N° d'ordre: 2589

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

François Pennaneac'h

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

Titre de la thèse :

UML : de l'action à la réflexion

À soutenir le ? novembre 2001 devant la commission d'examen

M. :	?	?	Président
MM. :	Denis	Caromel	Rapporteurs
	Pierre	Cointe	
	Jean-François	Perrot	
MM. :	Mireille	Ducassé	Examinateurs
	Pierre-Alain	Muller	
	Jean-Marc	Jézéquel	

Remerciements

Je remercie ? ?, , qui me fait l'honneur de présider ce jury.

Je remercie Denis Caromel, , Pierre Cointe, Professeur, École des Mines de Nantes, et Jean-François Perrot, Professeur, LIP6, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Mireille Ducassé, , et Pierre-Alain Muller, , d'avoir bien voulu juger ce travail.

Je remercie enfin Jean-Marc Jézéquel, Professeur, Université de Rennes 1, qui a dirigé ma thèse.

Résumé

Le langage de modélisation UML (Unified Modeling Language) est défini par une architecture de méta-modélisation à quatre niveaux. Le premier niveau (ou méta-méta-modèle) est défini comme un sous-ensemble du deuxième niveau (méta-modèle). Cette définition circulaire des concepts du langage aide à la compréhension de la notation UML.

Au-delà des aspects syntaxiques, nous suggérons d'utiliser la proposition d'un langage d'actions pour UML (*Action Semantics*) comme base d'une réflexion similaire pour la dynamique des modèles. L'utilisation de l'*Action Semantics* aux niveaux *méta* nous conduit à une clarification et une simplification de la sémantique de l'exécution des modèles UML. Conséquence immédiate, la conception d'interprétes ou de compilateurs de modèles est mieux maîtrisée car moins complexe et elle bénéficie naturellement des facilités offertes par UML en matière de gestion du développement logiciel. Cela nous conduit à l'ébauche d'un processus de développement établissant une séparation claire et une évolution indépendante entre le modèle de l'application et le modèle de son exécution. Nous explorons les divers avantages présentés par cette approche mise en œuvre dans l'outil de méta-modélisation UMLAUT.

Summary

The Unified Modeling Language (UML) is based on a four level meta-modeling architecture. The first level (also called meta-meta-model) is defined as a subset of the second level (meta-model). This circular definition of concepts helps the understanding of the UML notation.

Beyond syntaxic aspects, we think the Action Semantics Proposal for the UML may be a foundation of a similar approach for behaviours. We leverage the use of the Action Semantics at the meta levels and we clarify the semantics of execution of UML models. As a result, tool implementors benefit UML abilities to handle the software process for the realization of interpreters ou compilers for models. We then outline a software development process where the model of the application and the model of its execution are two distinct entities that are refined independantly. This approach is prototyped in the UMLAUT meta case-tool.

Table des matières

1 Introduction **13**

1.1 Logiciel et complexité 13
1.2 Le consensus UML 14
1.3 Processus : vers un consensus ? 15
1.4 Des ateliers logiciels performants 15
1.5 Contribution 16
1.6 Plan du document 17

2 UML : présentation et architecture **19**

2.1 L'approche à objets 19
2.1.1 Principales caractéristiques 19
2.1.2 Limitations et extensions 20
2.2 La notation UML 21
2.2.1 Une notation à ¡¡aspects¿¿ 21
2.2.2 L'exemple de la banque 22
2.2.3 Le méta-modèle UML 23
2.3 Vers une ingénierie des modèles 23
2.4 Architecture du MOF et de UML 24
2.4.1 Comparaison avec les autres architectures réflexives 28
2.5 Intérêts des architectures ¡¡méta¿¿ 29
2.5.1 Formaliser et manipuler le langage 29
2.5.2 Un support pour la méta-programmation 29

3 l'*Action Semantics* **31**

3.1 Pourquoi un langage d'actions ? 31
3.2 Présentation de l'*Action Semantics* 32
3.2.1 Le méta-modèle 33
3.2.2 Un modèle d'exécution 33
3.2.3 Une sémantique des actions 37
3.3 Exemple de la FIFO 39

4 Présentation et formalisation d'OCL **43**
4.1 Ce modèle est-il correct ? . . . 43
4.2 Présentation d'OCL . . . 43
4.2.1 Le paquet *UML_OCL* . . . 45
4.2.2 Naviguer dans un modèle . . . 48
4.2.3 Limitations et défauts d'OCL . . . 51
4.3 Formalisation d'OCL . . . 52
4.3.1 Un méta-modèle pour OCL . . . 52
4.3.2 Règles OCL . . . 56
4.3.3 Correspondance méta-modèles UML/OCL . . . 58
4.3.4 Caractéristiques de notre approche . . . 61
4.3.5 Sémantique d'OCL et modèle d'exécution . . . 62
4.3.6 Interface avec un langage de programmation . . . 63
4.3.7 Génération de code à partir d'OCL . . . 64
4.3.8 Un mot sur l'implantation . . . 65

5 Une sémantique pour UML **67**
5.1 De la non cohérence d'UML . . . 67
5.2 Simplifier et unifier . . . 68
5.3 Applications aux diagrammes de séquence . . . 69
5.3.1 Renforcer la sémantique des interactions . . . 70
5.3.2 Traduction des diagrammes de séquence vers l'*Action Semantics* 72
5.4 Construction de simulateurs UML . . . 73
5.4.1 AS et réflexion . . . 73
5.4.2 Sémantique opérationnelle pour l'*Action Semantics* . . . 74

6 UMLAUT : un outil pour le co-design **79**
6.1 OCL et modèle d'exécution . . . 79
6.1.1 Exemple élémentaire . . . 80
6.1.2 Le patron de conception ¡¡Singleton¿¿ . . . 80
6.1.3 Le patron de conception ¡¡Observateur¿¿ . . . 81
6.2 Des spécifications aux implantations . . . 82
6.2.1 Raffinage et Réusinage . . . 82
6.2.2 Un exemple de réusinage . . . 84
6.2.3 Définitions précises de raffinage et réusinage . . . 85
6.3 La simulation : un outil indispensable . . . 85
6.4 Raffinages et transformations . . . 86
6.4.1 Raffinage des transformations . . . 87
6.4.2 Transformation du modèle ou de l'interprète . . . 88
6.5 Des outils intégrés . . . 89
6.5.1 État de l'art des outils . . . 89
6.5.2 UML : un langage pour les transformations . . . 90
6.6 L'outil UMLAUT . . . 92
6.6.1 Architecture . . . 92

6.6.2 Contribution 92
6.6.3 Les visiteurs dans UMLAUT 94
6.6.4 Vers une auto-génération de l'outil 95

7 Conclusions et perspectives **97**
7.1 Contributions 97
7.1.1 Des outils OCL génériques 97
7.1.2 Des outils opérationnels de manipulation des modèles UML . . . 97
7.1.3 Une ébauche de processus dirigé par les outils 98
7.2 Perspectives 99
7.2.1 Expressivité accrue du langage OCL 99
7.2.2 Des outils encore plus évolués 99

8 Annexes
Contraintes OCL corrigées de UML 1.3 **101**

Table des figures

2.1 Notation UML et aspects 21
2.2 Cas d'utilisation de la banque 22
2.3 Diagramme des classes de la banque 23
2.4 Vers une ingénierie des modèles 23
2.5 Architecture ¡¡méta¿¿ à quatre niveaux 25
2.6 Le niveau méta-méta 26
2.7 Le niveau méta-méta est inclus dans le niveau méta 26
2.8 Architecture de UML 27
2.9 Un cœur réflexif pour UML 28

3.1 Intégration des méta-modèles UML et AS 33
3.2 Vue partielle du méta-modèle de l'AS 34
3.3 Positionnement de l'*Action Semantics* 35
3.4 Positionnement de l'AS dans l'architecture d'UML 35
3.5 Le cœur du modèle d'exécution : historiques et configurations 37
3.6 Modélisation de l'envoi de messages dans le modèle d'exécution 37
3.7 Spécification de l'étape `#execute` de l'exécution d'une *ReadAttributeAction* 39
3.8 Exemple de la file FIFO 39
3.9 Action d'un `push` 40
3.10 Diagramme d'objets spécifiant la méthode `push` 41

4.1 Un diag. de classe manifestement invalide 44
4.2 Exemple de la banque 45
4.3 Navigation de la propriété `a.b` 49
4.4 Navigations multiples, associations et typage 51
4.5 Interface OCL 54
4.6 Un méta-modèle pour OCL 55
4.7 Correspondance des types Eiffel, Java et OCL 64

5.1 Communications entre objets exprimées par un diagramme de séquence 70
5.2 Interactions et AS : deux représentations pour une opération 71
5.3 Compilation puis interprétation 73
5.4 Modélisation de l'envoi de messages dans le modèle d'exécution 75
5.5 Spécification AS d'une production 76
5.6 Formalisation des *life-cycles* avec l'AS 77

6.1 Un A est associé à 3 B au plus 80
6.2 Application du patron ¡¡Observateur¿¿ 81
6.3 Des spécifications à l'implantation : raffinages et réusinages 83
6.4 Un exemple de réusinage 84
6.5 Une implantation du patron ¡¡Observateur¿¿ 88
6.6 Principe du co-design 89
6.7 Spécialisation du domaine sémantique par héritage 91
6.8 Architecture de l'outil UMLAUT 93
6.9 La patron de conception ¡¡Visiteur¿¿ 94

Chapitre 1

Introduction

1.1 Logiciel et complexité

Les termes ¡¡plantage¿¿, ¡¡bug¿¿, ¡¡service pack¿¿ font désormais partie du vocabulaire courant de tout utilisateur *lambda* d'un ordinateur et semblent être entrés dans les mœurs comme des aléas liés à l'utilisation ¡¡normale¿¿ de programmes aux comportements parfois erratiques.

Cette situation met en avant le manque de qualité dont souffrent la plupart des logiciels actuellement développés pour le grand-public. Cela découle du fait que les systèmes informatiques croissent en complexité à un rythme tel que les techniques utilisées jusqu'à présent pour leur développement et leur mise au point s'avèrent désormais inadaptées. Pourtant, des méthodes existent pour répondre aux impératifs de qualité et de sécurité exigés par certaines branches de l'industrie pour leurs développements logiciels : l'aéronautique, les transports ou la banque recourent pour leurs systèmes les plus critiques aux méthodes formelles. Malheureusement ces techniques rigoureuses restent cantonnées à ces domaines spécifiques. Les raisons mises en avant pour expliquer leur diffusion confidentielle sont leur relative lourdeur de mise en œuvre et les fortes compétences qu'elles exigent ; de fait le surcoût qu'elles induisent les restreint à des systèmes où l'impact d'un dysfonctionnement est estimé à un coût supérieur à celui d'une correction après-coup.

Ce raisonnement ne s'applique évidemment plus lorsque des vies entre en jeu. Or, c'est de plus en plus souvent le cas. En effet, si nous considérons les utilisations récentes de l'informatique, celle-ci accapare insidieusement notre quotidien par l'intermédiaire des systèmes enfouis, dans tous les domaines de la vie courante, du lave-vaisselle à l'automobile, en passant par les systèmes médicaux. À cette omniprésence de l'informatique, s'ajoute une augmentation du caractère critique des tâches qui lui sont confiées. De plus, les systèmes sont de plus en plus souvent interconnectés, et des systèmes critiques peuvent être amenés à communiquer avec des systèmes non critiques.

Aussi est-il important d'établir un pont entre ces deux types de developpement, afin de garantir le respect des critères de qualité tout en maintenant les coûts à un niveau raisonnable, par exemple en favorisant l'interconnexion de ces systèmes ou la

réutilisation de composants ¡¡sur étagère¿¿, parfaitement documentés et de qualité mesurable. Un formalisme commun de description est à la base de toute communication, et UML tente de remplir cette fonction.

Cependant, un formalisme adapté à la réalisation des logiciels complexes n'est pas à lui seul un gage de qualité. Le développement doit nécessairement s'appuyer sur un processus de développement fiable et efficace. Ce processus doit offrir au concepteur des aides telles que traçabilité et raffinage. Ces concepts permettent d'assurer un passage quasi-continu d'une spécification à une implantation conforme, ainsi qu'une remontée efficace des dysfonctionnements de l'implantation vers l'un des modèles raffinés ou sa spécification.

Enfin, cette notation et ce processus de développement logiciel sont d'un intérêt limité s'ils ne sont pas outillés. La simulation des modèles, l'application automatique des patrons de conception, le réusinage d'une application ou encore le *round-trip engineering* devraient faire partie des fonctionnalités de tout bon atelier de génie logiciel. D'autant plus qu'au cours d'un développement logiciel, les concepteurs sont confrontés à bien des tâches répétitives, fastidieuses et purement mécaniques. Sans grand intérêt pour la conception, celles-ci sont coûteuses en temps et une source fréquente d'erreurs. Leur automatisation libère le concepteur qui peut dès lors se concentrer sur l'architecture de son application en faisant abstraction de détails de réalisation qui tendent à ¡¡polluer¿¿ les modèles, abaissent leur degré d'abstraction et nuisent à leur compréhension.

1.2 Le consensus UML

Pour répondre aux besoins de documentation et de spécification du logiciel, de nombreux langages graphiques ont vu le jour, en particulier au sein de la communauté des objets. Afin de privilégier la réutilisation de composants, d'architectures, ou de favoriser l'interconnexion entre des systèmes modélisés avec des notations différentes, il apparaît évident qu'une notation consensuelle est nécessaire.

De ce constat est née la notation UML, rapprochement entre les notations OMT [112], Booch [19] et OOSE [57], et quelques influences secondaires [105, 13, 11]. UML, empruntant le ¡¡meilleur¿¿ de chaque notation, a été standardisée à l'OMG en 1997 et s'est imposée comme ¡¡la¿¿ notation du moment ; les nombreux ateliers de génie logiciel qui l'ont adoptée témoignent de cet engouement.

Malheureusement, cette normalisation a été effectuée dans la précipitation, avec des considérations plus politiques et mercatiques que véritablement scientifiques. Sans doute sous la pression de vendeurs d'outils quelque peu empressés d'assurer une transition facile de leur outils existants vers la nouvelle norme.

À l'usage, il est rapidement apparu que la norme initiale était inexploitable pour le développement de logiciels fiables : la sémantique des modèles UML n'est pas toujours clairement definie et le document normatif comporte de nombreuses omissions et contradictions. Les milieux académiques ont saisi cette opportunité pour se raccrocher à la locomotive UML, partie sans eux. De nombreux travaux sur la sémantique d'UML ont permis au fil des révisions de la norme de supprimer petit à petit la plupart des

discordances. Diverses formalisations d'UML ont par ailleurs été proposées, exploitant des méthodes formelles antérieures à UML, mais mathématiquement éprouvées [2, 120].

Ces approches par traduction du domaine sémantique d'UML dans un formalisme autre comportent malheureusement des limitations que nous détaillons dans la section 3.2.2. De fait, elles se contentent souvent de formaliser un sous-ensemble restreint d'UML. De plus, elles exigent de la part de l'utilisateur UML un lourd investissement dans l'apprentissage d'une notation étrangère à UML, ce qui permet de douter de leur succès.

D'autres – à l'instar du groupe *Action Semantics* – ont suivi une approche différente, que nous pensons plus adaptée : ce groupe tente de définir le domaine sémantique d'UML grâce à la notation UML elle-même. Cette approche ¡¡réflexive¿¿ exploite le méta-modèle UML et accède directement aux concepts UML, facilitant leur manipulation par le concepteur, tout en restant parfaitement compréhensible par celui-ci car ne sortant pas du cadre UML. De plus, cette approche reste parfaitement transparente pour les concepteurs qui ont pris l'habitude d'insérer des morceaux de code dans leur modèle. Elle permet ainsi la réutilisation des modèles existants, l'*Action Semantics* définissant une traduction des principaux langages de programmation.

1.3 Processus : vers un consensus ?

L'agitation actuelle autour des processus de développement témoigne d'une prise de conscience de l'importance de ces derniers. L'apparition de nouveaux processus de développement (RUP [58, 73], Catalysis [38], OPEN[53]) rappelle le foisonnement des notations à objets qui a précédé le consensus UML. Ce rapprochement laisse supposer que le processus de développement devra lui aussi trouver un cadre adapté pour sa normalisation. Les travaux en cours à l'OMG autour du *Software Process Engineering Metamodel* (SPEM) [98] permettent d'envisager UML, ou plutôt le MOF (*Meta Object Facility* [94]) comme formalisme normalisé de description des processus.

1.4 Des ateliers logiciels performants

Enfin, il est urgent qu'une sémantique officielle soit adoptée pour UML, et que cette sémantique soit compréhensible par les concepteurs d'applications, mais aussi par les concepteurs des ateliers de génie logiciel. En effet, une sémantique officielle ouvre la voie à la réalisation de simulateurs de modèles UML, dès la phase de spécification et à toutes les étapes du processus de développement logiciel, facilitant ainsi la réalisation d'une implantation conforme à la spécification et la localisation au plus tôt des incohérences entre les spécifications et les besoins de l'utilisateur exprimés par des cas de tests.

Nous verrons comment le prototype UMLAUT, qui est un outil dédié à la manipulation de modèles UML, réalise certaines de ces fonctionnalités.

1.5 Contribution

Les travaux présentés dans cette thèse s'articulent principalement autour d'une proposition pour étendre UML par un langage d'action.

Dans un premier temps, nous avons participé à un groupe de travail dont l'objectif était de proposer à l'OMG un tel langage d'action appelé *Action Semantics*. Puis nous nous sommes intéressés à ses utilisations par delà une utilisation standard comme simple langage de description du comportement des modèles UML.

En particulier, nous avons cherché à faciliter la réalisation de simulateurs efficaces de modèles UML respectant la sémantique décrite dans la documentation officielle de l'*Action Semantics*. Cette démarche demande un lourd investissement, car la présentation de la sémantique de l'*Action Semantics* favorise sa compréhension par les concepteurs d'applications UML, mais n'est pas suffisamment opérationnelle pour simplifier la tâche des concepteurs d'ateliers de génie logiciel. Nous adoptons une approche originale, exploitant l'architecture ¡¡méta¿¿ en couches d'UML pour construire des implantations de l'*Action Semantics* conformes en utilisant un noyau minimal réflexif d'UML, de l'*Action Semantics* et d'OCL. L'*Action Semantics* n'étant que l'une des multiples vues exprimant la dynamique des modèles UML, nous étudions également la traduction des autres vues dynamiques vers l'*Action Semantics*. Ceci nous permet à la fois de formaliser leur sémantique et de simuler les comportements exprimés dans ces vues grâce à notre simulateur de l'*Action Semantics*.

En approfondissant cette approche, nous tirons parti de l'architecture à quatre couches d'UML et de l'*Action Semantics* pour proposer des améliorations au processus du développement logiciel. Cette démarche s'appuie sur un atelier de génie logiciel offrant à l'utilisateur l'accès aux niveaux ¡¡méta¿¿ et la possibilité de manipuler n'importe lequel de ces niveaux avec un seul et même langage, en l'occurrence UML. Nous illustrons l'automatisation du processus par l'intermédiaire de transformations de modèles écrites au niveau du méta-modèle et spécifiées avec l'*Action Semantics*. Cette étude nous amène à classer les transformations de modèles en catégories : d'un côté, celles qui sont spécifiques à l'application ; de l'autre côté celles qui sont liées à l'exécution de cette application. En effet, il est fréquent que lors du développement d'une application des concepts liés à l'implantation s'immiscent dans le modèle de l'application. Cela nous conduira à proposer le principe du ¡¡co-design¿¿ où le modèle d'exécution et le modèle de l'application évoluent indépendamment l'un de l'autre, dans des niveaux de modélisation distincts, et produisent finalement une implantation globale réalisant la fusion de ces différents aspects par un mécanisme de ¡¡tissage¿¿[1].

Ces études nous ont amenés à participer au groupe de travail sur l'*Action Semantics* de l'OMG, dans le cadre du projet RNRT Convergence [1] d'unification des approches SDL[92] et UML. Dans ce but, nous avons participé à la définition et la validation de la proposition du groupe *Action Semantics* [5] en réponse à la *Request for Proposal for an Action Semantics for UML* [93]. Nous nous sommes attachés plus particulièrement à la correction des contraintes OCL [126], langage à la base de la définition de la sémantique

[1] de l'anglais *weaving* utilisé pour décrire ce mécanisme dans la programmation par aspects.

de l'AS. Cette étude a exigé l'intégration à l'outil UMLAUT de fonctionnalités avancées de manipulation des contraintes OCL. Les travaux présentés dans cette thèse ont donné lieu aux publications [63, 60, 59, 61, 62, 128, 39, 55, 101, 100].

Enfin, l'approche présentée dans cette thèse pour la réalisation d'outils de méta-modélisation réflexifs et l'implantation de simulateurs de modèles UML est mise en œuvre dans le prototype UMLAUT.

1.6 Plan du document

Afin de guider le lecteur, nous décrivons brièvement l'organisation de ce document.

L'étude présentée dans ce document s'articule autour de la notation à objets UML et exploite les caractéristiques de son architecture ¡¡méta¿¿ à 4 couches. Dans le chapitre 2 nous présentons brièvement les caractéristiques de l'approche à objets, ainsi que ses limitations et les solutions proposées pour y remédier. Nous nous attachons ensuite à présenter brièvement la notation UML, ainsi que son architecture ¡¡méta¿¿ en couches. Nous détaillons les propriétés génériques de ces architectures en couches et les implantations qui existent. Enfin, nous explorons les possibilités que ces architectures offrent au génie logiciel pour résoudre les limitations intrinsèques au ¡¡tout objet¿¿ et améliorer le processus de développement logiciel par automatisation de certaines tâches.

Le chapitre 3 présente l'extension à UML appelée *Action Semantics* qui sera au centre de notre étude.

UML et l'*Action Semantics* s'appuient largement sur un langage pour la spécification de contraintes nommé OCL. Nous le présentons brièvement dans le chapitre 4 et comme ce langage est fondamental à notre étude, nous nous intéressons ensuite à sa formalisation. Nous utilisons le cadre d'UML pour cette formalisation.

Dans le chapitre 5 nous décrivons les problèmes de la notation UML, dûs à son aspect multi-vues, en particulier en ce qui concerne la sémantique comportementale des modèles. Nous verrons que l'*Action Semantics* peut servir de base fondatrice à la formalisation d'UML. Nous utilisons alors intensivement le langage OCL. Cette formalisation du comportement des modèles nous conduit naturellement à rechercher à les simuler. Nous en déduisons un mécanisme facilitant l'implantation des simulateurs du langage UML. Pour cela, nous déduisons une forme opérationnelle, facile à implanter, de la spécification de l'*Action Semantics*. Nous utilisons l'*Action Semantics* pour décrire à la fois cette implantation et le processus de raffinage de la spécification à l'implantation.

Enfin, la manipulation conjointe des modèles UML et de la sémantique d'exécution de ces modèles nous conduit à l'ébauche d'un processus de développement que nous appelons *codesign*. Cette approche est évoquée dans le chapitre 6 et a été donné lieu au prototype UMLAUT.

Finalement, nous concluons par un bilan de notre contribution et nous proposons quelques perspectives dans le chapitre 7.

Chapitre 2

UML : présentation et architecture

2.1 L'approche à objets

2.1.1 Principales caractéristiques

Le paradigme objet [84], apparu avec Simula67 [90], se prête particulièrement bien au développement des grands systèmes. Il permet une décomposition modulaire et structurée du problème en localisant les tâches dans des entités responsables et bien définies appelées objets. Les fonctionnalités de l'application sont réalisées par la coopération d'objets qui communiquent en s'échangeant des messages. Les principales caractéristiques des technologies à objets sont :

- masquer des informations, ce qui permet de cacher la structure d'un objet derrière une interface définissant les fonctionnalités, ou ¡¡services¿¿, assurés par cet objet ;
- l'héritage, qui permet de réutiliser des systèmes existants par spécialisation ;
- le polymorphisme (réalisé par le mécanisme de liaison dynamique), qui permet d'adapter à l'exécution le comportement d'une application en fonction du type d'objet considéré, sans modifier l'implantation de l'application ;
- la délégation qui permet la réalisation des fonctionnalités complexes en partageant une tâche en sous-tâches simples réparties entre plusieurs objets coopératifs.

Le paradigme objet s'appuie sur un grand nombre d'outils. Les nombreux langages à objets (C++ [121], Java [46], Eiffel [83, 84], Ada95 [12]), méthodes à objets, ateliers de génie logiciel objet, librairies, ou composants (Corba[87], COM[109], JavaBeans[86]), illustrent la large adoption de ces principes.

Les logiciels se complexifiant, il devenait urgent de proposer des mécanismes pour documenter l'usage des systèmes (pour leur réutilisation), mais aussi leur conception (pour réutiliser l'état de l'art en la matière, les ¡¡bonnes pratiques¿¿). Les méthodes graphiques OMT [112], OOSE [57], Booch [19], OORam [105], (Classes, Responsabilités, Collaborations) (CRC Cards) [13], ou Entité-Relation [11], pour ne citer que les principales, sont apparues pour répondre à ce besoin. Malheureusement, leur diversité est génante : le manque d'uniformisation, les difficultés à communiquer entre projets et

outils constituent en effet un frein au déploiement des méthodes de développement à objets. Il est donc souhaitable d'adopter un langage commun convenant à la modélisation des systèmes informatiques mais également suffisamment générique pour prendre en compte des problèmes d'autres domaines. De plus, chacune de ces méthodes se focalise sur des aspects précis de la modélisation et s'avère insuffisante sur d'autres points.

2.1.2 Limitations et extensions

Nous avons expliqué dans la section 2.1.1 que l'approche à objets permet de localiser les tâches dans des entités responsables et bien définies. Cette manière de découper le problème est appréciable pour gérer des problèmes de taille et de complexité importantes, mais elle a ses inconvénients. En particulier, cela entraine des artéfacts non souhaité dans l'implantation. Le plus ¡¡visible¿¿ d'entre eux est sans doute la notation pointée des langages à objets qui dans le cas d'une collaboration entre objets jouant des rôles ¡¡symétriques¿¿ oblige à une implantation rompant ce caractère de symétrie ou de régularité. Il n'est malheureusement pas toujours concevable d'étendre un système par réutilisation de l'existant sans casser les masques d'information, en raison des dépendances introduites entre cet existant et les nouvelles fonctionnalités par les mécanismes de surcharge, polymorphisme, héritage ou encore délégation. Comme le note Coplien dans [34], ce problème est le fait d'un paradigme ¡¡tout objet¿¿ qui ne permet pas d'exprimer toutes les interactions complexes du monde réel.

À titre d'exemple, considérons le cas d'une application bancaire permettant à un client d'effectuer des dépôts et des retraits sur un ensemble de comptes bancaires. Nous souhaitons ajouter la possibilité d'enregistrer toutes les transactions effectuées sur un compte, afin d'éditer des relevés de compte détaillés. Cela nécessite d'ajouter à toutes les opérations de la classe `Compte` du code spécifique à cette fonctionnalité, par exemple pour écrire le montant retiré ou déposé sur le compte dans un fichier. Cette dispersion de l'implantation d'une propriété au travers de plusieurs entités de l'application est génante car elle conduit au phénomène du ¡¡code spaghetti¿¿, incompréhensible et très difficile à maintenir.

Diverses approches ont été étudiées pour s'attaquer à ces problèmes. Toutes sont basées sur le principe de la *separation of concerns* [123, 40], littéralement ¡¡séparation des préoccupations¿¿. Ces techniques visent à focaliser l'attention du concepteur sur un point important à la fois. Il s'agit en particulier de séparer l'¡¡espace des problèmes¿¿ de l'¡¡espace des solutions¿¿. En effet, le premier recoupe les abstractions spécifiques au domaine, celles qui sont le plus importantes pour le concepteur. Le second contient les abstractions spécifiques à l'implantation de la solution.

Kiczales [69, 76] a introduit la notion de ¡¡programmation par aspect¿¿ [22, 40, 55] qui scinde la réalisation en deux : les caractéristiques fonctionnelles de l'application sont conçues séparément de ses caractéristiques non fonctionnelles (distribution, persistance, concurrence...). L'implantation est déduite en combinant des deux parties grâce à un ¡¡tisseur d'aspects¿¿. Cette approche permet de réutiliser les aspects non-fonctionnels, car ils sont par construction indépendants de l'application. Notons que la programmation par aspects n'est pas liée à la programmation par objets, mais elle nous intéresse

particulièrement ici en raison de l'architecture d'UML qui permet d'en tirer parti.

L'approche des ¡¡filtres de composition¿¿ [4] permet de modifier le comportement d'un objet en filtrant ses communications (messages reçus ou envoyés). Cela est réalisé en cachant l'objet initial derrière des interfaces qui interceptent et modifient les messages.

L'approche de la ¡¡programmation par sujet¿¿ [51] appelée encore ¡¡programmation par rôle¿¿ vise à réutiliser les objets en définissant leurs propriétés selon un certain point de vue : les objets qui collaborent jouent un certain rôle dans cette collaboration. Le comportement global d'un objet résulte de la composition de ses rôles dans les collaborations auxquelles il participe.

Dans la section 2.2.1, nous verrons que la notation UML possède des caractéristiques appropriées pour appliquer ces techniques. Nous verrons également dans la section 2.5.2 comment outiller celles-ci dans le cadre UML.

2.2 La notation UML

2.2.1 Une notation à ¡¡aspects¿¿

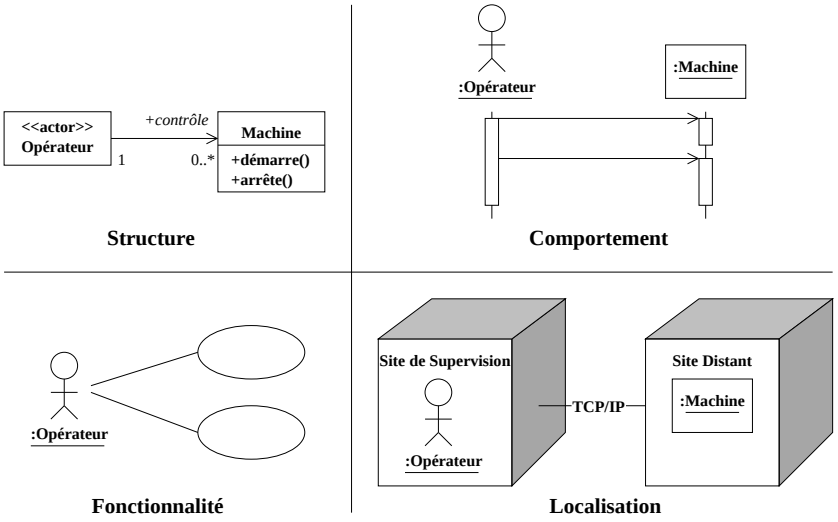


FIG. 2.1 – Notation UML et aspects

La recherche d'un consensus entre les nombreuses notations de modélisation à objets a conduit Booch, Rumbaugh et Jacobson – auteurs respectifs des méthodes Booch [19], OMT [112] et OOSE [57] – à proposer la notation UML. Elle reprend les principaux points forts de ces trois notations. Contrairement à ses prédécesseurs, UML impose un langage mais laisse libre le choix du processus de développement associé.

Les neufs vues proposées par la notation UML autorisent *a priori* une approche du développement similaire à la programmation par aspects [69, 76] et favorisent la ¡¡séparation des préoccupations¿¿. Chacune des vues focalise l'attention sur un as-

pect particulier du problème, évitant une dispersion sur des problèmes concomitants. Les cas d'utilisation représentent des aspects fonctionnels, les diagrammes des classes s'intéressent à la structure du modèle, les diagrammes des états et les collaborations décrivent des comportements, les diagrammes de déploiement la localisation des ressources... la Fig. 2.1 résume les quatre grandes dimensions de la notation UML.

2.2.2 L'exemple de la banque

Le lecteur trouvera une description complète de la notation UML dans les ouvrages [88, 20, 113]. À titre illustratif, nous présentons brièvement dans cette section un exemple simple de modélisation UML que nous utiliserons dans la suite de ce document. Il s'agit de modéliser une banque, dont les deux cas d'utilisation sont le retrait ou le dépôt d'argent sur un compte bancaire par un client. Ces cas d'utilisation sont représentés sur le diagramme de la Fig. 2.2. Un *Client*, représenté ici sous la forme d'un acteur UML, c'est-à-dire une entité extérieure au système modélisé (la banque), peut initier ces cas d'utilisation.

