paramètre. La machine à états associée à la classe réunion définit les différents *états* dans lesquels peut se trouver une réunion, ainsi que les conditions dans lesquelles les changements d'états se produisent, lors des *transitions*.

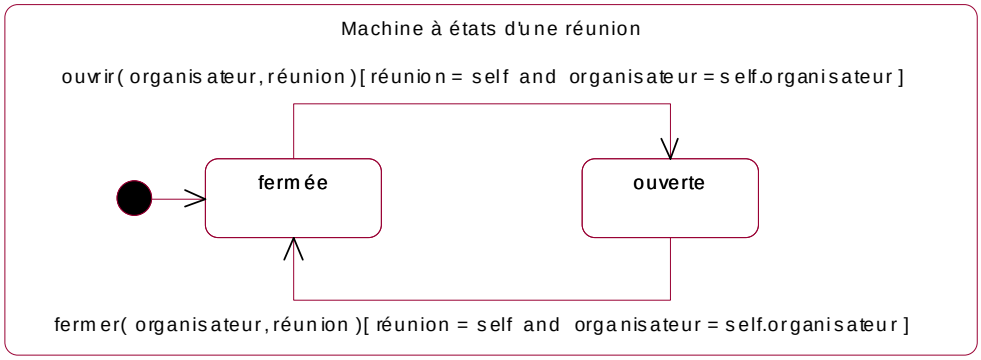


FIG. 2.8 – *Machine à états d'une réunion dans le contexte du système*

Ainsi, comme nous l'avons déjà vu, une réunion possède deux états, `fermée` ou `ouverte`. Lors de sa création, une réunion est initialement placée dans l'état `fermée`. Cela est indiqué par une transition sortant du pseudo-état initial représenté par le point noir et pointant vers l'état `fermée`. Elle peut passer d'un état à l'autre lorsque les opérations `ouvrir` et fermer du système sont appelées, et que le paramètre `réunion` lui correspond, dans la mesure où la personne tentant d'ouvrir la réunion en est bien l'organisateur. Les transitions portent toutes ces informations :

- Un événement déclencheur, le plus souvent un appel d'opération.
- Les paramètres de l'événement, qui correspondent alors à ceux de l'opération.
- Une garde, permettant de conditionner la transition.

– Un effet, ensemble d'actions exécutées lorsque la transition est tirée.

Un état E donné d'une machine à états attachée à une classe représente *un ensemble d'instances de cette classe ayant en commun certaines propriétés*. Un tel ensemble d'instances peut éventuellement être partitionné. Chaque sous-ensemble de la partition est alors un *sous-état exclusif* (ou XOR-state). La définition mathématique d'une partition impose en effet qu'une instance appartienne à un et un seul sous-ensemble de la partition, donc qu'elle soit dans un et un seul des sous-états.

Par exemple, une réunion de l'ensemble R des réunions est soit fermée, soit ouverte. Les états `fermée` et `ouverte` sont donc deux sous-états exclusifs de l'état englobant R (le "top-state" de la machine à états de la classe Réunion).

2.4.6 États des associations

Si les réunions possèdent intrinsèquement un état, fermée ou ouverte, le cas des personnes est un peu plus compliqué. Nous avons vu que lors d'une collaboration le rôle d'une personne pouvait changer. Cet "état" n'est cependant pertinent que vis-à-vis d'une réunion particulière par rapport à laquelle cette personne joue un *rôle* précis, d'organisateur, d'orateur, ou encore de simple participant.

Toutefois, en projetant une personne selon l'axe d'une réunion, nous pouvons décrire les différents états/rôles que cette personne va prendre successivement dans le cadre de la réunion. Dès lors, l'utilisation d'un statechart devient à nouveau tout à fait pertinente, comme en témoigne la figure 2.9. Le statechart est donc attaché non plus à une seule classe, mais à la fois à la classe Personne et à la classe Réunion, et représente l'états des associations entre une personne et une réunion. À notre connaissance, c'est la première fois que les statecharts d'UML sont utilisés de la sorte.

Lorsqu'il existe plusieurs critères *indépendants* de partitionnement, des sous-états dits *concurrents* (ou AND-state) sont intercalés dans la hiérarchie d'états, un pour chacune des partitions possibles. La notion de concurrence est à rapprocher ici de celle, plus adéquate, d'indépendance. Chaque état-partition est alors lui-même partitionné en sous-états exclusifs.

Ainsi, le fait d'être ou non organisateur est complètement indépendant du fait d'être ou non participant, voire orateur, d'une réunion. Vis-à-vis d'une réunion donnée, l'ensemble P des personnes peut donc être partitionné suivant plusieurs critères. On peut déjà deviner qu'il va exister autant d'états concurrents que d'associations `indépendantes` entre les deux classes. Au contraire, les associations qui dépendent d'une autre association (par exemple, le fait qu'il falle être participant pour pouvoir être orateur) se traduisent par des sous-états exclusifs.

Notons enfin qu'une même personne peut d'ailleurs très bien être organisateur d'une première réunion, orateur d'une seconde, et encore simple participant dans une troisième, de manière complètement indépendante.

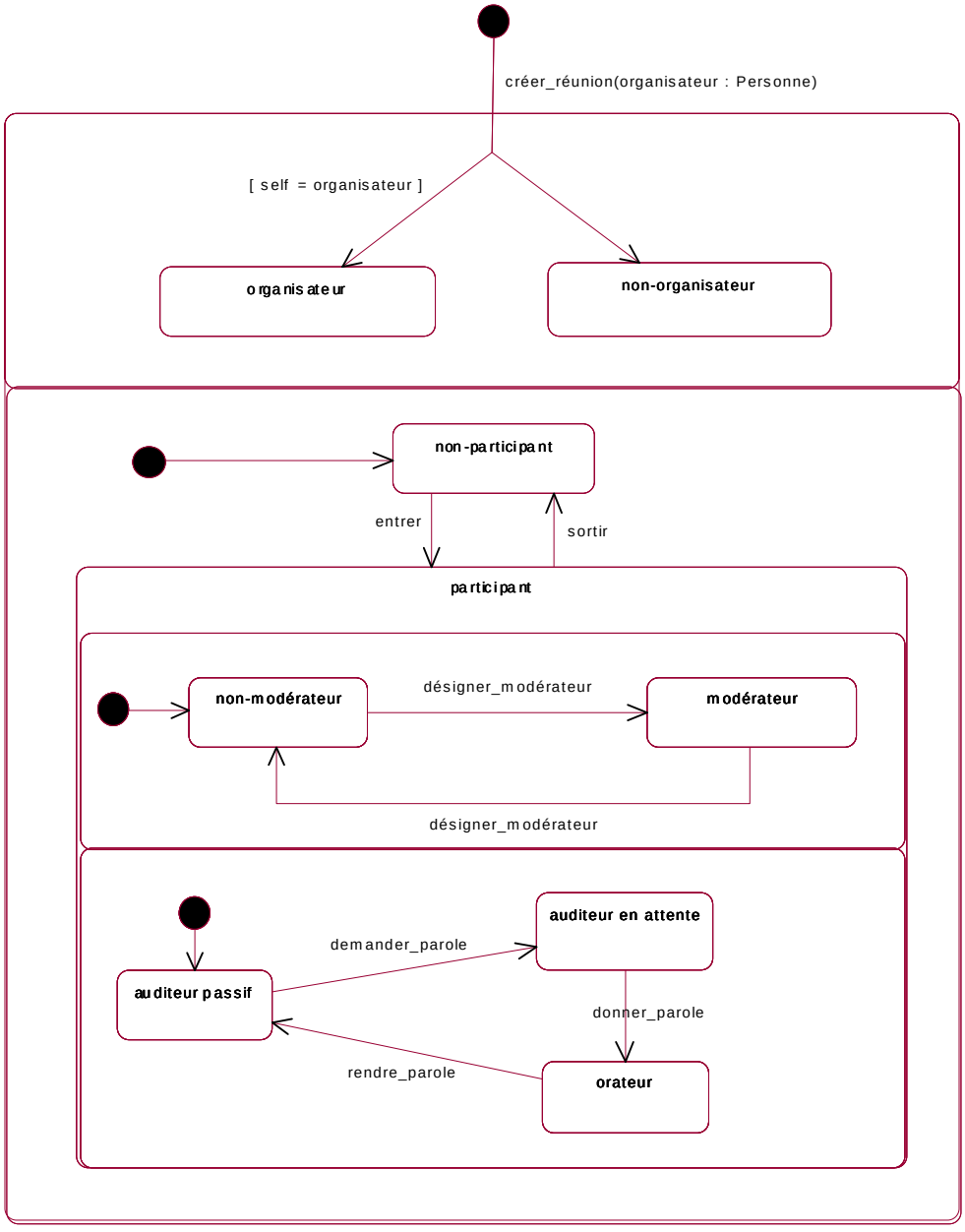


FIG. 2.9 – *Machine à états d'une personne dans le contexte d'une réunion*

2.5 De la spécification vers l'implémentation

2.5.1 Abstraction, Raffinements et Patrons de Conception : Une approche continue

À moins que le concepteur ne soit doté de la faculté de préscience, la spécification détaillée en UML d'un système ne peut pas être produite directement. Sa réalisation passe nécessairement par une suite de phases correspondant chacune à un niveau d'abstraction, du niveau le plus abstrait jusqu'au niveau le plus détaillé.

À chaque niveau d'abstraction correspond donc un modèle UML spécifique. Bien sûr, il existe une forte corrélation entre deux modèles décrivant le même système à deux niveaux d'abstraction adjacent. Pouvoir formaliser ce lien de traçabilité entre deux modèles présente de nombreux intérêts :

- Connaître l'impact qu'une modification faite à un niveau va avoir sur l'autre niveau
- Maintenir automatiquement la cohérence entre les deux niveaux d'abstraction, par des mécanismes de transformation de modèles ou l'application de patrons de conception
- Transposer certaines propriétés dont on aura prouvé la validité à un niveau vers un autre niveau.

La section 3.3.3 explique l'apport des techniques formelles dans ce contexte.

2.5.2 Patrons de conception

Les patrons de conception (*design patterns*), en caractérisant des problèmes récurrents de conception et en spécifiant des solutions claires et élégantes à ces problèmes, représentent le vocabulaire commun de l'expertise des concepteurs de logiciels [24, 27, 41]. Ce vocabulaire leur fournit un niveau de description adéquat pour discuter les choix de conception ou de restructuration d'un système, au-delà des détails connus de conception. De plus, l'explicitation des patrons de conception présents dans un système simplifie la compréhension de sa structure (voir par exemple [68]). Ils apportent au développement au moins deux bénéfices :

- D'une part, ils rendent accessibles les connaissances des informaticiens plus expérimentés aux concepteurs novices. Grâce à ces connaissances ceux-ci peuvent identifier une situation où l'utilisation suggérée par un patron de conception serait bénéfique.
- D'autre part, ils guident les développeurs dans le choix de la mise en œuvre la plus adaptée à cette solution. En effet, un patron regroupe sous une étiquette unique une variété de situations implémentatoires qui ont une logique de fonctionnement commune. Sa mise en œuvre est pilotée par différents compromis,

> et peut adopter différentes formes ou variations, sans pour autant déroger à la solution de conception qu'il définit.

Les patrons de conception trouvent leur origine dans les travaux de l'architecte Alexander [8, 7, 6].

Étant donné l'importance primordiale des patrons de conception, il aurait été indigne d'UML en tant que notation unifiée de ne pas proposer les concepts et notations adéquats pour leur représentation dans les modèles de conception. Les nombreux ouvrages traitant de modélisation avec UML comportent d'ailleurs quasiment tous un chapitre dédié aux patrons de conceptions. Malgré le traitement important qui leur est accordé dans ces ouvrages, UML se révèle étonnamment pauvre lorsqu'il s'agit de représenter précisément les patrons de conception, et encore plus pauvre lorsqu'il s'agir de les définir (en admettant que l'on s'entende enfin sur ce que définir "formellement" un patron de conception veut dire). Dans [95] et [49], nous montrons les limites dont souffre UML pour représenter les patrons de conceptions, et nous proposons quelques solutions afin de remédier à ces problèmes.

La figure 2.10 représente un diagramme de classes décrivant la couche réseau entourant le sous-système des réunions virtuelles. Ce diagramme de conception est assez proche d'une implantation en Java, comme en témoignent certaines classes directement extraites de la bibliothèque standard de ce langage (Thread, Socket, etc).

Cette bibliothèque fait justement un usage important des patrons de conception, et nous les avons indiqués sur le diagramme[2], car cela apporte une information tout à fait pertinente sur la structure du système et le rôle de ses composants. Les patterns Adapter et Decorator montrent comment les simples flots d'octets fournis par les sockets sont transformés en suites de lignes de caractères codés en unicode. Chaque ligne forme une commande textuelle complète qui pourra ensuite être décodée par la *thread* de connexion associée à la socket d'où proviennent les commandes. Après décodage, l'opération correspondante de l'interface du sous-système central des réunions virtuelles sera appelée.

2.6 Les mécanismes de communications en UML

UML dispose d'une riche gamme de mécanismes de communication inter-objets. La partie structurelle, avec les déclarations qu'elle comporte, donne un aperçu de l'ensemble de ces mécanismes. Chaque classificateur possède des *caractéristiques* (*Features*) qui composent sa signature. Ces caractéristiques se divisent en deux catégories :

– Les caractéristiques structurelles (*Structural Features*), c'est-à-dire les attributs du classificateur, faisant l'objet de la section 2.6.2.

2. L'outil utilisé pour dessiner ce diagramme ne possédant pas le symbole elliptique censé representer une occurrence d'un patron de conception, nous avons utilisé une note à la place.

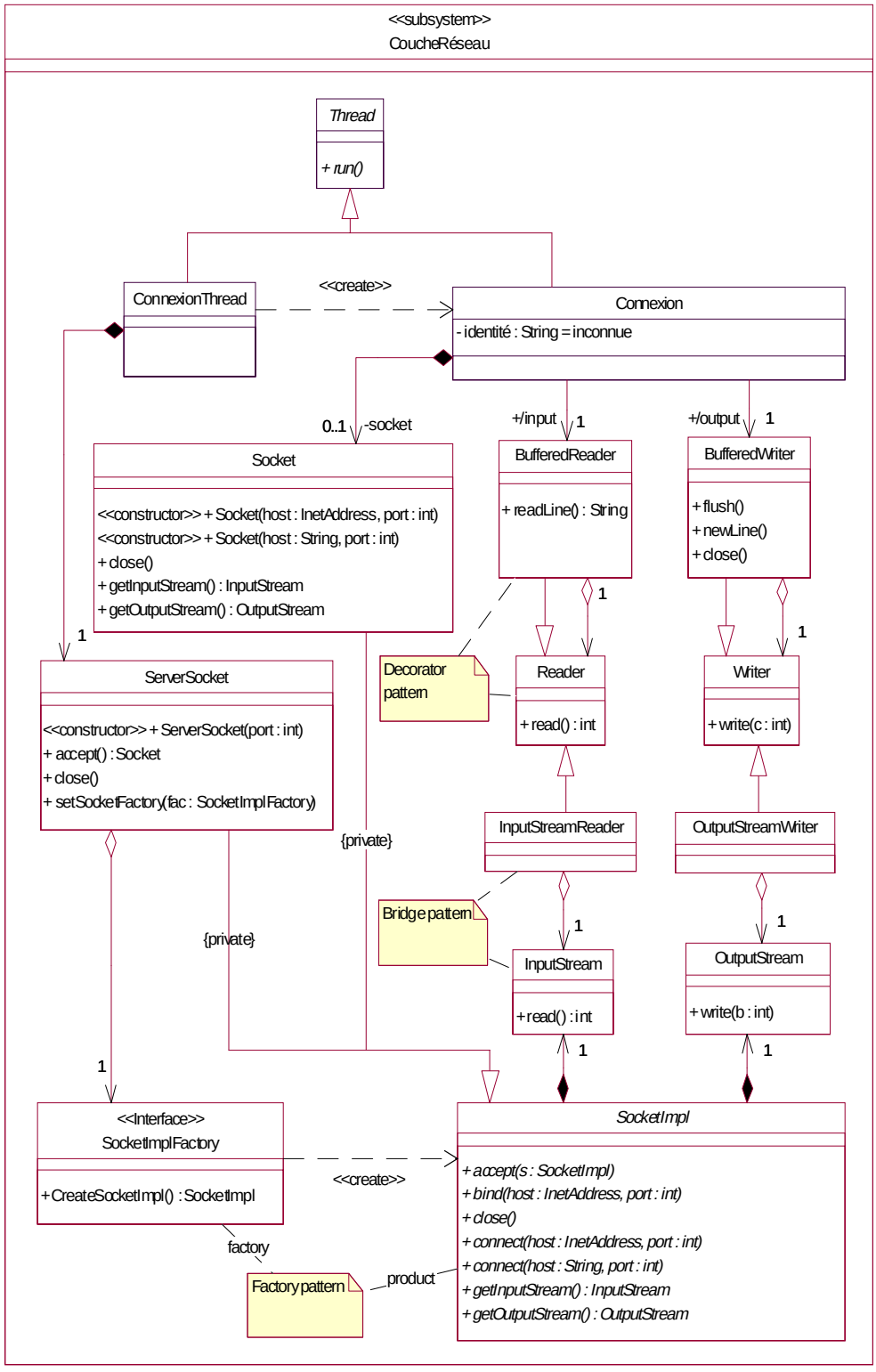


FIG. 2.10 – *Diagramme de conception des réunions virtuelles*

- Les caractéristiques comportementales (*Behavioral Features*), qui définissent les requêtes auxquelles les instances de ce classificateur seront capables de répondre :
  - Les *opérations* et *méthodes*, qui font l'objet de la section 2.6.3.
  - Les *récepteurs de signaux*, étudiés à la section 2.6.4.
  - Les *invitations*, étudiées à la section 2.6.5.

2.6.1 Accès aux caractéristiques d'une classe

Chacune des caractéristiques déclarées au sein d'une classe donne un point d'entrée qui peut être utilisé par ses clients. Il est possible de restreindre les autorisations d'accès :

- De manière globale pour tous les clients, en utilisant la propriété de visibilité $\{visibility = [public|protected|private]\}$ de la caractéristique. Ces différentes valeurs pour la visibilité sont représentées respectivement par les signes $+$, $\#$, et $-$ placés devant la caractéristique concernée. Nous avons déjà utilisé cette possibilité dans la section 2.3.3.
- En accordant des permissions (*Permissions*) d'accès spéciales à certaines classes clientes précises. Une permission a ainsi une sémantique similaire au mot clé *friend* du langage C++, ou à l'exportation sélective que l'on retrouve dans le langage Eiffel [78].
- En forçant un client à utiliser une interface donnée pour accéder au serveur, en attachant l'interface à l'extrémité de l'association permettant de contacter le serveur. L'accès se fait alors selon les caractéristiques de l'interface, et non plus directement selon les caractéristiques de la classe qu'elle abrite (nous reviendrons sur cette possibilité intéressante à la fin de la section 4.3.1).

2.6.2 Accès aux Attributs

En rendant certains de ses attributs publics, ou en utilisant le mécanisme de Permission, une classe permet à tout objet de classe quelconque (ou seulement des classes explicitement autorisées par une Permission) d'accéder directement à ces attributs.

Toutefois, cette forme simple (pour ne pas dire rudimentaire) de communication pose un certain nombre de problèmes :

- La lecture directe d'un attribut est fortement déconseillée car l'objet serveur n'a alors aucun contrôle sur ces accès et risque donc de fournir à son insu des informations incohérentes.
- L'écriture directe d'un attribut est encore plus dangereuse, car elle risque fort de placer l'objet serveur dans un état incohérent le rendant ultérieurement inutilisable.

Ces problèmes sont dus au fait qu'il est impossible d'identifier les responsabilités respectives du client et du serveur dans le cas des accès directs aux attributs. En effet, les assertions permettant de définir ces contrats ne peuvent être précisées que pour des routines, et non pas pour des attributs. Il convient donc d'encapsuler les accès externes aux attributs d'un objet à l'aide de routines "accesseurs", que l'on munira des contrats nécessaires pour assurer la cohérence des accès. Ceci nous amène naturellement à la deuxième forme de communication.

2.6.3 Appels d'opérations

Chaque classe définit un certain nombre d'opérations que ses clients peuvent appeler. Comme toutes les requêtes que peut recevoir un objet, un appel d'opération est accompagné d'*arguments* dont les valeurs respectives sont liées aux paramètres formels de l'opération (les types respectifs doivent bien sûr être compatibles). Les actions (voir section 2.8) effectuées lors d'un appel d'opération sont attachées à la méthode (le "*body*") réalisant l'opération. L'opération déclare la signature, et sa réalisation peut être redéfinie dans les sous-classes à l'aide de méthodes alternatives (par contre, UML n'autorise pas la redéfinition de la signature elle-même).

Les appels d'opérations sont normalement "synchrones", c'est-à-dire qu'ils ne passent pas par la file de messages éventuellement attachée à l'objet récepteur de l'appel si celui-ci est *actif*. Ces appels sont simplement traîtés comme les appels de routines "classiques" que l'on retrouve dans les langages à objets. C'est donc le mécanisme de communication le plus simple, et celui qui sera utilisé pour faire des requêtes vers des objets passifs.

2.6.4 Signaux

Une classe *active* peut aussi définir des récepteurs (*Reception*) pour certains signaux, déclarant ainsi que les objets de cette classe sont à même de recevoir ces signaux. La réception d'un signal par un objet se traduit par un événement-signal (*SignalEvent*) déposé dans la file d'attente de l'objet pour traitement ultérieur (la sémantique associée aux objets actifs sera étudiée à la section 4.6).

Le client n'est pas bloqué lors de l'émission et peut poursuivre sans attendre son exécution L'envoi de signal est donc un mécanisme de communication "asynchrone" et non-blocant. On parle aussi de flot de contrôle *plat* ("flat flow of control"). C'est le mécanisme le plus souvent adopté pour la communication avec un objet actif (communication de type "boîte aux lettres").

2.6.5 Invitation

Il existe un troisième mécanisme de communication, qui a la forme d'un appel classique, mais une sémantique plus proche d'un envoi de signal : Une *invitation* représente une demande de rendez-vous avec un autre objet qui doit être un objet actif. La communication est similaire à l'émission d'un simple signal, mais dans le cas d'une invitation, le client -courtois- se bloque en attendant que le serveur puisse répondre à sa requête. Lorsque le serveur a terminé de traîter la requête, les résultats éventuels sont transmis au client, et un terme est mis au rendez-vous (cette sémantique est très proche de la sémantique des rendez-vous dans le langage Ada95 [55]). Les deux intervenants peuvent alors reprendre leurs exécutions respectives.

Notons que la notion d'invitation n'est pas présente dans la version actuelle de UML. C'est une proposition du groupe de travail sur l'*Action Semantics*, faite dans le but de clarifier la sémantique des différents mécanismes de communication. Le simulateur UML que nous présenterons dans le chapitre 4 ne prend pas en compte ce nouveau mécanisme de communication.

2.6.6 Exceptions

Les exceptions sont un autre type de signal particulier. contrairement à l'émission d'un signal, la levée d'une exception se fait sans préciser de destinataire particulier. L'exception est propagée suivant des règles précises, jusqu'à ce qu'un gestionnaire d'exception approprié soit atteint.

Une caractéristique comportementale qui est susceptible de lever une exception peut le déclarer statiquement (UML ne précise pas si cette déclaration est obligatoire, comme c'est le cas en Java). Cette possibilité correspond à la clause *"throw"* d'une routine dans le langage Java qui permet d'indiquer la liste des *"checked exceptions"* que la méthode peut lever.

2.6.7 Émuler des appels d'opérations non-blocants à l'aide de signaux

Nous venons d'étudiés les mécanismes de communications disponibles dans UML. Nous avons notamment vu que les appels d'opérations étaient synchrones et que les envois de messages étaient asynchrones. Cette section a pour but de montrer comment on peut émuler des appels d'opérations asynchrones en utilisant des signaux, et ceci de manière transparente pour celui qui écrit la spécification.

2.6.7.1 De l'utilité des patrons de conception

En UML, les communications avec un objet actif sont généralement asynchrones. Les messages émis par les clients d'un objet actif ne sont pas exécutés immédiatement.

Ils sont placés dans une file de messages. L'objet actif fonctionne selon un mode cyclique : Au début de chaque cycle (appelé aussi *Run-To-Completion step*), un message est sélectionné depuis cette file et devient le message courant que l'objet va traité au cours de ce cycle.

Il existe un moyen élégant pour cacher cette complexité aux clients et rendre cette communication par envois de messages complètement transparente, comme s'il s'agissait d'une communication classique par appels de routines. Cela se réalise à l'aide d'une combinaison de deux patrons de conception classiques :

- Le patron de conception *Command* est utilisé pour associer à chaque opération de l'interface une classe de message correspondante. Les attributs de la classe de message servent à stocker les valeurs des paramètres de l'appel durant le transit par la file.
- Le patron de conception *Proxy* est utilisé pour masquer la création et l'envoi de messages. On intercale entre le client et le serveur un proxy qui a exactement la même signature (i.e. interface) que le serveur. Le proxy implémente chacune des opérations par une méthode qui consiste à créer une instance de la classe de message appropriée et à initialiser chacun de ses attributs par la valeur du paramètre correspond, avant d'envoyer le message vers la file du serveur.

Ainsi, le client appelle ce qu'il croit être le serveur. Le proxy crée le message, le place dans la file du serveur, et rend immédiatement le contrôle au client. Le message envoyé sera traîté plus tard par le serveur.

2.6.7.2 Impact sur le déploiement initial

Les transformations décrites ci-dessus doivent se reflèter dans les diagrames de déploiement du système (étudiés en section 2.9). Ces diagrammes montrent comment le système est configuré, en terme d'objets et de liens entre ces objets. Des objets proxies doivent être intercalés, en ajoutant les liens appropriés entre clients, proxies, et serveurs.

2.7 Gérer la concurrence

2.7.1 Les objets actifs

En UML, la concurrence est introduite par les objets actifs, c'est-à-dire les objets possédant leur propre flot de contrôle et donc capables d'initier des activités. Le fait qu'un objet est actif est indiqué par la propriété $\{isActive[= true]\}$ attachée à sa classe.

Les machines à états ont été adoptées dans UML afin de décrire le comportement des objets actifs La notion d'objet actif rejoint donc la notion de *processus* que l'on trouve dans des langages de description de *protocoles* comme LOTOS ou SDL.

Dans ce contexte d'objets actifs, les machines à états servent à décrire comment les objets réagissent aux événements qu'ils reçoivent et la manière dont ils échangent des messages pour se synchroniser. L'événement déclencheur d'une transition correspond donc à la réception d'un signal ou au début d'une invitation, notions que nous avons étudiées aux sections 2.6.4 et 2.6.5, respectivement.

En réaction à ces événements, les objets actifs exécutent les actions qui sont attachées en tant qu'effet de la transition. Ces actions peuvent comporter des communications avec d'autres objets (sous forme d'appels d'opérations, d'envois de signaux, ou encore d'invitations).

Bien qu'il ne soit syntaxiquement pas interdit d'attacher aussi une machine à états à un objet passif, il semble que les auteurs d'UML n'aient initialement pas envisagé cette possibilité. Cependant, associer des machines à états à des objets passifs revêt un intérêt non-négligeable, la notion de contrôle et celle d'état étant orthogonale. L'exemple de la machine à états de la classe Réunion vue à la section 2.4.5 en est la parfaite illustration. Autoriser la description d'objets passifs par des machines à états introduit malheureusement quelques problèmes sémantiques non triviaux à résoudre que nous mentionnons dans la section qui suit.

2.7.2 Objets passifs

Il existe plusieurs points assez flous quant à la sémantique des machines à états de UML lorsqu'elles sont attachées à des classes d'objets passifs :

- Un objet passif peut-il possèder une file de messages ? Si oui, alors la sémantique s'apparente aux "Controlled Types" d'Ada95 [55]. Sinon, alors la sémantique des gardes devient celle de pré-conditions qui doivent toujours être satisfaites si le système est correct.
- Peut-on faire une requête asynchrone à destination d'un objet passif ?
- La machine à états d'un objet passif peut-elle armer des *timers* ? Si oui, quelle *thread* prendra le contrôle lors de l'expiration du délai ?

Le plus raisonnable est encore d'éviter ces constructions problématiques, et de se contenter de donner aux gardes des transitions une sémantique de pré-conditions.

2.7.3 Récursivité et réentrance: la concurrence intra-objet

Alors qu'un objet actif contrôle parfaitement les traitements qui lui sont délégués (puisque ces traitements sont toujours sérialisés par la file de l'objet et exécutés par le flot de contrôle propre à l'objet), ce n'est absolument plus le cas pour les objets passifs, à qui le flot de contrôle de l'appelant est transmis lors de la requête afin d'exécuter le traitement correspondant (on parle alors de *nested flow of control*).

Ainsi, si plusieurs objets actifs envoient chacun directement ou indirectement une requête à un même objet passif tiers, il se peut que les traitements associés aux requêtes soient exécutés “en même temps”. Plusieurs flots de contrôle exécuteront alors simultanément des actions au sein d’un seul et même objet passif, ce qui pose alors le délicat problème de l’accès concurrent à des données partagées entre tous les flots de contrôle : il en va ainsi des variables globales bien sûr, mais aussi et surtout de tous les attributs de l’objet passif ainsi que des objets accessibles via des associations partant de cet objet.

Un problème similaire souvent négligé est le problème de la pseudo-concurrence intra-objet introduit par les *callbacks* inter-composants (problème mentionné dans [97]). Ainsi, pendant un traitement correspondant à l’une de ses opérations, un objet peut être conduit à appeler une opération d’un autre objet. Il se peut alors que cette seconde opération rappelle à son tour une opération du premier objet, ce qui fait que deux opérations sont alors en cours d’exécution en même temps sur ce même objet. Même s’il s’agit d’un seul flot de contrôle imbriqué (ou ici, “replié”), le même type de problèmes peut survenir que pour des accès réellement concurrents si les opérations n’ont pas été conçues en anticipant cette possibilité de réentrance. Ce genre d’appels réentrants est très fréquent en programmation objet. L’utilisation de *framework* repose notamment en grande partie sur cette possibilité.

UML propose un moyen pour contrôler les accès concurrents. Ce contrôle s’applique au niveau de chaque opération, à l’aide de la propriété suivante $\{concurrency = [concurrent|guarded|sequential]\}$.

concurrent signifie qu’aucune précaution particulière n’est nécessaire. L’opération fonctionnera correctement même si elle est appelée simultanément par plusieurs clients concurrents.

guarded signifie que l’opération est protégée par un verrou (similaire au «synchronized» de Java [46]), toute nouvelle requête restant bloquée tant que la précédente n’est pas terminée. Cela garantit l’absence d’accès concurrents, mais peut provoquer des blocages (*deadlock*) dans certaines circonstances. UML ne spécifie malheureusement pas si les verrous sont réentrants ou non (en Java, un verrou acquis par une *thread* ne sera pas bloquant si cette même *thread* essaye de se réapproprier le verrou lors d’un appel réentrant, ce qui autorise donc les *callbacks*). De plus, il n’y a qu’un seul verrou pour un objet donné qui protège l’ensemble des opérations marquées `guarded`, ce qui n’offre pas la granularité très fine parfois souhaitable. Rien n’est précisé pour les opérations de niveau classe (les opérations dites *static* dans certains langages comme C++ ou Java) : Y a-t-il un verrou de classe en plus des verrous d’instances, comme en Java ?

sequential signifie que l’effet ou le résultat de l’opération n’est garanti que si les accès sont correctement séquentialisés. L’objet accédé n’utilise donc pas de verrou. La synchronization est de l’entière responsabilité des clients, qui doivent se coor-

donner pour leurs accès. Si des accès concurrents survenaient malgré tout, alors le système doit être considéré comme incorrect.

À noter que la sémantique “run-to-completion” d’un objet actif (voir section 2.6.7) a le même effet qu’un verrou, et donc toute réentrance au travers d’un objet actif sera blocante !

La possibilité que des blocages ou des accès concurrents à des opérations `sequential` se produisent constitue une erreur dans la spécification. Une telle erreur doit pouvoir être détectée par vérification de la spécification afin d’être corrigée avant de passer à la phase d’implantation. La section 3.3.1 décrit des techniques de vérification formelles qui permettent de détecter ce genre d’erreurs.

2.8 Spécification opérationnelle : Les actions

Une Procédure est un ensemble d’actions prenant des objets en entrée et fournissant des objets en sortie. La notion de Procédure vient de la proposition de l’*Action Semantics*. Elle remplace la notion d’*ActionSequence* présente dans la version officielle d’UML.

UML possède un moyen de représenter graphiquement les flots de données et de contrôle entre les actions : Le diagramme d’activité.

En effet, il existe un type spécial d’état-activité appelé activité d’action (ActionState) qui se représente comme un état englobant une action. Les objets manipulés par les actions (ObjectFlowState) se représentent par de classiques rectangles, reliés aux actions produisant ou consommant ces objets.

Il est aussi possible de représenter ces actions sous forme textuelle, en utilisant une notation proche d’un langage de programmation classique.

Une procédure ainsi décrite peut être associée au service de l’objet qu’elle réalise de plusieurs manières.

2.8.1 Procédure attachée à une méthode

Une procédure peut être attachée à une méthode (*Method*) représentant la réalisation d’une opération. La méthode réalisant une opération peut être (re)déclarée dans une sous-classe de classe déclarant initialement l’opération. La sélection de la méthode adéquate, et donc de la procédure qui sera finalement exécutée, se fait suivant les règles de liaison dynamique habituelles. Ce cas simple d’association entre opération et procédure via une méthode est l’équivalent du corps d’une routine dans un langage de programmation classique.

2.8.2 Procédure attachée à une transition d'une machine à états

Alternativement, une procédure peut être attachée à une transition, et représente l'effet que la transition aura sur le système lorsqu'elle sera déclenchée par un événement.

2.9 Les diagrammes de composants et de déploiement

La figure 2.11 montre comment un système complet clients/serveur de réunions virtuelles se déploie sur un réseau. Dans cette configuration, trois connexions sont établies.

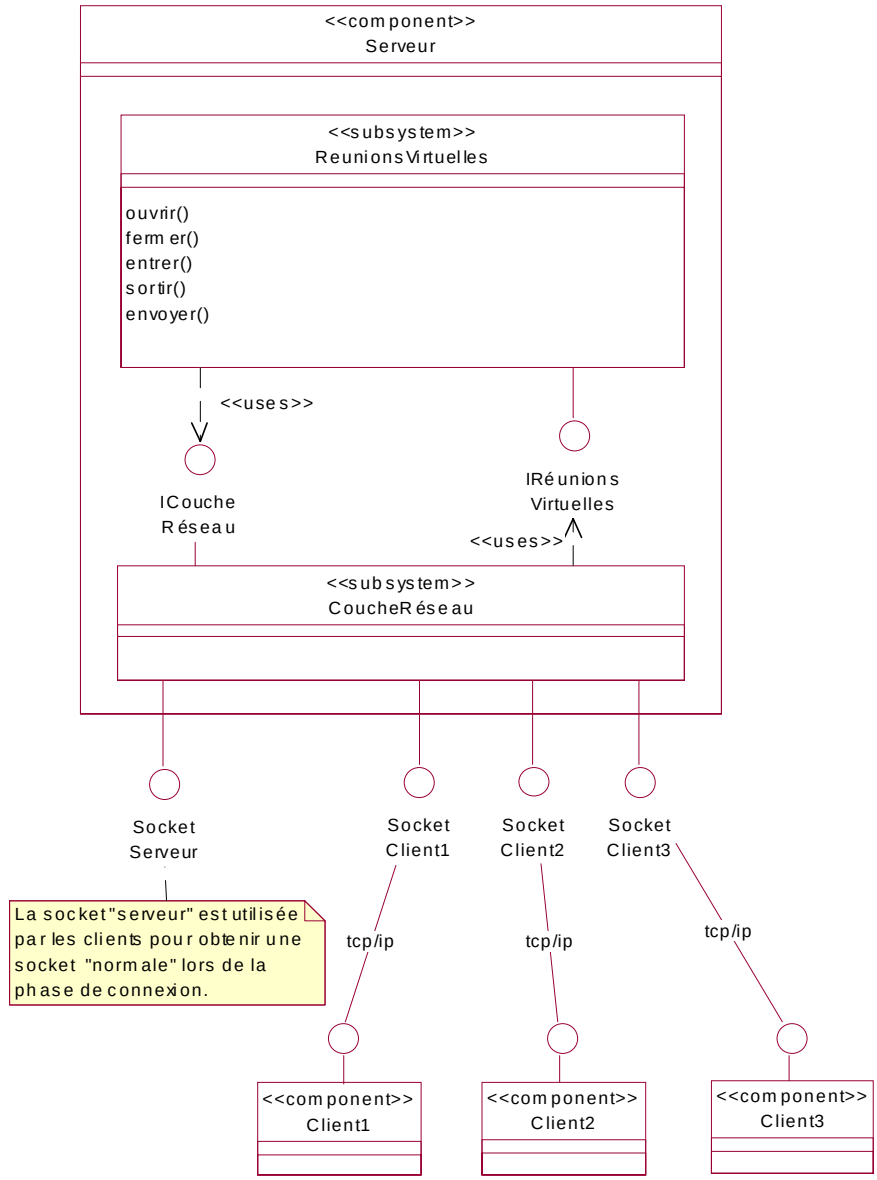


FIG. 2.11 – *Diagramme d'implémentation des réunions virtuelles*

Chapitre 3

Techniques Formelles avec UML

3.1 Introduction

Ce chapitre a pour objectif de montrer comment les techniques formelles introduites à la section 1.3 peuvent être mises en œuvre au sein d'un processus de développement basé sur UML. Nous avons vu lors du chapitre précédent comment spécifier un système en utilisant la notation UML. La spécification est au cœur du processus de développement. Elle va permettre :

- de s'assurer que les besoins exprimés initialement, le plus souvent de manière informelle, ont été bien compris, et qu'aucun cas n'a été oublié. La faculté de pouvoir simuler la spécification est alors particulièrement utile.
- de guider la phase d'implémentation, que celle-ci soit réalisée automatiquement à l'aide d'un générateur de code, ou bien manuellement par des développeurs qui suivront - au mieux - ses instructions.
- enfin, de formaliser le processus de test, en fournissant une référence formelle grâce à laquelle il sera possible d'établir ou d'infirmer la *conformité* d'une implantation par rapport à la spécification.

Le premier point est une étape indispensable dans le développement de logiciels complexes. Outre la complexité intrinsèque au système, notamment lorsqu'il est fortement concurrent, l'écart très important entre l'expression initiale des besoins (souvent sous forme de cas d'utilisation graphiques assez informels) et la spécification finale fait qu'il est parfois difficile d'être certain que le système spécifié fera bien ce que l'on attend de lui. À cette fin, des techniques complémentaires peuvent être utilisées :

- La simulation interactive permet d"'animer" la spécification, en déroulant son comportement afin d'observer ce qui se passerait. Des problèmes ou des oublis peuvent ainsi être détectés au plus tôt. L'animation permet aussi tout simplement de mieux comprendre le système que l'on est en train de spécifier, et d'éclaircir certains besoins exprimés de manière ambiguë.

– La vérification offre la possibilité de s'assurer qu'une certaine propriété - traduction formelle d'un besoin ou d'une contrainte initialement exprimés informellement - est toujours satisfaite par la spécification du système (propriétés de sureté, propriétés de vivacité, absence de blocage, etc). L'ouvrage collectif [35] propose une introduction complète et accessible aux techniques de vérifications formelles.

Enfin, le test est une étape indispensable, particulièrement dans le contexte d'une approche par composants logiciels. De plus en plus souvent en effet, les systèmes logiciels ne seront plus développés complètement, ex-nihilo. Ils seront (et sont déjà dans une large mesure) élaborés par assemblage de composants fournis par des tiers, disponibles dans de vastes collections ou bibliothèques de composants. Dès lors se pose le problème crucial d'établir le degré de confiance que l'on peut avoir dans un composant que l'on n'a pas réalisé soit même, et qui promet de se conformer à une spécification publiée, tout en restant le plus souvent fermé et opaque à toute inspection directe, pour des raisons commerciales évidentes. Une grande partie de ce chapitre sera donc consacrée au test fonctionnel, aussi appelé "test boîte noire", expression qui traduit bien l'opacité d'une implantation fournie par un tiers. Il existe cependant d'autre approche complémentaire du test (test boîte blanche, test de performance, test de charge), que nous ne traiterons pas ici.

3.2 État de l'art des techniques formelles appliquées à UML

Cette section a pour but de faire un inventaire le plus complet possible des travaux (définitions de sémantiques particulières, techniques, ou encore outils) permettant d'appliquer des techniques ayant une base formelle à des spécifications UML.

3.2.1 Génération de tests

Il n'existe qu'assez peu de publications traitant explicitement du problème de la génération de tests à partir de spécifications UML.

Fleisch [38] présente une méthodologie de test qui s'appuie sur des diagrammes décrivant la structure du système en utilisant la notation ROOM et sur un ensemble de cas d'utilisation accompagnés de scénari sous forme de diagrammes de séquences UML. Les diagrammes décrivant le modèle sont compilés en un modèle de simulation. Pour un scénario donné, on stimule le modèle de simulation avec les événement d'entrée et on compare la trace obtenue avec le scénario complet. Cependant, la fonction de comparaison de traces utilisée est trop naïve, en ce sens où elle ne prend pas en compte les autres événements indépendants pouvant provenir du système du fait notamment

de la concurrence. L'approche est donc limitée aux systèmes ayant des réactions complètement déterministes et séquentielles.

Dans [40] une technique est proposée pour générer des tests à partir de cas d'utilisation. Les cas d'utilisation (avec leur pré et post conditions) sont transformés en statecharts. Ces statecharts sont alors eux-mêmes réécrits dans un langage de planification, comme ensemble de contraintes. L'outil graphplan permet de générer des tests en résolvant les contraintes de planification.

Offutt [80] présente une technique permettant de générer des jeux de données permettant de tester les différentes conditions apparaissant dans les gardes des transitions des machines à états. L'article [2] propose d'utiliser les diagrammes de collaboration pour tester certaines propriétés des routines décrites à l'aide de ces diagrammes UML, à l'aide d'analyses classiques de flots de données et de contrôle.

Nous souhaitons pour notre part générer des tests permettant de vérifier la conformité d'une implantation vis-à-vis de sa spécification (test fonctionnel, de type boîte noire). Notre approche, que nous présenterons dans la section 3.3.4, ressemble un peu à la première approche mentionnée ci-dessus. Au contraire de celle-ci, les tests sont générés automatiquement, et l'algorithme de synthèse des cas de tests a été spécialement conçu pour traiter des systèmes fortement concurrents et non-déterministes.

3.2.2 Approches traductionnelles de la formalisation d'UML

De multiples techniques et algorithmes ainsi que plusieurs outils performants ont déjà été développés afin de répondre à des besoins similaires en techniques formelles, notamment dans le domaine spécifique des télécommunications. En général, ces outils sont associés à un langage de spécification dédié qu'il est nécessaire d'adopter pour les utiliser.

Comme nous l'avons déjà mentionné à la section 1.3.4, une approche dite "traductionnelle" peut donc sembler assez séduisante. Traduire un sous-ensemble d'UML vers un de ces langages permet en effet à la fois :

- de donner indirectement une sémantique au sous-ensemble en question;
- d'utiliser les techniques formelles offertes par l'outil associé.

Un certain nombre de travaux publiés récemment ont choisi une approche traductionnelle afin de permettre l'utilisation de techniques formelles avec UML. Dans la majorité des cas, ces travaux s'attachent plus particulièrement à la partie statecharts de UML.

3.2.2.1 Traduction vers Promela

Ainsi, il a été montré que les statecharts de UML pouvaient au moins partiellement être traduits vers le langage Promela, qui est le langage de spécification de l'outil SPIN [53, 54]. Ce langage permet en effet de décrire des automates communicants,

et SPIN permet de vérifier des propriétés de logique temporelle à partir de ces automates. Cette approche a notamment été adoptée par [83], [73] et [72]). Comme c'est le cas avec la plupart des approches existantes, les notions de références polymorphiques et de création dynamique d'objets ne sont pas prises en compte par ces traductions, ce qui les limite à des spécifications relativement statiques (SPIN n'autorise l'usage de ces constructions dynamiques que depuis peu, grâce à une extension nommée dSPIN [29]).

3.2.2.2 Traduction vers SMV

[70] présente une approche permettant de faire du model-checking à partir des statecharts de UML, grâce à une traduction en SMV [77]. Malheureusement, cette approche ne considère qu'un seul statecharts, en isolation, sans envisager de communications inter-objets, et sans intégrer les statecharts au reste du modèle objet. Surtout, il n'est tenu aucun compte des particularités des machines à états spécifiques à UML et qui les distinguent des statecharts classiques, ce qui en limite grandement l'intérêt.

3.2.2.3 Traduction vers LOTOS

L'article [17] aborde la traduction d'un sous ensemble de UML (en fait, du langage OBLOG sous-jacent à l'AGL du même nom) vers le langage de spécification LOTOS [57]. Cette traduction permet d'utiliser la boîte à outils CADP [36, 37, 42] sur les spécifications OBLOG. Cependant, les notions de références, de liens dynamiques entre objets, et de création d'objets ne sont pas prises en compte par la traduction, ce qui limite cette approche à des spécifications relativement statiques.

3.2.2.4 Traduction vers SDL

L'outil ObjectGeode contient un module permettant de compiler les statecharts de UML en processus du langage SDL. Il est ensuite possible de simuler le comportement de la spécification. Les deux langages sont en fait assez proches, ce qui rend cette traduction possible. Notons d'ailleurs que la dernière révision de SDL (adoptée en 2000) est définie en tant que *profile* (i.e. "spécialisation") de UML, rendant encore plus flagrante la convergence entre les deux langages.

3.2.3 Utilisation de Réseaux de Petri

[84] propose de modéliser les aspects dynamiques d'UML à l'aide de réseaux de Petri. Un aspect original de ces travaux est une méthode pour décomposer un sous-système en collaboration d'objets plus petits en analysant les "invariants de place" du réseau de Petri du sous-système (zone du réseau comportant un nombre constant de jetons). Les réseaux de Petri permettent aussi de représenter la concurrence intra-objets, ce qui est assez difficile à faire en UML. De plus, cette approche est bien outillée.

Par contre, la notion d'identité d'objet n'est pas représentée (par exemple par des jetons spéciaux), ce qui rend impossible toute connexion *dynamique* entre objets. La liaison dynamique due au polymorphisme n'est pas non plus prise en compte, pas plus que la création dynamique d'objets, ce qui, encore une fois, limite l'approche à des spécifications relativement statiques. Enfin, ces travaux présentent l'inconvénient majeur de reposer entièrement sur une extension non-négligeable de UML.

3.2.4 Les approches "méta"

Par opposition aux approches traductionnelles mentionnées ci-dessus, il existe un courant visant à formaliser UML à l'aide d'UML lui-même. Du moins, en modélisant le domaine sémantique d'UML en UML. C'est pourquoi nous avons décidé de regrouper ces travaux sous le qualificatif d"'approches méta".

On retrouve dans cette catégorie certains travaux initiés par le groupe *precise UML* (pUML) comme [23], ainsi que la proposition du groupe de travail sur l'*Action Semantics* [1] qui travaille à l'élaboration de "la" sémantique qui sera officiellement adoptée pour UML par l'Object Management Group. L'*Action Semantics* représente certainement l'un des efforts les plus significatifs visant à donner à UML les bases formelles permettant de lui appliquer avec succès des techniques formelles. Cependant, à l'heure actuelle aucun AGL pour UML n'implémente encore cette proposition.

L'approche que nous proposons au chapitre 4 fait aussi partie de cette catégorie, et possède de fortes similitudes avec l'*Action Semantics*, sans toutefois en avoir l'envergure. Néanmoins, un outil correspondant a été développé et est utilisable dès à présent.

3.2.5 Les problèmes inhérents à UML

3.2.5.1 Passer de l'informel au formel

Un problème majeur avec UML est de gérer la cohérence entre les nombreuses vues dont peut être composée une spécification. La plupart des travaux existants se concentrent sur une seule vue, le plus souvent la vue statecharts, sans vraiment essayer d'intégrer les différentes vues et les concepts qui les sous-tendent. Nous verrons à la section 4.3 qu'il est possible d'unifier plusieurs concepts importants d'UML normalement éparpillés entre plusieurs vues, afin d'assurer une meilleure cohérence des spécifications.