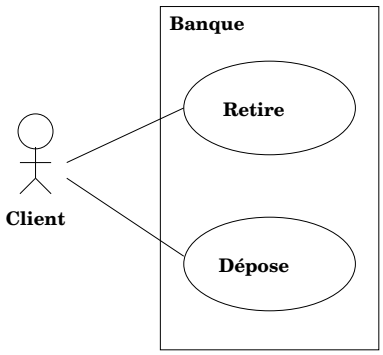


FIG. 2.2 – Cas d'utilisation de la banque

Une architecture à objets pour implanter ce système est représentée sur le diagramme des classes de la Fig. 2.3. Chaque classe est représentée par un rectangle comportant trois compartiments qui contiennent respectivement de haut en bas : le nom de la classe, les attributs (précédés de leur visibilité, + pour public ou - pour privé) et leur type, les opérations (précédées également de leur visibilité) et leur signature. Les associations entre classes sont représentés par des traits au bout desquels figurent les multiplicités des associations, * désignant une multiplicité quelconque. Par exemple, les multiplicités *1* et * de l'association entre les classes *Banque* et *Compte Bancaire* nous apprennent qu'une banque peut gérer un nombre quelconque de comptes et que chaque compte appartient à une et une seule banque. Enfin, ce diagramme indique que la classe *Compte Rémunéré* hérite de la classe *Compte Bancaire*.

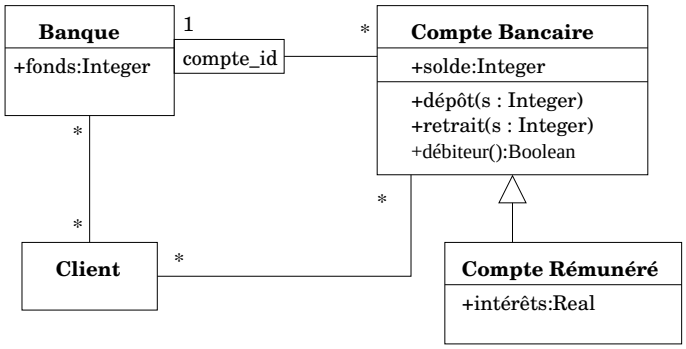


FIG. 2.3 – Diagramme des classes de la banque

2.2.3 Le méta-modèle UML

La caractéristique essentielle qui différencie la notation UML des autres notations réside dans sa base formelle appelée *méta-modèle* : la syntaxe de la notation est décrite très précisément en utilisant la notation elle-même. C'est en quelque sorte l'équivalent de la notation BNF des langages de programmation ; ce méta-modèle permet de définir l'arbre de syntaxe abstrait des modèles. Ce concept de méta-modèle nous semble fondamental. Non seulement il aide à réduire les ambiguïtés et les incohérences de la notation, mais il constitue aussi un précieux atout pour les concepteurs, dans le cadre d'une automatisation du processus de développement logiciel.

Dans les paragraphes suivants, nous présentons cette architecture de méta-modélisation dans le contexte plus général du *Meta-Object Facility* (MOF), qui vise à construire une véritable ingénierie de la modélisation.

2.3 Vers une ingénierie des modèles

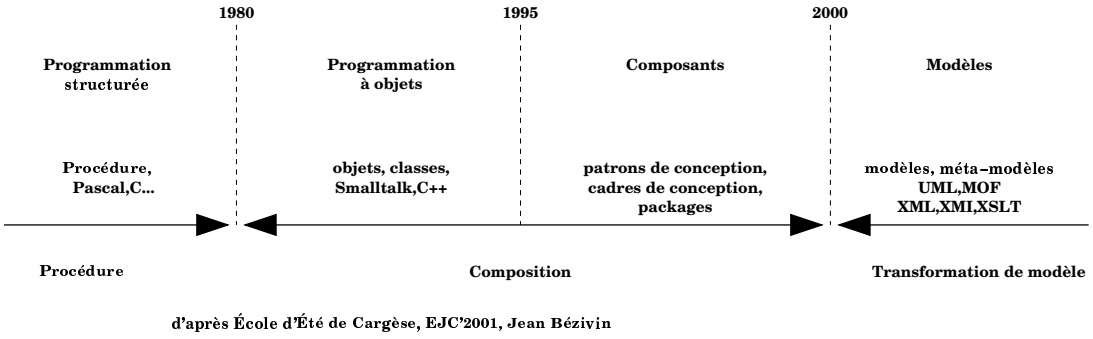


FIG. 2.4 – Vers une ingénierie des modèles

La Fig 2.4 résume selon Bézivin [14] une évolution future possible pour les tech-

nologies des objets. Il s'agit de l'ingénierie des modèles, qui d'après [15] initierait un rapprochement des ingénieries du logiciel objet et de représentation des connaissances. Les modèles prennent effectivement une place de plus en plus importante dans la conception d'applications, avec l'apparition des modèles d'analyse, des modèles de conception, des modèles métier, des modèles de processus, des modèles d'échange de données... La standardisation au sein de l'*Object Management Group* (OMG) de nombreux langages de description des modèles (UML [110], XML [31], Common Warehouse Metamodel (CWM) [95], Corba Object Model [87], Corba IDL [87], etc.) témoigne de cet engouement pour les modèles. Tous ces modèles sont définis par une notation particulière et une sémantique propre et sont adaptés à la modélisation d'un aspect particulier.

Néanmoins, ces modèles sont rarement indépendants. À la lecture de [94], nous constatons qu'ils partagent même un nombre important de notions semblables ou similaires. Par exemple, les concepts de type de données, d'association ou de spécialisation se retrouvent dans la plupart des modèles de l'OMG, mais avec des caractéristiques propres à chaque modèle. Par exemple, dans un modèle de classes UML, la notion de spécialisation se rapproche de la notion d'héritage des langages de programmation à objets.

Afin d'adopter un formalisme commun pour la description de ces différents modèles, de permettre la coopération et l'échange de données entre modèles, l'OMG est en train de bâtir une architecture appelée MDA, pour *Model Driven Architecture* [96], littéralement ¡¡Architecture Dirigée par les Modèles¿¿. La base fondatrice de cette architecture est le *Meta-Object Facility* (MOF), décrit dans [94]. L'une des promesses de MDA est de découpler les modèles d'un système de son implantation pour tel ou tel logiciel médiateur[1].

2.4 Architecture du MOF et de UML

Le but d'un programme est d'automatiser le traitement de données. Ces données appartiennent au niveau M_0. Elles sont modélisées au niveau M_1 par un modèle ; un programme est une implantation particulière de ce modèle. La syntaxe de ce modèle est elle-même décrite par un langage défini au niveau M_2, analogue à la description BNF d'un langage de programmation.

Par exemple, dans le cas de l'application de gestion bancaire de la Fig. 2.3, les objets du niveau M_0 sont les comptes courants – avec leur numéro de compte et leur solde, par exemple le ¡¡compte numéro 1001, solde 123¿¿ – et les clients – identifiés par leur nom, une chaîne de caractères.

Le modèle M_1 de ces objets permet de représenter tous les objets qui sont des comptes, par exemple en énumérant leurs propriétés et les relations entre ces objets. Dans ce cas, un compte est défini par une *classe* possédant les *attributs* `(numéro, solde)`, où `numéro` et `solde` sont des entiers. Ce modèle M_1 définit la *relation* ¡¡appartient¿¿ entre un utilisateur et les comptes, etc.

[1] ¡¡Logiciel médiateur¿¿ est le terme officiel pour *middleware*.

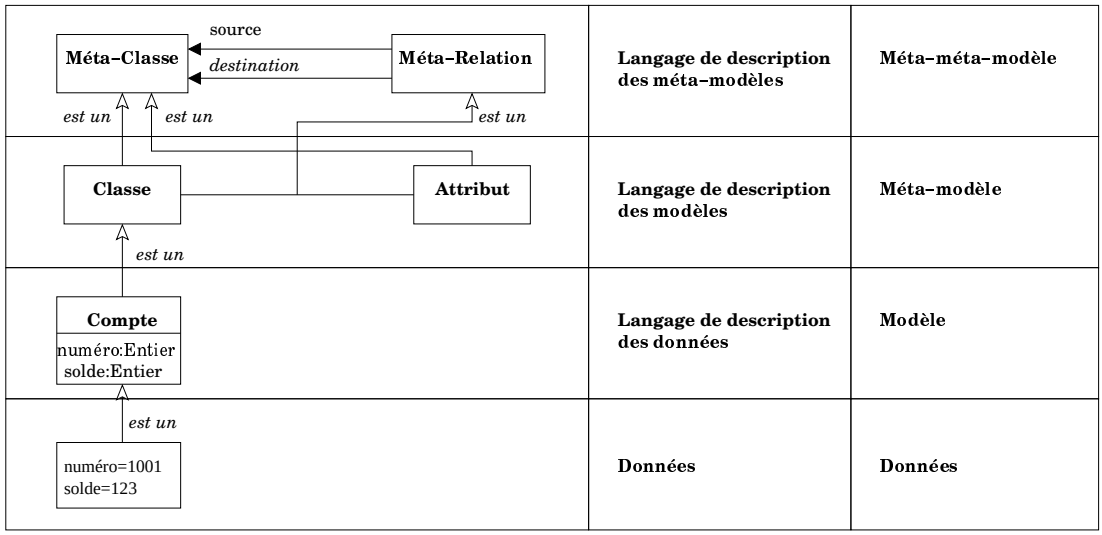


FIG. 2.5 – Architecture ¡¡méta¿¿ à quatre niveaux

D'une manière similaire, les concepts du niveau M_1 peuvent être définis au niveau M_2, appelé méta-modèle. Par exemple, les notions de *classe*, *attribut* et *relation* sont définies à ce niveau en fonction de concepts définis au niveau supérieur M_3, appelé méta-méta-modèle.

En généralisant, les concepts d'un niveau M_n sont définis en fonction de 1 ou plusieurs concepts du niveau M_{n+1}. En nous arrêtant au niveau M_3, nous obtenons l'architecture à quatre couches représentée sur la Fig. 2.5. Sur le schéma de la Fig. 2.5, nous avons représenté par des flèches la relation d'instantiation ¡¡est un¿¿ associée à toute architecture en couches [85, 6]. Cette relation associe à tout objet du niveau M_n sa définition au niveau M_{n+1}.

La question du nombre de couches et des relations entre ces couches se pose immédiatement ; il est *a priori* possible de construire une architecture avec un nombre quelconque de niveaux. La Fig. 2.6 représente un niveau M_4 possible pour définir le niveau ¡¡méta-méta¿¿ M_3 de l'architecture de la Fig 2.5. Sur la Fig. 2.6, la notation *source :Méta-Relation* indique que l'entité *source* du niveau M_3 est modélisé (¡¡est un¿¿) par une méta-relation.

Nous retrouvons dans ce schéma les éléments déjà définis au niveau M_3 ; cela se remarque par les relations ¡¡est un¿¿ qui se font exclusivement entre éléments du même niveau. Le méta-méta-modèle est réflexif et se décrit lui-même ; ceci est schématisé par la Fig 2.7. Il est donc inutile de poursuivre au delà de ce niveau, et quatre couches suffisent.

Le MOF exploite une architecture similaire à quatre couches. Si l'on se limite à la notation UML, il est même possible de ¡¡spécialiser¿¿ cette architecture en se contentant des trois niveaux M_0, M_1 et M_2. En effet, le méta-modèle UML M_2 est lui-même un modèle UML M_1 : il peut être décrit en utilisant un sous-ensemble restreint de la

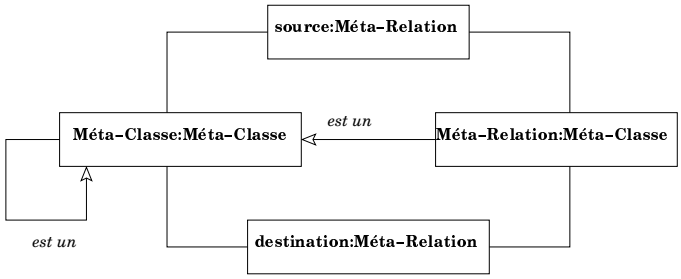


FIG. 2.6 – Le niveau méta-méta

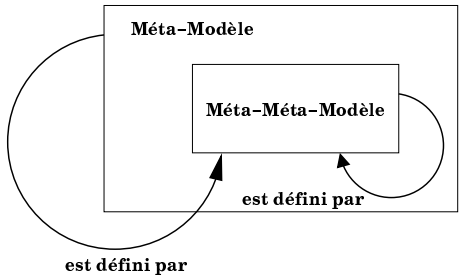


FIG. 2.7 – Le niveau méta-méta est inclus dans le niveau méta

notation UML (les diagrammes des classes). Cette inclusion du niveau M_2 dans M_1, ou du niveau M_3 dans le niveau M_2 – ce qui est équivalent mais à l'avantage de s'aligner sur l'architecture du MOF – apparaît clairement sur la Fig. 2.8, en grisé.

Conserver une architecture à quatre niveaux présente l'avantage de pouvoir manipuler plusieurs méta-modèles définis selon une base commune (en l'occurrence le MOF, dans le cas d'UML), voir [15]. Cela est utile pour faire coopérer des modèles basés sur des méta-modèles différents ; un méta-méta-modèle commun permet de définir des équivalence de concepts entre méta-modèles.

Notons d'ailleurs à propos de cet alignement sur le MOF, que MOF et méta-méta-modèle UML sont deux méta-modèles différents, mais avec des similarités importantes. Le *Core Package* du méta-modèle UML est quasiment identique au MOF. Cette ressemblance s'explique par le fait que des méta-modèles UML réflexifs ont été proposés avant la standardisation du MOF, et cela a eu une influence sur la conception du MOF, d'où cette ¡¡compatibilité¿¿ entre des sous-ensembles du méta-modèle UML et du MOF.

Pour des raisons pratiques, la plupart des ateliers de génie logiciel sont spécialisés dans la modélisation UML et leurs concepteurs ont fait le choix de confondrent les niveaux M_3 et M_2 et d'implanter une architecture à trois niveaux. Ces outils se restreignent même aux deux niveaux M_2 et M_1 lorsqu'ils ne permettent pas la simulation des modèles. En effet, les objets M_0 n'existent alors qu'en dehors de l'outil, lorque le code généré par l'outil à partir du modèle est exécuté.

L'intérêt majeur de cette réflexivité est que des outils initialement conçus pour manipuler des modèles pourraient être adaptés à moindre coût pour manipuler d'une manière identique des méta-modèles, ce que nous étudierons en détail par la suite.

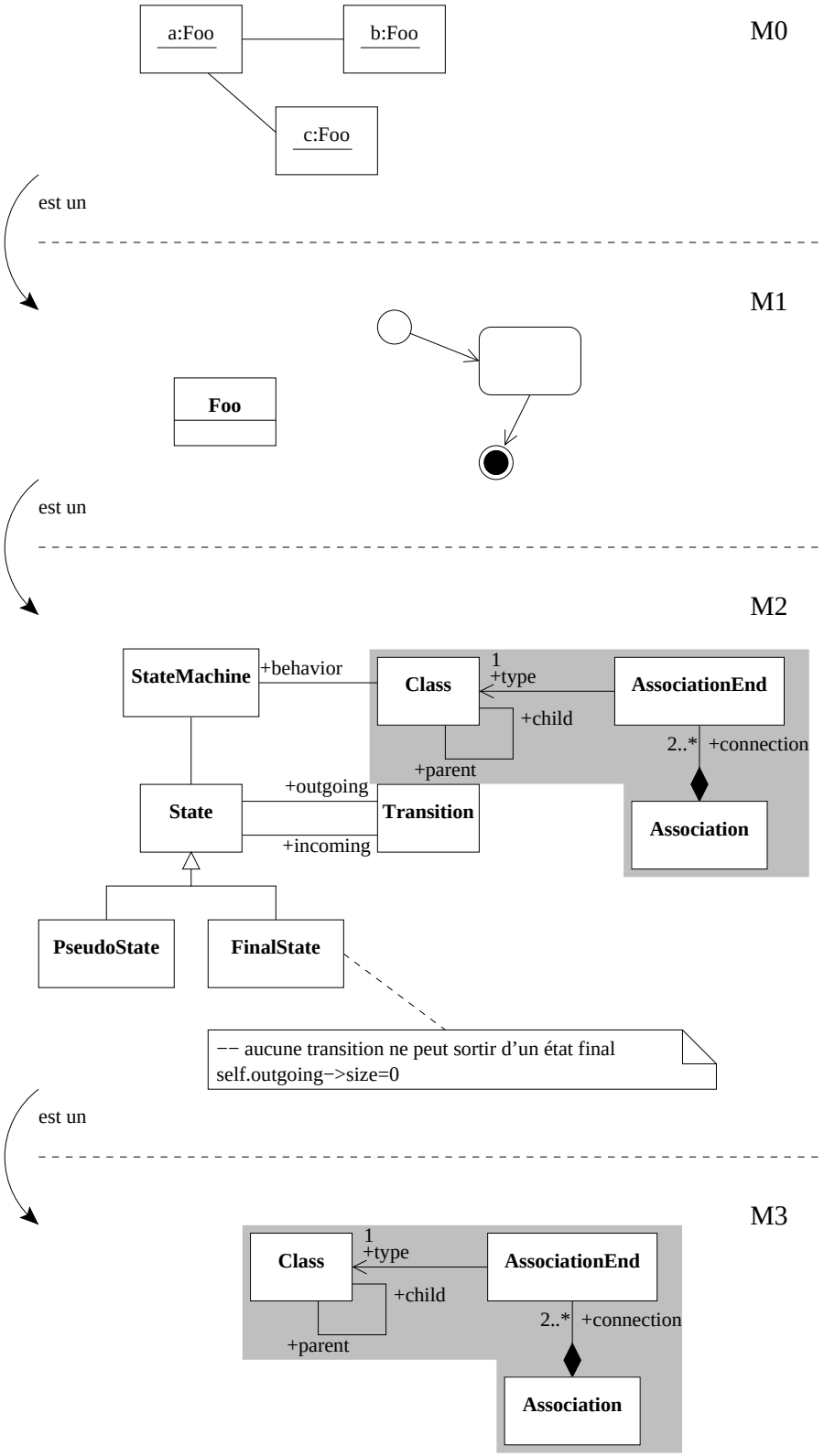


FIG. 2.8 – Architecture de UML

Cette approche réflexive de description des modèles UML est intéressante, mais malheureusement inexploitable dans l'état actuel du MOF et de UML. En effet, un certain nombre de définitions manquent à ces architectures. D'après [15] les concepts suivants doivent être clairement définis : entité, méta-entité, modèle[2], méta-modèle, ainsi qu'un ensemble de relations entre ces concepts, pour déterminer à quel modèle appartient une entité, quel est le méta-modèle d'un modèle, ou la méta-entité d'une entité, etc. Or ces concepts et ces relations ont été en partie ¡¡oubliés¿¿ par les concepteurs d'UML. Il existe bien une relation ¡¡est un¿¿ dans le méta-modèle entre la méta-classe *Object* et la méta-classe *Class*, deux concepts du niveau M_2, mais celle-ci n'a pas la même signification que le ¡¡est un¿¿ entre une entité d'un modèle et sa méta-entité dans un méta-modèle, comme l'explique clairement [15].

Néanmoins, il est facile de compléter l'architecture d'UML pour disposer de ces mécanismes dans un atelier de génie logiciel. En particulier, l'outil UMLAUT que nous présentons à la fin de ce document réifie le schéma de la Fig. 2.9.

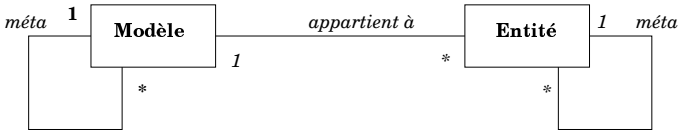


FIG. 2.9 – Un cœur réflexif pour UML

2.4.1 Comparaison avec les autres architectures réflexives

Des approches similaires existent dans les langages et systèmes réflexifs [25]. Par exemple, Loops [18] possède une architecture méta-classe/classe/objet où la méta-classe *Metaclass* est la racine de la hiérarchie des objets, jouant un rôle similaire à celui du MOF. La méta-classe *Class* définit la structure et le comportement d'une classe Loops, jouant ainsi un rôle semblable à celui de la classe *Class* du méta-modèle UML qui définit la syntaxe d'un modèle UML.

La question de savoir si l'architecture à quatre niveaux d'UML peut être ramenée à trois niveaux, suivant l'approche de Briot et Cointe pour Loops avec l'architecture ObjVLisp [23, 24, 30] se pose évidemment. Le méta-modèle UML est lui-même un modèle UML (un ensemble de diagramme des classes), tout comme la méta-classe *Class* d'ObjVLisp se décrit elle-même et est une instance d'elle-même.

Enfin, l'approche multi-vues d'UML fait que plusieurs éléments d'un modèle participent à la définition d'un objet en particulier. Par exemple, une *Class* va définir la structure de l'objet, une *StateMachine* va définir son comportement. Cela se retrouve également dans les architectures à niveaux ¡¡méta¿¿ des langages objets concurrents, par exemple CodA [81]. L'organisation des éléments du méta-modèle UML en neuf vues rappelle le canevas[3] AL-1/D [91], qui a été adopté par plusieurs logiciels médiateurs à

[2]Cette notion de modèle n'est pas à confondre avec l'élément du méta-modèle UML *Model* qui est un conteneur d'éléments d'un modèle UML, et non la définition de ce qu'est un modèle.

[3]¡¡Canevas¿¿ est la traduction de l'anglais *framework*.

objets et réflexifs, par exemple OpenORB [17], ou encore des systèmes d'exploitation comme μChoices [122].

2.5 Intérêts des architectures ¡¡méta¿¿

2.5.1 Formaliser et manipuler le langage

Le premier intérêt d'un méta-modèle pour UML est la formalisation de la syntaxe des modèles. Ceci ajouté au langage de contraintes *Object Constraint Language* (OCL) que nous présentons dans le Chapitre 4 permet de définir très précisément la structure des modèles.

Un autre avantage du méta-modèle est qu'il devient possible de manipuler les modèles. Par l'intermédiaire de la relation ¡¡est un¿¿, le modèle est considéré comme une collection d'objets qui réifient les concepts définis dans le méta-modèle. Ces techniques de manipulation de l'arbre abstrait des modèles sont regroupées sous le terme de méta-programmation. Nous expliquons cette approche dans la section 2.5.2.

2.5.2 Un support pour la méta-programmation

Ces manipulations du méta-modèle sont intéressantes si nous considérons les limitations intrinsèques des approches à objets que nous avons évoquées dans la section 2.1.2 et les approches envisagées pour contourner ces limitations, principalement les techniques de la ¡¡programmation par aspects¿¿. En effet, ces techniques nécessitent un outillage puissant pour effectuer l'opération de ¡¡tissage des aspects¿¿ qui réconcilie les aspects fonctionnels et non fonctionnels. Devant ¡¡mélanger¿¿ des modèles différents pour produire un modèle unique intégrant toutes leurs caractéristiques, cet outil doit pouvoir comprendre et manipuler ces modèles. Cette manipulation de modèle est désigné par le terme générique de ¡¡méta-programmation¿¿. Nous expliquons cette technique dans les paragraphes qui suivent.

Manipuler des programmes comme des données

La complexité croissante des logiciels fait que la méta-programmation devient une technique incontournable. En effet, l'amélioration du processus de développement logiciel passe par l'automatisation des tâches (raffinages, application de patrons de conception, génération de code, etc.). Les outils nécessaires pour réaliser cette automatisation sont des méta-programmes, c'est-à-dire des programmes qui manipulent des données qui sont des programmes ou des modèles. Par exemple, un générateur de code C++ pour UML est un méta-programme qui prend en entrée un modèle UML et produit en sortie du code C++.

Cette idée d'automatiser le développement d'un programme par un autre programme n'est pas nouvelle. En effet, tout compilateur de programme effectue une opération similaire en manipulant en interne une représentation du programme (un

arbre syntaxique abstrait). Cette représentation intermédiaire (le langage *Register Transfert Language* dans le cas du compilateur GCC, par exemple) est le plus souvent manipulable au travers d'une API. Le cas classique d'un compilateur C écrit en C consitue également un méta-programme, qui plus est ¡¡réflexif¿¿ si nous considérons que le langage utilisé au niveau méta pour décrire les transformations est le même que celui utilisé pour décrire les programme. Cela permet de ne pas introduire de nouveau méta-langage, de supprimer la distinction entre langage et méta-langage. Enfin, nous pouvons citer le cas du *template programming*, ou ¡¡programmation générique¿¿ ou la spécialisation du programme est effectuée par l'expansion des paramètres génériques lors de la compilation, comme c'est le cas avec le langage C++ [7, 36, 35].

Citons également le cas des MOP (*Meta-Object Protocols*) qui sont des interfaces de programmation que les programmeurs peuvent utiliser pour modifier le comportement ou l'implantation d'une application et permettent un accès à la définition des classes, des membres ou encore des appels de méthode [27].

Ces techniques peuvent être regroupés sous le nom de ¡¡programmation générative¿¿ [36, 114, 68], car elles visent à la production effective de programmes. Un système parfait permettant la réalisation de ces techniques est décrit par la notion d'*intentional programming* ou ¡¡programmation par intention¿¿ dans [36]. Ce type de système insiste sur la composition des transformations de programmes, donc leur conception générique à des fins de réutilisation [3].

Classification des approches de la méta-programmation

Plusieurs techniques sont possibles pour mettre en œuvre le mécanisme de composition des aspects. Cela peut se faire par transformation de programmes, transformation d'interprètes ou une approche mixte [22]. Cette classification est basée sur la constatation suivante : le comportement d'une application est décrit à la fois par le modèle de cette application et par l'interprète qui permet d'exécuter ce programme (cet interprète est définie par la sémantique d'exécution de la notation). Il est possible de recourir à deux types de transformations : les transformations du modèle et les transformations de l'interprète.

L'approche par transformation de programme (que nous pouvons appeler ¡¡transformation de modèles¿¿ dans le cas qui nous intéresse) consiste à transformer le modèle UML sans toucher au ¡¡moteur d'exécution¿¿ de ce modèle. L'approche par transformation d'interprète consiste à ne pas toucher le modèle mais à modifier (généralement à raffiner) la sémantique d'exécution du modèle, donc dans notre cas la sémantique d'UML. Les approches hybrides consistent à modifier à la fois le modèle et sa sémantique d'exécution pour obtenir le comportement souhaité. Nous verrons dans le chapitre 6 des exemples de transformation de modèles et d'interprètes et leur intérêt dans le processus de développement logiciel.

Chapitre 3

l'*Action Semantics*

3.1 Pourquoi un langage d'actions ?

Le concept d'*Action* apparaît à de nombreuses reprises dans la notation UML : action à exécuter en entrant dans un état ou en tirant une transition (diagramme des états), action à exécuter lors de l'envoi d'un message (diagramme de séquence), etc. La version 1.3 de la notation UML [110] présente très succintement quelques actions élémentaires telles la création ou la destruction d'un objet, mais fait l'impasse sur leur sémantique et ne propose pas de mécanisme efficace pour composer ces actions.

Cette spécification par actions est complémentaire des autre vues UML. En effet, le niveau d'abstraction des diagrammes des états ou d'activité s'avére parfois inapproprié. C'est le cas en particulier pour la description d'un algorithme qui nécessite la réalisation de structures de contrôle complexes ou une granuarité très fine pour exprimer par exemple des opérations de bas niveau telles que la modification de la valeur d'un attribut ou la création d'un objet.

Jusqu'à présent, l'absence de définition précise pour ces concepts obligaient les concepteurs d'applications à modéliser par des expressions *non interprétées* (cf. Fig. 3.1) – de simples chaînes de caractères sans syntaxe ni sémantique – les gardes des transitions d'une machine à états ou la spécification des méthodes. L'absence de définition précise pour ces concepts ajoutée au fait qu'il est quasiment impossible de modéliser sans les utiliser empêche la simulation des spécifications.

Pour contourner ces limitations, il est fréquent que les concepteurs UML ¡¡complètent¿¿ leurs spécifications par des morceaux de code écrits en C++, Java, etc. en lieu et place de ces actions. Certes, une implantation dans le même langage que celui utilisé pour les expressions peut alors être générée plus ou moins automatiquement en recopiant dans le code les fragments fournis par le concepteur et cette implantation peut être exécutée. Mais cette approche comporte de nombreuses limitations, la plus évidente étant bien sûr de repousser la simulation du modèle après la phase d'implantation. De plus, cela nuit énormément à la réutilisation du modèle et à son indépendance vis-à-vis de la plate-forme d'exécution, étant donné que la spécification n'est valide que dans le cadre du langage choisi, et non plus dans le cadre plus abstrait de la notation UML.

Dès lors, il est à craindre que le fait d'insérer du code spécifique à un langage ne force (inconsciemment ?) le concepteur à penser son modèle plus en tant que modèle d'implantation qu'en tant que modèle de conception. Enfin, l´intégration dans un cadre UML de fragments de code provenant d'autres contextes pose le problème de l'unification des concepts similaires des deux mondes, qui ne partagent pas forcément les mêmes sémantiques [48].

C'est pour ces raisons que la *Request for Proposal for an* Action Semantics [93] pour UML demande *une définition d'une sémantique qui autorise la spécification de la structure des actions dans un diagramme des états UML et dans les opérations d'un diagramme des classes UML*. La RFP exige que la spécification des actions offrent les caractéristiques suivantes :

- des modèles simulables dès la spécification initiale. Ceci afin de pouvoir vérifier le plus tôt possible l'adéquation entre la spécification, les modèles dérivés et leur correspondance avec les besoins de l'utilisateur. Il est dès lors possible d'appliquer des techniques de vérification extensive [48] ou de prouver des propriétés de la spécification ;
- un niveau d'abstraction plus élevé qu'un simple langage de programmation. Le but de l'*Action Semantics* n'est en effet pas d'être le nouveau langage à la mode. Par exemple, l'*Action Semantics* doit permettre d'exprimer un maximum de parallélisme ;
- une indépendance vis-à-vis des cibles logicielle et matérielle choisies pour l'implantation. Il doit donc être possible de générer du code pour des plates-formes différentes à partir d'une même spécification AS ;
- enfin, l'*Action Semantics* devra favoriser la réutilisation.

À l'opposé de la définition d'UML, qui insiste autant sur la notation que la sémantique, les concepteurs de l'*Action Semantics* se sont focalisés sur la sémantique. Ils ne cherchent pas à favoriser l'utilisation d'une syntaxe particulière, afin de laisser le libre choix aux concepteurs. La spécification se contente de fournir une syntaxe abstraite et de définir une traduction des principaux langages (C++, Ada ou SDL) vers l'*Action Semantics*. Cette approche facilite la réutilisation des modèles existants qui comportent des morceaux de code écrits dans ces langages.

3.2 Présentation de l'*Action Semantics*

Une seule réponse à cette RFP [5] a été soumise pour examen à l'OMG. Néanmoins, la liste des participants à la rédaction de cette soumission témoigne qu'elle est le résultat d'un consensus entre les grands acteurs industriels d'UML, en particulier tous ceux qui proposaient déjà des extensions propriétaires à UML pour l'expression des actions et la simulation des modèles comme les approches de BrigdePoint [102] ou xUML [64]. Nous avons participé à l'élaboration de cette soumission, au sein du projet RNRT *Convergence* [1] des approches UML et SDL regroupant Alcatel, l'Inria, Telelogic et Q-labs.

Les paragraphes suivants résument brièvement cette proposition qui s'articule au-

tour des trois axes suivants : une extension au méta-modèle UML, un modèle d'exécution et une sémantique des actions. Nous détaillons ensuite un exemple complet de modélisation avec l'*Action Semantics*, afin d'en illustrer les concepts et l'utilisation.

3.2.1 Le méta-modèle

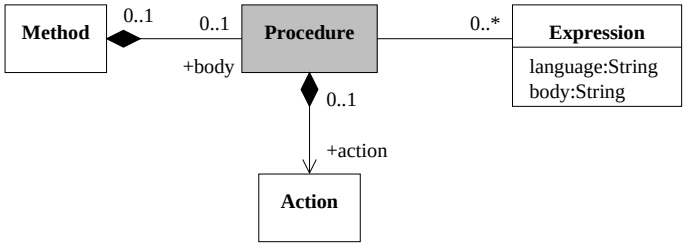


FIG. 3.1 – Intégration des méta-modèles UML et AS

Le but de ce méta-modèle est d'étendre le méta-modèle UML actuel afin de combler ses lacunes concernant la syntaxe de certaines constructions, telles les nombreuses *expressions* non interprétées : gardes des transitions, actions dans un état, etc. Il définit en particulier une syntaxe abstraite permettant de décrire précisément les actions et leur enchaînement sous forme d'ordres partiels (cf. Fig 3.2). La granularité est assez fine puisqu'il est possible de décrire le comportement par des actions aussi élémentaires que la création ou la destruction d'un objet, la lecture ou l'écriture d'un attribut, la création ou la suppression d'un lien entre objets, ou encore l'envoi d'un message... des dépendances de contrôle entre ces actions ou des dépendances de données entre les entrées et les sorties de ces actions élémentaires permettent ensuite d'imposer un ordre partiel sur leur exécution. À côté de ces briques de base, des structures plus complexes sont également disponibles : regroupement d'actions (actions composées, procédures) et structure de contrôle (boucle, choix). Cette syntaxe abstraite est renforcée par des règles de conformité écrites en OCL[1], par exemple pour exprimer l'absence de cycle dans les dépendances de données ou de contrôle entre actions.