3.2.5.2 Le problème des fragments de code natif

UML est de plus en plus utilisé tout au long du cycle de vie, depuis l'analyse à un haut niveau d'abstraction jusqu'à la conception détaillée, très proche de l'implantation finale, par exemple sous forme de code dans un langage à objets comme C++ ou Java. C'est une des raisons pour laquelle beaucoup d'utilisateurs d'UML souhaitent pouvoir

intégrer dans la spécification des fragments de code écrits dans le langage cible qui sera utilisé pour l'implantation.

Certes, la plupart des outils associés aux langages de spécification formels mentionnés précédemment permettent aussi d'insérer des fragments de code (souvent en langage C) au sein de spécifications écrites dans des langages de plus haut niveau. Cependant, cette possibilité est souvent fortement contrainte, à la fois en terme de langage utilisable (qui est généralement le langage cible du compilateur pour le langage de spécification), et en terme d'environnement d'exécution dans lequel ces fragments doivent s'insérer (environnement qui correspond au *run-time* de l'outil).

L'article [44] présente un outil capable d'explorer le graphe des états d'une *implantation* d'un système concurrent réalisée en langage C afin de vérifier certaines propriétés telles que la non-violation d'assertions ou l'absence de blocage. On se trouve donc dans le cas extrême où le code C de l'implantation fait office de spécification. Pour cette raison, l'outil ne sauvegarde pas les états globaux du système (sauvegarder ces états lorsque la spécification contient du code natif est une tâche assez ardue, comme nous le verrons bientôt dans la section 4.7). Des techniques de réduction par ordres partiels [43] y sont aussi mises en œuvre.

L'approche que nous proposons au chapitre 4 permet d'inclure des fragments de code en langage natif. Nous proposons en section 4.7 des solutions adaptées à plusieurs langages cibles, et qui nécessitent peu de connaissances de la part de l'utilisateur quant à la manière dont le *run-time* de notre outil fonctionne.

3.3 UML et techniques formelles : Une approche intégrée

Il existe aussi des outils qui reposent sur un formalisme mathématique de plus bas niveau que les langages de spécification mentionnés à la section 3.2.2, et vers lequel les langages de spécification de plus haut niveau doivent préalablement être "compilés". C'est notamment le cas de la boîte à outils CADP [36, 37, 42] qui repose sur le formalisme des systèmes de transitions étiquetés, et pour laquelle il existe des passerelles depuis les langages SDL [21] et LOTOS [57]. En effet, ce formalisme se prête bien aux manipulations formelles comme la vérification et la synthèse de tests de conformité que nous souhaitons mettre à la disposition des utilisateurs de UML.

Notre but étant de faire se rapprocher la modélisation objet des techniques formelles par le biais de la notation semi-formelle UML, nous avons choisi de reprendre le formalisme des systèmes de transitions étiquetés (*Labelled Transition Systems* ou LTS en anglais). Nous avons ainsi réalisé un *front-end* UML pour CADP, et l'avons intégré au sein de l'AGL UMLAUT développé à l'IRISA. Grâce à l'intégration entre UMLAUT et CADP, ces techniques s'intègrent aisément dans le cycle de vie du logiciel, comme l'illustre la figure 3.1.

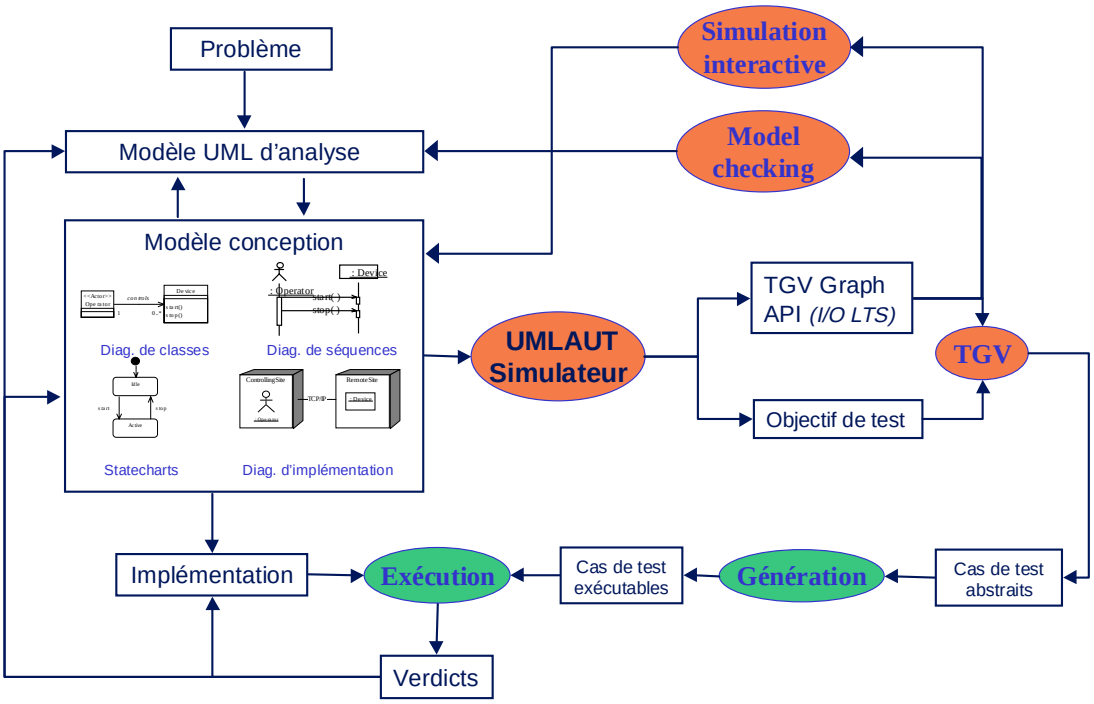


FIG. 3.1 – *Techniques formelles dans le cycle de développement*

Les techniques expliquées dans les sections qui vont suivre sont donc toutes basées sur les systèmes de transitions dont une définition est donnée ci-dessous :

Un LTS est un quadruplet $T = \{S, Obs, \longrightarrow, s_0\}$ où :

- S est un ensemble d'états ou configurations
- Obs est un alphabet fini d'actions observables, éventuellement partitionné en deux ensembles Obs_I et Obs_O afin de distinguer actions d'entrée et actions de sortie, distinction fondamentale pour l'activité de test.
- $\longrightarrow \subseteq S \times Obs \times S$ est la relation de transition
- et enfin s_0 est l'état initial du système.

La manière dont on obtient un tel système de transitions depuis une spécification UML ainsi que les restrictions que l'utilisation de ce modèle impose seront l'objet du chapitre 4.

3.3.1 Vérification

Établir la spécification formelle d'un système à partir des besoins exprimés par ses futurs utilisateurs est une tâche ardue. L'utilisation d'UML n'empêche pas l'existence d'un saut sémantique dans la manière utilisée pour décrire le système. Les exigences

sont le plus souvent exprimées de manière informelle, à l'aide de cas de tests accompagnés de texte en langage naturel.

3.3.1.1 Utilisation d'un model checker

Construire une spécification UML correcte impose donc de reformuler ces exigences de manière plus formelle. Certaines des propriétés attendues du système peuvent aussi être formalisées, notamment en utilisant une logique temporelle. L'utilisation d'un model-checker permet alors de vérifier que la spécification satisfait les propriétés formalisées. La boîte à outils CADP [42] propose ainsi deux model-checkers [75, 76] que l'on peut à présent utiliser pour des spécifications UML.

Cette formulation mathématique permet de mettre en lumière un certain nombre d'ambiguités ou de problèmes, qu'il faudra résoudre en communicant avec les utilisateurs.

Les propriétés formelles résultantes, exprimées en logique temporelle, sont donc plus précises, mais leur expression est très nettement plus abstraite et éloignée de la spécification UML de départ. Ceci peut entrainer un risque non-négligeable d'incompréhension entre les différents acteurs, voire même une réaction de rejet envers une notation jugée trop ardue à mettre en œuvre.

Notons qu'il est sans doute possible de réduire l'écart entre les propriétés en logique temporelle et la spécification UML en utilisant le mécanisme de *macros* offert par le model-checker *evaluator* de CADP. Nous n'avons cependant pas approfondi cette possibilité.

3.3.1.2 Vérifications des assertions

S'il n'est pas facile à l'heure actuelle d'exprimer des propriétés comportementales qui soient bien adaptées à des spécifications UML, certaines propriétés peuvent néanmoins être vérifiées dès à présent. Il s'agit des assertions classiques que sont les préconditions, postconditions et invariants, popularisées par le langage Eiffel [78], et exprimables en UML grâce au langage OCL (Object Constraint Language).

Ces assertions sont un cas particulier assez simple de propriétés. Les moments auxquels ces propriétés doivent être vérifiées sont connus. En effet, aucun opérateur temporel n'apparait explicitement, si ce n'est l'opérateur `@pre` de OCL, qui dénote la valeur d'une expression à l'état précédant immédiatement l'exécution d'une opération. Il serait néanmoins possible d'exprimer ces assertions au moyen de formules de logiques temporelles plus classiques, afin de mieux formaliser l'expression des instants en question (citons notamment les travaux sur le formalisme BOTL [31]).

Détecter les violations de préconditions et de postconditions ainsi que des invariants décrits en OCL "classique" est relativement aisé à réaliser. Les assertions OCL, qui doivent être atomiques et sans effet de bord, peuvent être facilement évaluées lors

de la construction du LTS. Toute violation d'assertion OCL résulte en une transition spéciale dont l'étiquette peut porter toute information nécessaire à l'identification du problème (nom de l'assertion, ou encore valeurs des variables participant à l'évaluation de l'assertion).

3.3.1.3 Contraintes de multiplicité

Le diagramme de classe définit implicitement un certain nombre de contraintes structurelles au travers des multiplicités associées aux associations. Les documents UML ne spécifient absolument rien en ce qui concerne la validité de ces contraintes. Il est évident que ces contraintes ne peuvent pas être constamment satisfaites. Dans certains états transitoires, comme par exemple les phases d'initialisations pendant lesquelles les objets à associer sont créés, il est impossible de satisfaire simultanément l'ensemble des contraintes.

L'utilisation d'une logique temporelle permettrait d'expliciter sans ambiguité les instants auxquels les contraintes structurelles doivent être satisfaites. Typiquement, une formule serait composée d'une partie temporelle et d'un atome représentant les contraintes structurelles à l'aide d'une expression écrite dans le langage OCL.

3.3.1.4 Contraintes de dimensionnement de files

Dans la plupart des applications, notamment dans le domaine des télécommunications, les ressources sont bien sûr limitées. Ainsi la taille des buffers et autres files de communications peut se révéler insuffisante, pouvant entraîner des pertes de messages. Le défi consiste alors à allouer judicieusement les ressources tout en s'assurant que certaines propriétés importantes resteront vérifiées. En rendant la taille des ressources visible et accessible depuis les formules en logique temporelle (et donc nécessairement dans le LTS sous-jacent), il devient possible d'écrire des formules mettant en jeu des contraintes sur les ressources.

Plus prosaïquement, il est possible de faire apparaître le débordement d'une file de communications comme une transition spéciale du LTS de la spécification menant dans un état puit identifiant cette erreur. On utilise alors le model-checker pour déterminer si cette transition peut être tirée sous certaines conditions ou non. C'est l'approche simple que nous avons adoptée, et qui sera expliquée dans le chapitre suivant qui traite du simulateur capable de construire le LTS d'une spécification UML.

3.3.2 Simulation interactive

Parfois, il est difficile d'appréhender le comportement d'une spécification par la seule analyse des propriétés qu'elle vérifie. La *simulation interactive* apporte une aide essentielle pour la compréhension. Elle permet d'explorer le comportement de la spécification, en avançant d'état en état en choisissant les transitions que l'on souhaite

déclencher. On peut donc considérer la simulation interactive comme une sorte de *debugger* fonctionnant au niveau de la spécification.

3.3.3 Raffinements entre spécifications UML

En plus de permettre la vérification de propriétés au sein d'une spécification UML, le fait de disposer d'un modèle sous forme de LTS permet de vérifier si une spécification UML donnée vérifie une certaine propriété vis-à-vis d'une autre spécification UML. Il existe en effet de nombreuses *relations* définies entre LTS (simulations, bi-simulations faibles ou fortes, testing-equivalence, etc). Ces relations permettent de savoir si deux spécifications sont équivalentes selon un certain critère ou type d'abstraction.

La notation UML permet de représenter des relations de raffinements entre spécifications. Ces relations de traçabilité peuvent indiquer qu'une spécification est une version plus détaillée (raffinée) d'une autre spécification plus abstraite.

Grâce au modèle sous-jacent des LTS, il devient possible de définir (et de vérifier dans certains cas) la nature exacte de la relation de raffinement existant entre deux spécifications UML.

3.3.4 Test de conformité

Comme nous l'avons évoqué en introduction, pouvoir synthétiser des test de conformité à partir d'une spécification est une nécessité si l'on adopte un développement par composants logiciels. Cette section est consacrée à l'étude de la synthèse de test de conformité [59] à partir du formalisme des LTS introduits précédemment, dans un contexte UML.

Nous n'apporterons pas dans cette partie de nouveaux résultats en ce qui concerne la théorie de la synthèse de tests de conformité, pour laquelle il existe une importante littérature. Citons notamment les travaux de Brinksma [16], de Tretmans [99], de Phalippou [85], et de Jéron [64]. Nous nous contenterons d'en reprendre les principaux résultats.

Notre contribution principale réside dans la jonction entre ces travaux basés sur le formalisme mathématique des LTS et un environnement de développement orienté-objet basé sur le formalisme graphique UML. Nous avons ainsi réalisé une intégration de l'outil de synthèse de tests TGV [64] avec l'Atelier de Génie Logiciel UML-LAUT [67], offrant aux utilisateurs d'UML la possibilité de synthétiser des tests de conformité à partir de leurs spécifications UML.

3.3.4.1 Principes

L'outil TGV permet de générer des cas de test à partir d'une spécification et d'un *objectif de test*. La théorie sous-jacente à TGV se fonde sur les systèmes de transitions étiquetées pour lesquels entrées et sorties sont distinguées (IOLTS). Les modèles et algorithmes de TGV sont présentés en détail dans [64].

L'objectif de test guide la génération du cas de test pour produire des cas de tests ciblés, en restreignant l'exploration du graphe d'accessibilité aux seules transitions que l'on souhaite voir explorées.

Cela est réalisé au travers d'un produit synchrone entre le IOLTS représentant la spécification et celui représentant l'objectif de test. Ce produit synchrone forme un graphe dirigé acyclique, dont les nœuds sont décorés par des verdicts.

3.3.4.2 Architecture de test

Pour vérifier la conformité d'une implantation vis-à-vis de sa spécification, on la place dans un *environnement de test*. Le testeur communique avec l'implantation sous test (*Implementation Under Test*, IUT) par des entrées et des sorties. Du point de vue du testeur, les sorties (ce qu'il envoie à l'implantation) sont contrôlables, alors que les entrées provenant de l'IUT ne sont qu'observables. Les actions internes à l'IUT ne sont quant à elles pas observables.

Le but du test de conformité est de voir si l'IUT plongée dans son environnement de test se comporte "de la même manière" que sa spécification plongée dans le même environnement. Il est donc fait l'hypothèse que le comportement réel de l'IUT (plongée dans son environnement) est lui aussi modélisable sous forme d'un IOLTS (*hypothèse de test*). La relation de conformité est alors définie comme une relation entre deux IOLTS. C'est un cas particulier de relations entre IOLTS telles que mentionnées dans la section 3.3.3. La relation de conformité *io-conf* que nous utilisons stipule que pour toute trace σ dans le LTS de la spécification, les sorties de l'implantation après cette trace sont incluses dans les sorties de la spécification après cette trace (le blocage est considéré comme une sortie particulière, et peut être détecté à l'aide de timers).

Exprimé informellement, cela signifie que si la spécification dit qu'au moins une sortie aurait due être observable, alors une implantation conforme ne doit pas se bloquer. De plus, une implantation conforme ne doit pas émettre d'événements qui ne soient pas prévus par sa spécification.

3.3.4.3 Objectif de test

Selon [56], un objectif de test est un critère permettant de construire un cas de test pertinent. Dans le contexte du test de conformité à l'aide de TGV, un objectif de test est aussi un IOLTS permettant de guider l'exploration de l'espace des états de la

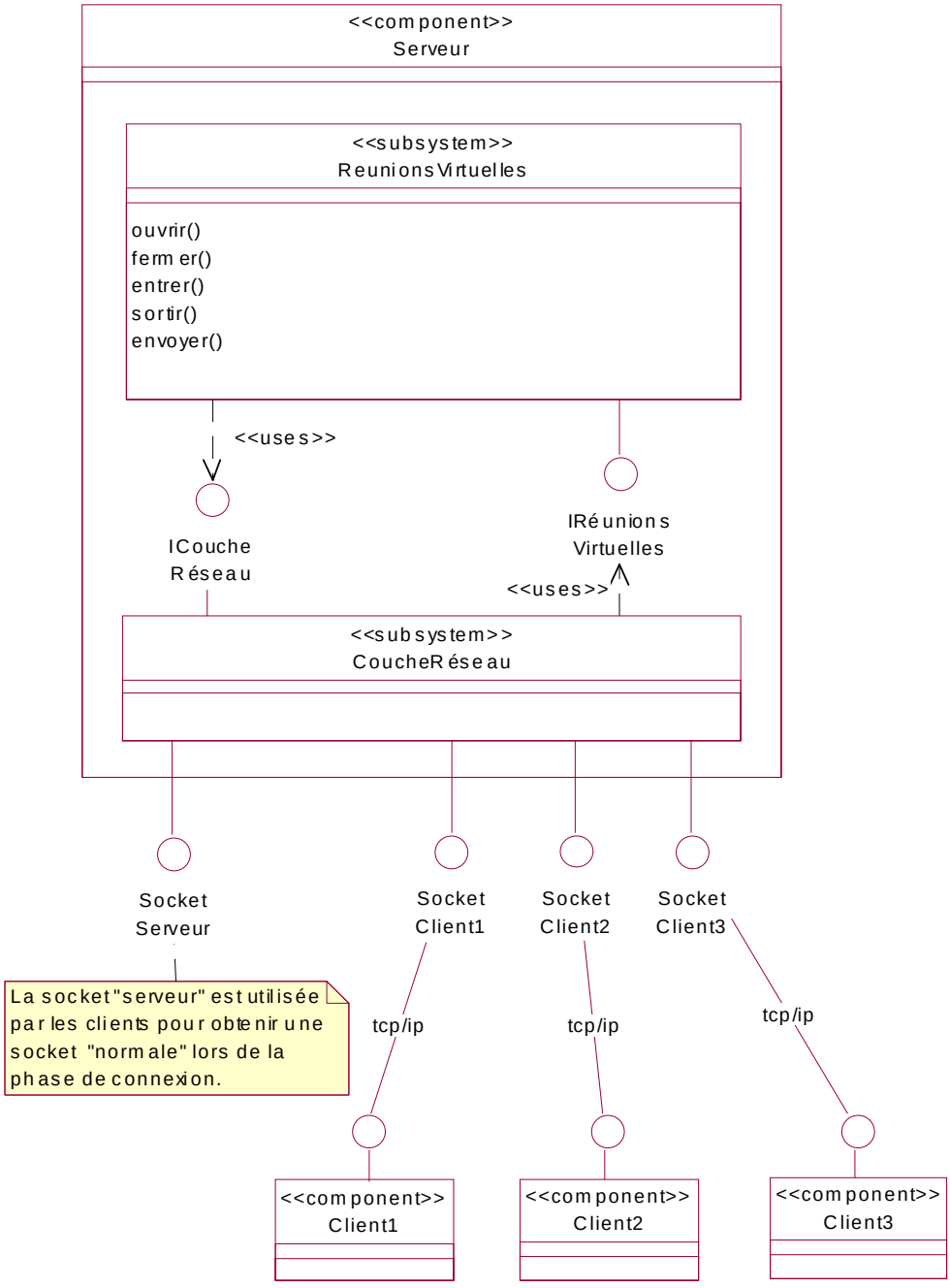


FIG. 3.2 – *Architecture de test pour les réunions virtuelles*

spécification en effectuant un produit synchrone avec celui-ci, à la volée, lors de sa construction.

Certaines transitions seront marquées “REFUSE”, ce qui a pour effet d’empêcher l’exploration des transitions correspondantes du IOLTS de la spécification, afin de limiter l’explosion combinatoire et donc le temps nécessaire à la synthèse du cas de test. Si lors de l’exécution du cas de test on ne peut empêcher l’implantation de tirer une transition marquée “REFUSE” par manque de contrôlabilité, alors le verdict sera “INCONCLUSIVE”.

3.3.4.4 Les objectifs de tests en UML

Pour pouvoir générer des tests, TGV a donc besoin d’objectifs. La spécification UML du système fournit déjà des objectifs de tests intéressants sous une forme abstraite : les cas d’utilisation. Un cas d’utilisation peut être décrit plus précisément par une collaboration et un diagramme de séquence représentant les interactions entre l’environnement et le système qui concernent directement la réalisation de ce cas d’utilisation. UMLAUT convertit ensuite ces interactions en IOLTS pour TGV.

3.3.4.5 Objectif de test avec valeurs instanciées

Le simulateur UML précédemment décrit construit le graphe d’atteignabilité de la spécification en utilisant des techniques d’énumérations exhaustives. Si l’on se rappelle que les étiquettes des transitions de ce graphe représentent des actions (qui peuvent être soit des action internes ou des actions d’interaction), alors cela signifie notamment que tous les noms (des événements) et toutes les valeurs (des variables ou des paramètres des messages) ont une valeur fixée, et ne sont plus représentés sous une forme symbolique.

Si l’on masque les actions internes inobservables, à une trace dans ce graphe d’atteignabilité correspond alors une séquence de stimuli telle que définie par un diagramme de séquences UML dit “de niveau instances”.

Aussi semble-t-il naturel de considérer les diagrammes de séquences UML de niveau instances comme l’artéfact UML le plus adapté lorsqu’il va s’agir de dériver des objectifs de tests pour TGV à partir d’une spécification UML.

3.3.4.6 Traitement de la concurrence

Notons cependant qu’il n’est nullement obligatoire de se limiter à des objectifs de test purement séquentiels. En effet, d’une part TGV prend en entrée un graphe, et non forcément seulement une séquence. D’autre part, les diagrammes de séquences de UML portent mal leur nom, puisqu’ils définissent non pas une séquence mais un ordre partiel

Il devient ainsi possible de prendre en compte des scénarios UML mettant en œuvre de la concurrence, et de générer pour TGV un graphe représentant la concurrence à l'aide d'entrelacements.

3.3.4.7 Traitement des choix

Des branchements peuvent apparaître dans le graphe d'entrée de TGV pour représenter un choix dans un scénario. Les diagrammes de séquences de UML permettent aussi de représenter les choix, qu'ils soient déterministes ou non. Ces choix apparaissent comme un point de décision sur la ligne de vie d'un objet d'où partent plusieurs messages gardés par des conditions qui peuvent ou non être mutuellement exclusives .

3.3.4.8 Actions internes, entrées, et sorties

Il est possible d'indiquer à TGV les actions du LTS qui doivent être considérées comme des entrées ou des sorties, et celles qui doivent être considérées comme des actions internes. C'est le rôle des fichiers `.io` et `.hide` qui peuvent accompagner le graphe d'entrée de TGV.

La notion de système ou de sous-système de UML permet de définir précisément les frontières d'un système, ce qui rend ces deux fichiers aisés à construire.

3.3.5 Représenter le cas de test généré par TGV en UML

3.3.5.1 Choisir le bon type de diagramme

TGV produit un graphe comme cas de test correspondant à l'objectif de test. Dans le cas où ce graphe ne comporte pas de boucle, il est possible de représenter le cas de test par un diagrame de séquence de niveau instances.

Les cas de test comportant des boucles sont plus difficiles à représenter en UML. Deux alternatives sont envisageables :

- Représenter le comportement du testeur par un statechart.
- Représenter le comportement du testeur par un diagramme d'activité.

Ces deux types de représentation autorisent en effet l'usage de boucles.

3.3.5.2 Les transitions implicites dans le cas de test

Tous les états intermédiaires du cas de test comportent implicitement des transitions sortantes "else FAIL", qui indiquent qu'un message non attendu représente une exécution caractéristique d'une implantation non conforme à la spécification. Mais UML ne permet malheureusement pas d'exprimer la notion de "n'importe quel autre événement".

Cela implique de définir une convention pour la réception d'événements non attendus par le statechart du testeur.

Une autre alternative à envisager est de représenter un cas de test aussi par un diagramme de séquences. Les problèmes de "else FAIL" se posent cependant aussi dans ce cas.

3.3.6 Exécution des tests

Une fois que l'on dispose d'une suite de tests, encore faut-il pouvoir appliquer ces tests à une implantation du système afin d'en vérifier la conformité. Cette section est donc dédiée à la description d'un environnement de test adapté au système des réunions virtuelles.

L'application réunions virtuelles prise dans son ensemble est un système distribué. L'outil TGV ne génère à l'heure actuelle que des cas de tests centralisés.

3.4 Contributions et conclusion

À l'heure actuelle, il existe peu de solutions permettant d'appliquer des techniques formelles à UML. La plus grande difficulté réside sans doute dans le fossé qui existe entre la notation UML et les notations plus formelles habituellement utilisées pour ces techniques. L'Atelier de Génie Logiciel UMLAUT [67] permet de construire des spécifications UML dont la cohérence inter-vues est garantie par construction, ce qui permet de leur donner une sémantique sous forme de systèmes de transitions étiquetés (cette sémantique est l'objet du chapitre suivant).

Grâce à cette sémantique et à la connexion entre UMLAUT et CADP qui en découle, il devient possible de :

- vérifier certaines propriétés sur une spécification (section 3.3.1)
- simuler le comportement de la spécification de manière interactive (section 3.3.2)
- générer des tests qui permettront de s'assurer de la conformité d'implantations vis-à-vis de la spécification (section 3.3.4).

Chapitre 4

Simuler des Spécifications UML

4.1 Introduction : Vers une sémantique pour UML

Ce chapitre a pour but de décrire les principes et la réalisation du simulateur qui va nous permettre de construire le système de transitions étiqueté qui est à la base des techniques et outils étudiés précedemment à la section 3.3. Les règles permettant de construire un LTS à partir d'une spécification UML constituent la *sémantique* de UML.

Si l'on suivait une approche dénotationnelle, une sémantique pour UML serait une fonction qui à toute spécification bien formée écrite en UML donnerait une *signification*, ici en terme de LTS.

L'approche que nous avons suivie est différente et plus opérationnelle : La fonction sémantique n'est pas explicitée directement. Nous donnons juste les règles décrivant les évolutions possibles d'un système en terme d'états globaux (des *snaphshots*) qui forment les nœuds du LTS et de transitions qui en forment les arrêtes.

En préambule, la section 4.2 explique comment une spécification UML se représente en terme de *syntaxe abstraite*. Nous expliquons notamment comment UML est défini à l'aide de son méta-modèle. Cependant, la syntaxe abstraite officielle de UML contient un nombre important de concepts redondants dispersés sur les différentes vues qu'offre la notation. La section 4.3 se propose d'unifier certains concepts redondants afin de simplifier et rendre plus cohérente la représentation abstraite d'une spécification UML. Grâce à cette syntaxe abstraite unifiée, il devient plus aisé de construire la représentation d'un état global, décrite dans la section 4.4. Un tel état global est un snapshot décrivant l'états de tous les objets du système ainsi que des connexions (liens) qui existent entre ces objets à un instant donné. Une originalité de l'approche est que nous avons aussi utilisé UML pour décrire la structure des états globaux. Enfin, la section 4.6 décrit la manière dont le système évolue en passant d'états globaux en états globaux. Une contribution importante est la prise en compte des actions décrites en code natif en section 4.7, où nous décrivons l'impact que cette caractéristique aura sur le LTS résultant. Nous concluons ce chapitre en donnant l'algorithme de construction

du système de transitions, en section 4.8.

4.2 Infrastructure et sémantique statique de UML

4.2.1 Une architecture à quatre niveaux

Contrairement à ses prédécesseurs (OMT, OOSE et la notation de Booch), la représentation graphique donnée par UML est sous-tendue par une syntaxe abstraite bien définie, qui se trouve être décrite elle-même en UML (récursivement, un peu comme un compilateur pour un langage classique serait lui-même écrit dans ce langage, et donc capable de se recompiler). La syntaxe abstraite de UML est ainsi donnée par une spécification UML un peu particulière appelée méta-modèle, qui donne la structure d'un arbre syntaxique abstrait pour une spécification UML. Cette définition "récursive" de la syntaxe de UML n'est pas un problème fondamental en soi, car seuls un faible nombre des concepts présents dans UML sont utilisés pour définir le méta-modèle :

- Les classes, pour définir la structure des nœuds de l'arbre syntaxique abstrait
- Les attributs, pour définir les propriétés attachées à chaque nœud
- Les associations, pour représenter les connexions entre ces nœuds.

Le méta-modèle est donc un schéma de construction pour tous les arbres syntaxiques abstraits possibles. Ainsi, toute spécification UML est une instance de ce méta-modèle.

Le méta-modèle étant lui-même défini par une spécification UML (certes un peu particulière), il peut donc être considéré comme une instance d'un *méta-méta-modèle*, qui serait isomorphe à lui-même, ou plutôt à un sous-ensemble de lui-même correspondant à sa partie centrale, puisque seuls un faible nombre de concepts (classes, attributs et associations) sont utilisés. Il n'est pas utile de remonter au delà dans les niveaux méta, le niveau n+1 étant dès lors isomorphe au niveau n. Il se trouve que le méta-méta-modèle officiel d'UML n'est pas une projection au niveau méta-méta de la partie "Core" du méta-modèle de UML. Le méta-méta-modèle en passe d'être utilisé est celui du MOF (Meta Object Facility), qui est systématiquement utilisé par l'OMG pour des raisons d'interopérabilité (l'utilisation du MOF pour UML nécessite cependant quelques alignements dans la définition d'UML, qui seront réalisées lors de la prochaine révision de la notation, UML 2.0). Ce méta-méta-modèle est toutefois très près d'être isomorphe à un sous-ensemble du méta-modèle UML. On pourra consulter [9] et [10] pour une discussion plus approfondie de ces concepts.

On retrouve ainsi la classique structuration en quatres niveaux :

M0 snapshot du système

M1 spécification

M2 méta-modèle

M3 méta-méta-modèle

avec toutefois quelques irrégularités, le niveau M1 permettant de représenter des modèles du niveau M0 grâce à la méta-classe Instance.

4.2.2 Sémantique statique

UML est essentiellement un langage graphique, qui permet de représenter une spécification à l'aide de diagrammes, dont il existe neuf sortes. Chaque sorte de diagrammes se concentre sur un point de vue ou aspect [69] spécifique du système, tel que sa structure statique, les collaborations entre ses constituants, ou encore les descriptions comportementales des différents objets pris séparément à l'aide de machines à états.

La manière exacte dont les différents diagrammes constituant une spécification UML se traduisent en arbre syntaxique abstrait (instance du méta-modèle de UML, comme nous l'expliquerons dans la section suivante) est définie de manière informelle dans le guide de notation de UML [89]. Nous ne nous intéresserons pas davantage à cet aspect dans cette thèse, sachant que l'outillage utilisé pour dessiner les diagrammes fournit par construction un arbre syntaxique censé traduire l'intention de l'utilisateur. Par la suite, nous nous placerons donc directement au niveau de l'arbre syntaxique abstrait, dont les nœuds sont des instances des classes du méta-modèle UML.

Comme souvent dès que l'on s'intéresse à un langage complexe, toute spécification syntaxiquement correcte n'est pas nécessairement bien-formée. C'est pourquoi on associe au langage un ensemble de règles ou contraintes qui doivent être vérifiées afin que la spécification soit acceptable. Cet ensemble de règles est ce que l'on appelle la sémantique statique du langage. La sémantique statique d'UML est partiellement définie par un ensemble de règles accompagnant le méta-modèle de UML et écrites à l'aide de l'Object Constraint Language (OCL)) [100]. Ces règles OCL sont énumérées dans le "guide sémantique" de UML ([89]). Toutefois, les règles ainsi définies dans la documentation officielle ne sont nullement suffisantes pour assurer qu'une spécification soit bien formée. L'intégration des différentes vues de UML est notamment à peine esquissée, et la section qui suit a pour but de combler certaines lacunes dans ce domaine.

4.3 Unification des notions de classes, états, rôles et interfaces

4.3.1 Rôles versus interfaces

Un système est structuré en terme de classes, d'associations, et d'interfaces. Peu de langages à objets possèdent la notion de rôle ou d'état intrinsèquement. Dans la

plupart de langages à objets à typage statique[1], un objet est normalement issu d'une seule classe, même s'il peut bien sûr jouer plusieurs rôles et se trouver dans plusieurs états. La classe dont est issu un objet est sa "matrice" : Elle lui fournit sa structure en terme d'attributs, de routines et d'associations potentielles avec d'autres objets.

La notion de rôle, bien que conceptuellement proche de celle de classe (toute deux sont dérivées de la méta-classe Classifier du méta-modèle de UML), ne sert pas de matrice pour créer de nouveaux objets. Un rôle sert à décrire un fragment (ou encore aspect [69]) de la structure et du comportement d'une classe d'objets, fragment qui représente seulement la partie pertinente de ces objets dans un contexte donné (ce contexte est limité aux seules collaborations en UML, ce qui est une restriction assez arbitraire comme le note [93]). On peut donc voir une classe comme la synthèse des rôles que ses objets vont pouvoir jouer, ou symétriquement, voir un rôle comme un fragment de classe, projection de celle-ci dans un contexte particulier. UML a fait le second choix : Tout rôle est défini en UML comme étant un fragment d'une (ou plusieurs) classes dites "de base". Notons que ce choix est discutable d'un point de vue modularité et réutilisabilité. En effet, cela empêche de réutiliser la définition d'un rôle indépendamment de sa classe de base, alors qu'on aurait pu imaginer réutiliser un rôle pour synthétiser plusieurs classes différentes [95, 93]).

Les descriptions structurelles étudiées précédemment dans le contexte des cas d'utilisation comportent de nombreuses définitions de rôles. Mais si l'on s'intéresse à l'identité (permanentes) des objets et non aux rôles (transitoires) qu'ils jouent, on en déduit que le système comporte en fait assez peu de classes : Une classe Personne (pouvant jouer les rôles d'organisateur, d'animateur, de participant, etc) et une classe Réunion.

Pour autant, l'information fournie par les définitions de rôles n'est pas perdue. On la retrouve en effet dans la définition des associations entre les classes. En effet, les extrémités d'une association sont nommées en fonction des rôles respectifs que les objets connectés joueront les uns vis-à-vis des autres. On observe ainsi une certaine redondance entre définition de rôles dans les collaborations et nommage des extrémités des associations dans les diagrammes de classes.

Ce n'est pas la seule résonance attribuable aux rôles. UML permet en effet de "plaquer" des interfaces sur les extrémités d'une association : les objets atteignables en traversant cette association ne sont alors manipulables qu'au travers des opérations définies dans les interfaces correspondantes, indépendamment de la classe à laquelle l'extrémité d'association est effectivement connectée. Il faut bien sûr que cette classe implémente l'ensemble des interfaces qui contraignent l'extrémité de l'association. Dès lors, il est possible de faire correspondre une interface à chaque rôle défini précédemment, en en extrayant la partie purement structurelle (une interface ne dispose d'aucun comportement en UML). En contraignant les associations judicieusement à l'aide de ces interfaces, il devient possible de s'assurer qu'un objet obtenu en traversant

1. UML est généralement considéré comme un langage statiquement typé, même s'il dispose des notions de classification multiple et dynamique, dont la sémantique est pour le moins floue dans UML.

l'extrémité "animateur" de l'association entre Réunion et Personne ne sera manipulé qu'au travers des opérations de l'interface Animateur, retranscription fidèle de la partie structurelle du rôle d'Animateur. Ce parallèle entre rôle et interface fait suggérer aux auteurs de [93] que les deux notions pourraient être unifiées, pour peu que soit levée la limitation interdisant tout comportement aux interfaces.

4.3.2 Classes-états

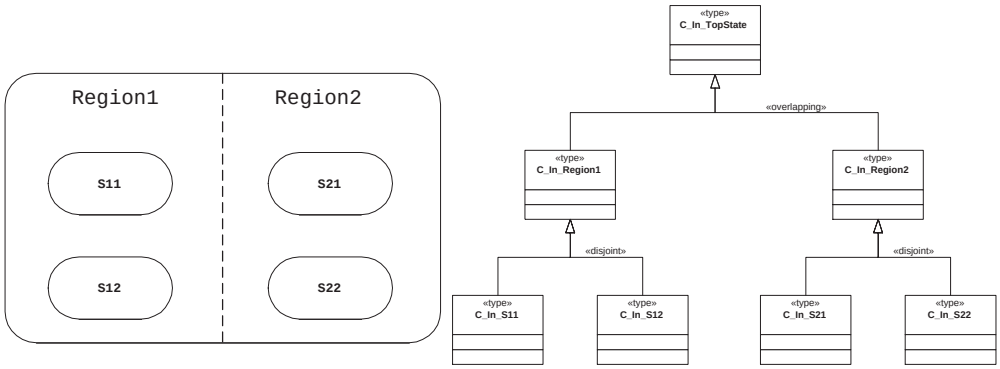
La notion d'état est assez proche de la notion de rôle. Les changements d'état comme les changements de rôle d'un objet modifient le comportement futur de l'objet. Tout se passe comme si l'objet adoptait temporairement un nouveau type, via un mécanisme de classification dynamique. La différence essentielle réside dans le fait qu'un changement de rôle est la conséquence d'une modification des liens entre objets (externes), alors qu'un changement d'état résulte d'une modification des attributs d'un objet (internes à celui-ci).

Alors qu'un rôle est une sorte de classe, avec des opérations et des associations, cela n'apparaît pas de manière aussi évidente pour la notion d'état. Pourtant, à chaque état d'une machine à états attachée à une classe, il est possible d'associer une classe-état ayant la même signature que la classe, mais avec des opérations redéfinies et spécialisées afin de traduire le comportement spécifique des objets se trouvant dans cet état.

En effet, bien que graphiquement identiques aux statecharts introduits par David Harel [50, 51], les machines à états de UML s'en différentient de manière subtile (et viennent ainsi s'ajouter aux innombrables variantes existantes [30]). Ces différences permettent de les intégrer harmonieusement au reste du système de types de UML basé sur les classificateurs (classes, rôles, interfaces). La différence principale réside dans l'affectation des priorités entre une transition sortant d'un état imbriqué par rapport à une transition sortant d'un état englobant lorsqu'elles sont toutes deux étiquetées par le même événement déclencheur. En UML, la première est plus prioritaire que la seconde.

Comme nous l'avons vu en section 2.4.5, la notion d'état sert à partitionner un ensemble d'instances. La section 2.3.5 a montré que la spécialisation entre classes permet de faire de même. Les deux notions s'organisent donc en hiérarchies. Dès lors, pourquoi ne pas considérer un état comme la classe des instances qui sont dans cet état ?

Fait peu connu, UML dispose en fait de cette notion de classe-état (la méta-classe correspondante s'appelle ClassifierInState). Mais rien n'est spécifié quant à la hiérarchie entre classes-états. UML n'impose pas que cette hiérarchie suive la hiérarchie entre états. Si l'on accepte cette hypothèse, l'état imbriqué devient simplement une sous-classe-état de la classe-état correspondant à l'état englobant, il est alors logique que la liaison dynamique donne priorité aux comportements attachées à la classe la



(a) state machine (b) ClassifiersInState

FIG. 4.1 – *Hiérarchie de classes-états*

plus spécialisée, et non l'inverse.

Supposons qu'une transition T sorte de l'état S lorsque qu'une opération O est appelée. Pour traduire ce comportement, il suffit de reporter les actions portées initialement par la transition T au sein du corps de la routine O redéfinié dans la classe correspondant à l'état S.

Ce mode de priorité est valable, que les opérations soit "localisées" (i.e. opérations attachées à la classe à laquelle la machine d'état est elle-même attachée), ou non localisées (i.e. opérations attachées au système, comme nous l'avons vu dans les exemples précédent). Dans le premier cas, on redéfinit les opérations dans la sous-classe-état, ce qui rentre dans le cadre classique de l'héritage. Dans le second, on surcharge l'opération au niveau du sous-système. Ce second cas rentre tout à fait dans le cadre de la théorie de la surcharge élaborée par Castagna [19, 20].

4.3.3 Predicate dispatching

Nous avons vu qu'il était possible d'associer des classificateurs à des prédicats, portant soit sur la valeur des attributs d'un objet (son état), soit sur ses liens avec d'autres objets (son ou ses rôles). Lorsqu'un objet change de classificateur parce qu'il satisfait de nouveaux prédicats, alors sa réaction aux événements change. Plutôt que la classe, l'état, ou le rôle, on peut considérer que la réaction d'un objet aux événements qu'il reçoit est déterminée par les prédicats qu'il satisfait. Ce mécanisme est parfois appelé *predicate dispatching* [22].

4.4 Modélisation en UML du domaine sémantique

Dans cette section, nous allons modéliser le domaine sémantique que nous proposons pour UML à l'aide d'UML lui-même. Cette approche "méta" (voir section 3.2.4) consistant à modéliser la sémantique d'UML à l'aide d'un sous-ensemble d'UML est similaire à celle choisie par le groupe de travail de l'*Action Semantics for UML* [1] qui devrait aboutir à une définition normalisée de *la* sémantique "officielle" d'UML. Le nom des classes d'entités constituant une configuration ont donc été choisis afin de correspondre autant que possible aux entités définies dans l'Action Semantics.

Les sections qui vont suivre décrivent le modèle d'exécution que nous avons adopté. Notons que nos travaux ont débuté bien avant que les premières propositions du groupe de travail sur l'*Action Semantics* ne soient disponibles. L'*Action Semantics* étend notamment la syntaxe abstraite de UML avec toute une partie actions (ou instructions) qui ne fait pas encore partie de la norme UML. Pour cette raison, nous n'avons souhaité faire aucune hypothèse sur la partie actions de UML, et ceci nous a conduit à faire certains choix permettant de se passer de ces actions, au profit de fragments de code écrits dans un langage natif[2].

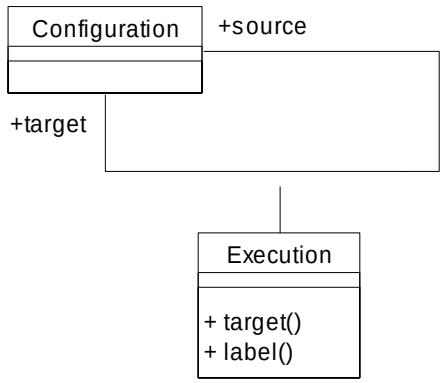


FIG. 4.2 – *Modèle simplifié du domaine sémantique*

Une `Configuration` d'un système spécifié avec UML représente un état global du système à un instant donné. Elle doit contenir suffisamment d'information pour permettre de calculer tous les états suivants possibles.

Une `Execution` représente l'enchainement des Configurations, en liant une Configuration source à une ou plusieurs Configurations target.

Ces 2 classes permettent de construire le LTS d'une spécification. À chaque configuration correspond un état global du LTS, et à chaque exécution correspond une transition.

2. L'Action Semantics offre aussi la possibilité d'inclure du code natif, mais sans définir comment ces fragment s'intègrent et interagissent avec la sémantique.

Bien sûr, le domaine sémantique est connecté au domaine syntaxique (c'est-à-dire au méta-modèle de UML). En effet, les objets contenus dans une configuration auront une structure définie par une classe, c'est-à-dire une instance de la méta-classe Classifier.

De même, une exécution se réfère à des actions modifiant l'état du système, ces actions étant définies dans le domaine syntaxique en tant qu'instances de la méta-classe Action, rattachées au corps d'une Méthode ou encore à l'effet d'une Transition.

4.4.1 Représentation d'un objet

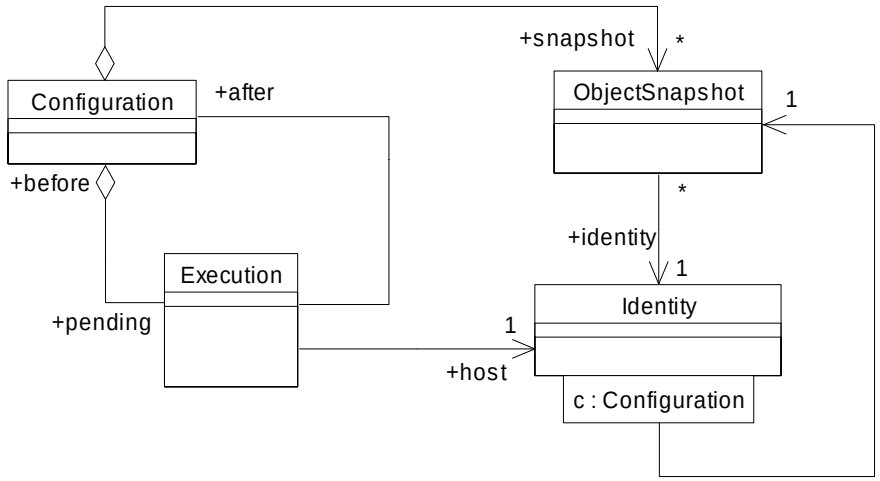


FIG. 4.3 – *Représentation d'un objet et de son identité*

Un système est composé d'un réseau d'objets évoluant au fil du temps. Une Configuration du domaine sémantique contient donc un ensemble d'ObjectSnapshots. Chaque ObjectSnapshot représente un objet à un moment donné. L'historique d'un objet se représente par une suite d'ObjectSnapshots se référant tous au même objet (qui a ainsi une identité persistante), répartis dans les Configurations formant l'historique du système.

Un objet possède une structuration en attributs. Chaque attribut est soit une valeur, d'un type fixe, soit une référence sur un autre objet. Si l'on considère les références comme des valeurs de types de données spécifiques, on obtient donc pour chaque objet un n-upplet de valeurs typées.

À chaque objet est associé un ensemble de types T (c'est un ensemble car UML autorise la classification multiple) pouvant varier à l'exécution (car UML autorise la classification dynamique).

Nous ferons l'hypothèse que le nombre d'attributs d'un objet est fixé une fois pour toute à la création de l'objet. Puisque les attributs sont définis comme appartenant à

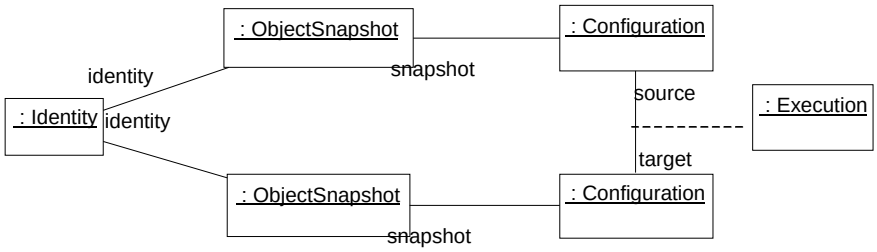


FIG. 4.4 – *Exemple d'un historique montrant l'évolution d'un objet à travers deux configurations*

un classifier, il faut donc restreindre la classification multiple et dynamique de façon à ce que l'ensemble des attributs d'un objet puisse être déterminé statiquement. De plus, le type d'un objet pouvant évoluer, certains attributs peuvent ne pas être accessibles (visibles) à un instant donné.

La figure 4.5 représente la classe ObjectSnapshot et ses connexions au domaine syntaxique.

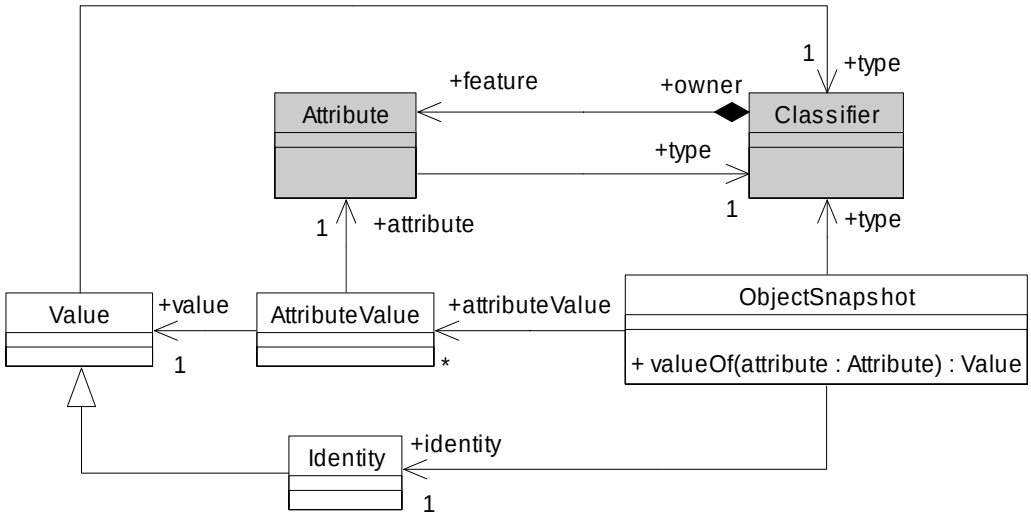


FIG. 4.5 – *Représentation d'un objet (vision interprétée)*

L'opération "valueOf(attr : Attribute) : AttributeValue" de la classe ObjectSnapshot représente la fonction sémantique renvoyant la valeur pour un attribut donné.