3.2.2 Un modèle d'exécution

Si nous reprenons l'analogie entre l'utilisation d'UML et l'utilisation d'un langage de programmation ¡¡classique¿¿, le méta-modèle est analogue à la grammaire (EBNF) utilisée pour la description de la syntaxe du langage ; un modèle UML peut alors être vu comme un programme, dont la configuration initiale est décrite par les diagrammes de déploiement et d'objets de la notation UML. Cette description n'est finalement rien de plus qu'un arbre abstrait, l'intérêt de la notation graphique d'UML résidant dans la séparation des concepts entre les neuf vues qui apporte une plus grande lisibilité, une plus grande souplesse et une maintenance facilitée, un peu à la manière de la programmation par aspects.

[1]Le langage OCL sera présenté en détails dans le chapitre 4.

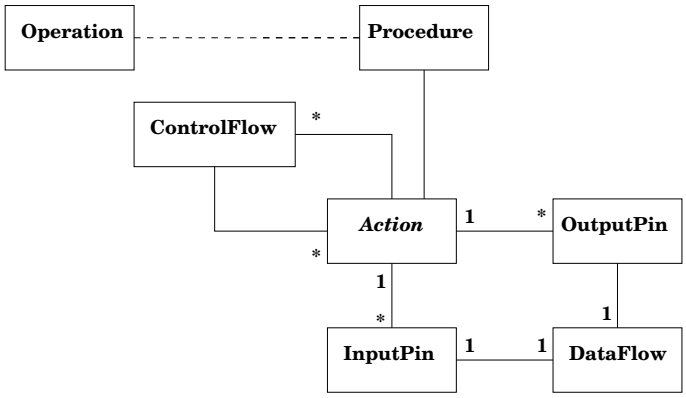


FIG. 3.2 – Vue partielle du méta-modèle de l'AS

Cette grammaire abstraite – augmentée de règles OCL – suffit à définir l'ensemble des modèles UML syntaxiquement corrects. Ces descriptions sont purement ¡¡statiques¿¿, elles décrivent des propriétés des objets du niveau M_0, mais ne renseignent pas sur l'exécution de ces modèles, c'est-à-dire l'évolution des objets du niveau M_0 lorsque des actions du modèles sont exécutées. Il faut associer une sémantique de l'exécution du modèle qui va permettre de décrire précisément l'évolution d'une configuration lorsqu'une action est exécutée.

La norme UML ne s'est malheureusement que très peu intéressée à cette sémantique, et celle-ci y est décrite de manière incomplète et imprécise. Cela donne lieu à des incohérences, et divers travaux ont visé à la formalisation de la sémantique d'exécution des modèles. Notons que beaucoup des formalisations proposées [127, 99, 45, 42] s'intéressent uniquement à la partie statique, et beaucoup moins à la difficile partie dynamique. L'approche la plus fréquemment choisie est la ¡¡compilation¿¿, ie. la traduction des concepts UML dans un autre formalisme. Par exemple, [56] utilisent les ASM, déjà utilisées pour la sémantique de SDL [41, 44], [127] utilise des systèmes de transition, [75] une logique temporelle, [115] les réseaux de Pétri . Si ces approches fournissent effectivement une sémantique à l'exécution des modèles UML, elles possèdent les deux inconvénients majeurs suivants :

- elles ne prennent généralement pas en compte les concepts de la modélisation objet et leur niveau de description est moins abstrait que celui utilisé dans la notation UML. Il en résulte que ces formalisations sont souvent longues, compliquées et le lien avec le concept UML correspondant n'est pas toujours évident. C'est très certainement pour ces raisons que ces approches se ¡¡contentent¿¿ souvent de la formalisation d'un sous-ensemble très restreint d'UML ;
- elles exigent de la part du lecteur un effort supplémentaire. En plus de la notation UML, celui-ci doit maîtriser le formalisme utilisé pour définir la sémantique. Cet effort peut être important et le public auquel s'adresse la norme UML ne possède pas forcément le bagage théorique nécessaire pour une bonne compréhension de ces formalismes. Cela risque de limiter l'acceptation de cette sémantique ;

– enfin, lorsque ces approches permettent une manipulation relativement aisée des concepts UML, il arrive que la mise en pratique soit remise en cause par les limitations des outils permettant de travailler avec ces formalismes. C'est le cas pour [56].

L'un des rôles de l'*Action Semantics* est de décrire précisément cette sémantique d'exécution. Afin d'assurer une large compréhension et diffusion auprès des concepteurs d'application à objets, la formalisation des mécanismes d'exécution d'un modèle UML repose en grande partie sur la notation UML elle-même.

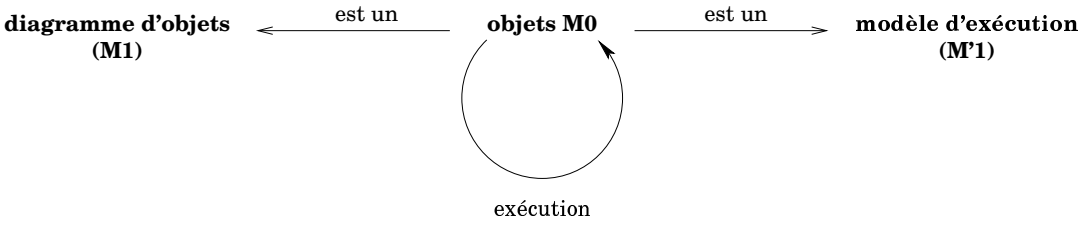


FIG. 3.3 – Positionnement de l'*Action Semantics*

L'approche choisie par les soumissionnaires de l'*Action Semantics* est en effet celle d'un interpréteur de modèles UML appelé *modèle d'exécution*. Ce modèle d'exécution est le modèle UML d'une machine virtuelle capable d'exécuter des spécifications UML, c'est-à-dire de manipuler des objets du niveau M_0 conformément à un *programme* défini au niveau M_1 par des actions. Ce modèle d'exécution peut donc être considéré comme de niveau M_1 et possède des liens unidirectionnels vers le méta-modèle (il a connaissance du ¡¡programme¿¿ qu'il doit exécuter), cf. Fig 3.4 et 3.3.

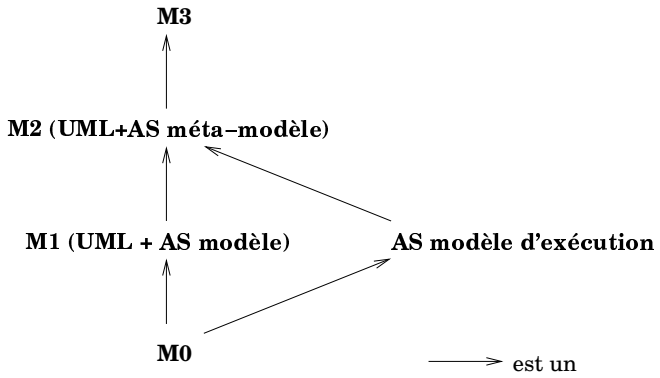


FIG. 3.4 – Positionnement de l'AS dans l'architecture d'UML

Notons que le but de l'*Action Semantics* n'est pas de fournir une implantation facile et efficace d'une machine virtuelle. C'est évident à la lecture de la spécification qui paraît plus destinée aux concepteurs d'applications avec UML qu'aux vendeurs d'outils UML : les descriptions cherchent à faciliter la compréhension, et non la réalisation d'une implantation conforme privilégiant l'empreinte mémoire ou la vitesse d'exécution.

Une implantation particulière sera donc un raffinage de la machine virutelle présentée ici. Nous étudierons dans la section 5.4 comment générer facilement ces implantations conformes et efficaces. Par exemple, la notion de pile des appels des méthodes n'est pas explicitement réifiée dans la spécification : c'est le mécanisme de la continuation qui est utilisé. Cependant, une implantation particulière peut très bien raffiner ce modèle et utiliser une pile pour stocker les appels de méthodes.

Dans le paragraphe suivant, nous présentons les concepts fondamentaux du modèle d'exécution. Comme tout modèle UML, celui-ci est contraint par des régles OCL.

Le modèle d'exécution

Le modèle d'exécution peut être considéré comme le modèle de la mémoire d'une machine virtuelle, c'est-à-dire une représentation des objets du niveau M_0. Notons que les diagrammes d'objets sont ici insuffisants, puisqu'au cours d'une exécution, le niveau M_0 ne contient pas uniquement les objets accessibles au concepteur (c'est-à-dire des instances des classes du modèle de son application), mais également les objets nécessaires à la définition du contexte d'exécution, c'est-à-dire par exemple les messages en transit entre les objets ou en attente de traitement dans les files d'attentes des objets actifs, etc. Ce domaine sémantique définit les notions importantes suivantes (les termes anglais de la documentation de l'*Action Semantics* sont indiqués entre parenthèses) :

- identité (*identity*) : à chaque objet du niveau M_0 est associé une identité, unique et constante pendant toute la durée de vie de cet objet ;
- configuration (*snapshot*) : une configuration définit l'état d'un objet, c'est-à-dire les valeurs des attributs et les identités des objets accessibles à partir de cet objet (via les liens qui réifient les associations de cet objet), entre deux changements de l'état de cet objet. Une configuration est immuable, c'est-à-dire que toute modification de l'état d'un objet crée une nouvelle configuration dans l'historique de cet objet ;
- historique (*history*) : la succession ordonnée dans le temps des configurations (ie. *snapshots*) d'un objet constitue l'historique de cet objet ;
- modification (*change*) : relie deux configurations successives dans un historique. La notion de configuration étant relative à un objet, des liens entre ces modifications permettent d'exprimer des synchronisations entre objets. Les synchronisations peuvent également se faire grâce à l'attribut *+time :Integer* de la classe *Change* ;
- exécution (*execution*) : l'exécution d'une action est réifiée par ce concept. L'exécution de certaines actions complexes se déroule en plusieurs étapes, et demande la création de plusieurs nouvelles configurations, une à chaque étape.

Ce modèle d'exécution est partiellement représenté sur la Fig. 3.5.

Nous avons signalé que la machine virtuelle d'exécution contenait d'autres objets, en plus de la configuration des objets M_0 réification des classes M_1 du modèle réalisé par le concepteur. En particulier, le modèle d'exécution du mécanisme assurant l'envoi de messages entre objets est représenté sur la Fig. 3.6. La classe abstraite *Packet* permet de représenter les données échangées entre objets ; l'attribut *status* détermine l'état du

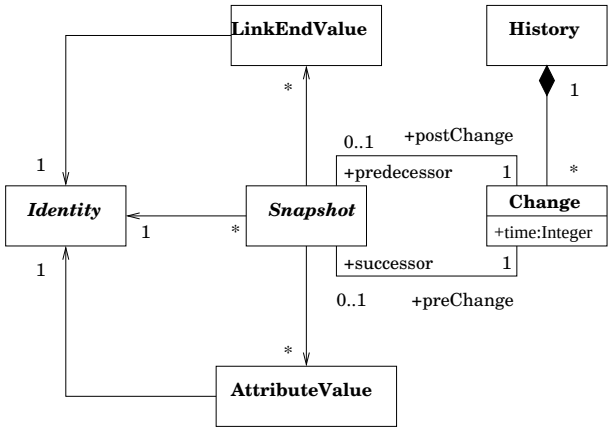


FIG. 3.5 – Le cœur du modèle d'exécution : historiques et configurations

message : `#ready` lorsque le message est prêt à être envoyé, `#transit` après son émission et avant sa réception dans la file du destinataire, `#arrived` lorque ce paquet est dans la file de l'objet destinataire, `#executing` lorsqu'il est retiré de la file et traité par l'objet récepteur, `#complete` lorsque ce traitement est terminé. Ce modèle est trop abstrait pour être implanté tel quel. Une réalisation particulière serait un raffinage implantant par exemple les caractéristiques d'un réseau de communication réel, c'est-à-dire avec des délais de transmission et d'éventuelles pertes de messages. Nous verrons dans la section 5.4.2 comment dériver un tel raffinage.

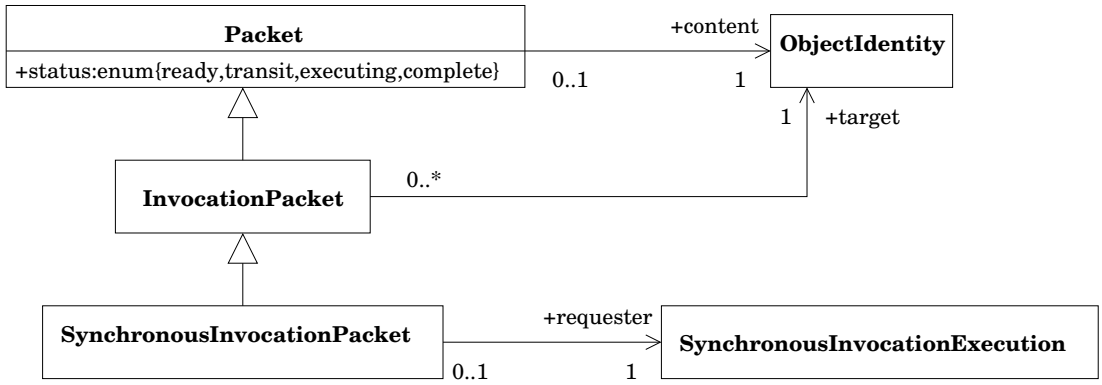


FIG. 3.6 – Modélisation de l'envoi de messages dans le modèle d'exécution

3.2.3 Une sémantique des actions

Le modèle d'exécution permet de représenter la mémoire d'une hypothétique machine virtuelle. Il reste à définir comment cette mémoire est modifiée par l'exécution d'une action.

Nous avons expliqué que le comportement d'un modèle UML est caractérisé par l'ensemble des historiques de ses objets, chaque historique étant constitué d'une succession de configurations. Ces configurations sont des entités immuables, c'est-à-dire que chaque exécution d'une action à effet de bord entraîne la création d'une nouvelle configuration qui est ajoutée à l'historique de l'objet. Les actions complexes sont décomposées en un ensemble d'étapes élémentaires qui donnent chacune naissance à une nouvelle configuration. L'ensemble des étapes élémentaires qui composent l'exécution d'une action est appelé dans l'AS ¡¡cycle de vie¿¿. Le cycle de vie ¡¡générique¿¿ d'une exécution est composée de quatre étapes : `#waiting`, `#ready`, `#execute` et `#complete`. Une exécution progresse de l'état `#waiting` à l'état `#ready` lorsque toutes les données dont elle a besoin sont disponibles (les arguments d'un appel de méthode, par exemple). L'état `#execute` réalise l'exécution proprement dite en créant une nouvelle configuration qui doit vérifier la postcondition de cet état. L'exécution progresse de `#execute` à `#complete` lorsque toutes les valeurs de ¡¡retour¿¿ ont été calculées et sont disponibles.

La nouvelle configuration dépend des configurations précédentes dans l'historique de l'objet et de l'action en cours d'exécution. L'*Action Semantics* ne dit pas comment calculer cette nouvelle configuration, mais indique précisément les propriétés qu'elle doit vérifier. En effet, chaque étape élémentaire de l'exécution d'une action est spécifiée par ses préconditions et postconditions. L'exécution ne peut progresser dans une étape que lorsque la précondition de celle-ci est vérifiée (sur la configuration courante). Lorsque l'exécution d'une étape élémentaire est terminée, la configuration qui vient d'être créée (si l'exécution avait un effet de bord) doit vérifier la postcondition.

Enfin, la spécification de l'AS documente les postconditions des étapes d'une exécution par des *assertions*. Mais cette description n'est pas formelle. À titre illustratif, la Fig. 3.7 recopie la description de l'étape `#execute` de l'exécution d'une action *ReadAttributeAction*, qui renvoie la valeur d'un attribut d'un objet.

Notons que l'*Action Semantics* permet la vraie concurrence [103], car plusieurs pas d'exécution parmi ceux dont les préconditions sont vérifiées à un instant donné peuvent être exécutés simultanément.

Les configurations initiales des objets M_0 peuvent se déduire (cf. Fig. 3.4) des diagrammes d'objets et de déploiement M_1 de l'application modélisée. Cela donne un point de départ à l'exécution d'un modèle.

Une manière simple de réaliser un interpréteur de modèle UML est donc d'utiliser un moteur d'inférence selon un mécanisme de chaînage avant : une base de faits initiaux est déduite des diagrammes d'objets et de déploiement du système. Lorsqu'une précondition (une des règles du moteur d'inférence) est vraie, un nouveau fait correspondant à la postcondition est ajouté à la base des faits, permettant ainsi éventuellement de poursuivre l'exécution.

```
Lifecycle
   the lifecycle is the same as the generic lifecycle
Execution Semantics
   context self:ReadAttributeActionExecutionSnapshot
Production 1 :
   the values put on the result output pin are those of the attribute in the
   predecessor snapshot, in order if the attribute is ordered.
Precondition:
   self.status=#ready
Postcondition:
   self.status=#completed
      and
   let attrVals : Collection(Identity) = self.attributeValues() in
      self.pinValue→select(p:Pin | p = self.executionID.action.result).value=
         if self.executionID.action.attribute.ordering = #unordered
            then attrVals.asSet()
            else attrVals.asSequence()
         endif
```

FIG. 3.7 – Spécification de l'étape `#execute` de l'exécution d'une *ReadAttributeAction*

3.3 Exemple de la FIFO

Nous présentons dans cette section un exemple de modélisation UML utilisant l'*Action Semantics* et le langage de contraintes OCL[2]. Cet exemple est celui de la file `FIFO` décrite sur le diagramme des classes de la Fig. 3.8.

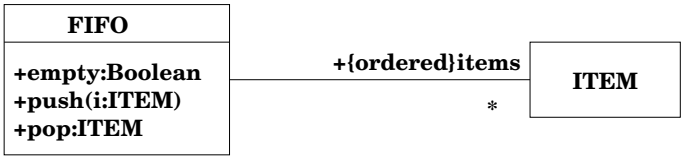


FIG. 3.8 – Exemple de la file FIFO

Les spécifications en OCL des opérations de la classe FIFO sont les suivantes :

```
context FIFO::empty:Boolean
   post: result = items→isEmpty

context FIFO::push(i:ITEM)
   post: items = items@pre→including(item)

context FIFO::pop:ITEM
   pre: not empty
   post: items=items@pre→excluding(items@pre→last)
```

[2]Celui-ci est présenté dans le chapitre 4. Néanmoins les concepts utilisés dans cet exemple sont suffisamment simples pour ne pas nécessiter la lecture préalable de ce chapitre.

Le comportement de l'opération `push` est décrit en terme de diagramme d'objets sur la Fig. 3.9. Ce comportement est très simple, puisqu'un `push` consiste en la création d'un lien entre l'objet `:FIFO` et l'objet `:ITEM` à ajouter à la fin de la file. Autrement dit, il suffit d'une action *CreateLinkAction* pour effectuer cette opération. D'après [5] une *CreateLinkAction* demande deux paramètres, qui sont des *LinkEndCreationData* : ils référencent les objets à relier et spécifient les indices désignant la position d'ajout dans l'association, en partant de chacun des bouts de l'association. Le premier paramètre – l'objet FIFO – est obtenu par la *ReadSelfAction*. Le deuxième paramètre – l'objet à ajouter dans la FIFO est passé par l'argument de la *Procedure* attachée à la méthode `push`.

Dans un pseudo-langage d'actions, cela pourrait se traduire pour l'opération `push` par :

---

context FIFO::push(i:ITEM)
action : self .items.insertAt(SequenceIndex.end, item)

Au regard du diagramme des objets de la Fig. 3.10 réifiant les concepts du méta-modèle de l'*Action Semantics* et de UML entrant en jeu dans la définition de la construction de ce lien, l'intérêt d'un langage de surface pour l'*Action Semantics* apparaît immédiatement.

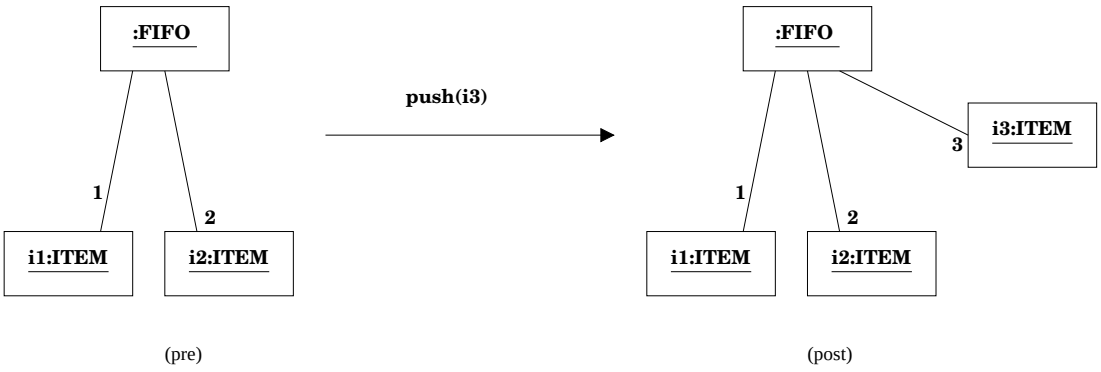


FIG. 3.9 – Action d'un `push`

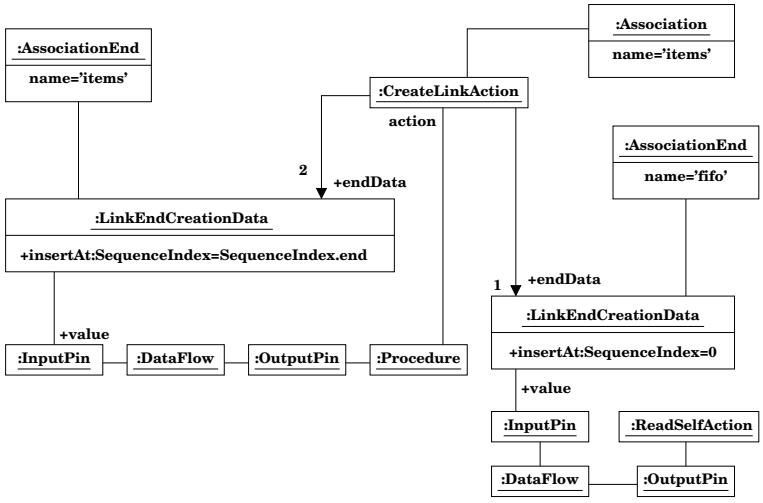


FIG. 3.10 – Diagramme d'objets spécifiant la méthode `push`

Chapitre 4

Présentation et formalisation d'OCL

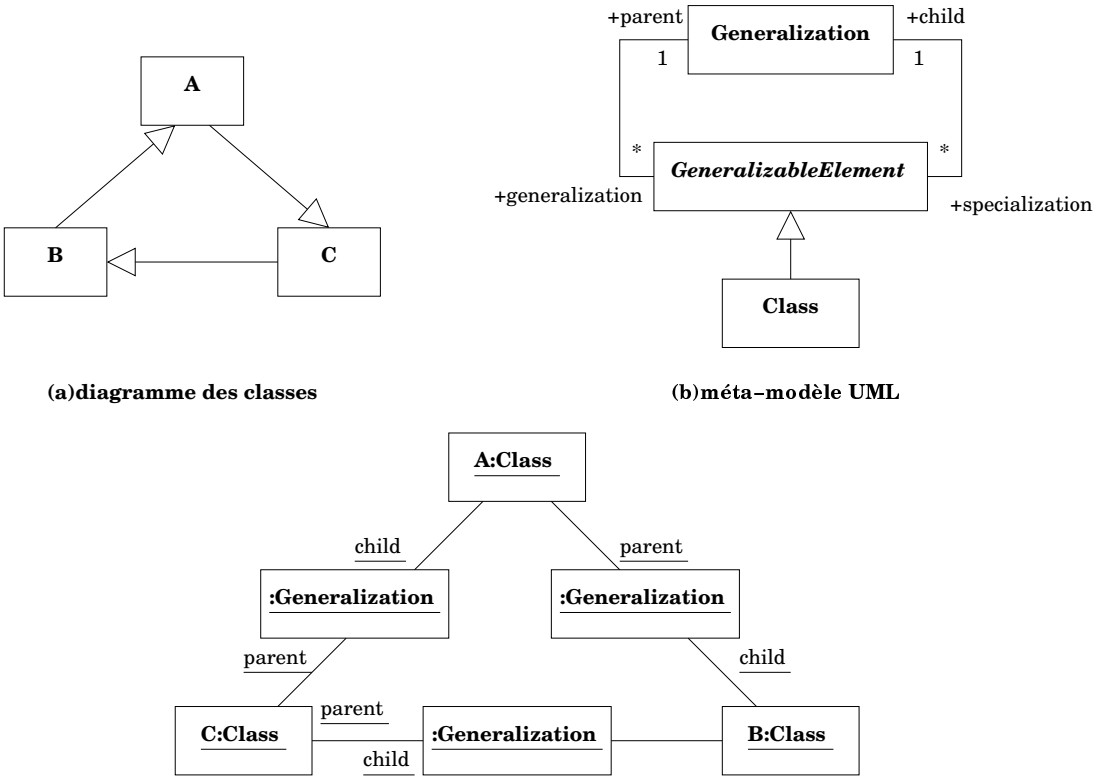
4.1 Ce modèle est-il correct ?

La satisfaction des contraintes structurelles imposées par le méta-modèle (typage, cardinalité des associations, etc.) ne suffit pas à assurer la correction d'un modèle UML. Il est en effet possible tout en respectant ces quelques restrictions de construire des modèles manifestement aberrants. C'est le cas du modèle présenté Fig 4.1a, erroné en raison du cycle formé par les relations d'héritages entre les classes. Si nous considèrons ce diagramme des classes au niveau ¡¡méta¿¿, c'est une réification des classes du méta-modèle UML. Il peut être représenté à ce niveau par le diagramme des objets de la Fig. 4.1c. Il est immédiat de constater que ce diagramme des objets vérifies les contraintes imposées par l'extrait du méta-modèle UML présenté Fig 4.1b ; ce modèle est donc *a priori* valide.

Pour rejeter ces modèles aberrants, la description du méta-modèle UML dans [110] a été complétée par des règles textuelles interdisant explicitement leur construction. Ces règles énoncent les propriétés que doivent vérifier les éléments d'un modèle (classes, associations, états, etc.) et sont parfois complexes. Aussi était-il nécessaire de définir un formalisme adapté à la rédaction et la compréhension de ces contraintes, tout en fournissant une interprétation sans équivoque.

4.2 Présentation d'OCL

Le langage OCL (*Object Constraint Language*) [110, 126] a été créé spécialement pour répondre à ce besoin de formaliser des propriétés sur la structure des modèles par la définition d'invariants sur les classes du méta-modèle [71]. Un modèle légal sera donc un modèle qui considéré comme un ensemble d'objets réifiant des éléments du méta-modèle UML vérifie ces invariants. Le méta-modèle UML étant un modèle UML, l'usage du langage OCL s'est rapidement étendu à la définition d'invariants au niveau des modèles également, la définition de préconditions et postconditions des opérations,



(a)diagramme des classes

(b)méta–modèle UML

(c)représentation de (a) dans le méta–modèle (b)

FIG. 4.1 – Un diag. de classe manifestement invalide

la spécification des gardes sur les transitions des diagrammes des états, etc. Nous verrons dans la section 4.2.3 que cet élargissement du domaine d'OCL pose quelques difficultés. Il s'agit de faire profiter la communauté UML des avantages de la conception par contrats [82, 84].

Nous allons présenter succintement les caractéristiques du langage OCL. Pour illustrer nos propos, nous nous basons sur le modèle UML simplifié d'une banque représenté sur la Fig. 4.2.

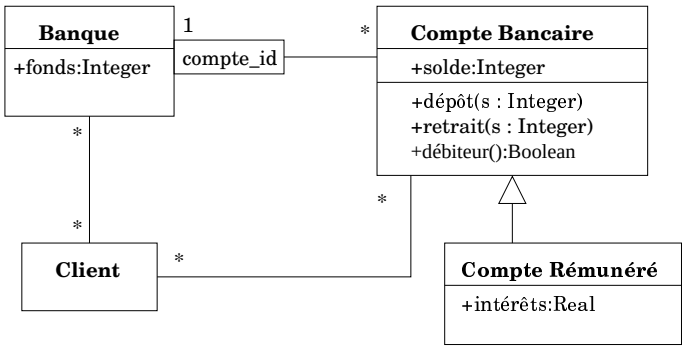


FIG. 4.2 – Exemple de la banque

Les concepteurs du langage OCL se sont attachés à fournir un langage simple à utiliser et aux constructions ¡¡intuitives¿¿. OCL dispose de constructions qui permettent d'exprimer facilement une ¡¡navigation¿¿ dans les modèles UML [50]. D'ailleurs la *Request for Proposal for OCL 2.0* [97] encourage l'usage d'OCL en tant que langage de requêtes sur les modèles.

4.2.1 Le paquet *UML_OCL*

Le langage OCL est fortement typé et sans effet de bord. Le langage consiste en un ensemble de types et d'opérations prédéfinis, regroupés et documentés sous la forme d'un paquet UML appelé *UML_OCL*. Le concepteur d'un modèle UML est censé importer ce paquet dans son modèle pour disposer des fonctionnalités d'OCL. Le paquet *UML_OCL* contient des types élémentaires tels les booléens, entiers, réels et chaîne de caractères, ainsi que des types plus complexes tels les types énumérés ou les collections. Tous les types OCL sont des sous-types du type `OclAny` qui définit les propriétés communes à tous les objets OCL.

À ces types prédéfinis, viennent s'ajouter des types ¡¡importés¿¿ qui correspondent aux classes, interfaces et type de données du modèle. Tous les types OCL, qu'ils soient prédéfinis ou importés sont accessibles via une instance du type OCL `OclType`. Cet `OclType` permet un accès restreint au niveau de modélisation ¡¡méta¿¿ supérieur.

Notion de *contexte*

Une expression OCL est toujours écrite dans un *contexte* qui peut être soit un type, soit une opération. Si le contexte est un type, le mot-clé `self` dans une expression se

rapporte à un objet de ce type. Si le contexte est une opération, `self` désigne le type qui possède cette opération.

---

context Type
-- ceci définit un invariant pour Type
inv: expression−ocl

OCL permet de spécifier le comportement d'une opération par des préconditions et des postconditions. Le mot-clé `result` ne peut apparaître que dans une postcondition, où il désigne le résultat de l'évaluation de l'opération :

---

context Type::opération(param1 : Type, ... , param2 : Type) : Type
-- ceci spécifie une opération OCL par ses pré et postconditions
pre: expression−ocl
post : expression−ocl

Les collections OCL

Le traitement des collections par le langage OCL est un peu particulier. Les collections sont partitionnées en trois sous-types du type abstrait `Collection` :

- `Set`, où chaque élément est présent au plus une fois ;
- `Bag`, où le même élément peut apparaître plusieurs fois ;
- `Sequence`, où les éléments sont ordonnés et peuvent être dupliqués.

Une collection d'éléments de type T est un sous-type d'une collection d'éléments de type T_2 si et seulement si les deux collections sont de la même sorte et T est un sous-type de T_2.

La définition d'OCL interdit la manipulation des collections de collections, les auteurs du langage ayant fait le choix d'¡¡aplatir¿¿ ces structures. Par exemple, nous aurons :

Set{ **Set**{1, 2}, **Set**{3, 4}, **Set**{5, 6}} = **Set** {1, 2, 3, 4, 5, 6 }

Nous verrons dans la section 4.2.2 que cette impossibilité à travailler sur des collections de collections se révèle génante lors de la manipulation d'objets d'un modèle UML. Elle a d'ailleurs été critiquée par les auteurs du langage OCL dans [33]. Aussi proposent-ils dans leur réponse à la *Request For Proposal for OCL 2.0* [97] de retirer cette limitation [125]. Néanmoins, la version normalisé d'OCL imposant cette limitation, nous nous y conformerons dans tout ce document.

Les opérations sur les collections qui sont prédéfinies dans le paquetage *UML_OCL* sont nombreuses (union, intersection, selection, etc.). Nous nous contentons ici de rappeler leurs principales caractéristiques. Syntaxiquement, les opérations portant sur des collections se distinguent par l'utilisation du symbole →. Cette notation est du simple sucre syntaxique dans la plupart des cas, sauf lorsqu'elle est utilisée comme raccourci pour indiquer une conversion implicite en collection, plus précisément en `Set`. L'exemple suivant illustre cette particularité :

-- dans l'expression suivante, size est l'opération String : :size :Integer

-- l'évaluation retourne la taille de 'toto' en nombre de caractères
`'toto'`.**size** = 4

-- dans l'expression suivante, size est l'opération Collection : :size :Integer
-- une conversion implicite asSet est effectuée, et le résultat est le nombre d'éléments de cette collection
`'toto'`→**size** = `'toto'`→**asSet**→**size** = **Set**{`'toto'`}→**size** = 1

L'opération la plus utilisée pour le traitement des collections est aussi la plus complexe. Il s'agit de l'opération `iterate`. Elle utilise la forme syntaxique suivante :

---

collection →**iterate**(elem : Type; acc : Type2 = expression |
expression−avec−elem−et−acc)

Où `elem` va prendre la valeur de chaque élément de la collection, et `acc` joue le rôle d'un ¡¡accumulateur¿¿ des résultats intermédiaires obtenus pendant le calcul. Nous donnons l'interprétation de l'opération `iterate` telle qu'elle est décrite dans [126] :

---

iterate(elem : T; acc : T2 = value)
{
acc = value;
for (**Enumeration** e = collection.elements() ; e.hasMoreElements();)
{
elem = e.nextElement(); acc = <expression−avec−elem−et−acc>
}
}

Si l'ordre de parcours de la collection est connue lorsqu'il s'agit d'une `Sequence` (du premier au dernier élément de la séquence), rien n'est spécifié dans le cas d'un `Bag` ou d'un `Set`. Il est donc possible d'écrire en OCL des expressions dont les résultats seront non déterministes.