Il est important de noter que le domaine sémantique est connecté au domaine syntaxique par des associations, ce qui veut dire que les objets et opérations du domaine sémantique manipule des objets du domaine syntaxique. Un simulateur qui implanterait ce modèle se comporterait donc comme un interpréteur pour le langage UML.

Or comme nous le verrons dans la section 4.7, il est possible de faire apparaître dans les spécifications UML des actions dites "non-interprétées" qui représentent des frag-

FIG. 4.6 – *Un exemple de modèle UML*

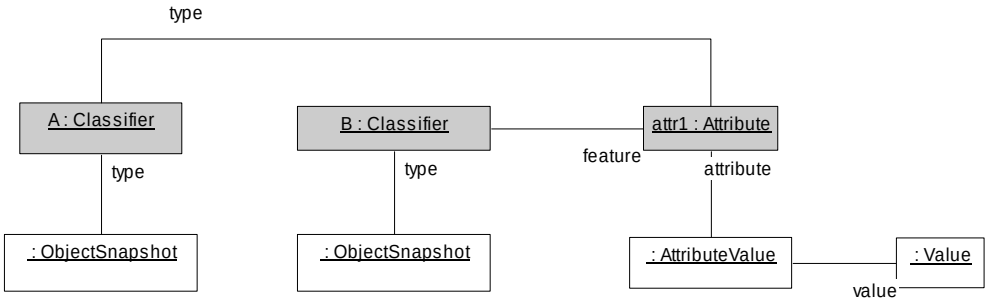


FIG. 4.7 – *Configuration pour le modèle UML de la figure 4.6*

ments de code écrits dans un langage de programmation classique, qu'il serait difficile de prendre en compte dans un interpréteur UML. Comme ces actions non-interprétées sont primordiales pour rendre la spécification exécutable, et donc simulable, cette limitation doit être contournée. La solution que nous avons développée est une approche par compilation : Le simulateur ne prend pas en paramètre lors de son exécution un "programme" (c'est-à-dire une spécification UML) qu'il manipulerait ensuite comme un interpréteur. Au contraire, la spécification est prise en compte lors de la génération du simulateur lui-même, afin de spécialiser le domaine sémantique en fonction des entités du domaine syntaxique présentes dans la spécification. Ensuite, ce modèle sémantique spécialisé est compilé, donnant le simulateur correspondant.

Il faut donc faire disparaître les associations explicites entre le domaine sémantique et le domaine syntaxique présentes sur la figure 4.5. La solution consiste tout simplement à remplacer chaque lien "type" entre un ObjectSnapshot et un Classifier en un lien d'héritage entre la classe ObjectSnapshot et une nouvelle classe du domaine sémantique qui héritera de ObjectSnapshot tout en reproduisant la structure de la classe du domaine syntaxique à laquelle le lien "type" faisait référence. Le résultat de cette transformation est représenté dans la figure 4.8.

Cette transformation revient en fait à peu près à "transposer" les classes présentes dans la spécification en classes du domaine sémantique. La fonction sémantique "valueOf" d'ObjectSnapshot n'est alors plus nécessaire, les attributs étant dès lors accessibles depuis le domaine sémantique directement. On notera que le type de l'attribut attr1 de la classe B du domaine sémantique n'apparaît pas sur le diagramme de la figure 4.6. Cet attribut représente une *référence* vers un autre objet de type A, au niveau du domaine sémantique, notion qui fait l'objet de la section suivante.

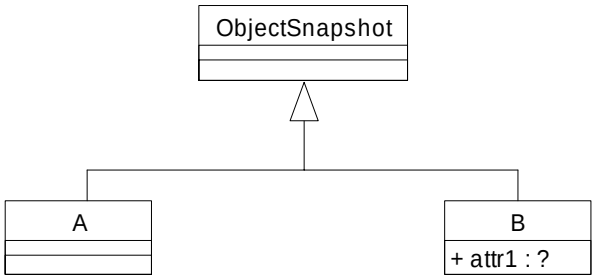


FIG. 4.8 – *Modèle sémantique spécialisé pour le modèle de la figure 4.6*

4.4.2 Valeurs, références, et identité

En UML, il existe deux genres de valeur (d'un attribut) :

- Les valeurs correspondant à des types de données (types entier, booléen, énumérés, etc...)
- Les valeurs correspondant à des références sur des objets d'une classe donnée.

Nous ferons l'hypothèse qu'à chaque type de données de la spécification UML correspond un type de données identique dans le domaine sémantique : On suppose donc que le modèle du domaine sémantique contient les types de données entier, booléen, énumérés, etc... Dans la transformation présentée dans la section précédente, un attribut ayant pour type un type de données au niveau syntaxique sera traduit en attribut ayant le type de données correspondant au niveau sémantique.

Le cas des attributs représentant une référence vers un autre objet d'une classe A donnée est plus intéressant. Deux choix sont possibles pour le type de l'attribut au niveau sémantique :

La première possibilité, représenté sur la figure 4.9, consiste à utiliser un type `Identity`, réification de la notion d'identité (il est même possible et souhaitable de définir une sous-classe de `Identity` pour chaque classe du modèle, afin d'obtenir des références typées). Cette solution introduit une indirection lorsque l'on souhaite accéder aux attributs d'un objet à travers une référence, puisqu'il faut retrouver l'ObjectSnapshot de la configuration courante correspondant à cette identité, en navigant au travers de l'association qualifiée.

La deuxième solution, représentée sur la figure 4.10, consiste à utiliser directement une référence vers un objet du domaine sémantique ayant pour classe la classe du domaine sémantique correspondant à A (et héritant donc de ObjectSnapshot). Cette solution apparemment plus simple évite certes le problème de l'indirection. Par contre, elle rend les ObjectSnapshots inter-dépendants (car ils sont alors connectés les uns aux autres explicitement), alors qu'une médiation par des Identities aurait permis de les isoler. Cela rend le partage d'ObjectSnapshots entre Configurations problématique.

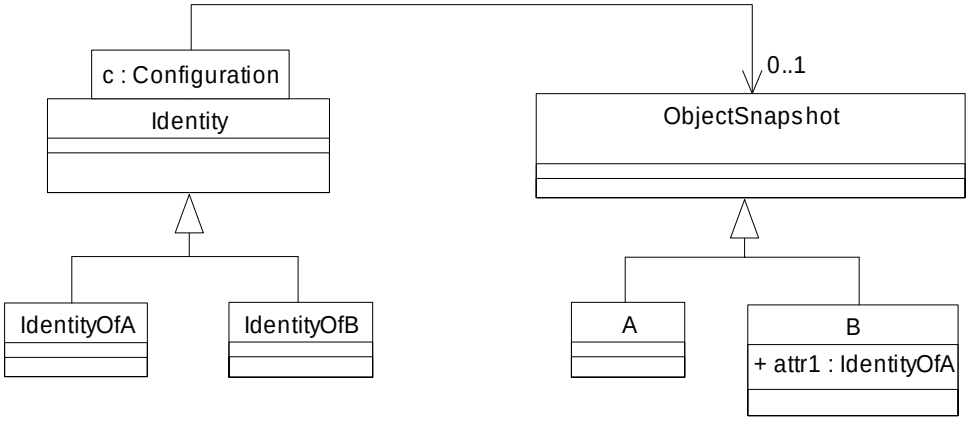


FIG. 4.9 – *Modèle sémantique spécialisé pour le modèle de la figure 4.6*

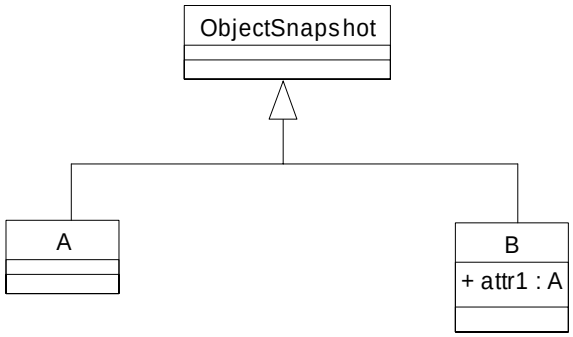


FIG. 4.10 – *Modèle sémantique spécialisé pour le modèle de la figure 4.6*

4.4.3 Représentations des liens entre objets

Les objets composant le système forment un réseau dont la topologie peut varier dynamiquement, par la création de nouveaux objets ou le changement de liens entre objets existants.

Un lien correspondant à une association d'arité n peut donc être représenté par un n-upplet d'identités d'objets. Comme nous l'avons vu précédemment dans la section 4.4.2, la notion d'identité peut être ou non réifiée dans le domaine sémantique.

Considérons l'exemple d'une association AB entre les classes A et B :

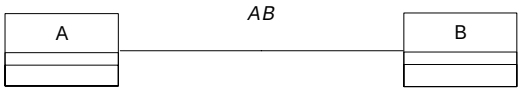


FIG. 4.11 – *Exemple d'association binaire*

De même qu'il existe de nombreuses façons d'implanter une association lorsque l'on traduit la spécification en un programme, il existe aussi plusieurs alternatives pour la représentation des liens dans le domaine sémantique. Nous en présentons deux, avec leurs avantages et inconvénients respectifs.

4.4.4 Les liens en tant que n-upplets d'identités

La figure 4.12 représente le modèle sémantique correspondant aux liens entre objets. On peut y voir qu'un LinkSnapshot contient une valeur par extrémité de l'association correspondant au lien.

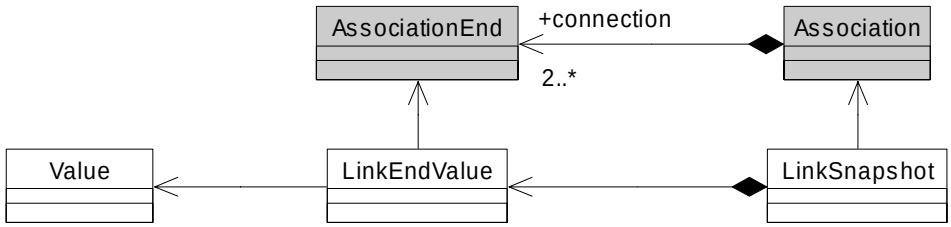


FIG. 4.12 – *LinkSnapshot (vision interprétée)*

La figure 4.13 représente le domaine sémantique pour l'exemple de la figure 4.11.

Ici aussi, des connexions explicites existent entre le domaine syntaxique (en grisé) et le domaine sémantique. Il est possible d'éliminer ces connexions par une approche "compilation", consistant à spécialiser la classe LinkSnapshot pour chaque association en faisant apparaître chaque élément du n-uplet en tant qu'attribut.

Comme nous l'avons déjà vu dans la section 4.4.2, le type de cet attribut peut être une identité réifiée ou une référence directe à l'un des ObjectSnapshot connectés par ce lien.

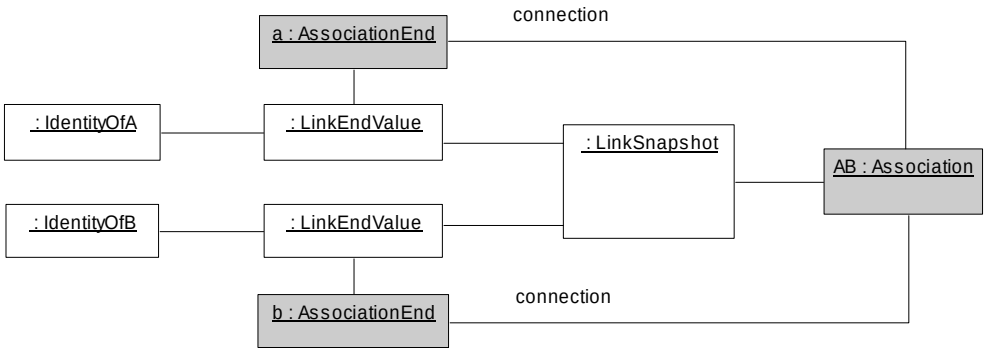


FIG. 4.13 – *configuration correspondant à l'exemple 4.11*

La première possibilité est présentée dans la figure 4.14.

FIG. 4.14 – *LinkSnapshot, spécialisé pour l'exemple 4.11*

La seconde est présenté dans la figure 4.15.

FIG. 4.15 – *LinkSnapshot, spécialisé pour l'exemple 4.11*

4.4.5 Les liens en tant que collections d'identités

Une autre possibilité, non applicable aux associations-classes, est de représenter une association par des ensembles d'identités : Depuis chaque objet *snapshot* jouant un rôle dans une association est accessible l'ensemble des objets jouant le rôle symétrique à cet objet dans la même association.

Comme les associations existantes depuis un objet dépendent de la classe de l'objet dont on part, cette solution n'est applicable que si l'on a préalablement spécialisé la classe ObjectSnapshot du domaine sémantique en fonction de la classe syntaxique dont les snapshot représentent des instances (c.f. figure 4.8).

Les objets accessibles en suivant une association sont obtenus à l'aide d'une nouvelle fonction sémantique dédiée, dont le nom correspond au rôle des objets ainsi obtenus.

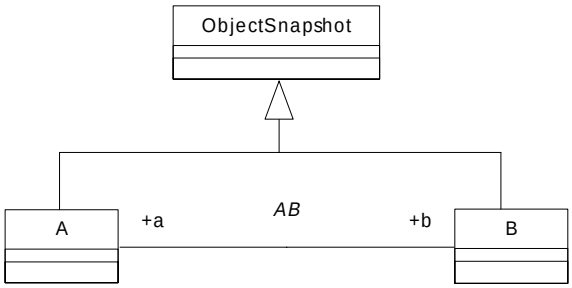


FIG. 4.16 – *Modèle sémantique spécialisé pour le modèle de la figure 4.11*

Ainsi que nous l'avons vu pour le cas de simples attributs à la section 4.4.2, l'accès aux objets au travers de l'association peut se faire soit en se référant directement aux snapshots en question (ce qui revient à transposer l'association directement du domaine syntaxique vers le domaine sémantique, comme on peut le voir sur la figure 4.16) soit via leurs identités respectives (solution présentée en figure 4.17), les deux solutions ayant les mêmes avantages et inconvénients respectifs que précédemment.

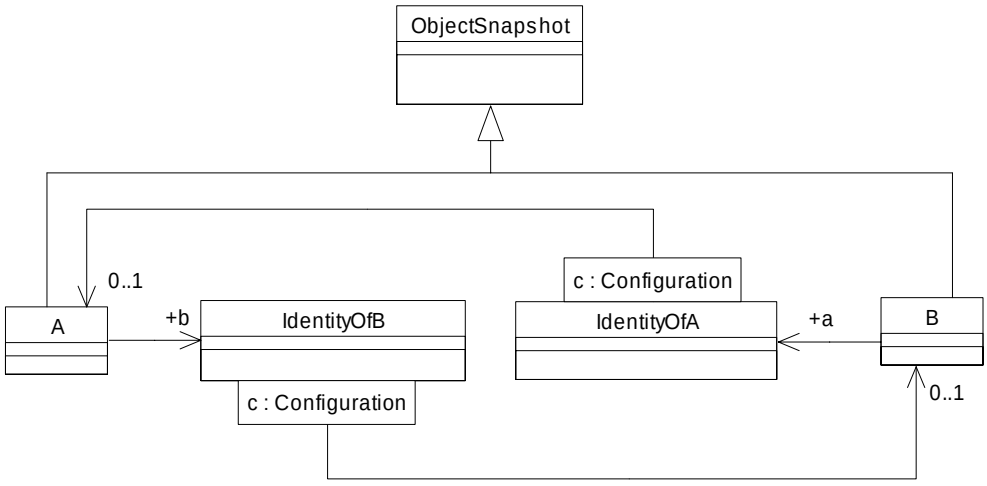


FIG. 4.17 – *Modèle sémantique spécialisé pour le modèle de la figure 4.11*

4.4.6 État des objets actifs

Les objets actifs auxquels est attachée une machine à état contiennent de l'information supplémentaire qui doit être représentée dans une configuration :

– L'état courant dans lequel se trouve le snapshot;

– L'état de la file de messages associée à ce snapshot.

La figure 4.18 représente une classe A à laquelle on a attaché une machine à états. Cette machine possède deux états S1 et S2, et on passe de S1 à S2 lors de la réception du signal `e` qui possède deux paramètres `p1` et `p2`.

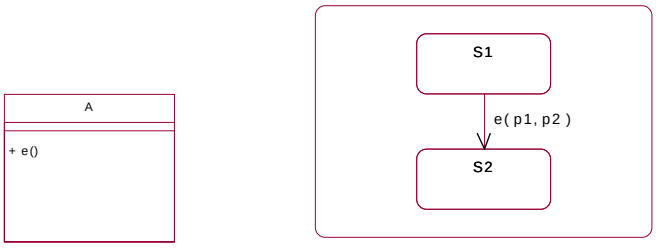


FIG. 4.18 – *Une classe avec une machine à deux états*

Voici en figure 4.19 le modèle sémantique adapté de celui de la figure 4.5 afin de prendre en compte les notions d'état et de file de message. On peut y voir que le snapshot d'un objet pointe vers son état courant (appartenant à la machine à états qui lui est associée). Le snapshot possède aussi une file de messages représentant les événements reçus (avec les valeurs correspondant à leurs paramètres respectifs).

Le figure 4.20 représente une configuration d'un objet de la classe A dans l'état S1 avec un message dans sa file correspondant au signal `e`.

Comme précédemment, nous souhaitons spécialiser et compiler le domaine sémantique de la figure 4.19 afin que les liens avec les classes du méta-modèle disparaissent. Le résultat de cette compilation est présenté sur la figure 4.21. On peut y voir que l'on a appliqué le patron de conception *State* afin de transformer chaque état en sous classe-état de la classe A. Ceci est en cohérence avec les idées d'unification entre les notions de classes et d'états énoncées à la section 4.3.2. Les actions associées à la transition déclenchée par la réception du signal `e` ont été intégralement reportées dans la méthode associée à `e` dans la classe-état `A_IN_S1`. Nous reviendrons plus longuement sur cet aspect en section 4.7.

Quant aux événements reçus, ils ont été réifiés en sous-classes de la classe `MessageInstance`. Nous avons donc cette fois appliqué le patron de conception `Command` : Chaque paramètre de l'événement donne lieu à un attribut du même type dans la sous-classe correspondante de `MessageInstance`. Une opération de niveau sémantique a également été rajoutée. Cette opération `dispatch` décrit ce qui se passe lorsque l'événement reçu est *dispatché*. On peut voir que la méthode correspondant à `e` dans la classe-état correspondant à l'état courant de l'objet récepteur sera appelée, ce qui aura pour effet d'exécuter les actions attachées à la transition, puis de passer dans l'état destinataire de la transition.

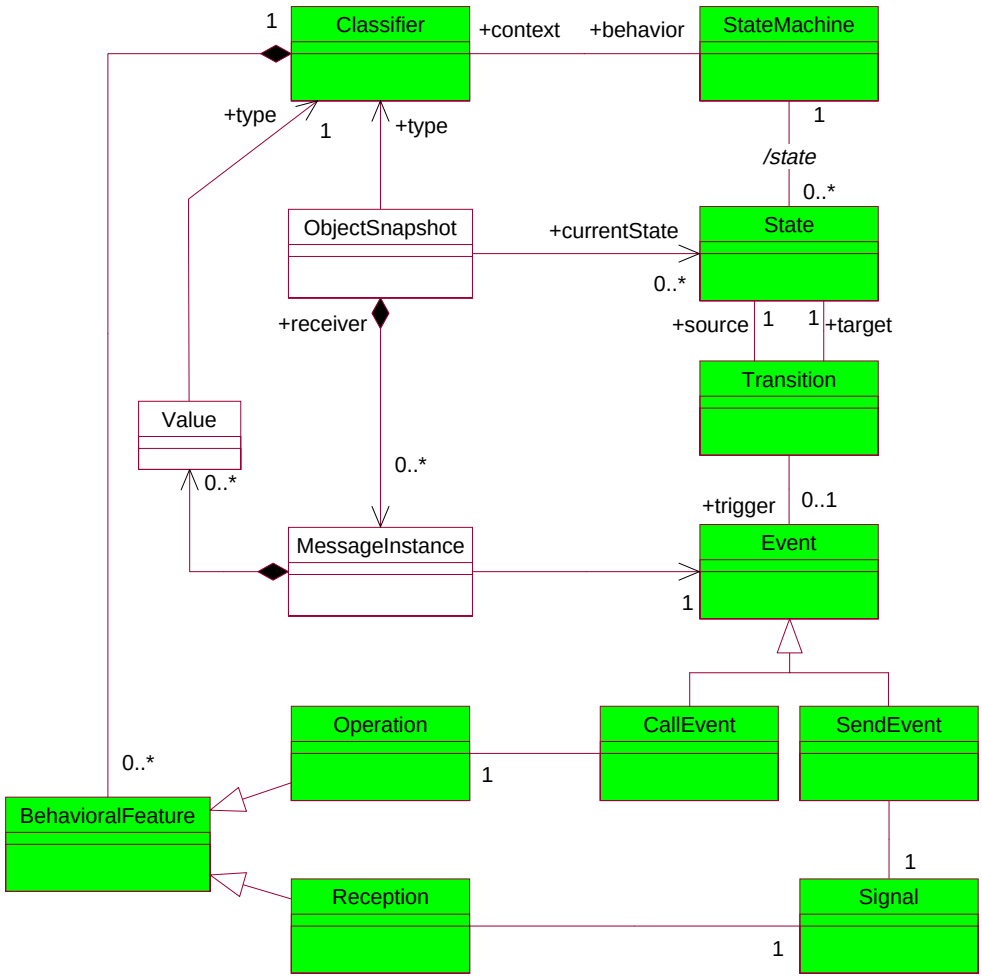


FIG. 4.19 – *Modèle sémantique pour les machines à états*

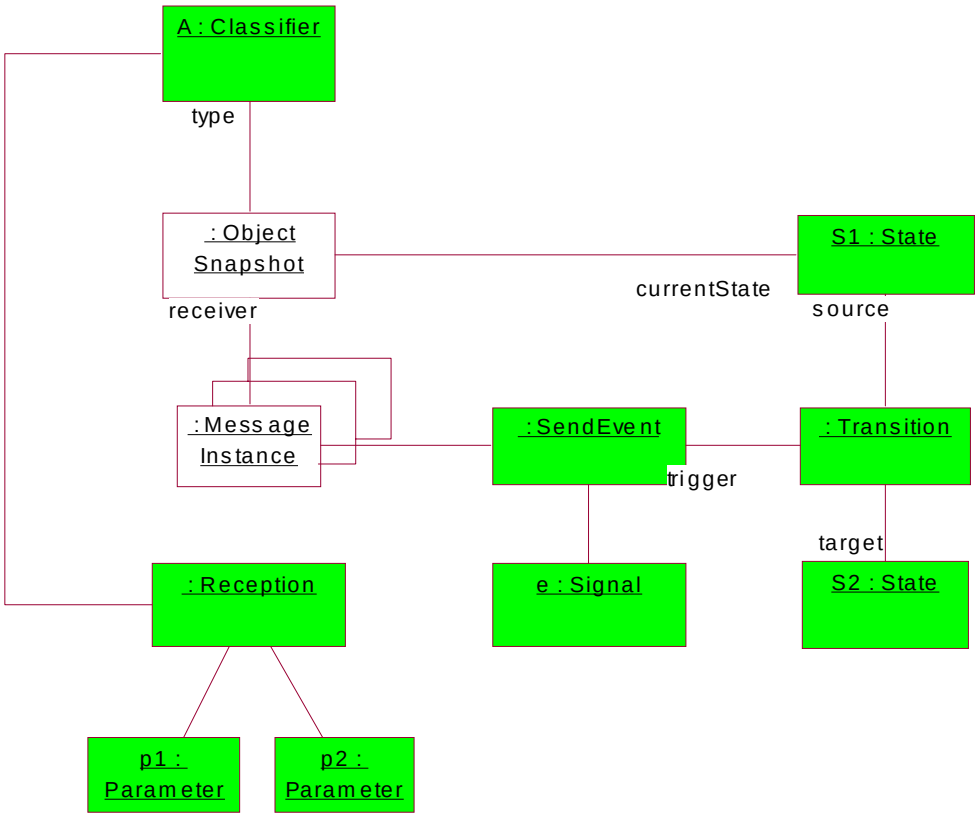


FIG. 4.20 – *Configuration avec un snapshot d'un objet actif*

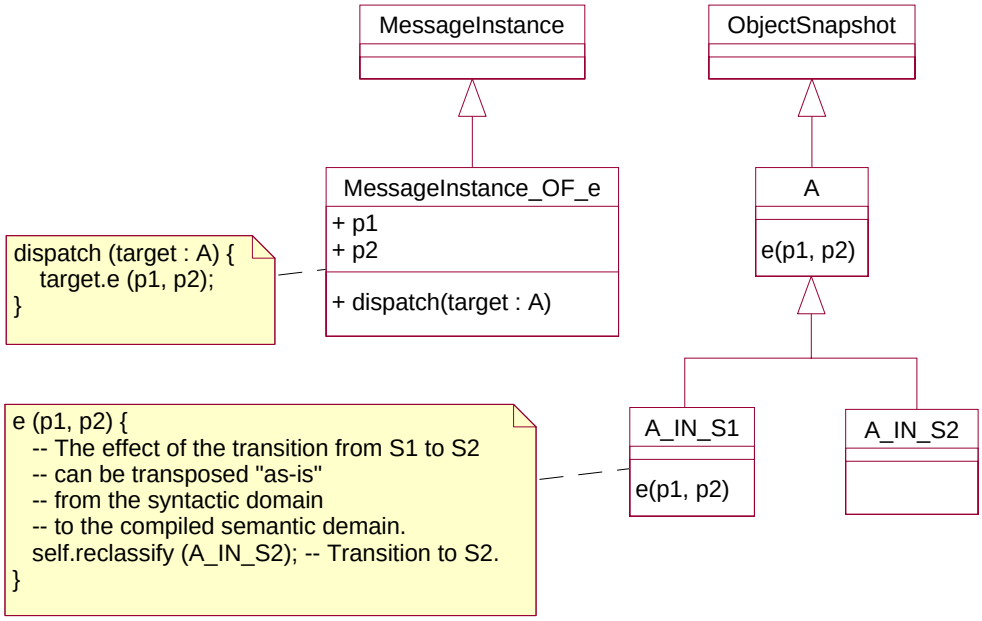


FIG. 4.21 – *Modèle sémantique compilé pour les machines à états*

4.4.7 Les piles d'exécution : Activations

Une configuration devrait aussi comporter un contexte d'exécution par flot de contrôle, appelé *Activations* et constitués de :

- Un pointeur sur la prochaine action à exécuter
- Un ensemble de bindings entre variables locales et valeurs

Nous verrons cependant en section 4.5.3 que les choix que nous avons effectués quant à l'atomicité des transitions font que les activations ne sont pas nécessaires dans notre domaine sémantique : Elles se trouvent en fait réduites à la valeur de l'état courant des objets actifs à l'origine des flots de contrôle.

4.5 Sémantique des expressions OCL

4.5.1 Exemple

Considérons la spécification UML de la figure 4.22, contenant un invariant de classe écrit en OCL.

La figure 4.23 est un diagramme d'objets se conformant au diagramme de classes de la figure 4.22.

L'invariant attaché à la classe A est clairement vérifié par ce diagrame d'objets. Nous avons vu à la section 3.3.1.2 qu'il est souhaitable que le simulateur UML puisse

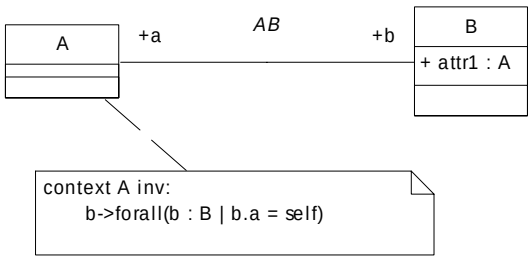


FIG. 4.22 – *Exemple de d'invariant de classe écrit en OCL*

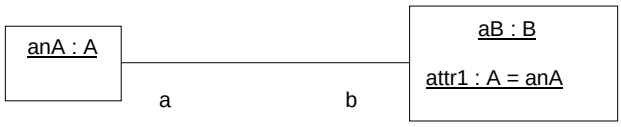


FIG. 4.23 – *Diagramme d'objets conforme à la figure 4.22*

détecter la violation de cet invariant. Pour cela, il faut être capable d'évaluer cette expression OCL dans le domaine sémantique tel que nous l'avons défini précédemment. La section suivante décrit la sémantique que nous donnons aux expressions OCL, tout d'abord dans le domaine sémantique interprété nous servant de référence, puis dans le domaine sémantique compilé que nous avons adopté pour le simulateur.

4.5.2 Méta-modèle OCL et modèle sémantique interprété

Avant de pouvoir évaluer une expression OCL, il faut d'abord pouvoir la représenter syntaxiquement. Or bien qu'elle inclue la définition d'OCL, la norme UML [89] ne donne qu'une BNF de la syntaxe d'OCL, et ne décrit qu'informellement la manière dont les expressions OCL sont connectées aux modèles UML qu'elles servent à compléter.

Plusieurs travaux comme [87] ont essayé de combler ce manque en proposant un modèle UML de la syntaxe d'OCL qui s'intègre au méta-modèle de UML (sous la forme d'un paquetage supplémentaire). Cette intégration d'OCL au méta-modèle UML rend la définition d'OCL dépendante de celle d'UML, ce qui en restreint l'utilisation dans un contexte UML. François Pennaneac'h propose dans sa thèse (qui n'a pas encore été publiée) un modèle général pour la syntaxe d'OCL indépendant d'UML (bien qu'aisément connectable à son méta-modèle). Il est à noter qu'une intégration poussée d'OCL dans UML est à l'agenda du groupe de révision chargé de définir UML2.0 (référence sur l'UML 2.0 OCL RFP accessible depuis l'URL http://cgi.omg.org/cgi-bin/doc?ad/2000-09-03)

La figure 4.24 présente une version alternative et nettement simplifiée d'un méta-modèle pour OCL intégré à celui d'UML.

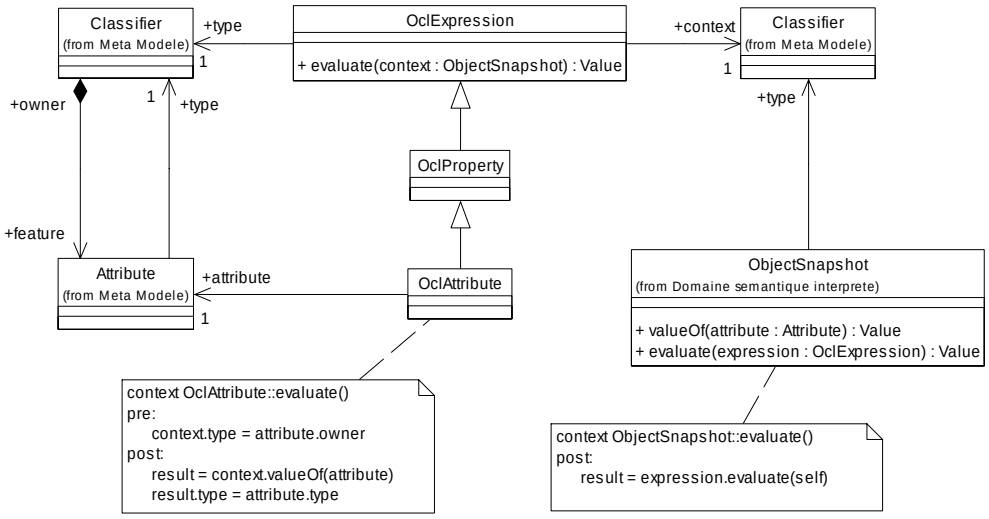


FIG. 4.24 – *Méta-modèle OCL*

La figure 4.25 représente une configuration contenant deux objets de classe A et B respectivement et qui se conforment au diagramme de classe de la figure 4.22.

L'arbre syntaxique de l'expression OCL `exp` y est également représenté. Cette expression donne la valeur de l'attribut `attr1` de la classe `B`, de type `A`.

context B exp : A = self . attr1

Cet arbre syntaxique est réduit à un unique nœud appelé `exp`, instance de la méta-classe `OclAttribute`. La valeur de cette expression dans un snapshot `s` donné est le résultat de la fonction sémantique `evaluate`, i.e. `exp.evaluate(s)`.

4.5.3 Atomicité des expressions OCL

Généralement, une expression OCL est atomique, c'est-à-dire qu'elle est complètement évaluable à l'aide de snapshots appartenant tous à une seule et unique configuration. Il existe cependant une exception : L'utilisation de l'opérateur OCL `pre` dans la post-condition d'une opération permet de se référer à l'évaluation d'une expression sur un snapshot appartenant à une autre configuration, celle correspondant à l'instant précédent l'appel.

Nous verrons par la suite que cette non-atomicité impose de prendre des précautions particulières dans le cas où le modèle sémantique est compilé.

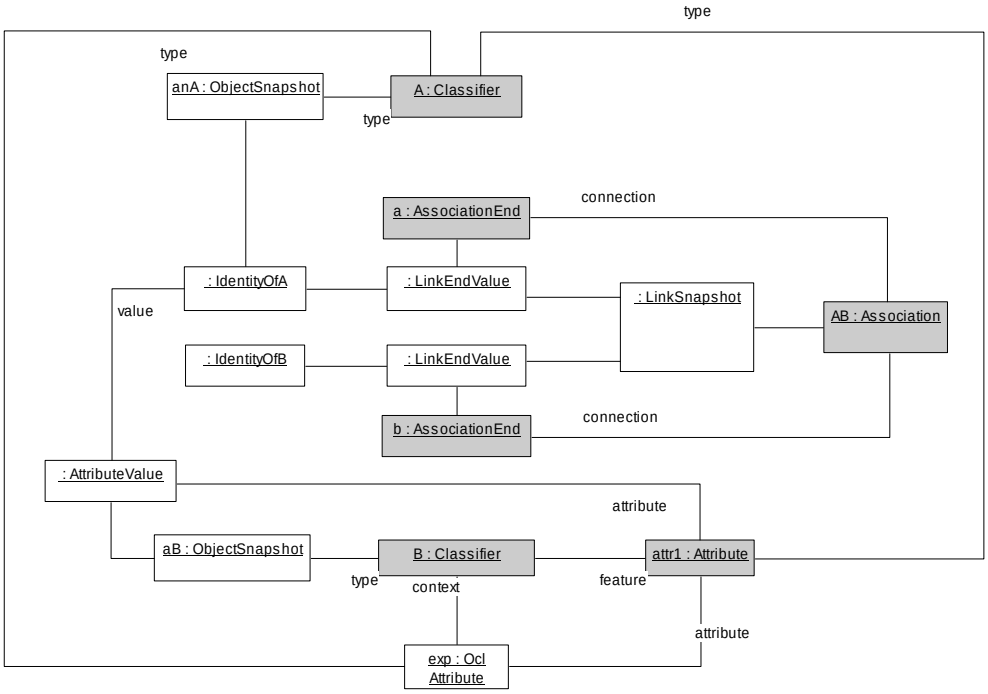


FIG. 4.25 – *Configuration correspondant aux figures 4.22 et 4.23*

4.5.4 Expression OCL mettant en œuvre des appels

Les expressions OCL attachées à une spécification peuvent aussi faire référence à des appels de fonctions définies dans la spécification. Ce point mérite un traitement particulier. En effet, une fonction de la spécification pourrait être implantée en terme d'actions, comme n'importe quelle autre routine. Mais alors, son apparition dans une expression OCL n'aurait plus aucun sens, car l'appel d'une telle routine modifie l'état du simulateur en créant de nouvelle configuration lors de l'exécution des actions. Deux conditions sont requises afin d'accepter un appel de fonction dans une expression OCL :

- La fonction doit être "sans effet de bord", i.e. elle doit posséder la propriété $\{isQuery[= true]\}$
- La fonction doit être elle-même complètement spécifiée en OCL, à l'aide d'une post-condition de la forme `result = une_expression_OCL`.

Nous verrons cependant dans la section 4.7 que cette seconde condition peut être relâchée lorsque le domaine sémantique est compilé : Il est alors possible d'implanter la fonction par un fragment de code natif, qui doit cependant rester sans effets de bord susceptibles de modifier l'état du simulateur.

4.5.5 Modèle sémantique compilé avec identités implicites

La figure 4.26 représente le domaine sémantique compilé et spécialisé pour la spécification de la figure 4.22. La version avec identités implicites est donc la fusion des figures 4.10 de la section 4.4.2 et 4.16 de la section 4.4.5.

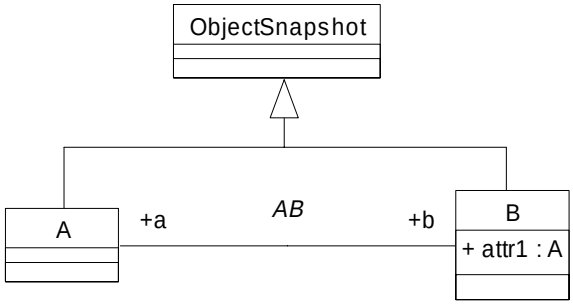


FIG. 4.26 – *Modèle sémantique avec identités implicites*

Un des intérêts d'avoir un modèle sémantique spécialisé et compilé est que les expressions OCL de type invariants de classe sont directement transposables du domaine syntaxique au domaine sémantique. Le cas d'OCL constitue en quelque sorte un préambule à l'introduction des fragments de code natif, objet de la section 4.7.

À noter que les expressions OCL mettant en œuvre des références/identités d'objets pour les comparer nécessitent certaines précautions. La comparaison de référence utilisée dans les expressions OCL de la spécification peut sous certaines conditions être transposée directement en comparaison de référence au niveau du domaine sémantique compilé avec identités implicites : L'évaluation des 2 opérandes de la comparaison et la comparaison elle-même doivent être réalisées atomiquement, c'est-à-dire ne manipuler qu'un seul état global. En effet, comme les références pointent directement sur les snapshots, et qu'un même objet sera représenté par autant de snapshots qu'il y a de configurations, une référence à un objet dans une configuration ne sera pas considérée égale à une référence au même objet dans une autre configuration. L'objet aurait en quelque sorte perdu son identité.

Il ne faut pas oublier que le @pre des post-conditions OCL peut se référer à un objet dans un autre état global ! Une solution permettant de contourner cet écueil est d'implémenter l'opérateur @pre à l'aide d'une "variable d'histoire" cachée qui serait recopiée d'état global en état global *en maintenant l'aliasing*.

4.5.6 Modèle sémantique compilé avec identités réifiées

Le modèle sémantique compilé avec identités réifiées ne souffre pas du problème de la comparaison de références appartenant à plusieurs configurations. Une référence

est en effet représentée directement par une valeur de type `Identity`, valeur qui est bien sûr la même pour un objet donné quelle que soit la configuration.

Par contre, l'indirection introduite par la classe Identity peut s'avérer gênante, car les expressions OCL ne sont alors plus (a priori) directement transposables du domaine syntaxique au domaine sémantique. Cette indirection peut cependant être rendue transparente en considérant une classe Identity comme un *proxy* ayant la même signature que la classe vers laquelle elle pointe (voir figure 4.27 ci-dessous).

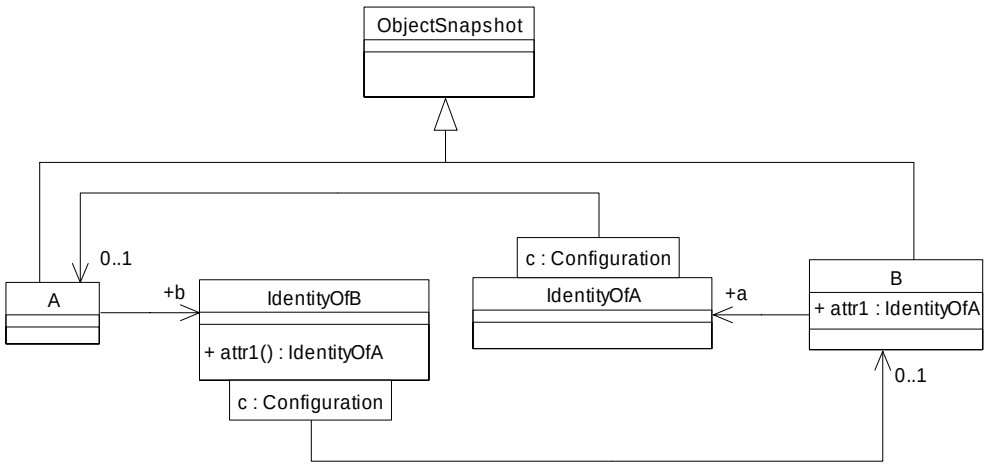


FIG. 4.27 – *Modèle sémantique avec identités réifiées*

Il y aura une instance de Identity pour chaque objet de la classe, qui recherchera automatiquement le snapshot correspondant de l'objet dans la configuration courante (il y en a au plus un, comme cela est indiqué par la qualification de l'association), puis ferait l'indirection de manière transparente.

Dans l'exemple de la figure 4.27, la classe `IdentityOfB` a été muni d'une opération appelée `attr1` qui représente la fonction sémantique permettant d'accéder à l'attribut `attr1` de l'ObjectSnapshot (de type B) correspondant à cette identité dans la configuration courante. Voici la spécification en OCL de cette fonction sémantique :

```
IdentityOfB :: attr1 : IdentityOfA =
  ( let
      snapshot : B = ( this . objectsnapshot [ currentConfiguration ]). oclAsType(B) in
    in
      snapshot . attr1
  )
```

4.6 Sémantique dynamique

4.6.1 Les statecharts de UML : synchrones ou asynchrones ?

A priori, les machines d'états de UML servent à représenter un ensemble de processus communiquant de manière asynchrone, via des files de messages. La sémantique des statecharts de Harel [50] a donc été adaptée de manière significative pour UML (la liste des principales différences est donnée dans la documentation officielle de UML [89], section 2.12.5, "*comparison to classical statecharts*"), ajoutant ainsi une nouvelle variante à celles présentées dans [30].

Cependant, la notion de file n'a de sens que pour les objets actifs, c'est-à-dire des processus, et la sémantique associée aux machines d'états des objets passifs est un problème ouvert. Ces objets étant normalement utilisés via des appels d'opérations classiques, c'est-à-dire synchrone, les machines d'états des objets passifs sont un moyen élégant de représenter les pré et post conditions de ces opérations. Cette approche est plus ou moins suggérée dans les Notes accompagnant le chapitre machines à états de la documentation officielle de UML mentionnées au paragraphe précédent.

À un plus haut niveau d'abstraction, il est cependant possible d'envisager une sémantique synchrone permettant de montrer l'évolution "simultanée" d'un ensemble d'objets réagissant en même temps à un même stimulus. On ne s'intéresse alors pas encore aux interactions de bas niveau entre les objets, nécessaires pour implanter le comportement global. Seule la réaction conjointe des objets présente un intérêt à ce niveau. Cette interprétation synchrone de plus haut niveau d'abstraction est directement liée à la notion de sous-système en UML (SubSystem), et à la notion de "joint action" telle qu'introduite dans la méthode Catalysis [32].

En effet, si l'on considère la machine à état d'une classe telle que la classe Réunion étudiée à la section 2.4.5, rien n'interdit syntaxiquement certaines de ses transitions d'avoir pour événement déclencheur (*trigger*) l'appel (*CallEvent*) d'une opération qui n'est pas attachée à la classe Réunion elle-même, mais à une entité plus globale.

On ne définit alors pas les opérations au niveau des objets constituant le sous-sytème "réunions virtuelles" pris individuellement, mais seulement au niveau de l'interface du sous-système. Les objets/classes composant le sous-système évoluent bien sûr lorsque les services du sous-système sont appelés. Aussi est-il naturel d'associer des machines d'états aux objets composant le sous-système, ayant pour *trigger* non pas des appels aux opérations de leur classes respectives (opérations qui n'ont guère d'intérêt à ce niveau) mais aux opérations du sous-système englobant dans son ensemble, qui délimite une frontière avec le reste du système. La file de message associée au sous-système tout entier (vu comme un seul gros objet actif composite) est ainsi partagée entre tous les objets constituant le sous-système, et l'on retrouve la sémantique de *broadcast* des événements qui est celle des statecharts de Harel.

Le système peut être vu comme un ensemble de sous-systèmes spécifiés de manière

synchrone et communiquant entre eux de manière asynchrone, ce qui rejoint tout à fait la philosophie du langage BDL [98]. Ce principe peut s'appliquer à tous les niveaux d'abstractions, par raffinements successifs.

Cependant, le simulateur que nous avons réalisé ne prend en compte que les statecharts de granularité la plus fine, c'est-à-dire où toutes les transitions sont localisées et sont donc déclenchées par l'appel de routines appartenant à la classe à laquelle la machine à états est liée. Cela correspond au niveau d'abstraction le plus bas, avec des automates communiquant de manière asynchrone par envois de messages pour les objets actifs et par appels de routines classiques pour les objets passifs. Le modèle sémantique correspondant a été présenté en section 4.4.6.

4.6.2 Atomicité des transitions

Une transition UML n'est normalement pas véritablement atomique : Elle est composée d'actions de plus faible granularité, qui pourraient interagir (ou au moins être entrelacées) avec d'autres actions portées par d'autres transitions UML.

Dans notre simulateur, une transition d'un statecharts est atomique au sens d'un "RTC step" (nous avons vu à la section 4.4.7 que notre domaine sémantique ne comportait pas de piles d'activations). Toutes les actions au sein d'une transition sont regroupées de manière atomique, et donc on n'observe d'entrelacements qu'entre transitions UML prises dans leur globalité, et non entre actions élémentaires.

Cette restriction empêche d'observer certains types d'interactions qui se feraient indépendamment du contrôle d'une machine d'états. C'est notamment le cas des accès à des variables partagées, dont font implicitement partie les attributs publics des objets (ils peuvent être consultés, voire même modifiés en UML, sans communication explicite avec leur objet hôte).

Cette restriction a pour origine la prise en compte des fragments de code écrits en langage natif. Dès lors que l'on autorise cette possibilité, il est impossible pour le simulateur de scinder l'exécution d'un tel fragment de code entre deux actions qu'il ne connaît pas. Nous avons commencé nos travaux à une époque où le langage d'actions d'UML (qui fait partie de la proposition de l'*Action Semantics*) n'existait pas encore, rendant toute analyse plus fine de l'atomicité impossible.

4.6.3 Evaluation des gardes

Contrairement à un langage comme LOTOS où les gardes ne peuvent référencer que les offres d'un rendez-vous, les gardes d'UML peuvent a priori référencer des attributs de l'objet courant, ou même d'autres objets aux travers des associations. Cela revient à l'utilisation de variables partagées, inexistantes en LOTOS (qui est purement fonctionnel).

La manière dont les gardes doivent être traîtées par le simulateur n'est pas très claire. La compilation des gardes des transitions nécessite a priori la notion d'expression évaluée atomiquement, ce qui impose très certainement des restrictions syntaxiques, et peut se révéler par ailleurs difficile ou impossible à transposer tel-quel dans une implantation de la spécification, notamment à cause d'accès non-locaux (mais cela n'est pas notre problème après tout, mais celui de la personne qui écrit la spécification, du moment que cela ait un sens au niveau de la spécification).

L'évaluation d'une garde d'une transition (voir d'un ensemble de gardes) peut-elle donc être considérée comme quelquechose d'atomique?

Dans la doc UML1.3, on peut lire ceci:

> Guards
>
> In a simple transition with a guard, the guard is evaluated before the transition is triggered.
>
> In compound transitions involving multiple guards, all guards are evaluated before a transition is triggered, unless there are choice points along one or more of the paths. The order in which the guards are evaluated is not defined.
>
> If there are choice points in a compound transition, only guards that precede the choice point are evaluated according to the above rule. Guards downstream of a choice point are evaluated if and when the choice point is reached (using the same rule as above). In other words, for guard evaluation, a choice point has the same effect as a state.
>
> Guards should not include expressions causing side effects. Models that violate this are considered ill formed.