L'opération `iterate` est importante car elle permet de construire la plupart des autres opérations OCL. Par exemple, l'opération `select` est spécifiée par :

---

context **Set**(T)::**select**(expr−avec−elem):**Set**(T)
post: result = self→**iterate**(elem; acc : **Set**(T) = **Set**{} |
if expression−avec−elem
then acc→**including**(elem)
else acc
endif)

Où l'expression `acc->including(elem)` renvoie un `Set` contenant tous les éléments de `acc` ainsi que `elem`.

De la même manière, le nombre d'éléments dans une collection est donné par l'opération prédéfinie `size` dont la spécification est la suivante :

---

context **Collection**::**size**:**Integer**
post: result = self→**iterate**(elem; acc : **Integer** = 0 | acc + 1)

L'opération `forAll`, qui réalise l'opérateur ∀ sur une collection :

context Collection(T)::**forAll**(expr−avec−elem): **Boolean**
  post: result = self→**iterate**(elem; acc : **Boolean** = **true** |
    acc **and** expr−avec−elem)

L'opération `exists`, qui réalise l'opérateur ∃ sur une collection :

context Collection(T)::**exists**(expr−avec−elem): **Boolean**
  post: result = self→**iterate**(elem; acc : **Boolean** = **false** |
    acc **or** expr−avec−elem)

Cette définition des opérateurs OCL en fonction de quelques opérations élémentaires nous sera très utile par la suite, puisque nous l'utiliserons à des fins de simplifications des outils de manipulation OCL (opérations complexes implantées directement en OCL, normalisation des expressions à des fins de génération de code, etc.).

Quelques exemples de calculs sur les collections OCL :

Set {1, 2, 3, 4, 5} →**select**(i : **Integer** | i > 3) = **Set**{4, 5}
Bag{1, 2, 3, −2, 4}→**forAll**(i | i > 0) = **false**

4.2.2 Naviguer dans un modèle

Nous avons dit qu'à chaque *Classifier* d'un modèle correspondait un type OCL ¡¡importé¿¿. Il est donc possible d'écrire des invariants pour les classes et types de données d'un modèle UML. Par exemple, la contrainte suivante a pour contexte la classe `CompteBancaire` du modèle UML de la Fig 4.2 et spécifie un invariant que les objets de cette classe devront respecter :

context CompteBancaire
  -- *les découverts ne sont pas autorisés*
  inv: solde ≥ 0

Les deux expressions OCL qui suivent ont pour contexte respectif les opérations `dépôt` et `retrait` de la classe `CompteBancaire`. Par exemple, les opérations `débiteur`, `dépôt` et `retrait` sont spécifiées par les préconditions et postconditions suivantes, où la notation `@pre` dans `solde@pre` désigne la valeur de l'attribut `solde` avant l'exécution de l'opération.

context CompteBancaire::débiteur:**Boolean**
  post : result = solde < 0

context CompteBancaire::dépôt(s : **Integer**)
  pre: s > 0
  post : solde = solde**@pre** + s

context CompteBancaire::retrait(s : **Integer**)
  pre: s > 0 **and** solde ≥ s
  post : solde = solde**@pre** − s

Comme OCL est un langage sans effet de bord, seules les opérations du modèle dont l'attribut (dans le méta-modèle) *isQuery* est vrai sont utilisables dans les expressions OCL. Si l'opération `debiteur` possède l'attribut *isQuery* = *true*, nous pouvons réécrire l'invariant sur la classe `Comptebancaire` :

---

context CompteBancaire
-- les découverts ne sont pas autorisés
inv: **not** débiteur()

En plus des attributs et des opérations *isQuery*, il est possible d'utiliser dans une expression OCL les noms figurant sur les bouts opposés des associations attachées à la classe qui sert de contexte ; si les noms sont omis, c'est le nom de la classe attachée au bout de cette association qui est utilisé. Le type de cette expression dépend des caractéristiques du bout de l'association (cardinalité et ordonnancement) ; nous préciserons ce typage dans la section 4.3.3. Ainsi, dans le contexte `Banque`, l'expression `self.comptebancaire` désigne l'ensemble des comptes bancaires de la banque, et son évaluation renvoie une collection d'objets, plus précisement un `Set(CompteBancaire)`. Inversement, dans le contexte `Comptebancaire`, l'expression `self.banque` est du type `Banque`. La manipulation des associations comme des scalaires ou des collections suivant leur cardinalité s'avère très pratique. Par exemple, pour conserver uniquement les comptes bancaires qui ont un solde supérieur à 1000, nous écrirons dans le contexte `Banque` :

---

self .compteBancaire→**select**(c : CompteBancaire | c.solde > 1000)

Un autre exemple, plus complexe :

---

context Banque
-- les fonds d'une banque sont égaux à la somme des soldes des comptes
inv: fonds = self .compteBancaire→**iterate**(c:CompteBancaire;acc:**Integer**=0|
acc + c.solde)

En résumé, les expressions OCL permettent la navigation dans un modèle en ¡¡traversant¿¿ les propriétés du contexte. Ces propriétés prennent l'une ou l'autre des trois formes suivantes : un attribut, une opération ou une association. Notons que d'un point de vue syntaxique, cela ne fait aucune différence dans l'écriture d'une expression OCL, qui s'écrira `a.b` quelle que soit la nature de la propriété `b` considérée. Ceci est résumé sur la Fig 4.3.

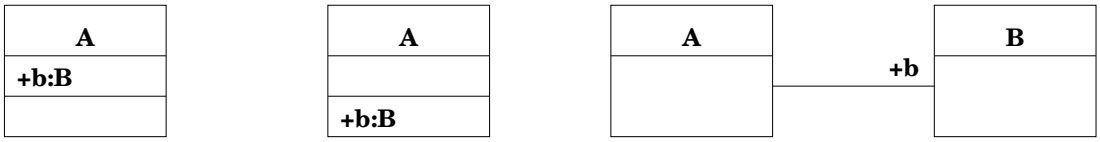


FIG. 4.3 – Navigation de la propriété `a.b`

Il est possible de combiner les traversées successives de plusieurs propriétés et cela permet de construire rapidement de puissantes expressions de navigation dans

le modèle. C'est l'opération `collect` qui accomplit cette fonction. Par exemple, si le contexte est `Banque`, l'expression ci-dessous retourne une collection de tous les comptes bancaires :

```
self . clients →collect(c : Client | c.compteBancaire)
```

Dans ce cas précis, cette expression est équivalente à l'expression :

```
self . clients →iterate(c : Client ; acc : Bag(CompteBancaire) = Bag{} |
                        acc→union(c.compteBancaire))
```

Pour améliorer la lisibilité des expressions comportant des `collect`, OCL autorise le raccourci suivant :

```
self . clients .compteBancaire
```

Le lecteur attentif notera que le type du résultat de cette expression est donné par l'expression `iterate` et est `Bag(CompteBancaire)`. Cette particularité est liée à la manière dont le type d'une navigation au travers d'une association est calculé. Nous avons indiqué que ce type dépendait de la multiplicité de l'association (*0..1*, *0..**) qui indique s'il s'agit d'un type scalaire ou d'une collection et des contraintes d'ordre sur cette association. Ainsi, la traversée d'une association *0..** retourne toujours un `Set`, sauf dans le cas où l'association est {*ordered*}. Dans ce cas, le résultat est une `Sequence`. Les résultats du type `Bag` apparaissent lors de la navigation successive de plusieurs associations *0..**. Intuitivement, la traversée d'une association *0..1* suivie de celle d'une association *0..** conduirait également à un `Set` plutôt qu'à un `Bag`. Ce typage est détaillé dans le tableau 4.4.

Malheureusement, nous verrons un peu plus loin dans ce chapitre que la définition de l'opérateur `collect` est erronée et conduit toujours à des résultats de type `Bag` lors de la traversée successive de deux associations. C'est en contradiction avec les choix effectifs des concepteurs d'UML dans la définition des règles de conformité du méta-modèle. Ils ont en effet utilisé implicitement les propositions du tableau 4.4. Nous nous conformerons à ces choix, plus intuitifs et fournissant un typage plus précis que le `collect`. Nous proposerons de redéfinir l'opération `collect` en accord avec le typage du tableau 4.4.

Pour conclure cette courte introduction au langage, nous revenons au but initial d'OCL qui est la formalisation des contraintes sur la structure des modèles, c'est-à-dire des expressions OCL écrites au niveau du méta-modèle. La contrainte interdisant les cycles dans les relations d'héritage entre classes (cf. Fig 4.1) s'écrit :

```
context GeneralizableElement
   -- les cycles dans les relations d'héritage sont interdits
   inv: not self . allParents→includes(self)
```

Cet invariant utilise une opération `allParents` spéficiée en OCL par la postcondition suivante :

```
context GeneralizableElement::allParents : Set(GeneralizableElement)
   post: result = self . parent→union(self.parent.allParents)
```

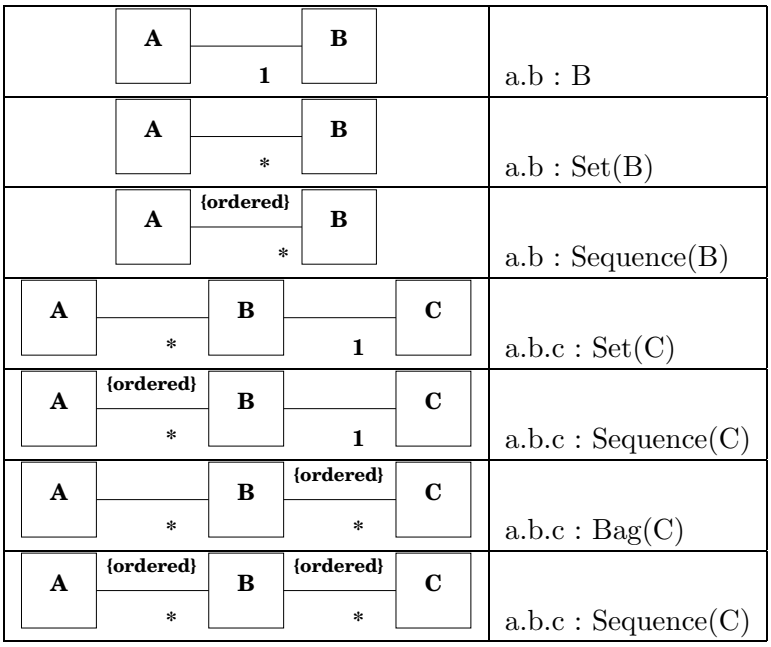


FIG. 4.4 – Navigations multiples, associations et typage

4.2.3 Limitations et défauts d'OCL

À l'usage, le langage OCL révèle ses limitations. Si des raisons pratiques justifient l'introduction d'extensions, avec par exemple la définition d'opérations supplémentaires dans le package *UML_OCL* [33, 116], ou encore le support du multi-contexte, ou la transformation d'OCL en un véritable langage d'actions [70], d'autres limitations s'avèrent plus génantes car liées à l'expressivité du langage [79, 124, 33]. En particulier, le choix des concepteurs de mettre à plat les collections – c'est-à-dire qu'une collection de collections est automatiquement transformée en collection – est génant lors des navigations successives des associations, car cela fait généralement perdre de l'information, en particulier sur le typage. La réponse du groupe pUML [125] à la RFP OCL 2.0 [97] tente de supprimer cette limitation fort critiquée.

Certaines de ces limitations sont une conséquence de l'origine d'OCL, initialement prévu uniquement pour la description de contraintes sur le méta-modèle UML. En effet, ce méta-modèle étant un sous-ensemble de la notation UML, il n'exploite pas tous les concepts de cette dernière. Il n'y a pas de types paramétrés, pas d'associations qualifiées au niveau du méta-modèle. C'est sans doute pour cette raison que la gestion des types paramétrés ¡¡importés¿¿ d'un modèle UML est complètement ignorée par le langage OCL.

À ces limitations s'ajoute l'inconsistance de certains concepts. La sémantique n'est pas toujours clairement définie en raison du manque de formalisme de la spécification[33], comme nous l'avons constaté avec l'opérateur `iterate` présenté section 4.2.1. Ainsi, il est possible en naviguant une association *0..1* d'obtenir des valeurs ¡¡non-définies¿¿, si la multiplicité du lien réifiant cette association est 0. La gestion de ces valeurs non

définies est malheureusement trop vague et peut conduire à des contradictions.

Notons également que l'exécution de certaines opérations OCL conduit à un résultat non déterministe. Nous avons déjà évoqué ce problème dans la section 4.2.1 lors de la présentation de l'opérateur `iterate`. C'est donc le cas également de l'expression suivante (l'opération `asSequence()` étant logiquement construire à partir d'une expression `iterate` non déterministe) :

Set{1, 2, 3}→**asSequence**()→**first**())

Du bon usage d'OCL

Notons que si OCL est destiné à spécifier et documenter des modèles UML, sa spécification ne fournit aucune information quant à son utilisation à bon escient. Il nous semble que dans un soucis de maintenir les bénéfices de la notation UML, les contraintes OCL doivent se plier aux mêmes règles de ¡¡bonne¿¿ conception que les modèles auxquels elles se rapportent. En particulier, nous veillerons à ce que les expressions de navigation OCL respectent les règles de visibilité imposées par UML. Nous nous interdisons donc l'écriture d'expression accédant à des propriétés (attributs, opérations ou bouts d'associations) cachées, par exemple un attribut privé. Pareillement, nous nous interdisons de naviguer une association uni-directionnelle dans les deux sens. Nous veillons également à ne pas utiliser dans une précondition des attributs, associations ou opérations privés ou protégés. Ceci afin d'éviter qu'une classe cliente ne voit des contrats liés à la structure des classes qu'elle utilise. Les règles à respecter pour une bonne conception sont celles décrites dans [84] et implantées dans le langage Eiffel [83].

4.3 Formalisation d'OCL

Dans les paragraphes suivants, nous cherchons à formaliser le langage OCL. Notre cadre de travail étant UML, nous veillons à établir cette formalisation dans ce cadre. Les travaux des articles [37, 107, 80, 26] fournissent également des sémantiques à OCL.

4.3.1 Un méta-modèle pour OCL

La syntaxe du langage OCL est définie dans [110] par une grammaire EBNF. Cette grammaire formalise la syntaxe ¡¡concrète¿¿ des expressions OCL mais est insuffisante pour décrire les expressions OCL valides. En particulier, le système de type n'est pas formalisé ; les types et les propriétés OCL sont simplement désignés par le terme générique `OclName`, une chaîne de caractères. La relation entre ces `OclName` et les classes, attributs, opérations d'un modèle UML n'est jamais explicitée. Nous proposons de formaliser la syntaxe du langage OCL en décrivant sa syntaxe ¡¡abstraite¿¿ sous la forme d'un modèle UML. Par comparaison à l'architecture en couches d'UML, une expression OCL est au niveau M_1 et la syntaxe ¡¡abstraite¿¿ d'OCL est au niveau M_2. Nous décrirons donc un méta-modèle pour OCL. Notons que cela est demandé par la *RFP for OCL 2.0* [97].

De plus un méta-modèle est utile pour proposer des alternatives à la notation textuelle d'OCL, dont la compréhension n'est pas toujours immédiate. C'est l'approche proposée par [21] qui utilise les collaborations d'UML pour visualiser les contraintes OCL, ou encore [65, 66]. Ces approches visent à améliorer la lisibilité de contraintes complexes en les intégrant aux diagrammes UML.

Ce méta-modèle nous sera utile pour la manipulation des expressions OCL, en particulier pour leur compilation. Nous étudierons ce point dans la section 4.3.7.

Approches existantes : des méta-modèles UML et OCL intégrés

Notons que plusieurs méta-modèles ont déjà été proposés en réponse à la *RFP for OCL 2.0*, en particulier dans [108, 8, 125]. Nous pouvons faire les reproches suivants à ces propositions : ces méta-modèles amalgament les classes d'UML avec le typage d'OCL, qui sont pourtant deux systèmes différents. Or nous avons dit que les types paramétrés d'UML ne sont pas gérés par OCL, que le mécanisme de surcharge ou de redéfinition en OCL diffère de celui d'UML (qui n'est d'ailleurs même pas défini !). Une opération UML peut renvoyer un nombre quelconque de résultats (paramètres inout, out, return), alors qu'une expression OCL est sans effet de bord et renvoie un seul résultat. Le langage OCL se contente d'un système de type assez simple, fort éloigné des possibilités complexes offertes en UML.

Enfin, les types prédéfinis OCL sont des types ¡¡mathématiques¿¿ (les entiers relatifs, par exemple) exempts de toute contrainte d'implantation. Ce ne sont pas nécessairement les types de données utilisés par le concepteur et il peut être utile de pouvoir définir précisément les correspondances.

Plus pragmatiquement, l'insertion d'un méta-modèle OCL dans le méta-modèle UML crée un couplage entre ces méta-modèles qui n'est pas justifié : par exemple, les notions d'attributs, d'associations, d'opérations sont spécifiques à UML et n'ont pas besoin d'être distinguées en OCL ou seule la notion de propriété (avec une signature) importe. Les méta-modèles proposés sont donc relativement compliqués. D'un autre côté, il est évident que cette approche favorise l'utilisation des classes, attributs et opérations d'un modèle UML dans le contexte OCL.

L'approche d'UMLAUT : découplage UML/OCL

L'approche que nous avons adoptée pour doter UMLAUT d'un environnement gérant OCL est différente ; elle se distingue par une complète séparation des méta-modèles UML et OCL. Le lien entre les entités de ces deux modèles est formalisé par un ensemble de règles OCL. Nous avons préféré cette solution pour les raisons suivantes :

L'intégration d'un méta-modèle pour OCL au méta-modèle UML nous semble limitative. En effet, elle restreint l'utilisation d'OCL à l'écriture de contraintes sur des modèles UML, et réciproquement en imposant OCL comme ¡¡partie¿¿ du méta-modèle UML – donc de la notation UML elle-même – il est à craindre que seul le langage OCL soit utilisé pour exprimer des propriétés sur les modèles UML. Or, l'expressivité d'OCL est limitée, et un autre langage peut s'avérer plus approprié. Par exemple, des travaux

de recherche [119, 37, 32, 104] visent à étendre OCL afin que des propriétés de logique temporelle puissent être exprimées. L'article [116] propose d'étendre la spécification des préconditions et postconditions dans OCL.

D'autre part, aucun méta-modèle pour OCL n'étant encore défini, il fallait découpler notre implantation de manière qu'elle puisse s'adapter aux futures propositions d'un méta-modèle pour OCL, mais aussi aux futures évolutions du méta-modèle UML. En effet, ces notations sont à l'heure actuelle loin d'être exemptes de contradictions, et il nous faut conserver le maximum de flexibilité pour d'éventuelles modifications de l'une ou de l'autre. L'exemple suivant illustre cette nécessité : lorsqu'une association est naviguée par une expression OCL, le type du résultat dépend des caractéristiques de l'association (cardinalité, ordonnancement, etc.). Cette correspondance n'est pas triviale, cf. Fig. 4.4. Les types de collection OCL `Bag`, `Sequence` et `Bag` ne permettent malheureusement pas de distinguer toutes ces caractéristiques. Par exemple, une association *0..** est traduite en un `Set`, alors qu'une association *0..** et {*ordered*} est traduite en `Sequence`, alors qu'un `OrderedSet` serait plus approprié. Mais ce type est inexistant dans la définition actuelle du langage OCL. Étant donné l'extrême complexité des associations dans UML et leur inconsistance (expliquée dans [9, 52, 10]), nous devons nous attendre à des adaptations d'OCL. La proposition pour OCL 2.0 de [125] est un bon exemple, puisque les collections de collections y apparaissent.

Enfin, notre mécanisme se distingue par sa capacité à être réutilisé pour d'autres méta-modèles que le méta-modèle UML. Pourquoi en effet limiter l'utilisation d'OCL à l'écriture de contraintes pour UML, alors que CDIF, la définition des interfaces CORBA, la notation OPEN ¡¡concurrente¿¿ d'UML ou bien un langage de programmation (Java, Eiffel) pourraient avantageusement bénéficier de l'apport d'OCL ? D'une manière générale, OCL pourrait s'appliquer au MOF.

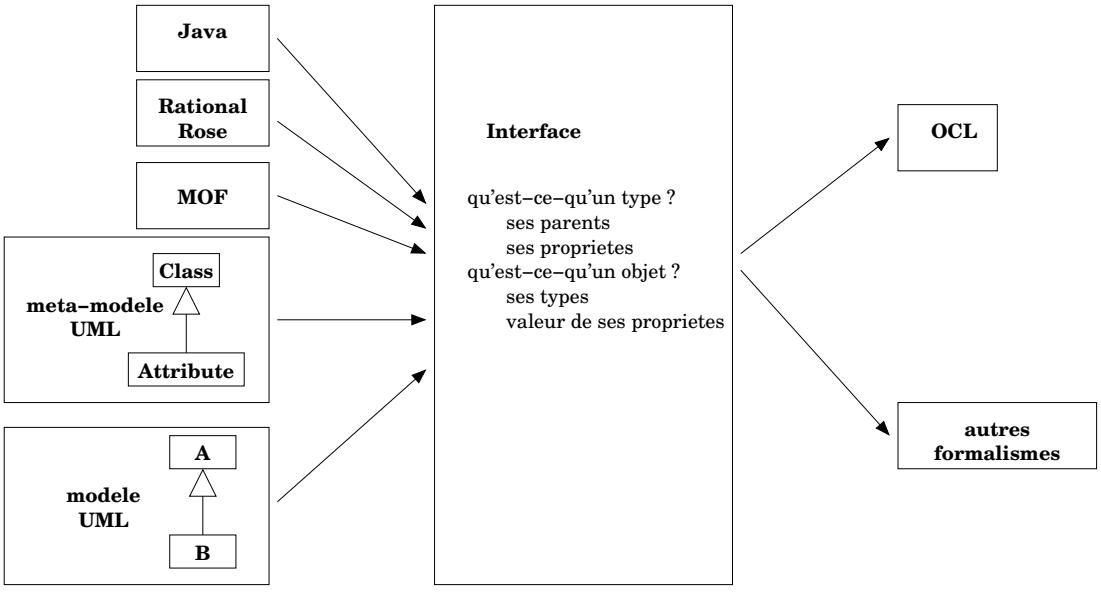


Fig. 4.5 – Interface OCL

L'implantation que nous avons réalisé avec cette approche est facilement réutilisable. Elle permet un ajout facile d'outils OCL dans les outils UML existants sans modification des méta-modèles implantés dans ces outils. Notons que comme ces outils possèdent souvent leur propre ¡¡interprétation¿¿ du méta-modèle UML, comme l'outil Rose de la société Rational, cela semble d'ailleurs être la seule approche viable.

La solution que nous proposons est illustrée sur la Fig. 4.5.

Le méta-modèle d'OCL dans UMLAUT

Ce méta-modèle est présenté sur la Fig. 4.6. Notons que notre méta-modèle ne respecte pas tout à fait les propositions d'OCL 2.0, puisqu'il a été développé alors que seule la version 1.0 existait. Néanmoins, les différences sont minimes. Le lecteur notera en particulier que *OclCollection* n'hérite plus de *OclAny* dans la version 2.0. Nous verrons que cette évolution est prise en compte à un coût très minime par notre implantation. En particulier, la gestion des collections de collections proposée pour UML 2.0 est prise en compte par notre méta-modèle, sans modification.

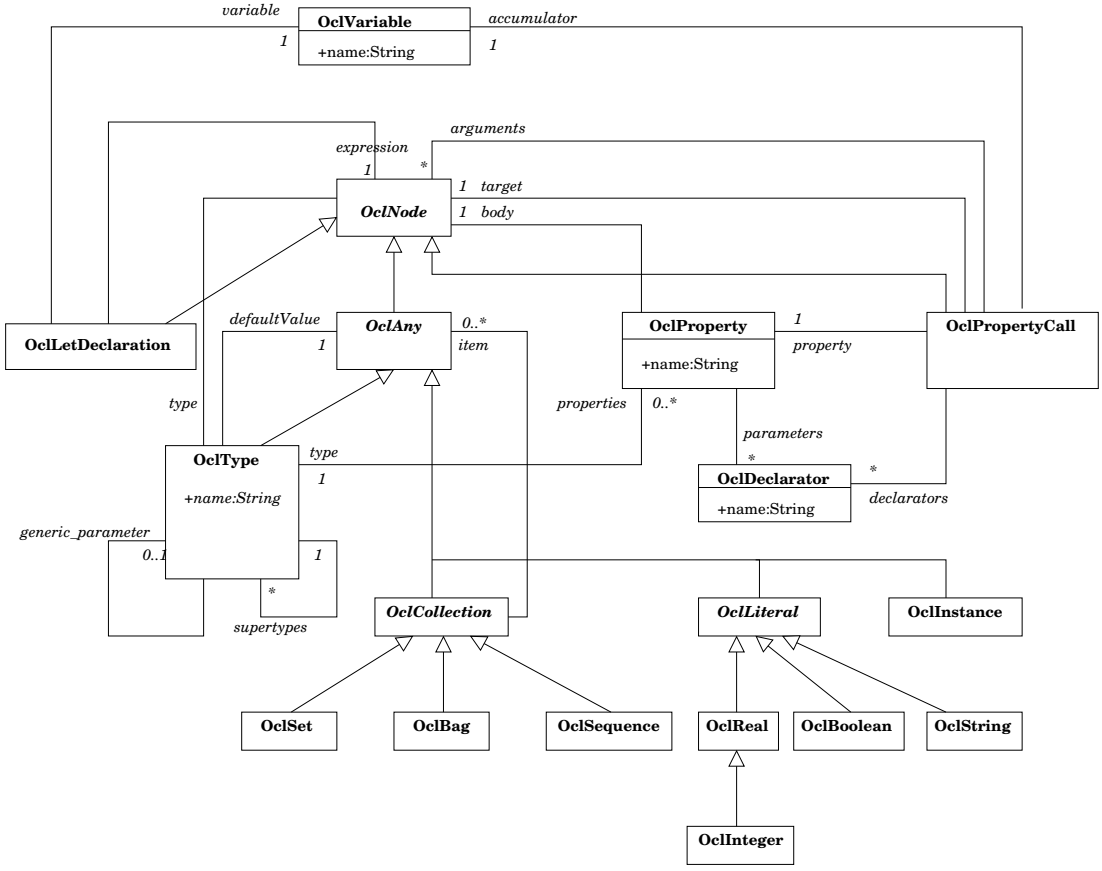


FIG. 4.6 – Un méta-modèle pour OCL

4.3.2 Règles OCL

Comme pour tout modèle UML, la structure de notre méta-modèle UML est précisée par des règles OCL.

Formalisation du système de types d'OCL

Ces règles permettent de formaliser les règles de typage OCL.

```
context OclType::isConforming(other : OclType) : Boolean
  -- cette opération renvoie vrai si self est égal à, ou un sous-type de, other
  post : result =
           self = other
         or
           if self.generic_parameter.is_empty
           then
             -- c'est un type scalaire
             self.allSupertypes→includes(other)
           else
             -- c'est un type de collection
             self.allSupertypes→includes(other)
             and self.generic_parameter.allSupertypes
               →includes(other.generic_parameter)
           endif
```

```
context OclType::allSupertypes : Set(OclType)
  -- cette opération calcule l'ensemble des parents d'un type
  post : result = result.supertypes→union(result.supertypes.allSupertypes)
    and result→includes(OclAny) -- OclAny est un supertype pour tous les types
```

Les opérations de typage d'OCL s'expriment en fonction de ces opérations. Par exemple :

```
context OclAny::oclIsKindOf(type : OclType) : Boolean
  -- est-ce-que le type de 'self' est conforme à type ?
  post : result = self.type.isConforming(other)
```

Formalisation de la grammaire EBNF

La contrainte suivante interdit les collections de collections, bien que le méta-modèle OCL que nous proposons autorise ces constructions :

```
context OclCollection
  -- le type d'une collection est paramétré
  inv: self.type.generic_parameter→size = 1
  -- le paramètre ne peut pas être lui-même paramétré
  -- (pas de Collection de Collections en OCL)
  inv : self.type.generic_parameter.generic_parameter→size = 0
```

Afin de formaliser aisément la syntaxe des expressions OCL, toutes les opérations du paquet standard *UML_OCL* sont ajoutées au méta-modèle OCL comme des sous-classes

de *OclProperty*. Cette approche nous permet de formaliser les opérations du paquet *UML_OCL*, en particulier en donnant un typage précis à toutes les expressions.

```
context SelectPropertyCall:
  -- le select utilise un déclarateur, qui est du même type ou un supertype du type du
  -- paramètre de la collection
  inv: self.target.oclIsKindOf(Collection)
         and self.declarator→size = 1
         and self.target.type.generic_parameter.allSupertypes
           →includes(self.declarator→first)
```

```
context ForAllPropertyCall:
  -- le select utilise n ≥ 1 déclarateurs, qui sont du même type ou des supertypes du type du
  -- paramètre de la collection
  inv: self.target.oclIsKindOf(Collection)
         and self.declarator→size > 1
         and self.declarator→forAll(d : Declarator |
           self.target.type.generic_parameter.allSupertypes→includes(d))
```

Formalisation de l'opération `collect`

Dans ce paragraphe, nous revenons sur la définition de l'opération `collect` qui s'avère peu pratique à utiliser dans sa définition actuelle. La formalisation du système de type d'OCL nous a amené à modifier le typage de certaines opérations OCL donné dans [110], car celui-ci est parfois trop peu précis et ne correspond pas à l'utilisation qui en est habituellement faite, en particulier lors des navigations successives. Nous avons vu dans le tableau de la Fig. 4.4 que la collection obtenue lors des navigations successives dépend des caractéristiques de chaque association (arité et ordonnancement). Or, la définition de l'opérateur `collect` dans [110] est en contradiction manifeste avec les informations fournies par ce tableau :

```
collection →collect(x : T | x.property)
-- is identical to :
collection →iterate(x : T; acc : T2 = Bag{} | acc→including(x.property))
```

D'une part, le résultat est toujours du type `Bag`, ce qui contredit les indications du tableau de la Fig. 4.4. D'autre part, cette définition est erronée puisqu'elle ne s'applique qu'au cas où `x.property` est un scalaire et non une collection, conformément à la signature de l'opérateur `including()`. Nous avons donc décidé de modifier cette définition pour être en accord avec le tableau de la Fig. 4.4. Une étude des règles de conformité du méta-modèle montre d'ailleurs que celles-ci ont été écrites avec ces suppositions implicites. Notre définition est la suivante :

```
context OclPropertyCall
  if self.property.name = 'collect' then
  -- il y a un seul argument, qui est la propriété à collecter
  self.argument→size = 1 and
  -- il y a un et un seul déclarateur,
```

```
-- et les éléments de la collection sont conformes à ce type
self.declarators→size = 1 and
self.target.type.generic_parameter.allSupertypes
        →includes(self.declarators→first.type)
-- il n'y a pas d'accumulateur
and self.accumulator→isEmpty and
-- le type du résultat dépend du type (scalaire/collection) de la propriété collectée
if self.argument.type.generic_parameter→isEmpty then
   -- c'est une propriété scalaire, le résultat est construit par une suite d'opérations
   -- including. Dans ce cas, on conserve le type de collection (Set, Bag, Sequence)
   -- mais la collection est désormais une collection de propriétés
   self.type = target.type and self.type.generic_parameter = self.argument.type
else
   -- la propriété est une collection. Dans ce cas, le résultat est construit par des
   -- opérations d'union, et le typage est donc déterminé par l'opérateur union
   -- c'est-à-dire que nous obtenons toujours un Bag sauf lorsque l'on collecte des
   -- séquences de séquences, ou le résultat est intuitivement une séquence
   target.type.generic_parameter = sef.argument.type and
   if target.type.name = 'Sequence'
      and target.argument.type.name = 'Sequence'
   then
      self.type.name = 'Sequence'
   else
      self.type.name = 'Bag'
   endif
endif
endif
```

À cette définition formelle du typage de l'opération `collect`, s'ajoute la définition suivante pour le calcul du résultat :

```
context Collection::collect(x : T | x.property)
   -- le type T2 de l'accumulateur acc est calculé avec la définition précédente
   -- et acc est initialisé avec une collection vide de ce type T2
post: if x.property.type.generic_parameter.isEmpty then
   -- la propriété est scalaire
   result = iterate(x : T; acc : T2 | acc→including(x.property))
   else -- la propriété est une collection
   result = iterate(x : T; acc : T2 | acc→union(x.property))
```

Ainsi, nous obtenons une définition de l'opération `collect` qui corrige celle de la documentation, et en adéquation avec la vision intuitive du typage de la navigation multiple évoqué dans le tableau de la Fig. 4.4.