Ces explications ne répondent pas à la question, malheureusement. Comme le modèle d'exécution de UML est a priori asynchrone, il se pose la question de savoir si l'on doit considérer l'évaluation d'une garde comme une série d'actions de bas niveau (lire tel attribut, faire telle opération, etc...) ou comme une expression d'un autre niveau (une fonction sémantique, i.e. évaluée par le simulateur atomiquement sur une et une seule configuration, correspondant à l'état global au moment de l'évaluation de la garde).

Dans le premier cas, cela voudrait dire que les sous-actions composant l'évaluation d'une garde pourraient être entrelacées avec les autres actions exécutées par ailleurs dans le système. Donc meme si une garde n'a pas d'effet de bord, son évaluation pourrait être perturbée par d'autres actions en ayant.

Dans le second cas, le problème ne se pose pas, et la notion d'effet de bord dans la garde n'a plus de sens. L'absence d'effet de bord dans les expressions OCL des gardes du modèle permet alors de les transposer en fonctions sémantiques comme nous l'avons étudié à la section 4.5. Il n'y a alors aucune communication pour évaluer ces expressions qui sont effectivement évaluées "instantanément" sur *une* configuration du système. Elles viennent alors entourer et protéger les actions, qui seront conjointement compilées dans le domaine sémantique comme nous l'expliquons ci-dessous.

4.6.4 Les actions

Les évolutions du système (ses changements d'états) sont causés par l'exécution d'actions (par le système lui-même ou par son environnement). Ces actions sont portées soit par une transition d'une machine à états, soit dans le corps d'une méthode implémentant une opération. Les transformations évoquées en section 4.3.2 permettent de ramener le premier cas au second.

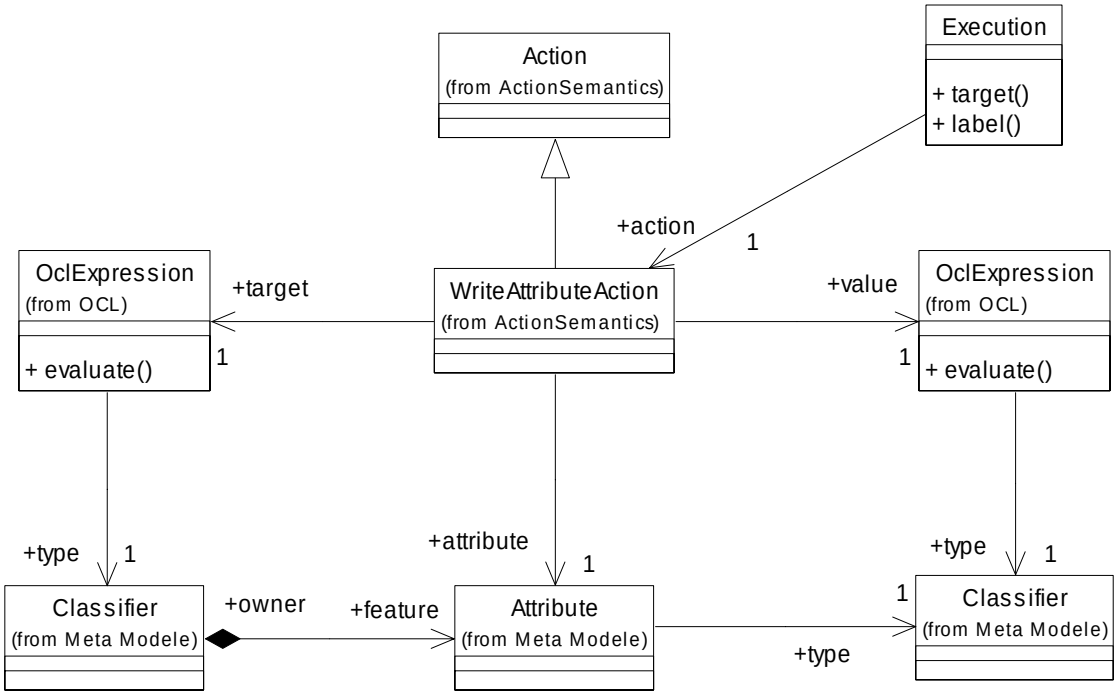


FIG. 4.28 – *Actions de mise à jour d'un attribut*

La figure 4.28 décrit la représentation d'une action servant à mettre à jour la valeur d'un attribut. En évaluant l'expression représentant la nouvelle valeur, il est possible de construire une nouvelle configuration contenant le snapshot de l'objet ainsi altéré mis à jour.

Cette construction de la nouvelle configuration n'est possible que si l'action est décrite dans le modèle sémantique, ce qui n'est justement pas le cas des actions écrites dans des fragments de code en langage natif. Le seul moyen de construire la nouvelle configuration est alors d'exécuter ce fragment de code (on parle parfois de *semantic loop-hole*). Ce fragment de code devra avoir été compilé en même temps que le modèle sémantique associé à la spécification (cette cohabitation est d'ailleurs tout l'intérêt de l'approche par compilation). Nous avons vu comment cela est réalisé à la section 4.4.6.

Un point supplémentaire à prendre en compte est que les actions "non-interprétées" construisent la nouvelle configuration en *altérant* la configuration courante au moment

de leur exécution, par effets de bord. La construction du LTS est donc légèrement différente dans cette situation. Plutôt que de construire une nouvelle configuration en utilisant une fonction prenant en paramètre la précédente, on duplique complètement la configuration courante et on laisse les actions non-interprétées effectuer leurs modifications directement sur cette copie, qui devient la configuration suivante à l'issue du processus.

Comme les snapshot des objets et des liens ont la même structure dans le domaine sémantique compilé que dans la spécification, les fragments de code en langage natif peuvent ainsi eux aussi être compilés "directement", car ils ont été écrits en tenant compte de la structure des entités (classes, attributs et opérations) présentes dans la spécification. Cette compilation "directe" n'est en fin de compte pas si directe que cela, même si la complexité est pour l'essentiel cachée à celui qui aura la charge d'écrire les fragments de code. La section 4.7 explique comment cela est réalisé.

4.7 Les actions non-interprétées

4.7.1 Le problème

Le problème majeur avec les spécifications UML actuelles, en l'absence d'un langage précis et standardisé pour les actions, est que les actions agissant sur les objets du modèle sont le plus souvent écrites à l'aide de langages de programmation classiques comme Java, C++, Ada ou Eiffel.

L'*Action Semantics* se propose de remédier à terme à ce problème en définissant une syntaxe abstraite ainsi qu'une sémantique pour un ensemble d'actions primitives dans lequel il sera possible de traduire les instructions de la plupart des langages suscités. L'*Action Semantics* n'offre pas de forme textuelle propre, mais définit la façon dont un langage textuel classique comme Java se projette sur sa syntaxe abstraite.

Ainsi, pour peu que l'*Action Semantics* définisse ces projections pour tous les langages utilisés dans les spécifications UML, et que les outils UML adoptent rapidement cette nouvelle partie du standard UML, le problème serait résolu, l'outil traduisant automatiquement le code Java entré par l'utilisateur en actions UML. Cependant, de tels outils appartiennent encore malheureusement au futur, et l'*Action Semantics* permettra malgré tout de contourner la traduction en actions UML en offrant une porte de sortie autorisant l'inclusion de code natif qui ne sera pas interprété par les outils. Le problème de ces fragments de code natif ne sera donc pas résolu.

4.7.2 Une approche par compilation au lieu d'interprétation

Pouvoir écrire des fragments de code Java, C++, Ada ou Eiffel au sein d'une spécification UML est un atout indispensable pour obtenir des spécifications précises et

complètement exécutables. Cela rend également la transition vers l'implantation plus facile, alors qu'imposer un langage d'actions différent aux seules fins de simuler la spécification pourrait rendre cette transition plus difficile.

Nous avons donc choisi d'adapter le modèle du domaine sémantique afin de permettre une intégration aisée des fragments de code écrits en langages "natifs". La spécification pourra être simulée en compilant ce modèle dans le langage natif correspondant. Rien n'empêche d'ailleurs d'utiliser plusieurs langages natifs simultanément : il faudra alors scinder le modèle du domaine sémantique en plusieurs sous-parties correspondant à un langage natif donné, puis compiler les sous-parties avec le compilateur correspondant.

Bien sûr, les fragments de code écrits par l'utilisateur ne doivent pas nécessiter de la part de celui-ci une connaissance poussée de la structure interne du simulateur UML. Ce code doit être écrit comme s'il s'agissait de code faisant partie de l'implantation de la spécification, en accédant aux objets et à leurs attributs de la manière la plus naturelle possible. Il ne serait notamment pas acceptable que l'utilisateur ait à introduire des indirections pour accéder aux objets et à leurs attributs à travers des objets qui appartiennent au domaine sémantique, c'est-à-dire des objets du run-time du simulateur UML.

4.7.3 Une intégration transparente pour l'utilisateur

Une partie importante de la modélisation du domaine sémantique consistera donc à pouvoir intégrer le code natif sans rajouter d'indirections et de délégations qui soient visibles de l'utilisateur. L'approche choisie dépend du langage dans lequel ces fragments de code sont écrits, chaque langage offrant ses propres idiomes nous permettant d'atteindre notre objectif. Nous avons déjà vu comment masquer ce type de problèmes lorsque nous avons traité les expressions OCL, à la section 4.5.6.

Il existe plusieurs approches pour que du code appartenant à une routine définie dans une classe B (une classe Transition par exemple) puisse accéder de manière transparente aux attributs d'un objet d'une classe A (une classe ObjectSnapshot par exemple).

Si B est connue statiquement, il suffirait que B hérite de A et que l'objet soit en fait une instance de la classe B. Cependant, dans notre cas, nous ne connaissons pas statiquement quelle est la classe B (il peut y avoir plusieurs candidates B1, B2, Bn).

Les sections suivantes expliquent comment procéder à l'aide d'idiomes spécifiques à certains langages de programmation :

- La délégation implicite à l'aide du mécanisme des "inner-classes" de Java.
- La création d'alias à l'aide du mécanismes des références de C++.
- La reclassification dynamique, pour une variante du langage Eiffel.

4.7.4 Le cas du langage Java : Inner-classes et délégation implicite

L'idée consiste à déclarer chaque classe B1, B2, Bn en tant que inner classe de A. Chaque instance d'une classe Bi est en relation avec une instance de la classe A appelée l'"outer object". Cet outer-object est d'ailleurs accessible explicitement au sein des routines (non statiques) de Bi à l'aide de l'expression A.this ("qualified this", [46], section 15.8.4).

Dans la mesure où aucun nom ne vient cacher le nom d'un attribut attr de la classe A, alors il est possible de remplacer "A.this.attr" par "attr". L'accès aux attributs de l'instance de A englobant l'instance de Bi est donc alors complètement transparent.

```
class A {
    // A's functions :
    public void reset1 () {
        attr1 = 0;
    }
    public static void reset2 () {
        attr2 = 0;
    }

    // A's attributes :
    private int attr1 ;
    private static int attr2 = 0;

    interface B {
        public void execute ();
    }

    class B1 implements B {
        public void execute () {
            // All members of A are directly accessible .
            // Code manipulating A's members can now be inserted here verbatim.
            reset1 ();
            reset2 ();
            attr1 ++;
            attr2 ++;
            // Note that the attributes are not actually declared in B,
            // to which an unqualified " this " refers , but in A.
            // This means that the following statement is not legal :
            //     this . attr1 ++;
            // It would have to be ( automatically ) replaced by:
            //     A. this . attr1 ++;
```

```
        }
    }

    public static void main(String s []) {
        A a = new A();
        B b = a.new B1();
        System.out.println (a.attr1 + ",␣" + a.attr2 );
        b.execute ();
        System.out.println (a.attr1 + ",␣" + a.attr2 );
    }
}
```

Pour exécuter le fragment de code contenu dans une routine d'une classe Bi devant accéder de manière transparente aux attributs de l'objet de la classe A, il suffit donc de créer une instance de la classe Bi avec l'objet de la classe A en utilisant une "qualified class instance creation expression" ([46], section 15.9), puis d'appeler la routine sur l'instance de la classe Bi.

4.7.5 Le cas du langage C++ : Alias et références

Contrairement au langage Java, C++ ne dispose pas des "inner classes", mais seulement des "nested classes" (équivalentes aux "inner classes" *static* de Java). Rendre la délégation transparente n'est donc pas aussi direct en C++ qu'en Java.

Cependant, C++ offre la possibilité de définir des alias, c'est-à-dire de définir un nouveau nom permettant d'accéder directement à une *lvalue* existante (pour simplifier, une lvalue est une entité ayant une adresse et pouvant donc être la cible d'une affectation).

Un alias est ainsi créé à l'aide du mécanisme des références pour chaque attribut non-statique que doit pouvoir manipuler le code que l'on souhaite insérer dans le modèle. L'accès aux fonctions non-statiques est également possible à l'aide d'une simple délégation. L'imbrication lexicale des classes assure quant à elle un accès transparent aux attributs et fonctions statiques.

```
#include <stdio.h>

class A {
    public:
    A() {
        attr1 = 0;
    }
    // A's functions :
```

```
void reset1 () {
    attr1 = 0;
}
static void reset2 ();
// A's attributes :
int attr1 ;
static int attr2 ;

// Everything is made visible to B and its descendents.
friend class B;

class B {
    public:
    B(A* a) {
        A_this = a;
    }
    // All non-static member functions delegate to the outer object.
    void reset1 () {
        A_this->reset1();
    }
    // The additional function to execute verbatim code.
    virtual void execute () = 0;
    protected:
    A* A_this; // The outer object.
};

class B1 : public B {
    public:
    B1(A* A_this) : B(A_this) {
    }
    void execute () {
        // Aliases are created for all non-static attributes of A.
        int& attr1 = A_this->attr1;
        // Code manipulating A's members can now be inserted here verbatim.
        reset1 ();
        reset2 ();
        attr1 ++;
        attr2 ++;
        // Note that the aliases are not real members of class B.
        // This means that the following statement is not legal :
        //     this->attr1++;
```

```
            // It would have to be ( automatically ) replaced by:
            //      A_this->attr1++;
        }
    };
};

// Definitions for static members of A:
void A::reset2 () {
    attr2 = 0;
}
int A::attr2 = 0;


int main() {
    A*     a = new A();
    A::B* b = new A::B1(a);
    printf ("%d, %d\n", a->attr1, a->attr2);
    b->execute();
    printf ("%d, %d\n", a->attr1, a->attr2);
}
```

4.7.6 La reclassification dynamique

Cette approche alternative est assez originale et n'est applicable que si le langage offre la possibilité de changer dynamiquement le type d'un objet. On appelle cette possibilité la reclassification dynamique.

L'idée consiste à faire en sorte que chaque classe B1, B2, Bn hérite de A, et à changer temporairement le type de l'objet : L'objet prend temporairement pour type une des classes Bi, le fragment de code contenu dans une des routines de la classe Bi est exécuté, puis l'objet reprend son type initial, A. Comme Bi hérite de A, le fragment de code a accès à tous les attributs de l'objet courant hérité de la classe A. L'exemple qui suit illustre cette approche dans le cas d'une variante du langage Eiffel basée sur le compilateur SmallEiffel.

```
class A
feature {ANY}

    attr1 : INTEGER

    reset1 is
```

```
        do
            attr1 := 0;
        ensure
            reset : attr1 = 0;
        end;

feature {ANY}

    reclassify (type : like Current) is
        require
            valid_prototype : type /= Void;
        do
            -- In the C code generated by SmallEiffel ,
            -- the 'Current' object is accessible through the 'C' struct ,
            -- while the first parameter is accessible through 'a1'.
            -- Both structs have an "id" field (an int) representing the type-tag.
            -- The type-tag of Current is modified to be that of the proto-type.
            c_inline_c ("C->id=a1->id;");
        ensure
            reclassified : Current.same_type(type);
        end

end -- class A
```

La routine `reclassify` mérite quelques commentaires. Le langage Eiffel ne dispose normalement pas de la classification dynamique. Cependant, il est aisé de lui rajouter cette caractéristique sous une forme simplifiée, en faisant l'hypothèse que tous les types qu'un objet pourra prendre ont la même structure. SmallEiffel représente en effet le type d'un objet par un champ spécial `id` de la structure C sous-jacente, qu'il suffit de modifier temporairement à l'aide d'un fragment de code C.

Ce "bricolage" n'est bien sûr pas portable, et nécessite de bien connaître la manière dont SmallEiffel compile le code Eiffel en C, mais ceci est bien isolé dans une seule routine `reclassify`. Cette astuce a au moins le mérite de résoudre notre problème pour le langage Eiffel.

```
class B
inherit A
feature {ANY}

    execute is
        do
```

```
            reset1 ;
            attr1  :=  attr1  + 1;
        end

end -- class B
```

```
class ROOT_CLASS
creation main
feature {ANY}

    main is
        local
            a : A
            b : B
        do
            !!a
            print(a.attr1 )
            a. reclassify (B)
            b ?= a
            b.execute
            b := Void
            a. reclassify (A)
            print (", ");
            print (a. attr1 );
            print ("%N");
        end

    A : A is
            -- Prototype object of class A.
        once
            !A!Result;
        end

    B : A is
            -- Prototype object of class B.
        once
            !B!Result;
        end

end -- class ROOT_CLASS
```

Ici encore, quelques commentaires sont les bienvenus : Les deux routines “once” permettent de créer des objets “prototypes” (référence au bouquin de Gamma à mettre ici). En donnant à ces routines le même nom que leur classe respective, cette convention permet d’émuler la notion d’“objet-classe” lors de l’appel de la routine `reclassify`.

Un objet `a` de type `A` est ensuite créé, avec un attribut `attr1` ayant la valeur initiale 0. Le type de `a` est alors changé dynamiquement en `B`. La variable `b` devient alors un alias sur l’objet `a`, dont la routine `execute` permet de modifier la valeur de `attr1`. Enfin, l’alias est annulé (`Void`) avant de restaurer le type original de `a`, afin de ne pas laisser accessible un alias de type `B` pour un objet de type `A`.

4.8 Conclusion

Forts de tous les travaux préliminaires présentés lors des sections précédentes, nous sommes à présent à même de contruire le LTS d’une spécification :

L’état initial du LTS est dérivé d’un diagramme de déploiement particulier (voir section 2.9) qui dénote l’état initial du système spécifié. À partir d’un état global donné, on peut construire l’ensemble des transitions du LTS sortant de cet état, et les états successeurs qui en résultent de la manière suivante :

1. On explore la configuration représentant l’état courant, et on prend successivement tous les objets actifs ayant une file de messages non vide.
2. Lorsque l’on a choisi un objet actif possédant cette propriété, on duplique alors complètement la configuration représentant l’état courant, qui deviendra par la suite celle de l’état successeur, par effets de bord des actions.
3. Le message en tête de file est retiré de la file, et devient l’événement courant.
4. L’opération sémantique `dispatch` est appelée sur ce message, ce qui a pour effet de faire exécuter par l’objet récepteur les actions associées à cet événement. La configuration courante est ainsi modifiée, et à l’issue de l’exécution des actions, elle représente l’état final de la transition dans le LTS.
5. On reprend alors au point 1 en choisissant un nouvel objet actif.

Cet algorithme repose sur l’hypothèse que les transitions des machines à états des objets actifs sont toujours déterministes, c’est-à-dire que si plusieurs transitions sortant d’un même état sont déclenchées par le même événement, alors leurs gardes respectives sont mutuellement exclusives. Cette hypothèse est nécessaire pour pouvoir compiler le domaine sémantique des machines à états en suivant les principes expliqués à la section 4.4.6 (il n’y a qu’une seule opération `e` dans la classe `A_IN_S1`).

Si aucune garde n’est satisfaite, alors d’après la façon dont sont compilées les gardes, aucune action ne sera exécutée lors de l’appel à `dispatch`. C’est conforme

au comportement par défaut prescrit par UML, qui est qu'un événement ne pouvant être traité est perdu. Cependant, UML offre aussi la possibilité d'associer à chaque état une liste d'événements qui seront conservés dans la file s'ils ne peuvent être traités au moment voulu (*deferred events*). Cette possibilité n'est pas actuellement supportée par notre simulateur.

Ces deux limitations significatives sont toutefois en passe d'être levées dans la nouvelle version du simulateur, dans laquelle un schéma de compilation légèrement différent a été choisi pour les machines à états. Dans ce nouveau schéma de compilation, chaque transition est réifiée en tant que sous-classe d'une classe `Transition` du domaine sémantique. Cette classe `Transition` dispose d'une opération sémantique appelée `fire` et chargée d'exécuter les actions représentant l'effet de la transition. Nous ne donnerons pas plus d'informations dans ce document quant à cette seconde version du simulateur UML, toujours en cours de développement au moment où nous écrivons ces lignes.

Chapitre 5

Mise en œuvre et applications

5.1 L'outil UMLAUT

5.1.1 Un environnement dédié à la manipulation de spécifications UML

UMLAUT [67] est un outil dédié à la manipulation de modèles UML. UMLAUT a été conçu dès le départ comme un outil ouvert, capable de lire des spécifications UML sauvegardées dans le format standardisé XMI et donc facilement interfaçable avec les ateliers de génie logiciel existants supportant la notation UML. Il peut toutefois être utilisé de manière autonome, via une interface graphique développée en Java. Cette interface supporte les types de diagrammes les plus importants (cas d'utilisation, diagrammes de classes, machines à états, déploiements, diagrammes de collaborations et diagrammes de séquences).

UMLAUT est un outil modulaire et extensible. Outre l'interface graphique et les modules d'importations de modèles, l'outil dispose d'une bibliothèque d'opérateurs de transformations pouvant être appliquées à des modèles UML, ainsi que d'un module permettant d'analyser et d'évaluer des expressions écrites en OCL [100]. Enfin, UMLAUT donne accès aux techniques formelles grâce à son module de simulation, capable de compiler une spécification UML sous la forme d'un système de transitions étiquetées construit à la volée en suivant les principes expliqués dans le chapitre 4. Une spécification UML compilée de la sorte est ainsi directement exploitable par les outils de la boîte à outils CADP [42], qui sont tous basés sur la même interface vers un système de transitions (l'API «graph» de CADP) qu'implémente UMLAUT. Parmi les outils disponibles, on peut citer un simulateur interactif permettant d'explorer le comportement du système, un *model checker*, et le générateur de tests TGV [64].

5.2 Le module de simulation

5.2.1 Duplication et sauvegarde de graphes d'objets

Nous avons vu au chapitre précédent, à la section 4.6.4, que pour pouvoir prendre en compte les actions non-interprétées, il était nécessaire que le simulateur puisse dupliquer une configuration. Une configuration du domaine sémantique compilé est un graphe d'objets. L'article [47] traite de la copie profonde de graphes d'objets, sans toutefois aborder le problème crucial des cycles dans les graphes d'objets.

Il est important pour pouvoir ensuite comparer deux configurations que le résultat soit isomorphe au graphe d'origine.

La version actuelle du simulateur de UMLAUT est écrite dans le langage Eiffel, qui dispose nativement d'une fonction de duplication de graphe appelée `deep_clone`, que nous avons décidé d'utiliser (et que nous avons d'ailleurs même implantée comme contribution au compilateur GNU SmallEiffel[1] qui avait omis cette routine jusqu'à présent).

5.2.2 Comparaison profonde de graphes d'objets

Déterminer si deux états globaux sont équivalents d'un point de vue de leurs comportements futurs est sans doute l'aspect le plus délicat dans la réalisation du simulateur.

Aspect correction : La fonction doit être suffisamment discriminante pour ne pas confondre accidentellement deux états qui ne sont pas équivalents au regard des propriétés que l'on souhaite vérifier. Cette fonction doit également être capable d'identifier correctement tout couple de graphes isomorphes, afin de détecter les comportements cycliques et donc autoriser la construction exhaustive d'un graphe d'accessibilité pour une spécification à nombre fini d'états mais comportant des cycles.

Aspect performance : La fréquence d'appel de cette fonction est élevée, donc sa performance est critique. Par ailleurs, la discrimination plus ou moins forte dont est capable cette fonction va avoir une conséquence directe sur le nombre d'états qui vont être explorés et donc sur la mémoire requise pour réaliser une simulation exhaustive (lorsque cela est possible).

Similairement à la routine deep_clone, Eiffel dispose d'une routine deep_equal dans Eiffel dont l'*intention* est de comparer si deux graphes sont isomorphes. Nous avons mis l'accent sur le mot *intention*, car la définition officielle du langage Eiffel [78] pour deep_equal ne garantit en fait pas l'isomorphisme (elle définit une relation plus faible). Une étude complémentaire, disponible sur le WWW à l'adresse

1. http://SmallEiffel.loria.fr/

http://www.irisa.fr/prive/aleguenn/deep_equal/deep_equal.html, explique en détails les problèmes que comporte la définition officielle et certaines implantations de deep_equal par des compilateurs Eiffel commerciaux.