4.3.3 Correspondance méta-modèles UML/OCL

Nous avons conçu notre modèle d'OCL indépendamment du méta-modèle OCL. Néanmoins, les classes, attributs, opérations et associations d'un modèle UML pouvant être utilisés dans des expressions OCL, nous devons définir les conditions de cette

utilisation. Pour cela, nous définissons à l'aide de règles OCL une correspondance entre les entités OCL et les entités UML. Dans ce but, nous aurons besoin des opérations OCL suivantes :

```
context OclType:class(): Class
  -- retourne la classe de même nom dans le modèle
  -- la redéfinition de cette opération permet de gérer la ¡¡mise à plat¿¿
  -- des hiérarchies des paquets UML. Utile pour traiter les règles de
  -- conformité du méta-modèle qui ne respectent pas cette hiérarchie. Par exemple,
  -- les règles utilisent Attribut au lieu de FoundationPackage : :Core : :Attribut
  post: result.allEnclosingNamespaces().name→iterate(s : String; acc : String
          | acc.concat(s.concat('::'))).concat(result.name) = self.name
```

```
context Element::allEnclosingNamespaces() : Sequence(Namespace)
  -- place dans une Sequence la hiérarchie des Namespaces dans
  -- laquelle l'élément est contenu
  post : result = if self.namespace <>#undefined
    then self.namespace.allEnclosingNamespaces()→union(self.namespace)
    else Sequence(Namespace){} endif
```

```
context OclProperty:element():Element
  -- retourne l'élément UML (attribut, association ou opération isQuery)
  -- correspondant à cette propriété
  -- la redéfinition de cette opération permet de gérer les conflits
  -- de nom d'opérations non précisés par la notation UML,
  -- en cas de surcharge ou de redéfinition
  post : result.name = self.name
```

Cette classe d'interface entre les méta-modèles UML et OCL est relativement simple. Nous définissons les types d'OCL en fonction des *Classes*, *Attributs*, *Associations* et *Opérations* du modèle UML :

- à chaque classe (ou interface) du modèle UML correspond un objet *OclType*, de même nom. Les super-types d'un type OCL sont les *OclTypes* des classes parentes de la classe UML, plus le type *OclAny*.
- à chaque attribut du modèle UML correspond une instance de la classe *OclProperty*, dont le type est l'*OclType* de l'association *type* de l'attribut ;
- en partant de chaque classe, nous faisons correspondre à chaque association du modèle UML (ou plus précisément aux bouts d'associations du côté opposé à la classe) une instance de la classe *OclProperty*, dont le type *OclType* est déterminé suivant la multiplicité et le caractère ordonné ou non de ce bout d'association (cf. tableau de la Fig. 4.4).
- à chaque opération du modèle UML (marquée *isQuery=true*) correspond une instance de la classe *OclProperty*, dont la signature est déterminée à partir de la signature de l'opération.

Nous avons donc pour la correspondance entre un modèle UML et les types OCL les contraintes suivantes (pour des raisons de clarté, le mécanisme de traduction des noms UML en OCL a ici été omis) :

```
context OclType
   -- pour chaque classe du modèle, il existe un et un seul type OclType de même nom
   inv: self.class()→size = 1 -- conversion implicite en Set(Class)
   -- il n'existe pas deux OclTypes de même nom
   inv: allInstances→forAll(o1, ,o2 : OclType |
                  o1.name = o2.name => o1 = o2)
```

```
context OclType
   -- l'héritage du modèle UML est conservé
   -- tout OclType est un sous type d'OclAny
   inv: self.supertypes.name = self.class().supertypes().name
        →union('OclAny')
```

```
context OclType
   -- les propriétés d'un type OCL sont les propriétés par défaut du paquet UML_OCL, ainsi
que les attributs, bouts opposés des associations, operations isQuery de la classe de même nom
dans le modèle
   inv: self.properties.name =
      self.class().attributs.name
      →union(self.class().allOppositeAssociationEnds().name)
      →union(self.class().operations→select(o : Operation |
                  o.isQuery)→collect(o | o.name))
```

```
context OclProperty
   -- les signatures sont calculées comme suit
   inv: if      -- il s'agit d'un attribut
      self.element().oclIsKindOf(Attribut) then
         let attribute = self.element()→oclAsType(Attribut) in
            (self.parameters→isEmpty()
            and self.type.class() = attribute.type)
      else -- il s'agit d'une opération
         if self.element().oclIsKindOf(Operation) then
         let operation = self.element()→oclAsType(Operation) in
            let    -- calcule la signature de la propriété en fonction des paramètres de l'opération
UML
               in_parameter = operation→parameters→select(p |
                  p.kind = #in or p.kind = #inout) in
               let    -- calcule le type du résultat. C'est le type du paramètre #return de
l'opération
                  -- pour être utilisée dans OCL, une opération doit posséder un et un seul
paramètre marqué #return
                  out_parameter = operation→parameters→select(p |
                     p.kind = #return)→first in
                  (self.parameters→size = in_parameters→size
                  and Sequence{1..in_parameters→size}→forAll(i1, i2 |
                     self.parameter→at(i1).name = in_parameter→at(i2).name
                     and self.type.class() = out_parameter))
      else  -- il s'agit d'un bout d'association
         let association_end = self.element()→oclAsType(AssociationEnd) in
            if association_end.multiplicity.upperBound() ≤1 then
```

```
            -- c'est un scalaire
            self.type.class() = association_end.type
        else -- c'est une collection
            self.type.generic_parameter→size = 1
            and
                if association_end.is_ordered then
                    -- c'est une association ordonnée, donc on obtient une Sequence
                    self.type.name = 'Sequence'
                else
                    -- l'association n'est pas ordonnée, on a un Set
                    self.type.name = 'Set'
                endif
        endif
    endif
```

4.3.4 Caractéristiques de notre approche

Gestion des espaces de noms

Notons que notre approche simplifie la gestion des espaces de noms. En particulier, la gestion des renommages en cas de conflits (si un type du modèle possède le même nom qu'un type OCL), la gestion de la surcharge des opérateurs ou la redéfinition des opérations ne sont pas clairement précisées en UML et OCL. Nous avons toute latitude pour adapter notre interfaçage en fonction des (futures) contraintes imposées au niveau d'UML ou d'OCL. La vérification des contraintes du méta-modèle UML (les *well-formedness rules*) a été simplifiée par cette approche, puisque les contraintes dans [110] et [5] supposent que le méta-modèle UML n'est pas découpé en une hiérarchie de paquets, mais est ¡¡à plat¿¿ : les contraintes dans [110] et [5] n'utilisent jamais la notation `paquet : :sous-paquet : :classe`, ce qui les rend effectivement plus lisibles.

Retour sur l'indépendance UML/OCL

Nous avons basé notre approche sur l'indépendance d'OCL vis-à-vis d'UML. Or, les opérations OCL `isAttribute`, `isAssociation`, `isOperation` font explicitement référence à des constructions spécifiques à UML. Nous recommandons de ne pas utiliser ces opérations ¡¡méta¿¿ (elles le sont très peu en pratique), d'autant plus que si le modèle contraint est effectivement un modèle UML il est facile de les définir comme des opérations sur le méta-modèle UML. Par exemple :

```
context ModelElement::isAttribute(): Boolean
  post: result = self.oclIsKindOf(Attribute)
```

Gestion des erreurs

Le seul inconvénient que nous avons pu rencontrer avec cette approche séparant les méta-modèles UML et OCL est une plus grande complexité (tout en restant raison-

nable) du mécanisme de gestion des erreurs. En effet, une erreur lors de l'évaluation d'une contrainte OCL implique de ¡¡remonter¿¿ cette erreur jusqu'à l'élément incriminé dans le modèle UML correspondant.

De même si nous générons du code à partir d'OCL, le code généré doit être en adéquation avec le code généré pour le reste du modèle UML. Typiquement, si une expression OCL réfère la lecture d'un attribut, l'accès à cet attribut doit être cohérent avec le codage de cet attribut. Nous verrons dans la section 4.3.7 une solution simple et élégante à cette difficulté.

Gestion de l'¡¡applatissement¿¿ des collections

Conformément à la définition du langage OCL, notre évaluateur OCL ne permet pas de manipuler des collections de collections, il les ¡¡applatit¿¿. Cependant, le lecteur notera qu'il est possible de représenter ces structures avec notre méta-modèle, mais que leur utilisation est interdite par des invariants OCL. Cet ¡¡applatissement¿¿ – et donc le respect des invariants – est garanti dans UMLAUT par le mécanisme de construction des collections. Des expressions littérales comme `Set{1,2}` sont d'abord transformées en `Set{}->append(1)->append(2)`. Cela facilite le typage et la construction de collections toujours ¡¡correctes¿¿, construites à partir d'un ensemble vide. Un `Set{1, Set{2,3}}` est d'abord transformé en `Set{1}->union(Set{2,3})`. De cette manière, il est impossible de construire une collection de collections dans UMLAUT. Ajouter la gestion des collections de collections pour être en conformité avec la proposition pour OCL 2.0 aura donc un coût minimal, puisqu'il suffira de supprimer certains invariants et le mécanisme de construction des collections littérales, sans toucher au méta-modèle.

Utilisation des postconditions

Nous avons vu que la plupart des opérations OCL sont dérivées d'opérations plus simples. Par exemple, les opérations sur les collections utilisent l'opérateur `iterate`. L'évaluateur OCL d'Umlaut utilise systématiquement cette propriété en évaluant directement cette expression OCL. Ceci s'applique à toutes les opérations dont la postcondition peut s'écrire sous la forme `result=...` (sans `result` dans le membre droit). Dans le méta-modèle OCL cette possibilité correspond à l'association body entre *OclPropertyCall* et *OclNode*. Toutes les opérations additionnelles OCL qu'un concepteur ajoute à son modèle son également évaluables sans aucun codage si leur postcondition à cette forme `result=...`.

4.3.5 Sémantique d'OCL et modèle d'exécution

Dans cette section, nous établissons une relation entre les expressions OCL et la sémantique d'exécution des modèles UML spécifiée par l'AS. Notre interpéteur OCL utilise cette sémantique.

La vérification des règles de conformité UML est simple, puisqu'il suffit de les évaluer sur le modèle UML considéré, qui est statique. Par contre, les règles OCL définies par le concepteur pour contraindre le modèle de son application doivent être vérifiées sur

les exécutions de ce modèle. Un invariant doit être vrai pour toutes les configurations des objets auxquels il s'applique, c'est-à-dire les objets qui ont pour classe le contexte de cet invariant. De même, les préconditions et les postconditions sont calculées sur les configurations de l'objet concerné.

4.3.6 Interface avec un langage de programmation

La formalisation d'OCL présentée ici (méta-modèle, formalisation des opérations) a servi de base à l'implantation d'un vérificateur syntaxique ainsi que d'un évaluateur pour OCL fonctionnant *à la fois au niveau du modèle et au niveau du méta-modèle.*

L'intérêt du découplage entre le méta-modèle UML et le méta-modèle OCL est apparu immédiatement dans notre implantation. En effet, le méta-modèle UML est implanté comme un ensemble de classes Eiffel dans le cœur d'UMLAUT et comme un ensemble de classes Java dans l'interface graphique.

Pour évaluer des contraintes sur un diagramme d'objets UML (vérification que ce diagramme d'objets est conforme au diagramme des classes), nous manipulons donc véritablement des objets UML, tels qu'ils sont codés dans UMLAUT (nous avons donc accès aux notions de classe, d'attributs, d'associations telles que définies par UML).

Par contre, l'évaluation des contraintes de niveau méta – pour vérifier par exemple que le modèle de l'utilisateur est un modèle correct (pas de cycle d'héritage, etc.) ou des contraintes relatives à notre implantation particulière du méta-modèle – implique un changement de niveau conceptuel : dans ce cas, nous n'avons plus accès aux objets UML, mais à des objets Eiffel (les notions manipulées sont celles d'objets Eiffel, d'attributs Eiffel, il n'y a pas d'associations...) , ou à des objets Java.

Nous aurions aussi pu nous contenter de coder ces contrats directement en Eiffel, ou en Java (il existe des outils permettant d'appliquer la programmation par contrat à Java, par exemple [72]), mais il était tentant d'exploiter la facilité de manipulation des collections en OCL et de pouvoir simplement recopier dans l'implantation les contrats tels qu'ils apparaissent dans la documentation de l'AS [5].

L'interface UML/OCL réalisée (cf. Fig. 4.5) s'est très facilement adaptée à ce schéma, et bien entendu, aucune modification dans le code OCL lui-même n'a été nécessaire. Nous avons donc spécifié à l'aide de règles OCL une correspondance Eiffel/OCL et Java/OCL, un travail similaire à la correspondance UML/OCL présentée dans la section 4.3.3. Pour cela, nous avons conçu un méta-modèle pour les constructions manipulées par chaque langage. Cela nous permet d'utiliser les types de base du langage, mais également les types que nous avons définis pour coder les associations du méta-modèle, par exemple. Nous résumons brièvement, cette traduction d'Eiffel vers OCL ou de Java vers OCL dans le tableau de la Fig. 4.7. Notons que cette correspondance n'est valide que pour notre implantation particulière dans UMLAUT du méta-modèle UML.

Eiffel	Java	OCL
attribut scalaire	attribut scalaire	propriété scalaire
ARRAY	Tableau	Sequence
SIMPLE_LIST		Set
ORDERED_LIST	Vector	Sequence
	Hashtable	Set

FIG. 4.7 – Correspondance des types Eiffel, Java et OCL

4.3.7 Génération de code à partir d'OCL

Dans cette section, nous nous intéressons à la génération de code à partir d'expression OCL. Il s'agit de vérifier que l'exécution d'une implantation ne transgresse pas les invariants, préconditions et postconditions définis sur le modèle. Les mécanismes de la programmation par contrat sont intégrés à certains langages, par exemple Eiffel [83], ou disponibles sous la forme d'extensions (librairies, préprocesseur...) dans d'autres langages, tels iContract pour le langage Java [72] ou les approches présentées dans [78] pour le langage C++. Enfin, les autres utilisations d'OCL (langages de description des gardes des transitions, langage de navigation dans les modèles) ont également besoin de générateurs de code.

Nous avons choisi de ne pas générer directement du code à partir du langage OCL, mais de passer par un langage intermédiaire, en l'occurrence l'*Action Semantics*. Cela simplifie grandement la génération de code à partir d'OCL, car la portabilité est assurée par le générateur de code AS. Enfin, cette démarche est logique si nous nous rappelons que les expressions OCL accèdent aux attributs et associations d'un modèle. Or, le codage de ces structures de données et de leurs accesseurs est réalisée par le générateur de code AS. Il est donc plus simple de réutiliser ce générateur.

Avant la traduction d'OCL vers l'AS, nous avons ajouté une étape intermédiaire de normalisation des expressions OCL. Cette normalisation découle directement du fait que la plupart des opérateurs OCL sont définis à partir de l'opérateur `iterate` (cf. section 4.2.1). Cette traduction permet de réduire fortement la complexité de l'étape de traduction OCL vers l'AS, avec cependant un impact négatif sur la vitesse d'exécution. Il est vrai que nous n'avons pas cherché à optimiser l'arbre normalisé, et le remplacement d'une expression de type `collection->size` qui pourrait être implantée directement grâce à un opérateur natif par l'expression `collection->iterate(e ; acc : Integer = 0 | acc + 1)` n'est sans doute pas optimal. Un drapeau dans le compilateur OCL d'UMLAUT permet d'éviter la normalisation et de générer un code plus performant, pourvu que les opérateurs non normalisés sont disponibles nativement.

Notons que l'utilisation de l'AS en tant que langage intermédiaire est facilitée par le fait que l'AS dispose de primitives de manipulation des collections (cf. chapitre 7 de [5]). En effet, nous disposons dans l'AS des actions *MapAction*, *FilterAction*, *IterateAction* et *ReduceAction* qui implantent les opérateurs d'ordre supérieur `map`, `filter`, `reduce` du formalisme de manipulation de listes BMF (Bird-Meertens Formalism) [16].

La génération de code à partir d'OCL pose certains problèmes sémantiques. En

effet, une expressions OCL utilisée comme garde d'une transition est supposée être évaluée atomiquement et en un temps nul. Ces hypothèses ne sont pas réalistes et ne peuvent être maintenues dans le cas d'une implantation pratique, surtout si une contrainte réfère plusieurs objets, éventuellement distants. Notons que ces hypothèses sont vérifiées lorque les expressions sont évaluées par notre simulateur de modèle UML, puisque dans ce cas, notre simulateur ¡¡bloque¿¿ le temps (qui est discret) et évalue les contraintes sur des configurations figées.

4.3.8 Un mot sur l'implantation

Dans l'outil UMLAUT, les implantations du vérificateur syntaxique et de l'interpréteur OCL ont été réalisées en Eiffel. L'interface au méta-modèle UML, ainsi que l'interface au modèle UML représentent chacune quelques centaines de lignes. Nous avons également partiellement testé notre approche en écrivant des interfaces pour les outils commerciaux de modélisation UML TogetherJ et Rose. Nous n'avons pas rencontré de problèmes particuliers lors de ces tentatives.

Le passage par une conversion d'un modèle UML vers notre méta-modèle OCL pourrait faire craindre une perte inutile de temps pour l'analyse des contrats OCL. Il n'en est rien, puisqu'un mécanisme de cache (une implantation du schéma de conception *Proxy* [43]) implique que la conversion a lieu au plus une fois pour chaque élément du modèle UML traduit dans le méta-modèle OCL. À titre d'exemple, la vérification syntaxique de l'ensemble des contrats du méta-modèle UML (environ 200 expressions OCL) est de l'ordre de la seconde sur un PC. Notre méta-modèle étant adapté à OCL, il permet sans doute un travail plus rapide qu'une manipulation directe du méta-modèle UML.

Notons que grâce à ces outils, nous avons pu corriger les règles de conformité de la norme UML [110]. Plus d'une centaine d'erreurs a été détecté, de la simple erreur de frappe à l'utilisation d'opérations inexistantes en passant par de nombreuses erreurs de typage. Les règles ainsi corrigées sont présentées dans le chapitre 8.

Chapitre 5

Une sémantique pour UML

5.1 De la non cohérence d'UML

Les neuf vues de la notation UML (cf. 2.2.1) peuvent être considérées comme des projections du modèle. Malheureusement, ces projections ne sont généralement pas orthogonales mais dépendent les unes des autres. Ainsi, une collaboration met en relation des rôles dont les propriétés (opérations) sont définies sur un diagramme des classes et les associations définies entre ces classes déterminent quels sont les rôles de la collaboration succeptibles de communiquer. À cette connexion entre vues, s'ajoutent parfois des redondances. Il peut en effet être souhaitable, à des fins de documentation ou pour une meilleure compréhension, de représenter plusieurs fois le même aspect sous des angles différents par des vues équivalentes, complètes ou partielles. Par exemple, les scénarios sont couramment utilisés comme des cas de tests, représentant une interaction particulière entre des objets dont les comportements sont définis par ailleurs complètement par des diagrammes des états ou l'*Action Semantics*. Il est même ensisageable de reconstruire le comportement des objets à partir de ces spécifications partielles, comme il est montré dans [67].

Le développement par aspect de modèles UML pâtit de ces recoupements entre vues. Maintenir entre les différents aspects une séparation franche qui autorise leurs développements indépendants semble illusoire. D'un autre côté, le concepteur d'une application peut souhaiter s'appuyer sur ces dépendances entre aspects pour promouvoir un développement itératif et conjoint des vues. Cette démarche nécessite un outillage adéquat ; il est alors crucial que l'outil assiste en permanence le concepteur afin de garantir la cohérence entre aspects, les interactions étant nombreuses, pas toujours évidentes, parfois même implicites.

C'est l'historique de la notation UML, union des notations OMT [112], Booch [19], Jacobson [57] et diverses autres influences, qui a conduit à cette situation : le méta-modèle actuel ressemble à un *patchwork*, où chaque vue possède son propre méta-modèle plus ou moins bien intégré à l'ensemble. Les incohérences ne sont donc pas de simples problèmes ¡¡de surface¿¿, la conséquence d'une notation parfois imprécise et redondante [118]. Elles sont à notre avis le reflet de problèmes plus profonds et trouvent leur origine

dans la constitution du méta-modèle UML actuel qui ne favorise pas la cohésion des concepts : sa complexité, la redondance de certaines constructions et les lacunes de la sémantique [118] nuisent à la réunification des vues en un tout cohérent. Ces problèmes ne concernent pas uniquement la cohérence entre vues ; des redondances apparaissent au sein d'une même vue [117].

Les nombreuses règles OCL qui tentent de réconcilier les concepts des différentes vues en attestent ; malheureusement elles sont complexes et leur complétude n'est pas assurée. Les vues dynamiques sont les plus nombreuses et les plus complexes : diagramme des états, collaborations et maintenant spécification AS sont interdépendantes ; leur compréhension souffre donc tout particulièrement de cette situation.

Des simplifications du méta-modèle UML ont été envisagées, par la recherche d'un méta-modèle minimal et canonique [29, 28, 106, 77]. Le but est de pouvoir déduire toutes les constructions complexes du méta-modèle par des compositions de ces éléments de base.

Une autre approche ensisageable serait de réduire les problèmes d'incohérence entre la notation et son méta-modèle en limitant le couplage entre modèle et vue : actuellement, presque tous les éléments graphiques sont représentés par un élément spécifique du méta-modèle. Une approche de type MVC [43] permettrait de découpler vue et modèle.

5.2 Simplifier et unifier

Nous proposons dans un premier temps de donner une définition précise et non ambiguë aux concepts présents dans chaque vue, après quoi nous nous intéressons à la cohérence entre les vues. Nous nous attachons ici au cas des vues comportementales, que nous proposons d'unifier. Cela passe par la définition d'une base sémantique formelle commune entre les concepts des différentes vues [47]. Diverses approches ont été étudiées et UML dispose ainsi de sémantiques exprimées avec les formalismes des ASM [56], des réseaux de Petri [115], les notations B [74], Z [42] ou encore PVS [89]. La plupart de ces tentatives ont deux inconvénients majeurs. D'une part, elles sont conceptuellement différentes de la notation UML et exigent un apprentissage parfois lourd que les concepteurs de modèles ne sont sans doute pas prêts à assumer. D'autre part, elles sont souvent inadaptées à la manipulation des concepts UML : elles ne s'appuient pas sur les principes de la conception objet, ou bien sont de trop haut ou de trop bas niveau par rapport aux concepts manipulés en UML. Cela entraine une certaine lourdeur dans leur description d'UML et un lien avec les concepts UML pas toujours évident. C'est sans doute pour cette raison que ces tentatives se restreignent généralement à un sous-ensemble très limité d'UML (une vue, le plus souvent il s'agit des diagrammes de classes) auquel elles fournissent une sémantique précise. Notons que dans le cas des ASM, qui est une notation suffisamment structurée et de ¡¡haut-niveau¿¿ pour manipuler les concepts UML assez facilement, c'est le manque d'outils de qualité qui limite la faisabilité de l'approche [56].

L'AS nous semble une bonne alternative pour toutes ces raisons : il dispose évidemment

d'une très bonne intégration à UML et les primitives de maniement des concepts UML qu'il propose sont par construction appropriées. De plus, le niveau d'abstraction qu'il présente nous semble adéquat. Certes, il peut être reproché à l'AS de ¡¡ressembler¿¿ un peu trop à un langage de programmation, mais cela nous semble un avantage dans la mesure où l'AS se pose entre les autres diagrammes UML de haut niveau d'abstraction (diagramme des états, collaborations...) et les constructions rencontrées dans les langages de programmation à objets usuels. L'AS semble donc être un langage intermédiaire adéquat pour exprimer un raffinage itératif des modèles en implantations. De plus, raffiner toutes les vues dynamiques en spécification AS donne immédiatement à ces vues une sémantique d'exécution fondée sur celle de l'*Action Semantics*.

Dans les paragraphes qui suivent, nous montrons comment l'*Action Semantics* nous sert à formaliser les diagrammes de séquence. Pour cela nous établissons des équivalences entre des éléments du méta-modèle des diagrammes de séquence et des éléments du méta-modèle de l'*Action Semantics*, à l'aide de règles OCL. Après avoir établi cette correspondance, qui autorise la traduction des diagrammes de séquence en spécification AS, nous montrons comment construire des simulateurs de ces spécifications AS.

5.3 Applications aux diagrammes de séquence

Les diagrammes de séquence sont représentés dans le méta-modèle UML par les *Interactions*. D'après la documentation UML1.4r2 [110], *l'objectif d'une interaction est de spécifier les communications entre un ensemble d'objets collaborant à la réalisation d'une tâche particulière. Une interaction fait partie d'une collaboration, c'est-à-dire que la collaboration définit le contexte de l'interaction.* Il existe deux types d'interaction en UML : les interactions de niveau ¡¡spécification¿¿ qui représentent des communications entre des rôles (notion de même niveau que la notion de classe) et s'appliquent à tous les objets conforment à ces rôles, et les interactions de niveau ¡¡objet¿¿ qui représentent des communications entre des objets particuliers. Cette deuxième catégorie d'interaction est une réification de la première pour des objets donnés conformes aux rôles. La Fig 5.1 montre un exemple de diagrammes de séquence entre objets.

La relation entre un diagramme de séquence et une spécification AS est immédiate : l'*Action Semantics* décrit complètement le comportement, alors qu'une interaction n'est qu'une vue partielle sur l'ensemble des comportements possibles : *une collaboration spécifie une restriction, une projection des comportements possibles d'un modèle...* d'après [110] page 2-127.

La documentation officielle de UML est malheureusement bien insuffisante quand il s'agit de donner une sémantique précise aux diagrammes de séquence. La correspondance entre la notation graphique et le méta-modèle n'est pas toujours bien explicitée et la notation est parfois imprécise. Par exemple, il n'est pas toujours possible de déterminer à la seule vue d'un diagramme de séquence si la notation exprime un branchement conditionnel ou l'envoi concurrent de plusieurs messages [118], le méta-modèle est également insuffisant pour décrire précisément les constructions valides dans

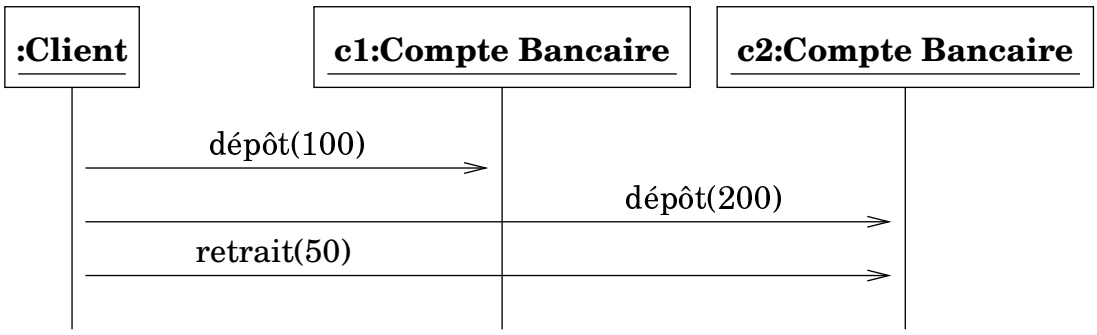


FIG. 5.1 – Communications entre objets exprimées par un diagramme de séquence

un tel diagramme. Sans l'apport de l'*Action Semantics*, l'exécution des diagrammes de séquence est impossible, à cause des nombreuses ¡¡expressions¿¿ textuelles utilisées dans ce type de diagramme. Ainsi, la notation permet la définition de branchements, d'alternatives ou de boucles. Mais leur traduction en terme d'éléments du méta-modèle n'est pas clairement précisée : le méta-élément utilisé pour spécifier une boucle est l'attribut *iterateExpression* de la méta-classe abstraite *Action*, encore une fois une expression sans syntaxe ni sémantique définies. Quelles sont les variables utilisables dans ces expressions, quels sont les objets accessibles ? Toutes ces règles ne sont que très informellement décrites sous forme textuelle. Pour une collaboration attachée à une opération, nous apprenons page 2-135 de la spécification [110] : *une collaboration décrivant une opération inclut uniquement les paramètres et les variables locales de l'opération, ainsi que les associations attachées à la classe possédant l'opération.* Cette phrase témoigne de l'incomplétude de la documentation UML : la notion de variable locale existe en AS mais pas dans UML. De plus, la sémantique des interactions doit s'accorder harmonieusement avec celle de l'AS car des actions (appel de méthodes, création ou destruction d'objets, etc.) peuvent être attachées aux messages. Nous pensons pour cette raison qu'une traduction des interactions en AS semble naturelle.

Enfin, une sémantique précise pour les diagrammes de séquence permettrait également de définir précisément ce qu'est le raffinage d'un diagramme de séquence.

5.3.1 Renforcer la sémantique des interactions

La sémantique des interactions a été approfondie dans [111, 129]. Nous explorons dans cette section un ensemble de propriétés déduites de l'utilisation conjointe des interactions et de l'*Action Semantics*. Nous nous intéressons dans un premier temps à un sous-ensemble des interactions restreint à la spécification d'une opération. Ces propriétés sont exprimées en OCL, au niveau des méta-modèles de la notation UML et de l'AS, détaillés dans [110] et [5] respectivement. Les premières propriétés que nous décrivons formalisent les contraintes brièvement évoquées dans la section 5.3. Nous nous intéressons ensuite aux propriétés des messages qui composent une interaction, par exemple nous exprimons que les messages échangés entre les rôles d'une même inter-

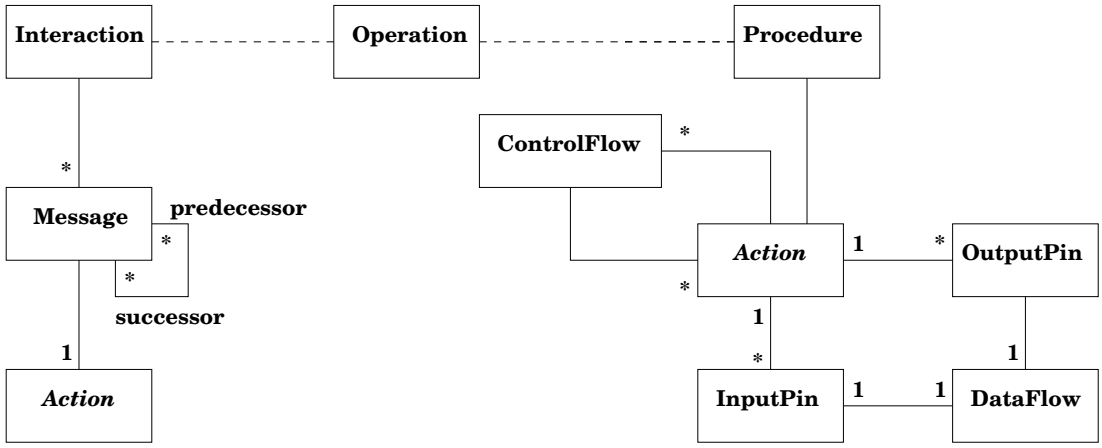


FIG. 5.2 – Interactions et AS : deux représentations pour une opération

action respectent l'ordre partiel imposé par les émissions et les réceptions de messages décrites dans la spécification AS de l'interaction. Ces contraintes expriment des propriétés d'équivalence entre deux types de descriptions (ici, les diagrammes de séquence et les descriptions AS). Elles sont un premier pas vers la traduction des diagrammes de séquence en description AS (et réciproquement). Nous explorons ensuite l'automatisation de ce mécanisme de traduction dans la section 5.3.2.