5.3 Projet OURAL

5.3.1 Objectifs du projet

Le projet OURAL est un projet pré-compétitif RNRT. Ce projet s'est déroulé sur une durée de deux ans, à partir de Février 1999. Nos partenaires étaient Softeam, Alcatel et Thomson LCR.

L'objectif de ce projet était de fournir un ensemble d'outils destinés à accélérer et améliorer le processus industriel de développement de produit par le biais de réutilisation de composants logiciels. Dans ce processus, nous nous focalisions essentiellement sur les phases "amont" que sont : la définition et analyse d'architecture, la validation de l'architecture, l'exploitation des informations en vue de la génération de scénarios exécutables et la génération de squelettes de code. Sur une échéance de deux ans, le projet avait pour objectif la conception et le développement d'un environnement autour d'UML permettant à la fois :

- de décrire une application du point de vue de son architecture dans le formalisme standard UML,
- de valider fonctionnellement cette architecture
- de valider, notamment par la simulation, des aspects non fonctionnels de l'architecture
- de dériver cette description sur une plate-forme générique basée sur l'architecture de Corba

Le projet a permis avec succès de fédérer autour du langage UML les différents outils des partenaires, et de promouvoir un processus de développement industriel basé sur une modélisation unique de l'application.

L'outil UMLAUT et sa connexion avec la boîte à outils CADP ont été directement mis en oeuvre pour la partie validation fonctionnelle du projet, dont voici le résumé :

> Le sous projet outils d'ingénierie fonctionnelle a pour but de définir et fournir un outillage destiné à faire une validation formelle d'une architecture logicielle décrite en UML/T. Ces travaux s'articulent autour des extensions d'UML destinées à enrichir la description fonctionnelle du comportement d'une classe. L'environnement propose un simulateur de comportement et un générateur de tests. L'utilisation de tels outils nécéssitent une modélisation fonctionnelle de la plate-forme d'exécution générique.

5.3.2 LTS d'un système de contrôle de trafic aérien

Une modélisation UML d'un système de contrôle de trafic aérien (ATC) a servi de cas d'étude pour démontrer les possibilités de l'outil UMLAUT. Ce cas d'étude est décrit en détail dans l'article [48], et nous avons choisi de ne présenter ici que les conclusions de cette étude.

Le tableau 5.1 résume les résultats obtenus en utilisant UMLAUT pour construire le graphe d'accessibilité de l'ATC, en bornant la taille des files à 1, puis à 2.

Taille des files	1	2
Nombre d'états	13094	560216
Nombre de transitions	45396	2290713
Taille mémoire nécessaire	74MB	4GB

TAB. 5.1 – *Résultat du simulateur sur l'exemple de l'ATC*

Notons que la taille mémoire qui peut sembler importante nécessaire à la construction du graphe complet pourrait être réduite d'un facteur nb-transitions/nb-états : Ce facteur est dû au fait que le simulateur n'est pas prévenu lorsque certains états ne sont plus utilisés et ne peut donc pas libérer la mémoire correspondante. Ce problème n'est pas fondamental mais technique, et est sur le point d'être résolu.

5.4 Collaboration avec Gemplus

UMLAUT a également été utilisé dans un projet de recherche mené par la société Gemplus. L'étude [74], réalisée par Hugues Martin et Lydie Dubousquet, consistait à comparer la correction et la pertinence de cas de tests générés automatiquement à partir de spécification avecdes cas de tests générés manuellement.

Le système spécifié est une application de porte monnaie électronique pour Java Card, UML fut utilisé comme langage de modélisation et UMLAUT comme Atelier de Génie Logiciel pour la saisie de la spécification. Les tests furent synthétisés en utilisant la connexion entre UMLAUT et CADP, plus précisément à l'aide de l'outil de synthèse de tests TGV qui fait partie de CADP.

L'utilisation d'UML est apparue comme étant un bon compromis entre formalisme et contraintes industrielles, cette notation étant déjà connue des développeurs. De plus, le fait que TGV fontionne à partir d'objectifs de tests à permis de produire des tests en se basant sur l'expertise des développeurs, qui ont su cibler les parties importantes à tester.

Cette étude est aussi à l'origine de travaux complémentaires visant à améliorer l'expressivité du langage de test. Ces travaux en sont encore à un stade préliminaire

et seront évoqués dans la section 6.2.2 en perspectives de cette thèse. Les objectifs de tests tels qu'ils sont actuellement spécifiables dans UMLAUT souffrent en effet d'un certain manque de paramétrisation. Beaucoup de motifs récurrents dans les objectifs ont été observés au cours de cette étude, et il aurait été appréciable de pouvoir spécifier un objectif de test "générique", à partir duquel il aurait été possible de décliner ensuite automatiquement les objectifs de tests concrets correspondant à toutes les valeurs possibles des paramètres. Cette possibilité de générer des ensembles d'objectifs de tests à partir de schémas de test n'existe pas à l'heure actuelle dans UMLAUT, et a donc du être réalisée indépendamment.

Cette étude a également comparé les résultats obtenus grâce à des tests manuels par rapport à des tests générés automatiquement par UMLAUT. Les tests générés automatiquement ont mis en évidence autant d'erreurs dans l'implantation que les tests produits manuellement. Cependant, les erreurs trouvées étaient de nature différente. Notamment, le modèle UML a dû être simplifié (par abstraction de certaines données) afin d'être traitable par UMLAUT/TGV. Certaines erreurs mettant en œuvre des valeurs précises des données qui ont été abstraites n'ont donc pû être détectées par les tests générés automatiquement. Cependant, les tests générés présentaient la qualité d'être systématiques, ce qui a permis de trouver des erreurs dans des fonctionnalités qui ont été négligées par les tests manuels.

Chapitre 6

conclusions et perspectives

6.1 Contributions majeures

6.1.1 Une chaîne opérationnelle d'outils formels pour UML

Nous avons montré que des méthodes formelles étaient applicables à des spécifications écrites en UML. Nous nous appuyons sur le modèle des systèmes de transitions étiquetées pour donner une sémantique précise et suffisante à un sous-ensemble d'UML pour les techniques que l'on souhaite mettre en œuvre. Le simulateur inclus dans UMLAUT est déjà en mesure d'explorer exhaustivement des graphes de taille respectable et certaines améliorations devraient nous permettre d'en repousser les limites. Il peut aussi fonctionner à la volée, lorsque les outils qui lui sont couplés savent exploiter ce mode et peut donc potentiellement travailler sur des graphes infinis (y compris des graphes à branchement infini).

L'utilisation de la boîte à outils CADP, et notamment de TGV, permet de produire automatiquement des cas de tests à partir de spécifications UML et d'objectifs de test pertinents.

6.1.2 Intégration sémantique des vues d'UML

Simuler des spécifications UML en exprimant leur comportement sous forme de LTS n'est pas chose aisée a priori. Les différents diagrammes de UML ne forment pas un tout nécessairement complet ni même cohérent.

Il nous a donc fallu montrer comment il était possible d'unifier certains concepts redondants de UML (objet de la section 4.3), afin de garantir une meilleure cohérence des spécifications UML. Ces propositions ont été partiellement concrétisées au sein de l'outil UMLAUT, qui permet de réaliser des spécifications UML dont les différents diagrammes sont cohérents *par construction*, ce qui est loin d'être toujours le cas dans les outils UML existants.

6.1.3 Formalisation des patrons de conception

Enfin, au cours de notre étude, les patrons de conception nous sont apparus comme des outils très précieux :

- Leur utilisation permet d'aider le développeur à réaliser ses spécifications UML, en réutilisant l'expertise d'architectes logiciels plus expérimentés, comme nous le montrons dans la section 2.5.2.
- Ces même patrons de conceptions se sont aussi révélés utiles au niveau méta, par exemple lorsqu'il s'est agit d'unifier les concepts de classe et d'états (section 4.3), et lors de l'étude des mécanismes de communication (section 2.6.7).

Il devenait donc intéressant, si ce n'est même nécessaire, de mieux formaliser les patrons de conception. Nous fûmes donc relativement surpris en étudiant la littérature UML de constater que les patrons de conceptions y étaient traités de manière très superficielle. Nous avons donc dans un premier temps étudié en détail les propositions existantes et leurs nombreuses limites. Les résultats de cette étude ont été publiés dans [95]. Nous avons ensuite proposé une approche originale consistant à modéliser les patrons de conceptions en tant que contrainte sur la structure des spécifications UML. Ces contraintes sont similaires (et dans un certain sens complémentaires) aux règles OCL de niveau méta formant la sémantique statique d'UML. Ces travaux étant assez indépendants de la thématique principale de cette thèse, nous ne les développerons pas davantage ici. Le lecteur intéressé trouvera tous les détails de l'approche que nous proposons dans [49].

6.2 Perspectives

6.2.1 Une logique temporelle adaptée à UML

Il n'existe pas à l'heure actuelle de logique temporelle adaptée pour UML qui permettrait de diminuer la "distance" qui sépare les formules en logique temporelle de la spécification UML. L'apparition d'OCL dans UML offre l'opportunité de changer cet état de fait.

L'Object Constraint Language [100] est un petit langage textuel inspiré de Z [92] permettant d'exprimer des assertions (préconditions, postconditions, invariants) que le système ne doit pas enfreindre, en utilisant la théorie des ensembles.

OCL possède la particularité intéressante d'être très bien intégré à UML. Les attributs et associations définis dans une spécification UML sont "navigables" à l'aide d'OCL, dénotant ainsi les objets atteints sous forme de collections d'objets manipulables via un ensemble complet d'opérations prédéfinies (test de présence, union, intersection, etc...).

OCL constitue donc en quelque sorte un premier pas vers l'utilisation des langages formels pour exprimer des contraintes ou des propriétés dans un contexte UML. OCL

est à présent relativement familier aux utilisateurs d'UML, qui sont de plus en plus enclins à l'utiliser en phase de spécification. Certaines méthodes basées sur UML préconisent d'ailleurs d'utiliser intensivement OCL, et participent ainsi à son adoption. C'est notamment le cas de la méthode Catalysis [32].

Il est remarquable qu'OCL constitue, malgré ses limites, l'une des parties les plus formelles d'UML. Cependant, une part conséquente des exigences porte sur le comportement et les réactions du système et non pas seulement sur son état à un instant donné. Seule l'utilisation d'une logique temporelle permet d'exprimer ce type de contraintes. Étendre OCL avec des opérateurs de logique temporelle permettrait d'introduire en douceur l'utilisation d'une logique temporelle au sein d'une méthodologie à objets reposant sur UML, en se basant sur la syntaxe à présent familière d'OCL pour l'expression des propositions atomiques.

Certains travaux comme [86] vont d'ailleurs dans ce sens, même s'il reste encore beaucoup à faire. Les travaux présentés dans [34] visent à identifier les motifs récurrents dans les formules de logiques temporelles classiques, ce qui permettra peut-être de définir un nouveau langage plus simple d'utilisation car mieux adapté aux cas les plus fréquemment rencontrés. Une autre piste intéressante est présentée par Corbett dans [28]. Il y est question d'une logique temporelle spécialement adaptée au langage Java. Ces travaux pourraient avoir un impact sur UML en raison de la similarité des deux langages sur certains aspects (orientation objets, liaison dynamique, création dynamique d'objets, etc).

Notons enfin qu'une logique temporelle bien adaptée à UML pourrait aussi s'avérer très utile pour spécifier les aspects comportementaux des patrons de conception.

6.2.2 Vers une plus grande expressivité du langage de test

Une piste intéressante consiste à poursuivre la formalisation des cas d'utilisation et des collaborations de UML commencée par [82] ainsi que leur lien avec les objectifs de test tels que sait les exploiter TGV, afin d'offrir une aide à la conception des *objectifs* de test, ou mieux encore, de partiellement automatiser cette tâche.

En effet, si les objectifs de tests avec valeurs instanciées que nous avons étudiés à la section 3.3.4.5 ont l'avantage d'être assez facilement traduisibles dans le format d'entrée de TGV, ils sont en contre partie d'une expressivité assez faible, car ils ne sont en effet aucunement paramétrés.

Quelques exemples de problèmes illustrent les limites de cette approche :

- Représenter la propagation d'une valeur *arbitraire* dans un scénario (on envoie x, on s'attend à recevoir $x + 1$)
- Représenter des choix portant sur des domaines de valeurs étendus.
- Représenter un ensemble de scénarios similaires où seuls certains type de messages varient (variante du premier cas, ou l'arbitraire n'est plus une valeur mais un message).

Une solution permettant de résoudre de manière satisfaisante ces problèmes consiste à rendre les objectifs de tests paramétrables. Notons que bien que l'approche décrite ci-dessous présente des similitudes avec des approches de test symbolique, elle n'en n'a pas toute la puissance. En effet, seuls ici les objectifs de tests seront "symboliques", le graphe d'accessibilité et donc les cas de test finaux restant quant à eux énumératifs, c'est-à-dire avec valeurs instanciées et non symboliques.

6.2.2.1 Paramètres formels d'un objectif de test

UML offre deux mécanismes de paramétrisation :

- Les *templates*, à la base de la généricité.
- Les rôles, à la base des diagrammes de séquences de niveau spécification, et plus généralement des collaborations et interactions entre objets.

Il est frappant que ces deux notions soient également à la base des patrons de conception de UML. On peut donc bien parler de schémas ou *patterns* d'objectifs de tests.

6.2.3 Simulation sans description opérationnelle des actions

Afin de pouvoir simuler une spécification UML et ainsi construire le système de transition étiqueté, nous avons vu dans la section 4.6.4 que notre approche nécessitait de donner une description opérationnelle des routines, au travers de la partie "actions" de UML. Les routines peuvent pourtant être équipées d'assertions (pré et post conditions), mais celles-ci ne sont utilisées qu'à des fins de vérifications. Malheureusement, les descriptions opérationnelles impliquent bien souvent une *sur-spécification* et obligent à rentrer à un niveau de détail très élevé. Une amélioration notable consisterait à pouvoir simuler des spécifications UML en utilisant la vision déclarative offerte par les assertions, et non plus la vision opérationnelle offerte par les actions.

Cela pourrait s'envisager en couplant plusieurs techniques complémentaires :

- Transformation d'équations OCL en système de réécriture lorsque cela est possible
- Énumération des valeurs vérifiant les équations dans le cas contraire
- Utilisation d'expressions intermédiaires visant à "guider" la résolution de contraintes pour éviter l'explosion combinatoire entraînée par l'énumération exhaustive des valeurs

Les résultat existants en résolution de contraintes pourraient s'avérer fort utiles dans ce contexte.

Bibliographie

[1] Updated joint initial submission against the action semantics for UML RFP, available at http://cgi.omg.org/cgi-bin/doc?ad/00-08-03.

[2] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 383–395. Springer, 2000.

[3] J. R. Abrial. *The B reference manual*. Edinburgh Portable Compilers, 17 Alva Street, Edinburgh EH2 4PH, UK, 1991.

[4] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[5] B. Achee and D. Carver. Object Extensions to Z: a Survey. *Int. Journal on Software Engineering and Knowledge Engineering*, 6(3):507–530, 1996.

[6] Christopher Alexander. *A Pattern Language: towns, buildings, construction*. Number 2 in Center for Environmental Structure series. Oxford University Press, New York, 1977.

[7] Christopher Alexander. *The Timeless Way of Building*. Number 1 in Center for Environmental Structure series. Oxford University Press, New York, 1980.

[8] Christopher Alexander. The origins of pattern theory: The future of the theory and the generation of a living world. *IEEE Software*, pages 71–82, September-/October 1999. Keynote speech at OOPSLA'96.

[9] Jean Bezivin and Richard Lemesle. Ontology-based layered semantics for precise OA&D modeling. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 31–37. Technische Universität München, TUM-I9725, 1997.

[10] Jean Bezivin and Richard Lemesle. Towards a true reflective modeling scheme. In *OORaSE*, pages 21–38, 1999.

[11] D. Bjørner. The VDM principles of software specification and program design. In *TC2 Work.Conf. on Formalization of Programming Concepts*, pages 44–74, LNCS Vol. 107, 1981. IFIP, Springer-Verlag.

[12] B. W. Boehm. The high cost of software. In Ellis Horowitz, editor, *Practical Strategies for Developing Large Software Systems*. Addison-Wesley, 1975.

[13] B. W. Boehm. Improving software productivity. *Computer*, 20(9):43–57, September 1987.

[14] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[15] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[16] Ed Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*. North Holland, 1989.

[17] Paulo J. F. Carreira and Miguel E. F. Costa. Automatically verifying an object-oriented specification of the steam-boiler system. In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, *Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000)*, pages 345–360. GMD, 2000.

[18] D. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier Science Publishers (North-Holland), 1990.

[19] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[20] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

[21] CCITT. *SDL, Recommendation Z.100*, 1987.

[22] Craig Chambers. Predicate classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[23] Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[24] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, 1992.

[25] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, NJ, 2nd ed. edition, 1991.

[26] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[27] James O. Coplien. A generative development-process pattern language. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 183–237. Addison Wesley, Reading, MA, 1995.

[28] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In K. Havelund, J. Penix, and W. Visser, editors, *Proceedings of the seventh SPIN Software Model Checking Workshop*, volume 1885 of *LNCS*, pages 205–223. Springer, 2000.

[29] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dspin: A dynamic extension of spin. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking. 5th and 6th International SPIN Workshops, Trento, Italy, July 1999, Toulouse, France, September 1999. Proceedings*, volume 1680 of *LNCS*, pages 261–276. Springer, 1999.

[30] M. Von der Beeck. A comparison of statecharts variants. *Lecture Notes in Computer Science*, 863:128–??, 1994.

[31] Dino Distefano, J.-P. Katoen, and Arend Rensink. On a temporal logic for object-based systems. Technical report CTIT 00-06, University of Twente, 2000.

[32] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[33] E. H. Dürr and J. van Katwijk. VDM++ – A Formal Specification Language for Object-oriented Designs. In Bertrand Meyer Georg Heeg, Boris Magnusson, editor, *Technology of Object-oriented Languages and Systems*, pages 63–78. Prentice-Hall International, 1992. Proceedings of Tools Europe '92.

[34] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press / ACM Press, 1999.

[35] Philippe Schnoebelen et. al. *Vérification de Logiciels: Techniques et outils du model-checking*. Vuibert, April 1999.

[36] J.-C. Fernandez, H. Garavel, L. Mounier, C. Rodriguez A. Rasse, and J. Sifakis. A toolbox for the verification of programs. In *International Conference on Software Engineering, ICSE'14, Melbourne, Australia*, pages 246–259, May 1992.

[37] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102, pages 437–440, August 1996.

[38] Wolfgang Fleisch. Applying use cases for the requirements validation of component-based real-time software. In *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, May 1999.

[39] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[40] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 472–491. Springer, 2000.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, 1995.

[42] Hubert Garavel. Open/caesar: An open software architecture for verification, simulation and testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384. Springer-Verlag, Lecture Notes in Computer Science, 1998.

[43] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996.

[44] Patrice Godefroid. Model checking for programming languages using VeriSoft. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[45] Martin Gogolla, Oliver Radfelder, and Mark Richters. A UML semantics FAQ - the view from bremen. In S. J. H. Kent, A. Evans, and B. Rumpe, editors, *Proc. ECOOP'99 Workshop UML Semantics FAQ*. University of Brighton, 1999.

[46] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[47] Peter Grogono and Markku Sakkinen. Copying and comparing: Problems and solutions. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 226–250. Springer, 2000.

[48] Alain Le Guennec. Méthodes formelles avec uml. In *CFIP'2000 : Colloque Francophone sur l'Ingénierie des Protocoles*. Hermes, oct 2000.

[49] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 482–496. Springer, 2000.

[50] David Harel. Statecharts: a visual formalism for complex systems. *Sciences of Computer Programming*, 8(3):231–274, June 1987.

[51] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[52] Wai Ming Ho. *Contribution à la Réification d'un processus de Conception*. PhD thesis, Université de Rennes 1, 2001.

[53] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[54] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[55] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.

[56] ISO. *Information Processing Systems, Open Systems Interconnection, OSI Conformance Testing Methodology and Framework*. ISO 9646.

[57] ISO. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. ISO/ DP 8807, March 1985.

[58] ISO. *Estelle: a Formal Description Technique based on an Extented State Transition Model*. ISO 9074 TC97/SC21/WG6.1, 1989.

[59] ISO. Proposed ITU-T Z.500 "framework: Formal methods in conformance testing", April 1997.

[60] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[61] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley/ACM Press, 1992.

[62] Claude Jard, Jean-Marc Jézéquel, Alain Le Guennec, and Benoit Caillaud. Protocol engineering using UML. *Annales des Telecoms*, 54(11–12):526–538, November 1999.

[63] Thierry Jéron, Jean-Marc Jézéquel, and Alain Le Guennec. Validation and test generation for object-oriented distributed software. In *IEEE Proc. Parallel and Distributed Software Engineering, PDSE'98, Kyoto, Japan*, April 1998.

[64] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *CAV'99, Trento, Italy*, pages 108–122. Springer, LNCS 1633, July 1999.

[65] J.-M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts*. Addison-Wesley, October 1999. ISBN 1-201-30959-9.

[66] Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Validating distributed software modeled with UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation.*

First International Workshop, Mulhouse, France, June 1998, pages 331–340, 1998.

[67] Jean-Marc Jézéquel, Wai Ming Ho, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[68] Ralph E. Johnson. Documenting frameworks using patterns. In SIGPLAN Notices, editor, *OOPSLA'92*, volume 27(10), pages 63–76, Vancouver BC, 1992.

[69] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, N.Y., June 1997.

[70] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.

[71] K. C. Lano. Z^{++}, an object-orientated extension to Z. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 151–172. Springer-Verlag, 1991.

[72] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal Aspects of Computing*, 11:637–664, 1999.

[73] Johan Lilius and Ivan Porres Paltor. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 430–445. Springer, 1999.

[74] Hugues Martin. *Une methodologie de generation automatique de suites de tests pour applets Java Card.* PhD thesis, Université de Lille I, March 2001.

[75] Radu Mateescu. Vérification des propriétés temporelles des programmes parallèles, April 1998.

[76] Radu Mateescu and Hubert Garavel. Xtl: A meta-language and tool for temporal logic model-checking. In Tiziana Margaria, editor, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark)*, pages 33–42. BRICS, July 1998.

[77] K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[78] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[79] Pierre-Alain Muller. *Modélisation Objet avec UML*. Number 2. Eyrolles, 2000.

[80] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 416–429. Springer, 1999.

[81] OMG. Action semantics for the UML RFP, available at http://cgi.omg.org/cgi-bin/doc?ad/98-11-01, 1998.

[82] Gunnar Övergaard. A formal approach to collaborations in the unified modeling language. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 99–115. Springer, 1999.

[83] Ivan Paltor and Johan Lilius. vUML: A tool for verifying UML models. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[84] Mario Paludetto and Jérome Delatour. UML et les réseaux de petri : vers une sémantique des modèles dynamiques et une méthodologie de développement des systèmes temps réel. *L'objet*, 5:443–467, 1999.

[85] M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties*. PhD thesis, Université de Bordeaux, 1994.

[86] Sita Ramakrishnan and John McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*, 1999.

[87] Mark Richters and Martin Gogolla. A metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 156–171. Springer, 1999.

[88] D. Bjørner and C. Jones. *The Vienna Development Method*. Springer Lecture Notes in Computer Science 61, 1978.

[89] UML RTF. *OMG Unified Modeling Language Specification, Version 1.3, UML RTF proposed final revision*. OMG, June 1999.

[90] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.

[91] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[92] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[93] Friedrich Steimann. A radical revision of UML's role concept. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 194–209. Springer, 2000.

[94] Perdita Stevens. On Use Cases and their relationships in the Unified Modelling Language. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029 of *LNCS*, pages 140–155. Springer, 2001.

[95] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 44–62. Springer, 2000.

[96] Gerson Sunyé, François Pennaneac'h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML Action Semantics for executable modeling and beyond. In *Proceedings of CAiSE 2001*, LNCS. Springer, 2001.

[97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[98] J.-P. Talpin, A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane, and H. Canon. BDL, a language of distributed reactive objects. In *ISORC'98, The 1st IEEE International Symposium on Object-oriented Real-time Distributed Computing*, Kyoto, Japan, April 1998.

[99] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[100] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.