Propriété 1 *Seuls des paramètres, des variables locales appartenant à l'opération, ou des associations attachées au* Classifier *possédant l'opération peuvent être des rôles.*

```
context Interaction
  let operation = self.context.representedOperation in
    self.context.ownedElement→forAll(cr: ClassifierRole |
      cr.base→exists(c : Classifier |
        -- le rôle est conforme à l'un des paramètres de l'opération
        operation.parameter→includes(c)
        -- ou le rôle est conforme à une variable locale
        or operation.method.procedure.action.variable→type→includes(c)
        -- ou le rôle est conforme à une association attachée au Classifier
        or operation.owner.allOppositeAssociationEnds()→type→includes(c)))
```

Propriété 2 *Si dans une interaction, un message est émis par le rôle A vers le rôle B, alors il existe une* InvocationAction *dans la spécification AS de l'opération où l'une des bases de A possède l'action et l'une des bases de B peut être destinataire du message. Une* ReceiveAction *dans B traite le message, et s'il s'agit d'une invocation synchrone, alors il existe une* ReplyAction *dans B.*

```
context Interaction
self.message→forAll(m : Message |
m.sender <>m.receiver implies
```

```
( self .context.operation.method.procedure.allActions()→exists(ac:Action|
   ac.oclIskindOf(InvocationAction) and
   m.sender.base→exists(a|a.oclIsKindOf(ac.context.operation.owner))
   and m.receiver.base→exists(b|b.oclIsKindOf(ac.target.classifier))
   and (ac.oclIsKindOf(SynchronousInvocationAction) implies
      m.receiver.base→exists(b | b.allMethods()→exists(m |
      m.procedure.allActions()→exists(ac2 | ac2.oclIsKindOf(ReplyAction)))))
   and m.receiver.base→exists(b | b.allMethods()→exists(m|
      m.procedure.allActions()→exists(ac2 | ac2.oclIsKindOf(ReceiveAction)))))))
```

Propriété 3 *Si dans une interaction il existe un rôle qui envoie un message m1 avant un message m2 (c'est-à-dire que l'envoi de m2 a lieu après la réception de m1), alors il existe deux* InvocationActions *dans la spécification AS avec des récepteurs dont les types sont conformes et qui ont la même causalité.*

```
context Interaction
   self.message→forAll(m1, m2 : Message |
   -- m2 est après m1 et les 2 messages ont le même émetteur
   m2.allPredecessors()→includes(m1) and m1.sender = m2.sender implies
   self.context.operation.procedure.allActions()→exists(a1, a2 : Action |
   a1.oclIsKindOf(InvocationAction) and a2.oclIsKindOf(InvocationAction) implies
   (m1.receiver.base→exists(a : Classifier | a.oclIsKindOf(a1.target.classifier))
   and m2.receiver.base→exists(b : Classifier | b.oclIsKindOf(a2.target.classifier))
   and a2.allPredecessors()→includes(a1))))
```

5.3.2 Traduction des diagrammes de séquence vers l'*Action Semantics*

Les règles OCL énoncées précisent la sémantique des interactions, relativement à la sémantique de l'AS. Elles nous permettent de déterminer l'équivalence de diagrammes de séquence représentés par des *Interactions* ou des spécifications AS. Nous pouvons également chercher à construire une spécification AS à partir d'un diagramme de séquence. Dans ce cas, les règles énoncées sont vues comme des posconditions de l'opération `sequenceDiagramToAS(sd : Interaction) : Procedure`, qui est une transformation de modèles s'appliquant à un diagramme de séquence de niveau spécification et retournant une procédure AS :

```
context Interaction::sequenceDiagramToAS(): Procedure
   pre: -- le diagramme de séquence est valide
   post: -- les propriétés d'équivalence interaction/AS sont vérifiées
```

Il faut noter qu'une conséquence directe d'une telle traduction est de rendre les diagrammes de séquence attachés à une opération exécutable, car ils bénéficient alors de la sémantique d'exécution de l'AS. Ce mécanisme de traduction permettra donc également une simplification notable d'un interprète/compilateur de modèles. Les paramètres de l'opération constituent les *InputPins* de la procédure créée. Nous déduisons les traductions suivantes des règles énoncées :

- la **propriété 1** implique que les rôles participant à une collaboration qui ne sont ni des paramètres de l'opération, ni des associations de la classe possédant l'opération se traduisent en variables locales ;
- la **propriété 2** implique que tout *Message* envoyé d'un rôle A à un rôle B est traduit en une *InvocationAction* dans l'une des bases de A, et dont la *target* est conforme à B. Une *ReceiveAction* doit également exister dans B. Une *ReplyAction* est de plus nécessaire dans le cas d'un appel synchrone.
- la **propriété 3** implique que l'ordre partiel des messages d'une interaction se retrouve dans des dépendances de contrôle de l'AS.

Bien sûr, l'opération `sequenceDiagramToAS()` étant une transformation du méta-modèle UML, donc d'un modèle UML, il est possible d'utiliser l'*Action Semantics* lui-même pour décrire cette opération.

5.4 Construction de simulateurs UML

5.4.1 AS et réflexion

En suivant une démarche similaire, il semble possible de traduire toutes les vues dynamiques de la notation UML en leur équivalent AS. Cela motive notre choix de l'*Action Semantics* en tant que formalisme fondateur pour la sémantique de la dynamique des modèles UML.

Une conséquence de cette approche par compilation des diagrammes vers l'AS est que l'exécution d'un modèle ne dépend que de la définition de l'exécution des spécifications AS.

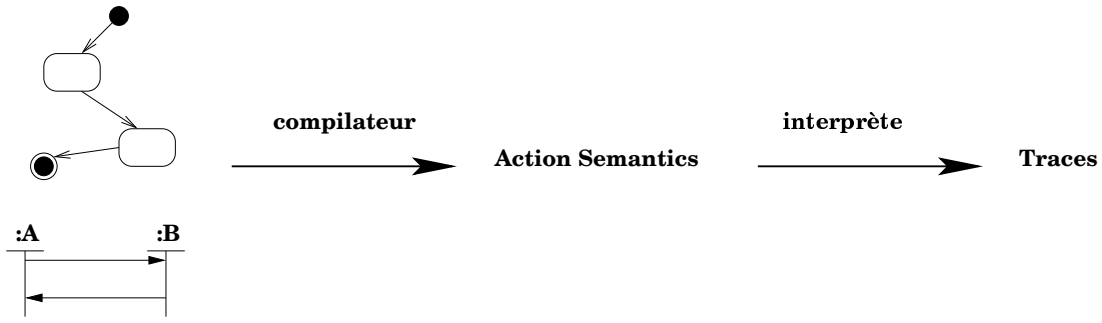


FIG. 5.3 – Compilation puis interprétation

Simuler un modèle UML revient dans notre cas à réaliser les deux fonctions suivantes : d'une part réaliser un compilateur capable d'assurer la traduction des vues dynamiques vers le sous-ensemble du méta-modèle dynamique défini par l'*Action Semantics*, d'autre part contruire un interprète capable d'exécuter les spécifications écrites avec l'*Action Semantics*, cf. Fig 5.3. Nous étudions dans la section 5.4.2 comment construire ces entités tout en restant dans un cadre UML. Une fois de plus, nous allons constater que l'*Action Semantics* semble ici un choix particulièrement adapté.

En effet, si à l'origine l'*Action Semantics* a été conçu pour une description du comportement des modèles, nous pensons que son rôle peut être étendu au delà de celui d'une simple vue dynamique UML. Un interprète UML, c'est-à-dire une implantation du modèle d'exécution de l'AS (mémoire de la machine virtuelle) est à l'origine destiné à la manipulation d'objets M_0 d'un modèle UML. Ces manipulations sont spécifiées au niveau M_1 et font partie du modèle dynamique de l'application. Mais comme le méta-modèle UML et le modèle d'exécution de l'*Action Semantics* sont eux-mêmes des modèles UML, nous proposons également d'utiliser l'*Action Semantics* pour spécifier les évolutions de ces modèles :

- dans le premier cas, grâce à l'architecture à quatre niveaux de UML, une spécification AS est écrite au niveau M_2 du méta-modèle et manipule des objets du niveau M_1, c'est-à-dire des modèles UML. Une spécification AS décrit alors une transformation de modèles (méta-programmation). Un compilateur transformant les diagrammes dynamiques de UML en leur équivalent AS est un tel exemple ;
- dans le second cas, une spécification AS est écrite au niveau M_1 du modèle d'exécution de l'AS et manipule des objets M_0, c'est-à-dire les objets présents à l'exécution (une représentation des objets du niveau M_0 appelée configuration, cf. 3.2.2). Dans ce cas, une spécification AS décrit la transformation appliquée à une configuration et résultant en une nouvelle configuration, c'est-à-dire la sémantique des actions de l'AS lui-même. Il s'agit de réflexion appliquée à la spécification du moteur d'exécution, nous obtenons un interprète. Nous détaillons la construction de ce type d'interprète dans la section 5.4.2.

Nous proposons de combiner ces deux approches dans des outils de méta-modélisation réflexifs : le même moteur d'exécution s'applique alors à la fois à l'exécution des transformations de modèles et à la simulation du modèle. Nous détaillons cette approche au chapitre 6 où nous l'appliquons à l'outil de modélisation UMLAUT.

5.4.2 Sémantique opérationnelle pour l'*Action Semantics*

Formaliser l'exécution des actions

Il faut reconnaître que la manière dont la sémantique de l'*Action Semantics* est décrite dans le document proposé à l'OMG [5] est en partie inadaptée aux besoins des concepteurs d'outils UML. En effet, si les nombreuses préconditions et postconditions de la sémantique des actions fournissent toutes les précisions nécessaires pour vérifier la conformité d'une implantation, leur forme déclarative n'aide en rien à la construction d'une implantation correcte, ainsi que nous l'avons vu à la section 3.2.3. De plus, la grande complexité de certaines de ces préconditions et postconditions fait craindre un certain nombre d'erreurs ou d'incomplétudes dans le document normatif lui-même. Or, l'acceptation et la diffusion de l'*Action Semantics* nécessite un support important des fournisseurs d'ateliers de génie logiciel UML et la confiance des utilisateurs. Aussi, nous pensons qu'aux documents de description de l'*Action Semantics* devrait être ajoutée une sémantique opérationnelle facilitant la mise en œuvre et l'implantation de simulateurs conformes. Nous proposons de réaliser cette description opérationnelle à l'aide du

modèle d'un interprète écrit en AS et travaillant sur le modèle d'exécution.

Afin d'illustrer cette approche, nous détaillons dans un premier temps la formalisation des actions présentée dans la spécification. Nous traitons ici un cas simple, celui de l'action *SynchronousAction* utilisée pour envoyer de manière synchrone un message à un objet. Le sous-ensemble du modèle d'exécution qui nous intéresse, déjà présenté sur la Fig. 3.6, est rappelé sur la Fig. 5.4. Nous avons traité cet exemple en détails dans [100].

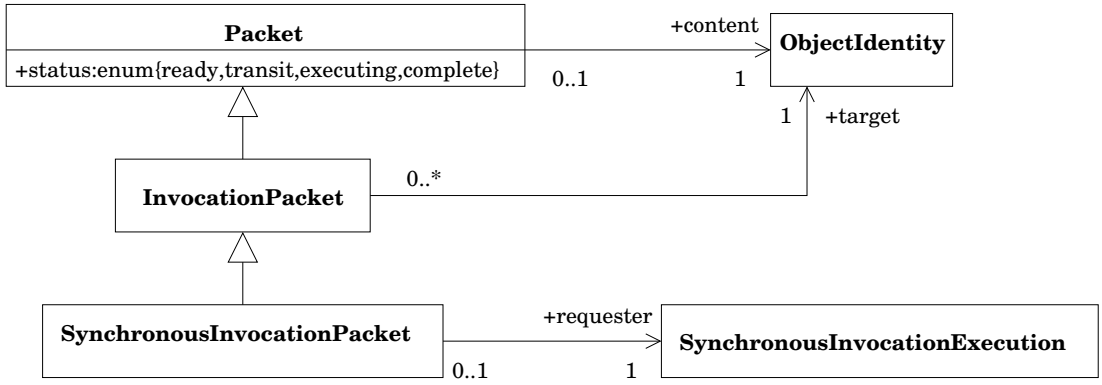


FIG. 5.4 – Modélisation de l'envoi de messages dans le modèle d'exécution

L'exécution de cette action est décrite dans [5] de la façon suivante : *l'émetteur crée un message de requète et l'envoie à l'objet destinataire au moyen d'une action d'invocation synchrone, après quoi l'exécution de l'émetteur est bloquée.* Comme les autres exécutions, son comportement est décrit par un ensemble d'étapes élémentaires appelées *productions*. L'extrait du méta-modèle de la Fig 5.4 représente le modèle utilisé par le moteur d'exécution pour l'exécution de cette action. Voici un exemple de production[1] :

Production 1 :
-- *synchronous invocation action generates a synchronous invocation packet and blocks itself.*
Precondition :
self .status = #ready
Postcondition :
self .status@post = #executing -- *put call in executing state while subordinate execution proceeds*
packet:SynchronousInvocationPacket.isNew **and** -- *create a call packet*
packet.requester@post=self **and**
packet.status@post= #ready **and**
packet.target@post=self.target **and**
packet.content@post=self.argument

Cette production indique que la nouvelle configuration créée lors de l'exécution (les objets suivis de la notation `@post`) contient un nouveau *SynchronousInvocationPacket*

[1] la notation `obj@post` utilisée dans la postcondition de la production représente la valeur de `obj` dans la nouvelle configuration créée lors de l'exécution de la production courante.

```
do in sequence {
   packet := new SynchronousInvocationPacket -- a CreateObjectAction
   do in parallel {
      packet.requester := self -- a CreateLinkAction
      packet.status := #ready -- a WriteAttributeAction
      packet.target := self.target -- a ReadLinkAction then a CreateLinkAction
      packet.content := self.argument -- a ReadLinkAction then a CreateLinkAction
   }
}
```

FIG. 5.5 – Spécification AS d'une production

correctement initialisé.

Une description opérationnelle équivalente (c'est-à-dire qu'en partant de la précondition de la production, l'exécution de cette description doit impliquer la postcondition) est décrite Fig 5.5, dans un langage imaginaire qui pourrait convenir comme syntaxe concrète de l'*Action Semantics*. Ce mini programme construit un paquet vérifiant les propriétés requises par la postcondition. La correspondance entre notre syntaxe concrète et les actions du méta-modèle de l'AS utilisées est indiquée dans les commentaires. Dans un soucis de concision, nous avons choisi un exemple où la déduction d'une spécification AS est relativement simple, mais ce n'est bien sûr pas toujours le cas. Le lecteur consultera [5] pour se rendre compte de la complexité de certaines des assertions (nous avions également donné l'exemple d'une production de l'exécution d'une *ReadAttributeAction* sur la Fig. 3.7). Nous pouvons noter que la spécification opérationnelle obtenue peut être très efficace pourvu qu'elle exploite les possibilités d'ordres partiels offertes par l'AS. Nous cherchons en effet à fournir une spécification opérationnelle sans perdre les abstractions de l'AS afin que celle-ci puisse ensuite être raffinée plus précisément par un concepteur d'outils soucieux d'atteindre des objectifs spécifiques à son implantation, entre autres en terme de performance.

Nous pouvons établir un parallèle entre ces spécifications AS des productions et l'architecture en couches d'UML. En effet, lorsque ces productions sont exécutées, les objets manipulés sont les objets du modèle d'exécution de l'AS. Si nous comparons ces actions aux actions utilisées par un concepteur pour décrire les comportements de son application, ce sont en quelque sorte des ¡¡méta-actions¿¿ : ces actions décrivent l'exécution d'autres actions, tout comme le méta-modèle UML décrit le modèle UML.

Formaliser le moteur d'exécution

Notons que si la sémantique des actions (cf. section 3.2.3) est clairement définie dans [5] par le biais de préconditions et de postconditions, cette spécification présente tout de même quelques lacunes sur d'autres points. Il s'agit en particulier des mécanismes fondamentaux du ¡¡moteur de l'exécution¿¿ (les *life-cycles*), qui permettent à l'exécution d'une action de progresser par étapes quand les préconditions de celles-ci sont vérifiées. Ceux-ci ne sont pas formellement décrits. Nous pensons qu'il est important que ces

descriptions, bien que précises, bénéficient également d'une formalisation. Dans ce but, nous adoptons toujours la même démarche : utiliser d'abord des préconditions et post-conditions, puis raffiner les spécifications données par ces contrats en spécifications AS exécutables.

Une fois de plus, il y a un lien entre cette description des mécanismes d'exécution et l'architecture d'UML. Cette fois ci, nous cherchons à décrire le niveau ¡¡méta-méta¿¿ de l'architecture d'exécution des modèles UML. Bien entendu, nous utilisons l'AS pour cela et notre niveau ¡¡méta-méta¿¿ d'exécution se retrouve donc être un sous-ensemble du niveau ¡¡méta¿¿, tout comme le méta-méta-modèle UML est un sous-ensemble du méta-modèle UML. Cette architecture peut éventuellement être ramenée à trois niveaux seulement comme dans le cas d'UML (cf. chapitre 2).

Nous montrons brièvement comment le mécanisme d'enchaînement des productions peut-être décrit avec l'AS. Une production est définie par sa précondition et sa postcondition, et ne peut s'exécuter que si la précondition est vraie. Nous proposons de modéliser les *life-cycles* par une action conditionnelle (*ConditionalAction*) de l'AS, comportant autant de branches qu'il y a de productions. Les clauses de test de la conditionnelle sont les préconditions et le corps de chaque alternative est l'action de la production. Ceci est illustré sur la Fig. 5.6.

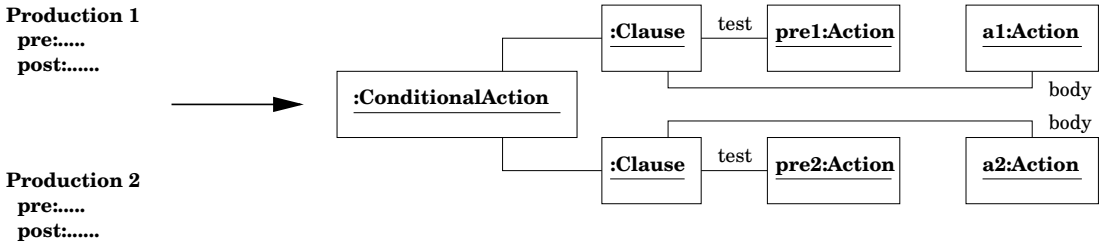


FIG. 5.6 – Formalisation des *life-cycles* avec l'AS

Dans le mécanisme d'exécution des modèles UML de l'AS, une action peut être exécutée à partir du moment où elle dispose de toutes les données qui lui sont nécessaires. La quantité et le type de ces données sont spécifiés via des flots de données rattachés aux plots d'entrée (*InputPins*) des actions (voir à ce sujet le méta-modèle de l'AS de la Fig. 3.2). Dans le modèle d'exécution, l'exécution d'une action n'est donc possible que lorsque cette *ActionExecution* est reliée à un nombre adéquat de valeurs (*PinValues*). La précondition des productions qui permet de passer de l'étape d'exécution `#waiting` à l'étape d'exécution `#ready` est en fait équivalente à cette contrainte sur la disponibilité des données utilisées par l'exécution de l'action :

context ActionExecution
  inv: self.status = #ready **implies**
    self.pinvalue→**size** = ae.action.inputPin→**size**

Notre interprète peut donc à tout instant obtenir l'ensemble des *ActionExecutions* qui sont `#ready` et donc succeptibles d'être exécutées (étape `#execute`) en utilisant la

fonction suivante :

```
ready_action_executions () : Set(ActionExecution)
   post:
      result→forAll(ae : ActionExecution |
                ae.pinValue→select(pv | pv.oclIsKindOf(InputPin))→size
                   = ae.action.inputPin→size)
```

Il reste à choisir une ou plusieurs exécutions et à exécuter leur étape `#execute`.

```
next_executions () : Set(ActionExecution)
   post:
      -- toutes les actions choisies sont #ready, ie. disposent de
      -- toutes leurs valeurs en entrée
      ready_action_executions()→includesAll(result)
```

```
execute_next_executions()
   do in sequence {
      executions = next_executions()
      forAll execution in executions {
         execution.execute()
      }
   }
   post:
      -- après l'exécution, on passe dans l'étape #complete
      -- ce #complete est équivalent à écrire que toutes les données sont disponibles
      -- sur les pins de sorties de l'ActionExecution
      executions→forAll(ae : ActionExecution | ae.status = #complete)
      -- est donc équivalent à
      executions→forAll(ae | ae.pinValue→select(pv | pv.oclIsKindOf(OutputPin))→size
         = ae.action.outputPin→size)
```

Notons que l'interprète d'UMLAUT implante ce schéma en Eiffel. Il ne réalise donc pas la vraie concurrence de l'AS mais uniquement des entrelacements.

Chapitre 6

UMLAUT : un outil pour le co-design

Dans ce chapitre, nous explorons les améliorations au processus de développement logiciel que peut apporter un outil de manipulation des niveaux ¡¡méta¿¿ permettant de manipuler les modèles UML de l'utilisateur, mais également le méta-modèle UML et le modèle d'exécution de l'*Action Semantics* qui permet de réaliser la simulation des modèles.

Dans la section 6.1 nous décrivons tout d'abord comment améliorer les spécifications logicielles par l'écriture de contraintes portant non plus sur le modèle de l'application, mais sur son modèle d'exécution. Nous verrons que cela est utile pour la spécification des comportements.

Nous verrons ensuite dans la section 6.2 quels sont les principes qui permettent de passer graduellement – et si possible continûment – des spécifications aux implantations. Nous tentons de formaliser cette évolution des modèles en nous appuyant sur l'approche décrite dans la section 6.1.

Dans la section 6.3 nous verrons que la démarche présentée dans les sections 6.1 et 6.2 nécessite l'utilisation d'outils de simulation des modèles, afin d'assurer la conformité des implantations par rapport aux spécifications.

Puis dans la section 6.4 nous montrons comment ce processus d'évolution peut être décrit et automatisé par des techniques de méta-programmation.

Enfin, dans la section 6.5 nous verrons comment mettre en œuvre ces approches dans un outil de développement intégré, comme c'est le cas pour l'outil UMLAUT présenté dans la section 6.6.

6.1 OCL et modèle d'exécution

Nous avons évoqué dans la section 4.2.3 les limites à l'expressivité des propriétés d'un modèle UML imposée par la notation OCL. En particulier, ce langage ne permet pas d'exprimer des propriétés de logique temporelle et les articles [119, 37, 32, 104] proposent des modifications du langage pour autoriser de telles contraintes.

Néanmoins, nous pensons que le langage OCL peut dans sa version actuelle être utilisé pour exprimer des propriétés temporelles dans le cadre d'un outil permettant l'accès au modèle d'exécution de l'*Action Semantics*, ce qui est le cas du simulateur de modèles UML implanté dans UMLAUT. Il est en effet possible d'exprimer le comportement temporel des modèles par la spécification de contraintes sur les configurations de l'historique d'un objet M_0. Ainsi un invariant sur une classe d'un modèle UML se traduit au niveau des objets M_0 qui réifient le modèle d'exécution par une propriété qui doit être vérifiée pour toutes les configurations des objets de cette classe. Nous avions déjà évoqué dans la section 4.3.5 cette ¡¡dualité¿¿ entre les invariants et les propriétés sur les objets M_0 du modèle d'exécution.

Nous illustrons ce principe par quelques exemples simples.

6.1.1 Exemple élémentaire

Nous considérons deux classes A et B reliés par une association de A vers B, dont la multiplicité du côté de la classe B est 0..3 (cf. Fig. 6.1). Cette multiplicité est équivalente

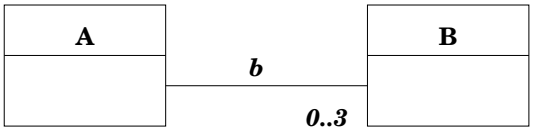


FIG. 6.1 – Un A est associé à 3 B au plus

à l'invariant OCL suivant sur la classe A :

context A
-- *un A est attaché à 3 B au plus*
inv: self.b $\leq$ 3

Sur le modèle d'exécution, cela se traduit par la contrainte suivante qui – partant d'un objet de la classe `A` – parcourt tous les objets attachés et calcule le nombre d'objets attachés par l'association `b` :

context ObjectIdentity
-- *un A est attaché à 3 B au plus*
inv: **let** class_A : Class = Class.**allInstances**()→**select**(c | c.**name** = 'A')→**first** in
self. classifier →**includes**(class_A) **implies**
let attached_B_objets = self.linkEndValue.linkValue.linkEndvalue→**select**(lv |
lv.link.association.**name** = 'b') in
attached_B_objets→**size** $\leq$ 3

6.1.2 Le patron de conception ¡¡Singleton¿¿

Le patron de conception ¡¡Singleton¿¿ [43] est utilisé pour exprimer qu'une classe doit être réifiée au plus une fois. Cela se traduit par la contrainte OCL suivante :

Singleton.**allInstances**()→**size** ≤ 1

Ce qui se traduit sur le modèle d'exécution par :

```
context ObjectIdentity
  inv: let classe_Singleton = Class.allInstances()→select(c |
    c.name = 'Singleton')→first in
      self . classifier →includes(classe_Singleton)→size ≤ 1
```

6.1.3 Le patron de conception ¡¡Observateur¿¿

Le cas du patron de conception ¡¡Observateur¿¿ est plus intéressant, car les propriétés comportementales de ce patron de conception ne sont pas exprimables comme les précédentes à l'aide d'invariants OCL. Le patron ¡¡Observateur¿¿ fait collaborer une entité `Sujet` et un nombre quelconque d'entités `Observateurs`. Sa définition ¡¡abstraite¿¿ est que les observateurs doivent être prévenus de toute modification de l'état du `Sujet` (ou des modifications d'un sous-ensemble des états si l'observateur n'est pas intéressé par la connaissance de l'état complet de l'observé).

L'application du patron ¡¡Observateur¿¿ à deux classes (un sujet et un observateur) se représente en UML par le diagramme de la Fig. 6.2.

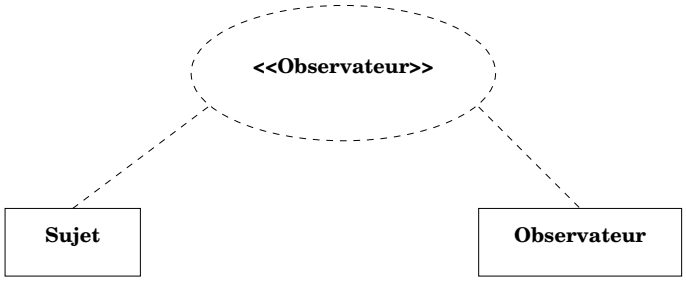


FIG. 6.2 – Application du patron ¡¡Observateur¿¿

Sur le modèle d'exécution, nous pouvons traduire cette propriété par le fait que toute modification de l'état de l'objet `Sujet`, c'est-à-dire toute exécution d'une *WriteAttributeAction* sur cet objet (écriture d'une valeur dans un attribut) devra être suivie d'un appel à une méthode `notify()` chargé de ¡¡prévenir¿¿ les observateurs (par l'envoi d'un message). L'appel à l'opération `notify` est effectuée par l'exécution d'une *CallOperationAction* sur cette opération. Cette notification peut être asynchrone. Les configurations créées pendant l'exécution de cette *CallOperationAction* sont donc postérieures aux configurations créées pendant l'exécution de la *WriteAttributeAction* dans l'historique de l'objet `Sujet`.

```
context WriteAttributeActionExecutionSnapshot
  inv: ( self .status = #complete and self.action. classifier .name = 'Sujet') implies
    -- completionTime indique quand l'écriture de l'attribut est terminée
    -- cf. 3.2.2 et l'attribut 'time' de la classe 'Change'
    let completionTime = self.postChange.time in
      let sujet = self .objectId in -- récupère un 'pointeur' sur le 'Sujet'
      -- récupère tous les appels de méthodes postérieurs à l'écriture de l'attribut
```

```
CallOperationActionExecutionSnapshot.allInstances()→select(coaes |
   -- sélection des appels qui nous intéressent
   -- ils concernent l'objet 'sujet'
   coaes.objectId = sujet
   -- ils concernent l'opération 'notify'
   and coaes.operation.name = 'notify'
   -- ils ont eu lieu après l'écriture de l'attribut
   and coaes.status = #waiting and coaes.preChange.time ≥completionTime)
-- les appels sélectionnés sont les 'notify' sur l'objet 'sujet'
-- il DOIT y avoir au moins un appel à 'notify'
→notEmpty
```

6.2 Des spécifications aux implantations

6.2.1 Raffinage et Réusinage

Le processus de développement logiciel vise à produire une implantation conforme à la spécification initiale. Les ateliers de génie logiciel qui existent actuellement assurent rarement cette conformité de l'implantation par construction ; citons tout de même le cas de l'Atelier B qui est un outil basé sur la méthode formelle B [2]. Les outils travaillant avec la notation UML ne permettent malheureusement pas une telle approche. Le lien entre deux modèles n'est ni explicité, ni formalisé. UML introduit pourtant les concepts de *dépendance*, d'*abstraction*, de *dérivation*, de *réalisation* ou de *trace*, etc. ainsi que des stéréotypes (¡¡refine¿¿, ¡¡derive¿¿) pour indiquer que certains éléments d'un modèle se ¡¡déduisent¿¿ d'autres éléments, mais la signification précise de ces notions reste à définir.

UML fournit donc des concepts utiles à la définition d'une méthodologie de conception *continue*. Il s'agit de faire évoluer les modèles – depuis les spécifications jusqu'à une implantation – par une succession d'évolutions aussi petites que possible, de manière *incrémentale*. L'impact limité de chacune de ces étapes sur le modèle doit permettre de vérifier plus facilement que le nouveau modèle obtenu à chaque étape conserve les propriétés (issues de la spécification initiale) du modèle dont il est une évolution.

Les opérations qui permettent de faire évoluer une spécification vers une implantation sont le raffinage et le réusinage. Le réusinage consiste à modifier un modèle sans en modifier les comportements. Le raffinage consiste à modifier un modèle en supprimant certains comportements. Par exemple, renommer une classe dans un modèle UML est un réusinage. Remplacer un comportement non déterministe par un comportement déterministe est un raffinage. Nous pouvons dire qu'en terme de comportements, le réusinage conserve la même abstraction que le modèle qu'il raffine, alors que le raffinage donne un modèle plus précis, plus ¡¡concret¿¿. Cette évolution des spécifications initiales vers des implantations est schématisée sur la Fig. 6.3. Les points situés sur une même ligne horizontale représente des modèles qui sont des réusinages les uns des autres ; ils sont strictement les mêmes comportements, et leur niveau d'abstraction sont identiques. Les points situés sur une même ligne verticale représente des modèles de

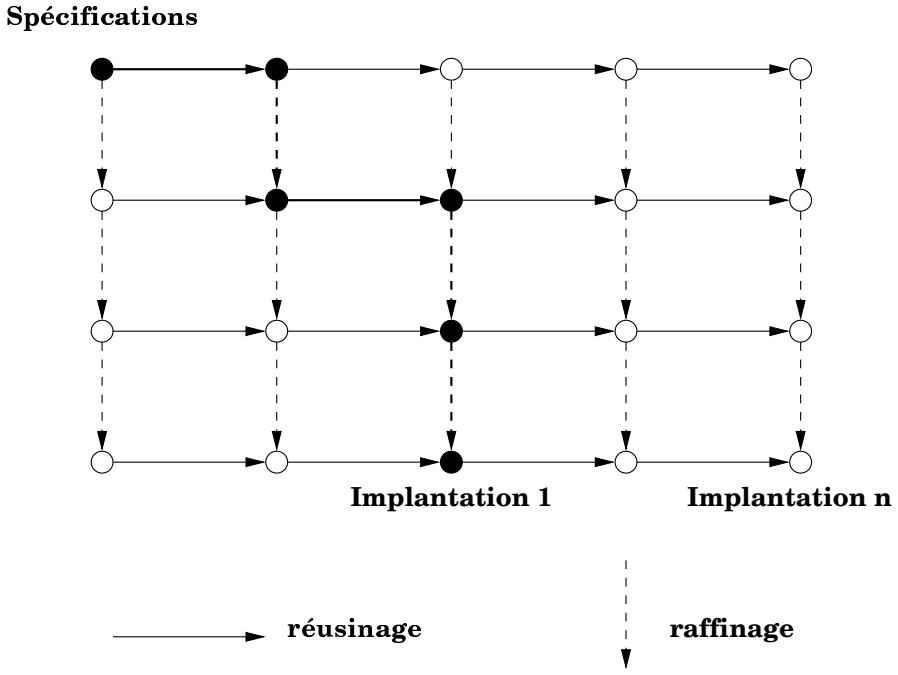


FIG. 6.3 – Des spécifications à l'implantation : raffinages et réusinages

plus en plus précis en allant vers le bas, jusqu'aux implantations. Plusieurs chemins sont évidemment possibles pour parvenir à une même implantation particulière.

Il est pourtant difficile d'éviter l'apparition de discontinuités lors du processus de raffinage : elles surviennent lorsque l'évolution du modèle conduit à un changement du niveau d'abstraction. C'est particulièrement flagrant lors du passage d'une spécification (exprimée généralement dans un langage manipulant des abstractions de haut-niveau, tel UML) à une implantation utilisant des mécanismes d'abstraction moindre, car liés à la plateforme matérielle, par exemple. Le fossé entre deux niveaux d'abstraction est souvent lié au changement de langage de description : ainsi, lors de la génération du code, nous passons d'un modèle UML à un modèle C++ ou Java.

C'est pour cette raison qu'il est utile de poursuivre le développement d'une application au sein d'une seule et même notation : l'uniformité des concepts manipulés et leur cohérence sémantique sont alors assurées. UML permet une approche de ce type en proposant à la fois des concepts de haut-niveau (par exemple, les collaborations) et des concepts de niveau d'abstraction moindre, plus ¡¡proches¿¿ des langages d'implantation, comme l'*Action Semantics*. Les notions ¡¡refine¿¿, ¡¡derive¿¿ d'UML permettent par exemple d'exprimer qu'une collaboration raffine un cas d'utilisation. Notons qu'il est néanmoins peu probable (et sans doute peu souhaitable) qu'une notation intègre les concepts de toutes les architectures cibles. Cela surchargerait la notation et la rendrait inutilisable et non standardisable.

6.2.2 Un exemple de réusinage

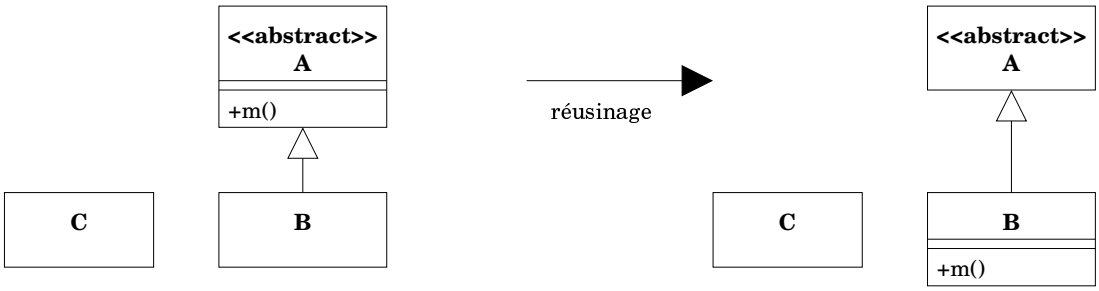


FIG. 6.4 – Un exemple de réusinage

Nous présentons ici un exemple de réusinage d'un modèle de conception. Nous nous intéressons à un modèle décrit par le diagramme des classes de la Fig. 6.4. Nous ne considérons pas les autres vues (statiques ou dynamiques) dans cet exemple. Dans ce modèle, la classe A est abstraite. Aucun objet de la classe A ne peut dont être créé et l'opération `A : :m()` est donc toujours exécutée pour un objet de la classe `B`. Il est donc légitime de déplacer cette opération de la classe `A` vers la classe `B` sans affecter les comportements du modèle.

D'après ce que nous avons vu dans la section 6.1, nous pouvons définir les comportements d'un modèle par des expressions OCL exprimées sur le modèle d'exécution (pour les propriétés temporelles), ou sur le modèle de l'application dans les cas les plus simples (invariants).

Pour le réusinage de la Fig. 6.4 les propriétés qui nous intéressent sont des invariants que nous exprimons sur le modèle de l'application. Elles sont une conséquence du caractère abstrait de la classe A :

---

context A
  inv: *-- il n'existe aucun objet de la classe A*
    self .**allInstances**()→**isEmpty**

En considérant la spécification de la dynamique de cette application (exprimée avec l'*Action Semantics*, par exemple), cette propriété pourrait également s'exprimer :

---

context CreateObjectAction
  inv:
    -- il n'existe pas de création d'objet de la classe A
    self .**allInstances**()→**forAll**(coa | coa.class.**name** <>'A')

Le modèle réusiné conserve ces propriétés du modèle initial, c'est pourquoi il s'agit a priori d'un réusinage *par rapport à cette propriété.*

6.2.3 Définitions précises de raffinage et réusinage

Nous avons lié les notions de raffinage et de réusinage aux comportements d'une application. Ces comportements sont représentés dans le modèle d'exécution par des historiques des configurations. Nous considérons un modèle M défini par sa spécification P (un ensemble de propriétés sur les historiques du modèle d'exécution).

Soit un modèle M'.

- si $\forall H'$ historique pour le modèle M', H' vérifie P alors M' est un raffinage de M. Nous notons $\mathit{raffine}(P, M, M')$;
- si $\mathit{raffine}(P, M, M')$ et $\mathit{raffine}(P, M', M)$ alors M et M' sont des réusinages l'un de l'autre. Nous notons $\mathit{réusinage}(P, M, M')$.

Avec ces définitions, tout réusinage est également un raffinage.

6.3 La simulation : un outil indispensable

Afin de vérifier que les modèles vérifient les propriétés de la spécification, exprimées sous la forme d'expressions OCL sur les historiques du modèle d'exécution, il est souhaitable que les ateliers de génie logiciel disposent de simulateurs de modèles UML.

Malheureusement, cette fonctionnalité est absente de la plupart des outils disponibles sur le marché. Cela est bien entendu en partie lié à la notation UML qui ne donne pas toujours des modèles exécutables. Il faut donc souvent se contenter de l'exécution d'une implantation particulière (souvent il s'agit d'exécuter le code généré par l'outil à partir du modèle), et non pas de l'exécution des spécifications. Cette démarche se rapproche plus du test d'intégration que d'une véritable simulation de l'application. Il est évident que cette approche augmente le coût des corrections des modèles, puisque les problèmes détectés sur une implantation peuvent avoir pour origine une erreur de conception bien en amont dans le processus de développement.

Le second problème de cette approche est qu'elle ne répond pas vraiment à la question de savoir si l'application est correctement modélisée en UML. En effet, cette méthode ne permet pas de vérifier la conformité d'une spécification par rapport à la sémantique d'exécution de l'*Action Semantics*, mais sa conformité par rapport à un modèle d'exécution bien déterminé, lié à la cible logicielle et matérielle visée (langage, système d'exploitation, matériel, etc.). De plus, le lien entre le modèle UML exécuté sur une machine virtuelle *Action Semantics* et le modèle d'implantation exécuté sur la plate-forme d'implantation n'est pas clair : le mécanisme d'exécution de la plate-forme cible est-il un raffinage du modèle d'exécution de l'*Action Semantics* ? Cette traçabilité entre modèles d'exécution est impossible au sein des outils car la plate-forme d'exécution n'y est pas modélisée. Cela entraîne les discontinuités du processus de développement dues au ¡¡saut¿¿ sémantique que nous évoquons dans la section 6.2.

En effet, dans ce cas la sémantique d'exécution ne fait pas partie du modèle manipulé par le concepteur, mais est ¡¡à part¿¿ : elle est définie par le support d'exécution sous-jacent (le système d'exploitation, le langage cible et ses biblothèques). La cohérence de cette sémantique avec celle d'UML se pose donc immédiatement. De plus, cette sémantique est généralement décrite avec un formalisme différent de celui d'UML, d'ou

des problèmes de compréhension et d'intégration. Cela apparaît clairement dans le cas d'une ingénierie à rebours [1], où seul le code de l'application est repris et transformé en modèle. Les autres connaissances sur l'application – et en particulier le modèle de son environnement d'exécution – sont perdues.

À titre d'exemple, nous considérons le cas des objets actifs. Ces objets possèdent en UML et dans l'*Action Semantics* une queue de réception pour les messages en attente de traitement. Ces queues de messages ont une capacité infinie dans la spécification *Action Semantics*. Il est évident qu'une implantation ne disposera en pratique que de files de taille bornée. Le concepteur a deux possibilités. La première possibilité consiste à ne pas modéliser le concept de queue dans le modèle de l'application, puisque ce concept fait partie du mécanisme d'exécution d'UML et est donc implicite dans tout modèle UML où des objets actifs communiquent. Mais dans ce cas, une implantation conforme est impossible en raison du caractère non borné des files UML. La deuxième possibilité consiste à raffiner le modèle UML en y faisant apparaître explicitement la notion de file telle qu'elle est définie dans le langage cible, avec sa taille (ou en généralisant, une file paramétrée par sa taille maximale). Mais alors le modèle UML de l'application est surchargé par des concepts liés à l'exécution, au détriment de sa clarté et de sa réutilisation.

Le mélange du modèle de l'application et du modèle de l'exécution risque de conduire à un modèle sur spécifié : dans le cas où la cible matérielle ou logicielle est modifiée, il n'est pas possible de déterminer si le modèle continuera à s'exécuter correctement ou bien si une reprise complète de la conception est nécessaire. Par exemple, une application concurrente mal conçue pourra très bien fonctionner correctement sur une plate-forme ou l'exécution est séquentielle.

Nous décrivons dans la section suivante (6.4.2) une approche de ¡¡co-design¿¿ qui apporte des éléments de réponse à ce problème en réifiant le passage du modèle d'exécution UML au modèle d'exécution de la plate-forme cible, tout en séparant ces évolutions de celles du modèle de l'application.

6.4 Raffinages et transformations

Plutôt que de vérifier *a posteriori* (par simulation) qu'un modèle est un raffinage, il nous semble plus judicieux d'assurer le raffinage par construction, tout au moins quand cela est possible. Il est envisageable de décrire les raffinages par des transformations de modèles.

Nous considérons une transformation de modèles qui transforme un modèle M satisfaisant l'ensemble des propriétés P en un modèle M'. Cette transformation est définie par ses préconditions et ses postconditions.

Si cette transformation est un raffinage, c'est-à-dire $\mathit{raffinage}(P, M, M')$, alors le modèle transformé M' vérifie également les propriétés de P :

---

transformation_de_raffinage (m : Model) : Model
pre : -- *m vérifie les contraintes P*

---

[1] De l'anglais *reverse engineering*.

post : -- *result vérifie les contraintes P*

6.4.1 Raffinage des transformations

Dans le contexte des méthodes à objets que nous étudions, il est souhaitable que les transformations puissent également profiter des acquis de ces méthodes. En particulier, nous souhaitons pouvoir réutiliser des transformations, par exemple par héritage ou redéfinition. Nous pouvons donc définir la notion de raffinage d'une transformation en nous calquant sur la conception par contrats [82] : la transformation héritée à une précondition plus faible et une postcondition plus forte que la transformation dont elle hérite. Avec cette définition, une transformation qui hérite d'un raffinage pour les propriétés P reste un raffinage pour ces propriétés :

raffinage_de_transformation (m : Model) : Model
pre : -- *(m vérifie les contraintes P) OU (d'autres contraintes)*
post : -- *(result vérifie les contraintes P) ET (d'autres contraintes)*

Nous reprenons l'exemple du patron de conception ¡¡Observateur¿¿ présenté dans la section 6.1.3, pour lequel la propriété essentielle est que toute modification de l'état de l'objet doit être suivie par un appel à l'opération `notify()`. Une transformation possible pour raffiner un modèle devant implanter un observateur est tout simplement d'ajouter dans la spécification AS du modèle une action *CallOperationAction* d'appel de l'opération `notify()` immédiatement après chaque action *WriteAttributeAction*, et avant l'exécution de toute autre action. Ceci est illustré par le diagramme d'objets (de niveau ¡¡méta¿¿) de la Fig. 6.5.

```
raffinage_observateur (m : Model) : Model
  -- cette opération ajoute un appel à notify() après chaque action WriteAttributeAction
  -- d'un Sujet d'une application du patron ¡¡Observateur¿¿
  -- bien sûr nous pouvons écrire le corps de cette transformation avec l'AS
  post :
    -- result vérifie AU MOINS la propriété de l'observateur, cf.6.1.3
    -- nous définissons ici une propriété plus forte que celle de 6.1.3

    -- sélectionne toutes les WriteAttibuteActions dans toutes les opérations de
    -- la classe 'Sujet'
    Class.allInstances()→select(c | c.name = 'Sujet')→forAll(c |
      c.allOperations().procedure.allActions())
      →select(a | a.oclIsKindOf(WriteAttributeAction))
      -- toutes les WriteAttributeActions doivent être suivies (dépendance 'ControlFlow')
      -- par une CallOperationAction sur l'opération 'notify'
      →forAll(a | a.consequent.successor→exists(next_action |
        next_action.oclIsKindOf(CallOperationAction)
        and next_action.oclAsType(CallperationAction).operation.name = 'notify'))
```

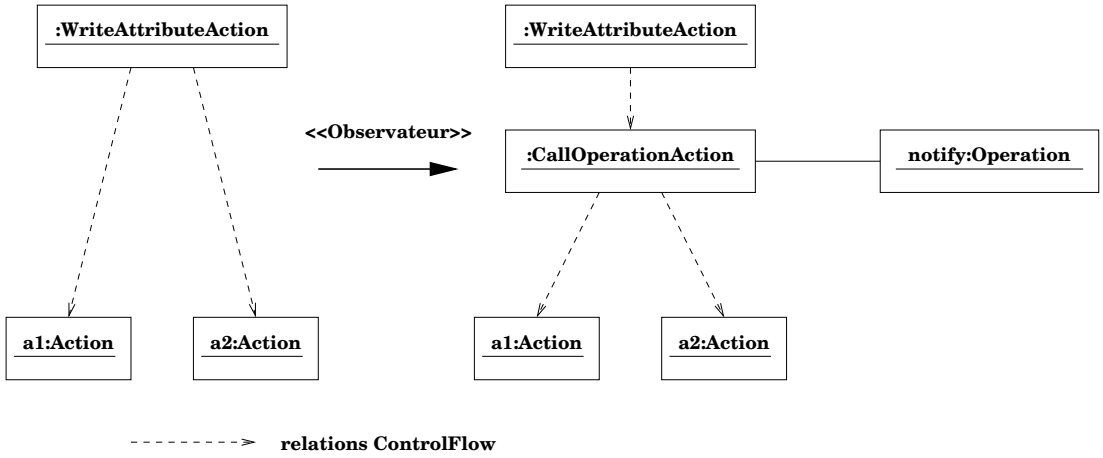


FIG. 6.5 – Une implantation du patron ¡¡Observateur¿¿

6.4.2 Transformation du modèle ou de l'interprète

Les exemples de raffinage que nous avons présentés sont tous basés sur le principe de la transformation de modèles de la programmation par aspects. Or, nous avons vu dans la section 2.5.2 que le tissage des aspects pouvaient également être réalisé par une transformation de l'interprète. Dans le cas d'UML, cette transformation porte sur le modèle d'exécution de l'*Action Semantics*.

Il est possible de raffiner et réusiner ce modèle d'exécution comme tout autre modèle UML. L'intérêt premier de cette approche est de combler le fossé sémantique entre les exécutions de modèles UML et les exécutions de leurs implantations que nous avons décrit dans la section 6.2. Enfin, cette approche permet une meilleure abstraction des modèles de l'application, puisque ceux-ci sont désormais débarrassés des contraintes d'implantation. Cette approche est illustré sur la Fig. 6.6, qui est à comparer au processus de développement présenté sur la Fig. 6.3. L'évolution de la spécification vers une implantation est maintenant décomposée selon trois plans : l'application, son modèle d'exécution, l'implantation.

Si nous reprenons l'exemple du patron de conception Observateur, la transformation d'interprète correspondant à l'application de ce patron consiste à modifier la sémantique d'exécution de l'action *WriteAttributeAction*. Dans le modèle d'exécution, il s'agit donc de redéfinir l'étape `#execute` de la classe *WriteAttributeActionExecutionSnapshot* (cf. section 3.2.3) pour que lors de l'exécution de cette étape une *CallOperationActionExecutionSnapshot* soit exécutée immédiatement après.

Les transformations qui ne font pas partie du domaine de l'application mais sont spécifiques à l'exécution des modèles sont nombreuses. Nous pouvons citer la modélisation des communications entre objets actifs : le concepteur est uniquement intéressé par l'association entre les classes de son diagramme des classes. La réalisation effective sous forme de files est un problème d'implantation. De même, un objet actif est trop souvent

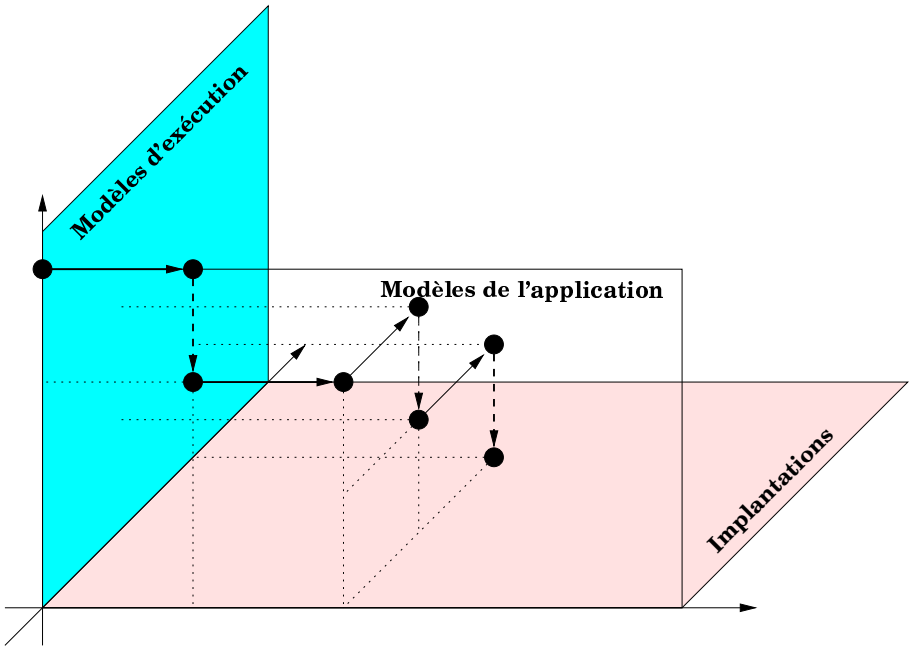


FIG. 6.6 – Principe du co-design

assimilé à un processus léger, qui est un concept d'implantation.

6.5 Des outils intégrés

Les sections précédentes ont énuméré des techniques utiles pour l'amélioration du processus logiciel par une automatisation des transformations des modèles et la vérification des propriétés des raffinages par simulation. Toutes ces approches sont réalisables en pratique dans un outil disposant d'un accès au niveau ¡¡méta¿¿ de la notation UML ainsi qu'au modèle d'exécution de l'*Action Semantics*. Si de plus nous utilisons le caratère réflexif de cette architecture, il est possible de décrire les évolutions du modèle en utilisant la notation UML elle-même.

6.5.1 État de l'art des outils

Cette approche se distingue des outils actuels qui souvent n'appliquent pas les principes de la programmation à objets à leur langage de manipulation. Les langages utilisés sont souvent peu performants (Visual Basic dans le cas de l'outil Rational Rose), non standardisé (langage J de l'atelier Objecteering). Ils nécessitent un investissement supplémentaire de la part du concepteur d'applications et leurs concepts ne sont pas ceux d'UML, ni même parfois ceux de la programmation à objets (dans le cas de Visual Basic). De fait, la manipulation des modèles UML est alourdie par ces décalages conceptuels. Les concepteurs de l'outil Scriptor de la société Sodifrance ont reconnu ce décalage en proposant le langage Java comme langage de manipulation des modèles.

La quantité et la qualité des transformateurs disponibles dans ces outils s'en ressent. Nous nous intéressons aux générateurs de code, qui peuvent être classés selon trois grandes catégories :

- approche struturelle : la génération du code est effectuée à partir de la structure des objets (ie. les diagrammes des classes), et pour un canevas d'applications bien particulier (C++, Ada, un RTOS spécifique) ;
- approche comportementale : la génération du code est effectuée à partir des machines des états complétées par une spécification des actions (du code dans le cas d'UML). C'est le cas de SDL ;
- approche par traduction : les modèles de l'application et de l'architecture sont indépendants l'un de l'autre. L'application est modélisée avec une notation comme OOA (Schlaer et Mellor). Un langage spécifique permet de décrire certains modèles d'architecture, souvent en paramétrant des motifs fournis (par exemple, un modèle de concurrence réalisé par des processus légers, un modèle de la persistence, de communication, etc...). Un moteur de traduction génère le code de l'application en suivant un ensemble de règles de correspondance définies dans l'architecture, ce qui permet une réutilisation plus facile que dans les deux autres approches.

6.5.2 UML : un langage pour les transformations

Nous proposons d'utiliser UML comme langage de description des transformations de modèles, et même de description du processus de développement logiciel. Le concepteur doit bien sûr avoir accès à ces transformations de niveau ¡¡méta¿¿ au sein de l'outil, soit pour les consulter (le modèle UML de ces transformations sert alors de documentation), soit pour les modifier.

Cette démarche apporte toute la connaissance du monde UML à la conception des transformations et leur organisation en paquets, dans le but de construire des bibliothèques de ¡¡composants de transformation¿¿ réutilisables.

Nous associons cette démarche au principe du ¡¡co-design¿¿ en faisant une distinction claire entre les transformations de l'application (par exemple, construction du comportement d'une classe par un diagramme des états à partir d'un ensemble de diagramme de séquence) et les transformation du domaine sémantique d'exécution (par exemple, la manière dont sont réalisés les envois de messages entre classes reliées par une association). Nous explorons l'intérêt de cette approche par quelques exemples.

Raffiner les associations

La Fig. 6.7 présente un exemple de spécialisation du domaine des exécutions. Il s'agit de raffiner les bouts associations suivant l'une des stratégies précisée dans les sous-classes. Nous utilisons le concept d'héritage pour indiquer cette spécialisation.

```
selection_association (ae : AssociationEnd) : AssociationEnd
  -- cette opération 'choisit' la spécialisation à appliquer
  post:
    result =
      if ae. multiplicity .upperBound ≤1
```

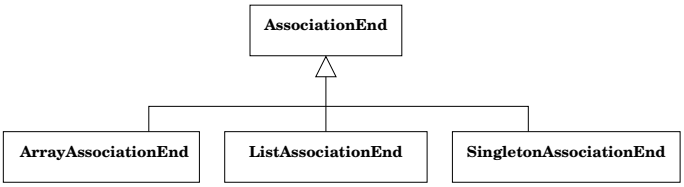


FIG. 6.7 – Spécialisation du domaine sémantique par héritage

```
    -- il s'agit d'une association scalaire
    then result.oclIsKindOf(SingletonAssociationEnd)
else if ae.isordered = true
    -- l'association est ordonnée, utiliser un tableau qui permettra
    -- un accès plus rapide aux éléments par leur indice
    then result.oclIsKindOf(ArrayAssociationEnd)
else
    -- utiliser une liste
    result .oclIsKindOf(ListAssociationEnd)
endif
endif
```

Notons que cette approche par transformation de modèles au sein d'UML permet de libérer les concepteurs de certains schémas d'implantation qui n'ont plus lieu d'être. C'est par exemple le cas du patron de conception ¡¡État¿¿ qui permet de réaliser une machine des états en réifiant le concept d'état. Il nous semble aberrant pour un concepteur de modèles UML d'utiliser ce patron, c'est-à-dire de raffiner un diagramme des états (concept de haut niveau fourni par la notation UML) en un ensemble de classes et de transitions. Cela conduit à un code lourd (nombreuses classes) et relativement difficile à optimiser. Ce patron est souvent associé au patron de conception ¡¡Singleton¿¿ qui permet à chaque état de n'être créé qu'une fois, mais impose une certaine gymnastique pour réaliser l'implantation. Ce patron n'est utile qu'à des fins de maintenance, si la machine à états est sujette à de nombreuses modifications (ou à des modifications à l'exécution). Dans un modèle UML, il est évidemment plus simple de modifier directement le diagramme des états et de générer automatiquement un code optimisé (par exemple sous forme de matrices codant pour chaque état ses successeurs).

De même, le patron ¡¡Visiteur¿¿ a pour seul but d'implanter le mécanisme de liaison dynamique double *double-dispatch* dans les langages qui n'en disposent pas. Au niveau du modèle et de l'outil, il est souhaitable que le passage de la dispersion des fonctionnalités dans les classes ¡¡visitées¿¿ à leur regroupement dans une seule classe appelée ¡¡Visiteur¿¿ (ou vice-versa) puisse être réalisé automatiquement par une transformation de modèle. Nous reviendrons sur cet exemple dans la section 6.6.3.

Accès au processus

L'accès aux niveaux ¡¡méta¿¿ au sein de l'outil offre également des possibilités intéressantes pour la maîtrise du processus de développement lui-même. Il est en effet

possible de modéliser le processus au niveau du méta-modèle, par exemple en exprimant le raffinage d'un cas d'utilisation en collaborations, voire même au niveau du méta-méta-modèle, ce qui permettrait d'exprimer ce qu'est une transformation de modèle UML, comment les choisir ou les appliquer, ou encore définir des métriques de conception.

6.6 L'outil UMLAUT

6.6.1 Architecture

L'outil UMLAUT [128, 55, 62] est un cavenas de conception spécialisé dans la manipulation de modèles UML. Il est conçu autour d'un noyau qui est une implantation du méta-modèle UML dans le langage Eiffel. Ce noyau possède des interfaces qui permettent de manipuler les modèles UML ou de connecter des modules réalisant des fonctionnalités diverses. UMLAUT dispose ainsi de modules pour importer des modèles UML dans les formats les plus répandus (XMI ou le format MDL de l'outil Rational Rose). Cet outil dispose aussi d'un moteur de transformation de modèles UML [54], des générateurs de code (vers Eiffel et Java) à partir d'OCL et de l'AS, d'une connection à l'outil de validation CADP [48], ainsi qu'un prototype d'interprète de spécifications OCL et AS. L'ensemble de ces fonctionnalités est accessible par l'intermédiaire d'une interface graphique d'édition des modèles UML réalisée en Java. L'architecture générale de l'outil est représentée sur la Fig. 6.8.

6.6.2 Contribution

Notre contribution à l'outil UMLAUT a consisté à implanter les fonctionnalités suivantes :

- implantation de l'architecture de méta-modélisation présentée sur la Fig. 2.9. Cette étape est nécessaire pour que le méta-modèle UML puisse être manipulé comme un modèle UML, de manière réflexive. Cette représentation coexiste avec la schéma initial d'UMLAUT qui est un codage en Eiffel du méta-modèle UML. Il est donc actuellement possible de manipuler un méta-modèle comme une collection d'objets Eiffel ou comme une collection d'objets UML. Notons que si le cœur de l'outil était implanté en Java, l'introspection permettrait un accès à la description du méta-modèle mais cette implantation ne serait pas plus satisfaisante que l'implantation en Eiffel en ce qui concerne la facilité à manipuler les concepts d'UML. Avec notre implantation, les notions d'associations ou de paquets sont immédiatement accessibles à l'utilisateur ;
- implantation du méta-modèle OCL et d'un interprète OCL. En utilisant l'architecture réflexive et la séparation des méta-modèles UML et OCL (cf. chapitre 4) cet outil permet à la fois de vérifier les contrats de niveau utilisateur et les contrats de conformité de la notation UML. C'est à notre connaissance le seul outil disposant à l'heure actuelle de cette fonctionnalité OCL sur les deux niveaux ;
- implantation du méta-modèle de l'AS, ainsi que de son moteur d'exécution et de la sémantique des actions. Cette implantation utilise intensivement les outils

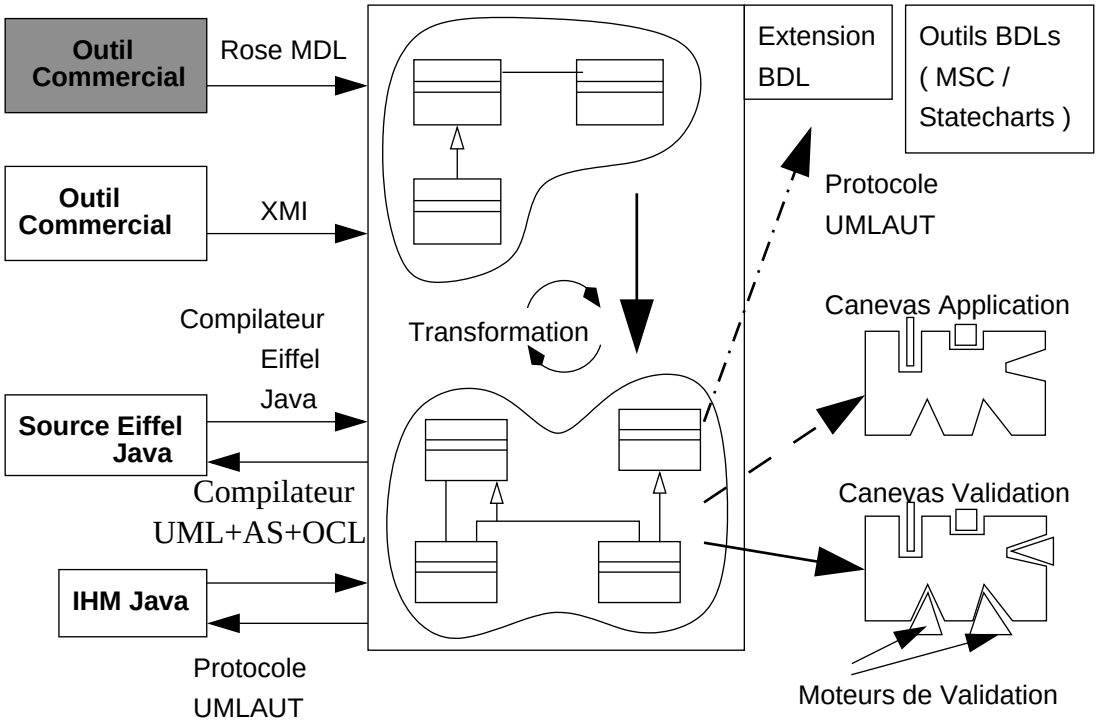


FIG. 6.8 – Architecture de l'outil UMLAUT

OCL. Les premières versions réalisées en Eiffel ont progressivement évoluées vers des implantations en AS, profitant ainsi de l'architecture méta réflexive de l'outil. Un générateur d'OCL vers l'AS et de l'AS vers Eiffel est également disponible pour un sous-ensemble de l'AS ;
– enfin, nous avons également participé au développement de l'interface graphique de l'outil.

L'ensemble de ces contributions représente environ 40000 lignes de code sur un total de 150000.

6.6.3 Les visiteurs dans UMLAUT

À titre illustratif, nous présentons l'intérêt d'utiliser directement l'AS et OCL (plus particulièrement OCL dans notre exemple) pour l'écriture de l'outil. Nous nous intéressons aux Visiteurs d'UMLAUT, qui sont des implantations du patron de conception ¡¡Visiteur¿¿ [43]. Ces entités définissent un parcours de modèle (elles traversent tous les éléments suivant une stratégie définie par un patron ¡¡Itérateur¿¿ [43]) ainsi qu'une opération à appliquer à chaque élément en fonction de son type. Ce patron permet de simuler le mécanisme du *double-dispatch* pour les langages qui n'en disposent pas. La Fig. 6.9 présente ce patron appliqué à l'exemple de la banque que nous avons présenté dans le chapitre 2.

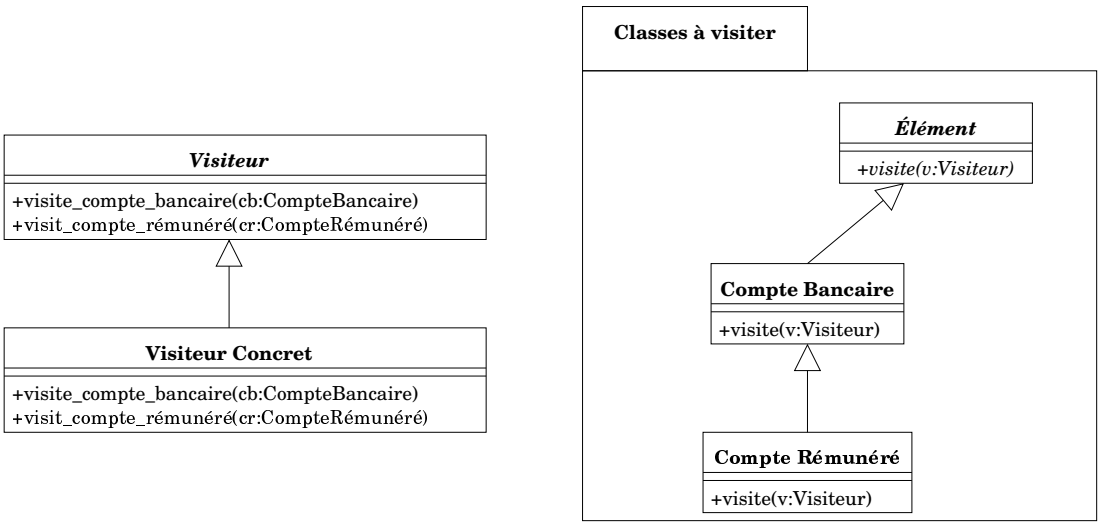


FIG. 6.9 – La patron de conception ¡¡Visiteur¿¿

Ce principe est largement utilisé dans UMLAUT – en particulier pour la génération de code – malgré les défauts de ce patron : violation des principes objets, puisque un comportement spécifique à chaque classe est déplacé dans une autre classe, difficultés liées à la réutilisation des comportements hérités, surtout en cas d'héritage multiple, réentrance difficile à mettre en œuvre... Il s'agit d'un patron que nous pou-

vons qualifier de ¡¡patron d'implantation¿¿ car effectivement lié à l'absence de gestion du *double-dispatch* par le langage cible, en l'occurrence Eiffel. Cependant, l'usage du visiteur est logique si on considère que ce visiteur est l'implantation d'un aspect (par exemple, l'aspect sortie du modèle en XMI). Dès lors, mettre le code dans chaque classe conduirait effectivement à du code spaghetti. Cependant, nous pensons que ce problème est typiquement un problème du langage et doit être pris en charge par l'outil de modélisation, par l'intermédaire d'une transformation de modèles. Ainsi, le pseudo-code suivant définit l'ajout de la méthode `visit_<nom_de_la_classe>` dans chaque classe ¡¡visitable¿¿. Nous utilisons OCL comme un langage de requête sur les modèles UML, selon l'usage demandé par [97] :

```
visitor :Class
p:Package
sc:Set(Class)
sc := p.ownedElement→select(m:ModelElement |
      m.oclIsKindOf(Class))
for c in sc→select(c : Class | not c.isAbstract) {
   Operation m := new Operation();
   m.name.body := 'visit_' + c.name.body
   add_link( visitor . feature , m)
}
```

6.6.4 Vers une auto-génération de l'outil

Nous avons vu qu'il était possible de décrire des applications logicielles à l'aide de transformations de modèles UML. Il est envisageable d'appliquer cette technique à l'outil UMLAUT lui-même. Ce mécanisme – appelé *bootstrap* – a déjà été mis en œuvre dans [54]. Cependant dans cette approche, les transformations sont codées ¡¡en dur¿¿ dans l'outil et écrites en Eiffel. Nous pensons que ce mécanisme pourrait être appliqué en utilisant UML et l'AS pour la description des transformations. De plus, un interprète permet d'exécuter ces transformations de manière plus interactive que l'approche par compilation de programmes Eiffel utilisée dans UMLAUT. Dans le cas d'une auto-génération de l'outil, les spécifications de l'application sont les spécifications de UML et de l'AS.

Notons que cette approche pourrait être couplée à celle de la réalisation d'une application. Dans ce cas, l'outil et l'application évoluent simultanément. Cette approche semble logique si nous reprenons le schéma du ¡¡co-design¿¿ de la Fig. 6.6 : c'est l'outil qui sert à modéliser l'application et possède la connaissance de la sémantique d'UML par le biais de son interprète de modèle, qui partant du modèle d'exécution de l'AS est progressivement raffiné en une implantation. L'outil pourrait en effet évoluer dans le ¡¡plan des modèles d'exécution¿¿ tandis que les transformations qu'il fournit permettent à l'application d'évoluer dans le ¡¡plan des modèles de l'application¿¿.

Chapitre 7

Conclusions et perspectives

7.1 Contributions

Nous résumons dans ce chapitre les travaux présentés dans ce document. L'étude du langage *Action Semantics* nous a conduit à divers travaux autour de la notation OCL et des techniques de la méta-programmation. Nous avons cherché à exploiter l'architecture en couches réflexives d'UML pour construire des outils génériques de manipulation des modèles UML utilisables à tous les niveaux de l'architecture. Ces outils utilisent la notation UML pour décrire les manipulation de modèles, ce qui simplifie la conception de ¡¡composants de transformation¿¿ réutilisables.

7.1.1 Des outils OCL génériques

Les techniques de la conception par contrat sont largement exploitées par la spécification UML. Les travaux présentés dans cette thèse nous ont donc naturellement conduits à la réalisation d'outils exploitant le langage OCL de spécification des contraintes. Ces réalisations s'appuient sur une formalisation du langage OCL, dont la spécification présente certaines lacunes. La solution originale que nous proposons s'appuie sur un méta-modèle UML des expressions OCL indépendant du méta-modèle UML. Cette indépendance permet une grande flexibilité de nos outils. En particulier le même outil permet une vérification de contrats au niveau du modèle (ce diagramme d'objets est-il conforme au diagramme des classes ?), mais également une vérification des contrats au niveau du méta-modèle (ce modèle de conception respecte-t-il la syntaxe imposée par la notation UML ?). Cette dernière vérification est indispensable pour assurer des modèles corrects.

Enfin, la génération de code à partir des contrats permet d'assurer la traçabilité des contrats de la conception jusqu'à l'implantation.

7.1.2 Des outils opérationnels de manipulation des modèles UML

L'un des points faibles de la notation UML est la sémantique dynamique des modèles. L'arrivée du langage d'actions *Action Semantics* fournit un formalisme de

description des comportements alternatif aux diagrammes des états, collaborations, etc. Un effort important a été consacré à la sémantique lors de l'écriture de cette spécification AS. C'est pourquoi nous envisageons d'utiliser ce formalisme comme fondement à la sémantique d'UML, y compris les autres vues dynamiques. Nous avons donc étudié la traduction des autres vues (en particulier des diagrammes de séquence) vers ce formalisme.

Cette compilation des spécifications UML en spécification AS est intéressante pourvu que les spécifications AS puissent être simulées. Cela nous a conduit à étudier les aspects pratiques de la construction de simulateurs UML. Cette tâche n'est pas aisée car la spécification de l'AS, si elle définit précisément les caractéristiques et propriétés d'une machine virtuelle d'exécution des modèles UML, présente un caractère peu pratique pour les concepteurs d'ateliers de génie logiciel. La machine virtuelle UML de l'AS étant elle-même un modèle UML, il nous a paru naturel de générer des implantations de cette machine virtuelle en suivant un processus de développement UML. Cette implantation est donc vue comme un raffinage de la spécification initiale de l'AS.

Nous sommes donc confrontés au problème du raffinage de modèles UML. Plutôt que faire ces raffinages à la main, un travail fastidieux et source d'erreur, nous avons cherché à automatiser ce processus, en nous appuyant sur les techniques de transformation de modèles ou méta-programmation.

Nous nous sommes aidés des caractéristiques de l'architecture en couches d'UML et du fait que son méta-modèle est un modèle UML : nous manipulons des modèles UML grâce à d'autres modèles UML. Les transformations de modèles sont donc de véritables modèles UML. Cela permet de disposer des possibilités de gestion du développement logiciel offertes par cette notation. Nous pouvons en particulier imaginer de raffiner des transformations, hériter ou redéfinir d'autres transformations, les faire collaborer, construire des bibliothèques de transformations classées dans des paquets ou profils UML, etc...

Dans notre approche, les objets, les modèles ou les méta-modèles se manipulent indifféremment, ce qui permet une exploitation des mêmes développements à tous les niveaux : un simulateur de modèle UML peut-être utilisé pour l'éxécution d'une transformation de modèles.

7.1.3 Une ébauche de processus dirigé par les outils

L'automatisation du processus de développement logiciel apparaît de plus en plus souhaitable avec la montée en complexité des problèmes modélisés. Cette automatisation nécessite des outils puissants de transformation de modèles. Ces transformations se divisent en deux catégories : les transformations qui raffine le modèle de l'application et les transformations liés à l'implantation de cette application. Il est évident que ces dernières monopolisent l'attention du concepteur sur des activités annexes et sans intérêt pour la conception de l'application.

La classification devient évidente si nous nous intéressons aux concepts manipulés par chaque catégorie de transformations :

– les raffinages du modèle de l'application manipulent les concepts définis dans la

spécification initiale de l'application, c'est-à-dire des cas d'utilisation, des collaborations entre objets du domaine de l'application ;
– les autres transformations raffinent les exécutions des modèles UML, c'est-à-dire la machine virtuelle modélisée par le modèle d'exécution de l'AS. Ces transformations agissent sur le codage d'une association, les caractéristiques des communications entre objets, etc.

7.2 Perspectives

7.2.1 Expressivité accrue du langage OCL

Le langage OCL se base sur la théorie des ensembles et reste un langage simple à utiliser qui semble bien accepté par la communauté des concepteurs UML. Son utilisation est fortement recommandé par la méthode Catalysis [38]. Il constitue un premier pas vers l'utilisation de langages formels plus puissants pour l'expression des propriétés des spécifications UML. En effet, OCL est malheureusement un langage dont l'expressivité est limitée [33, 116, 79, 124] et dont la fonction principale est de définir l'état du système à un instant donné. Des propriétés plus complexes sur les comportements exigent l'utilisation d'une logique temporelle. Divers travaux [119, 37, 32, 104] proposent d'ailleurs des extensions temporelles au langage OCL. Cette logique temporelle serait utile au niveau du modèle, mais également au niveau du méta-modèle où elle pourrait servir à une meilleure spécification des transformations, en particulier pour la spécification des aspects comportementaux des patrons de conception [49].

Néanmoins, nous avons vu qu'OCL dans sa définition actuelle pouvait être utilisé pour exprimer des propriétés de logique temporelle en utilisant le modèle d'exécution.

7.2.2 Des outils encore plus évolués

Simuler des spécifications abstraites

L'utilisation de plus en plus répandue du langage OCL et la nécessité de simuler ¡¡au plus tôt¿¿ une spécification afin d'éviter la propagation coûteuse d'erreurs de conception montre qu'il est souhaitable de disposer de simulateurs de modèles capables de prendre en charge des modèles ou les comportements ne sont pas décrits sous une forme opérationnelle (avec l'AS ou des diagrammes des états) mais sous la forme d'invariants, de préconditions et de postconditions.

Nous avons déjà évoqué cette idée dans la section 3.2.3 en proposant un mécanisme similaire au chaînage avant des moteurs d'inférence.

Ceci éviterait le recours à des spécifications opérationnelles et donc supprimerait le risque de sur-spécification.

Nous pensons que les systèmes de résolution de contraintes pourraient apporter une aide précieuse aux phases préliminaires de la conception ou seules ces descriptions non opérationnelles sont disponibles. Il est également envisageable d'utiliser des systèmes de preuve à partir de ces spécifications OCL.

Rôle des outils

Enfin, nous pensons que le rôle des outils est amené à changer. Nous pensons qu'outil et application doivent évoluer simulatanément pour réaliser une implantation. En effet, c'est l'outil qui possède la connaissance de la sémantique d'UML alors que c'est l'application qui possède la connaissance du domaine modélisé. Nous pensons que l'implantation est le résultat du raffinage et de la composition de ces deux entités.

Chapitre 8

Annexes
Contraintes OCL corrigées de UML 1.3

---

context ActionState
-- [1] An action state has a non-empty entry action.
inv: self.entry→**size** > 0

context ActionState
-- [2] An action state does not have an internal transition, exit action, or a do activity.
inv: self.internalTransition→**size** = 0 **and** self.exit→**size** = 0 **and**
self.doActivity→**size** = 0

context ActionState
-- [3] Transitions originating from an action state have no trigger event.
inv: self.outgoing→**forAll**(t | t.trigger→**size** = 0)

context ActivityGraph
-- [1] An ActivityGraph specifies the dynamics of (i) a Package, or (ii) a Classifier (including UseCase), or (iii) a BehavioralFeature.
inv: (self.**context**.**oclIsTypeOf**(Package) **xor**
self.**context**.**oclIsKindOf**(Classifier) **xor**
self.**context**.**oclIsKindOf**(BehavioralFeature))

context Actor
-- [1] Actors can only have Associations to UseCases, Subsystems, and Classes and these Associations are binary.
inv: self.associations→**forAll**(a |
a.connection→**size** = 2 **and**
a.allConnections→**exists**(r | r.type.**oclIsKindOf**(Actor)) **and**
a.allConnections→**exists**(r |
r.type.**oclIsKindOf**(UseCase) **or**
r.type.**oclIsKindOf**(Subsystem) **or**
r.type.**oclIsKindOf**(Class)))

context Actor
-- [2] Actors cannot contain any Classifiers.
inv: self.contents→**isEmpty**

context Association
-- the AssociationEnds must have a unique name within the Association.
inv: self.allConnections→**forAll**(r1, r2 | r1.**name** = r2.**name implies** r1= r2)

context Association
-- at most one AssociationEnd may be an aggregation or composition.
inv: self.allConnections→**select**(ae | ae.aggregation <>#none)→**size** ≤1

context Association
-- if an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition.
inv: self.allConnections→**size** ≥ 3 **implies**
self.allConnections→**forAll**(ae | ae.aggregation = #none)

context Association
-- the connected Classifiers of the AssociationEnds should be included in the Namespace of the Association.
inv: self.allConnections→**forAll**(r |
self.namespace.allContents→**includes**(r.type))

context Association::allConnections():**Set**(AssociationEnd)
-- the operation allConnections results in the set of all AssociationEnds of the Association.
asl : self.connection→**asSet**
post: result = self.connection→**asSet**

context AssociationClass
-- the names of the AssociationEnds and the StructuralFeatures do not overlap.
inv: self.allConnections→**forAll**(ar |
self.allFeatures→**forAll**(f |
f.**oclIsKindOf**(structuralFeature) **implies** ar.**name** <>f.**name**))

context AssociationClass
-- an AssociationClass cannot be defined between itself and something else.
inv: self.allConnections→**forAll**(ar | ar.type <> self)

context AssociationClass::allConnections():**Set**(AssociationEnd)
-- the operation allConnections results in the set of all AssociationEnds of the AssociationClass, including all connections defined by its parent (transitive closure).
asl : self.connection→**union**(self.parent→**select**(s | s.**oclIsKindOf**(Association))
→**collect**(a:Association|a.allConnections))→**asSet**
post: result = self.connection→**union**(self.parent→**select**(s | s.**oclIsKindOf**(Association))
→**collect**(a |a.**oclAsType**(Association).allConnections))→**asSet**

context AssociationEnd
-- the Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end.
inv: (self.type.**oclIsKindOf**(Interface) **or**
self.type.**oclIsKindOf**(DataType)) **implies**
self.association.connection→**select**(ae |
ae <> self)→**forAll**(ae | ae.isNavigable = **false**)

context AssociationEnd
-- an Instance may not belong by composition to more than one composite Instance.
inv: self.aggregation = #composite **implies** self.multiplicity.**max** < 1

context AssociationEndRole
-- [1] The type of the ClassifierRole must conform to the type of the base AssociationEnd.
inv: self.type.**oclAsType**(ClassifierRole).base = self.base.type
or
self.type.**oclAsType**(ClassifierRole).base.allParents→**includes** (self.base.type)

context AssociationEndRole
-- [2] The type must be a kind of ClassifierRole.
inv: self.type.**oclIsKindOf** (ClassifierRole)

context AssociationEndRole
-- [3] The qualifiers used in the AssociationEndRole must be a subset of those in the base AssociationEnd.
inv: self.base.qualifier →**includesAll** (self.availableQualifier)

context AssociationEndRole
-- [4] In a collaboration an association may only be used for traversal if it is allowed by the base association.
inv: self.isNavigable **implies** self.base.isNavigable

context AssociationRole
-- [1] The AssociationEndRoles must conform to the AssociationEnds of the base Association.
-- [FP] connection association is not ordered in the metamodel!
inv: **Sequence**{ 1..(self.connection→**size**) }→**forAll** (index |
self.connection→**at**(index).base =
self.base.connection→**at**(index))

context AssociationRole
-- [2] The endpoints must be a kind of AssociationEndRoles.
inv: self.connection→**forAll**(r | r.**oclIsKindOf** (AssociationEndRole))

context AttributeLink
-- The type of the Instance must match the type of the Attribute.
inv: self.value.classifier →**union** (
self.value.classifier.allParents)→**includes** (
self.attribute.type)
context BehavioralFeature
-- all Parameters should have a unique name.
inv: self.parameter→**forAll**(p1,p2 | p1.**name** = p2.**name** **implies** p1 = p2)

context BehavioralFeature
-- the type of the Parameters should be included in the Namespace of the Classifier.
inv: self.parameter→**forAll**(p |
self.owner.namespace.allContents→**includes**(p.type))

context BehavioralFeature::hasSameSignature(b:BehavioralFeature):**Boolean**
-- the operation hasSameSignature checks if the argument has the same signature as the instance itself
asl : (self.**name** = b.**name**) **and** (self.parameter→**size** = b.parameter→**size**)
and Sequence(**Integer**){1..(self.parameter→**size**)}→**forAll**(index:**Integer**|
b.parameter→**at**(index).type = self.parameter→**at**(index).type
and b.parameter→**at**(index).kind = self.parameter→**at**(index).kind)
post: result = (self.**name** = b.**name**) **and** (self.parameter→**size** = b.parameter→**size**)
and Sequence(**Integer**){1..(self.parameter→**size**)}→**forAll**(index:**Integer**|
b.parameter→**at**(index).type = self.parameter→**at**(index).type
and b.parameter→**at**(index).kind = self.parameter→**at**(index).kind)

context BehavioralFeature::matchesSignature(b:BehavioralFeature):**Boolean**
-- the operation matchesSignature checks if the argument has a signature that would clash with the signature of the instance itself (and therefore must be unique). Mismatches in kind or

any differences in return parameters do not cause a mismatch.
asl : (self .**name** = b.**name**) **and** (self.parameter→**size** = b.parameter→**size**)
and Sequence(**Integer**){1..(self.parameter→**size**)}→**forAll**(index:**Integer** |
b.parameter→**at**(index).type = self.parameter→**at**(index).type
or (b.parameter→**at**(index).kind = #return
and self. parameter→**at**(index).kind = #return))
post: result = (self .**name** = b.**name**) **and** (self.parameter→**size** = b.parameter→**size**)
and Sequence(**Integer**){1..(self.parameter→**size**)}→**forAll**(index:**Integer** |
b.parameter→**at**(index).type = self.parameter→**at**(index).type
or (b.parameter→**at**(index).kind = #return
and self. parameter→**at**(index).kind = #return))

context Binding
-- the argument ModelElement must conform to the parameter ModelElement in a Binding. In an Instantiation it must be of the same kind.
inv: **true**

context Binding
-- a binding has one client and one supplier.
inv: (self . client →**size** = 1) **and** (self. supplier→**size** = 1)

context Binding
-- a ModelElement may participate in at most one Binding as a client.
inv: Binding.**allInstances**→**forAll**(b1, b2 |
(b1 <>b2) **implies** (b1.client <>b2. client))

context CallAction
-- [1] The number of arguments be the same as the number of the Operation.
inv: self .actualArgument→**size** = self.operation.parameter→**size**

context CallState
-- [1] The entry action of a call state is a single call action.
inv: self .entry→**size** = 1 **and** self.entry .**oclIsKindOf**(CallAction)

context Class
-- if a Class is concrete, all the Operations of the Class should have a realizing Method in the full descriptor.
inv: **not** self .isAbstract **implies** self .allOperations→**forAll**(op|
self .allMethods→**exists**(m | m.specification = op))

context Class
-- a Class can only contain Classes, Associations, Generalizations, UseCases, Constraints, Dependencies, Collaborations, DataTypes, and Interfaces as a Namespace.
inv: self .allContents→**forAll**(c |
c.**oclIsKindOf**(Class) **or**
c.**oclIsKindOf**(Association) **or**
c.**oclIsKindOf**(Generalization) **or**
c.**oclIsKindOf**(UseCase) **or**
c.**oclIsKindOf**(Constraint) **or**
c.**oclIsKindOf**(Dependency) **or**

```
                c.oclIsKindOf(Collaboration) or
                c.oclIsKindOf(DataType) or
                c.oclIsKindOf(Interface))

context Classifier
   -- no BehavioralFeature of the same kind may match the same signature in a Classifier.
   inv: self.feature→forAll(f,g |
   (
     (
       ( f.oclIsKindOf(Operation) and g.oclIsKindOf(Operation)) or
       ( f.oclIsKindOf(Method) and g.oclIsKindOf(Method)) or
       ( f.oclIsKindOf(Reception) and g.oclIsKindOf(Reception))
       ) and
       f.oclAsType(BehavioralFeature).matchesSignature(g.oclAsType(BehavioralFeature))
       )
       implies f = g)

context Classifier
   -- no Attributes may have the same name within a Classifier.
   inv: self.feature→select(a | a.oclIsKindOf(Attribute))→forAll(p,q |
                p.name = q.name implies p = q)

context Classifier
   -- no opposite AssociationEnds may have the same name within a Classifier.
   inv: self.oppositeAssociationEnds→forAll(p, q | p.name = q.name implies p = q)

context Classifier
   -- the name of an Attribute may not be the same as the name of an opposite AssociationEnd
or a ModelElement contained in the Classifier.
   inv: self.feature→select(a | a.oclIsKindOf(Attribute))→forAll(a |
                not self.allOppositeAssociationEnds→union(self.allContents)
               →collect(q | q.name)→includes(a.name))

context Classifier
   -- the name of an opposite AssociationEnd may not be the same as the name of an Attribute
or a ModelElement contained in the Classifier.
   inv: self.oppositeAssociationEnds→forAll(o |
            not self.allAttributes→union(self.allContents)
               →collect(q| q.name)→includes(o.name))

context Classifier
   -- for each Operation in a specification realized by the Classifier, the Classifier must have a
matching Operation.
   inv: self.specification.allOperations→forAll(interOp |
                        self.allOperations→exists(op |
                        op.matchesSignature(interOp)))

context Classifier
   -- all of the Generalizations in the range of a Powertype have the same discriminator.
   inv: self.powertypeRange→forAll(g1,g2|g1.discriminator=g2.discriminator)
```

context Classifier
-- Discriminator names must be distinct from attribute names and opposite AssociationEnd names.
inv: self . allDiscriminators →**intersection**(self. allAttributes .**name**
→**union**(self.allOppositeAssociationEnds.**name**))→**isEmpty**

context Classifier :: allFeatures (): **Set**(Feature)
-- the operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features.
asl : self . feature→**union**(self.parent→**collect**(c |
c.**oclAsType**(Classifier).allFeatures))→**asSet**
post: result = self . feature→**union**(self.parent→**collect**(c |
c.**oclAsType**(Classifier).allFeatures))→**asSet**

context Classifier :: allOperations (): **Set**(Operation)
-- the operation allOperations results in a Set containing all Operations of the Classifier itself and all its inherited Operations.
asl : self . allFeatures→**select**(f| f.**oclIsKindOf**(Operation))
post: result = self . allFeatures→**select**(f| f.**oclIsKindOf**(Operation))

context Classifier :: allMethods():**Set**(Method)
-- the operation allMethods results in a Set containing all Methods of the Classifier itself and all its inherited Methods.
asl : self . allFeatures→**select**(f | f.**oclIsKindOf**(Method))
post: result = self . allFeatures→**select**(f | f.**oclIsKindOf**(Method))

context Classifier :: allAttributes (): **Set**(Attribute)
-- the operation allAttributes results in a Set containing all Attributes of the Classifier itself and all its inherited Attributes.
asl : self . allFeatures→**select**(f | f.**oclIsKindOf**(Attribute))
post: result = self . allFeatures→**select**(f | f.**oclIsKindOf**(Attribute))

context Classifier :: associations (): **Set**(Association)
-- the operations associations results in a Set containing all Associations of the Classifier itself
asl : self .associationEnd.association
post: result = self .associationEnd.association

context Classifier :: allAssociations :**Set**(Association)
-- the operation allAssociations results in a Set containing all Associations of the Classifier itself and all its inherited Associations.
asl : self . associations →**union**(self.parent→**collect**(g |
g.**oclAsType**(Classifier).allAssociations))→**asSet**
post: result = self . associations →**union**(self.parent→**collect**(g |
g.**oclAsType**(Classifier).allAssociations))→**asSet**

context Classifier :: oppositeAssociationEnds:**Set**(AssociationEnd)
-- results in a set of all AssociationEnds that are opposite to the Classifier
asl : self . associations →**select**(a|a.connection→**select**(ae |
ae.type = self)→**size** = 1)→**collect**(a |
a.connection→**select**(ae | ae.type <> self))→**union**

```
        (
        self . associations →select(a | a.connection→select(ae |
        ae.type = self ).size <>1)→collect(a |
        a.connection))→asSet
    post: result = self . associations →select(a|a.connection→select(ae |
        ae.type = self)→size = 1)→collect(a |
        a.connection→select(ae | ae.type <> self ))→union
        (
        self . associations →select(a | a.connection→select(ae |
        ae.type = self ).size <>1)→collect(a |
        a.connection))→asSet

context Classifier :: allOppositeAssociationEnds:Set(AssociationEnd)
    -- the operation allOppositeAssociationEnds results in a set of all AssociationEnds, including
the inherited ones, that are opposite to the Classifier.
    asl : self .oppositeAssociationEnds→union(self.parent→collect(g |
        g.oclAsType(Classifier).allOppositeAssociationEnds))→asSet
    post: result = self .oppositeAssociationEnds→union(self.parent→collect(g |
        g.oclAsType(Classifier).allOppositeAssociationEnds))→asSet

context Classifier :: specification :Set( Classifier )
    -- the operation specification yields the set of Classifiers that the current Classifier realizes.
    asl : self .clientDependency→select(d |
        d.oclIsKindOf(Abstraction)
        and d.stereotype.name = 'realization').supplier
        →collect(g | g.oclAsType(Classifier))→asSet
    post: result = self .clientDependency→select(d |
        d.oclIsKindOf(Abstraction)
        and d.stereotype.name = 'realization').supplier
        →collect(g | g.oclAsType(Classifier))→asSet

context Classifier :: allContents:Set(ModelElement)
    -- the operation allContents returns a Set containing all ModelElements contained in the
Classifier together with the contents inherited from its parents.
    asl : self .contents→union(self.parent→collect(g |
        g.oclAsType(Classifier).allContents→select(e |
            e.elementOwnership.visibility = #public or
            e.elementOwnership.visibility = #protected)))
    post: result = self .contents→union(self.parent→collect(g |
        g.oclAsType(Classifier).allContents→select(e |
        e.elementOwnership.visibility = #public or
        e.elementOwnership.visibility = #protected)))

context Classifier :: allDiscriminators :Set(Name)
    -- the operation allDiscriminators results in a Set containing all Discriminators of the Ge-
neralizations from which the Classifier is descended itself and all its inherited Features.
    asl : self . generalization . discriminator→union(self.parent→collect(g |
        g.oclAsType(Classifier).allDiscriminators))
    post: result = self . generalization . discriminator→union(self.parent→collect(g |
        g.oclAsType(Classifier).allDiscriminators))
```

context ClassifierRole
-- [1] The AssociationRoles connected to the ClassifierRole must match a subset of the Associations connected to the base Classifiers.
inv: self . allAssociations →**forAll**(ar |
self . base. allAssociations →**exists** (a | ar = a))

context ClassifierRole
-- [2] The Features and contents of the ClassifierRole must be subsets of those of the base Classifiers.
inv: self . base. allFeatures →**includesAll** (self. allAvailableFeatures)
and
self . base.allContents→**includesAll** (self.allAvailableContents)

context ClassifierRole
-- [3] A ClassifierRole does not have any Features of its own.
inv: self . allFeatures →**isEmpty**

context ClassifierRole :: allAvailableFeatures (): **Set**(Feature)
-- [1] The operation allAvailableFeatures results in the set of all Features contained in the ClassifierRole together with those contained in the parents.
post: result = self . availableFeature→**union**(self.parent→**collect**(cr |
cr . **oclAsType**(ClassifierRole).allAvailableFeatures))

context ClassifierRole :: allAvailableContents : **Set**(ModelElement)
-- [2] The operation allAvailableContents results in the set of all ModelElements contained in the ClassifierRole together with those contained in the parents.
post: result = self . availableContents→**union**(self.parent→**collect**(cr |
cr . **oclAsType**(ClassifierRole).allAvailableContents))

context Collaboration
-- [1] All Classifiers and Associations of the ClassifierRoles and AssociationRoles in the Collaboration must be included in the namespace owning the Collaboration.
inv: self . allContents→**forAll** (e |
(e . **oclIsKindOf** (ClassifierRole) **implies**
self . namespace.allContents→**includes** (
e . **oclAsType**(ClassifierRole).base))
and
(e . **oclIsKindOf** (AssociationRole) **implies**
self . namespace.allContents→**includes** (
e . **oclAsType**(AssociationRole).base)))

context Collaboration
-- [2] All the constraining ModelElements must be included in the namespace owning the Collaboration.
inv: self . constrainingElement→**forAll** (ce |
self . namespace.allContents→**includes** (ce))

context Collaboration
-- [3] If a ClassifierRole or an AssociationRole does not have a name then it should be the only one with a particular base.

```
inv: self.allContents→forAll ( p |
       (p.oclIsKindOf (ClassifierRole) implies
       p.name = '' implies
        self.allContents→forAll ( q |
       q.oclIsKindOf(ClassifierRole) implies
       (p.oclAsType(ClassifierRole).base =
       q.oclAsType(ClassifierRole).base implies
       p = q) ) )
          and
       (p.oclIsKindOf (AssociationRole) implies
       p.name = '' implies
        self.allContents→forAll ( q |
       q.oclIsKindOf(AssociationRole) implies
       (p.oclAsType(AssociationRole).base =
       q.oclAsType(AssociationRole).base implies
       p = q) ) )
       )
```

```
context Collaboration
-- [4] A Collaboration may only contain ClassifierRoles and AssociationRoles, and the Generalizations and the Constraints between them.
inv: self.allContents→forAll ( p |
   p.oclIsKindOf (ClassifierRole) or
   p.oclIsKindOf (AssociationRole) or
   p.oclIsKindOf (Generalization) or
   p.oclIsKindOf (Constraint) )
```

```
context Collaboration
-- [5] A role with the same name as one of the roles in a parent of the Collaboration must be a child (a specialization) of that role.
inv: self.contents→forAll ( c |
                 self.parent→collect(g : GeneralizableElement |
                     g.oclAsType(Collaboration).allContents)→forAll ( p |
                     c.name = p.name implies c.allParents→includes(p) ))
```

```
context Collaboration::allContents:Set(ModelElement)
-- [1] The operation allContents results in the set of all ModelElements contained in the Collaboration together with those contained in the parents except those that have been specialized.
post: result = self.contents→union (
                self.parent→collect(g |
                 g.oclAsType(Collaboration).allContents)→reject ( e |
                     self.contents.name→includes(e.name) ))
```

```
context Component
   -- a Component may only contain other Components
   inv: self.allContents→forAll(c | c.oclIsKindOf(Component))
```

```
context Component
   -- a Component may only implement DataTypes, Interfaces, Classes, Associations, Dependencies, Constraints, Signals, DataValues and Objects.
```

```
inv: self.allResidentElements→forAll(re |
                re.oclIsKindOf(DataType) or
                re.oclIsKindOf(Interface) or
                re.oclIsKindOf(Class) or
                re.oclIsKindOf(Association) or
                re.oclIsKindOf(Dependency) or
                re.oclIsKindOf(Constraint) or
                re.oclIsKindOf(Signal) or
                re.oclIsKindOf(DataValue) or
                re.oclIsKindOf(Object))

context Component::allResidentElements:Set(ModelElement)
  -- the operation allResidentElements results in a set containing all ModelElements resident
in a Component or one of its ancestors.
  asl : self.resident→union(self.parent→collect(g |
    g.oclAsType(Component).allResidentElements)→select(re |
      re.elementResidence.visibility = #public or
      re.elementResidence.visibility = #protected))
  post: result = self.resident→union(self.parent→collect(g |
    g.oclAsType(Component).allResidentElements)→select(re |
      re.elementResidence.visibility = #public or
      re.elementResidence.visibility = #protected))

context Component::allVisibleElements:Set(ModelElement)
  -- the operation allVisibleElements results in a set containing all ModelElements visible
outside the Component.
  asl : self.allContents→select(e |
        e.elementOwnership.visibility = #public)→union(
                self.allResidentElements→select(re |
                re.elementResidence.visibility = #public))
  post: result = self.allContents→select(e |
        e.elementOwnership.visibility = #public)→union(
                self.allResidentElements→select(re |
                re.elementResidence.visibility = #public))

context ComponentInstance
-- [1] A ComponentInstance originates from exactly one Component.
inv: self.classifier →size = 1
     and self.classifier .oclIsKindOf (Component)

context CompositeState
-- [1] A composite state can have at most one initial vertex.
inv: self.subvertex→select (v | v.oclIsKindOf(Pseudostate))→
  select(p | p.oclAsType(Pseudostate).kind = #initial)→size ≤1

context CompositeState
-- [2] A composite state can have at most one deep history vertex.
inv: self.subvertex→select (v | v.oclIsKindOf(Pseudostate))→
    select(p | p.oclAsType(Pseudostate).kind = #deepHistory)→size ≤1
```

context CompositeState
-- [3] A composite state can have at most one shallow history vertex.
inv: self.subvertex→**select**(v | v.**oclIsKindOf**(Pseudostate))→
select(p | p.**oclAsType**(Pseudostate).kind = #shallowHistory)→**size** ≤1

context CompositeState
-- [4] There have to be at least two composite substates in a concurrent composite state.
inv: (self.isConcurrent) **implies**
(self.subvertex→**select**
(v | v.**oclIsKindOf**(CompositeState))→**size** ≥2)

context CompositeState
-- [5] A concurrent state can only have composite states as substates.
inv: (self.isConcurrent) **implies**
self.subvertex→**forAll**(s | s.**oclIsKindOf**(CompositeState))

context CompositeState
-- [6] The substates of a composite state are part of only that composite state.
inv: self.subvertex→**forAll**(s | (s.container→**size** = 1) **and**
(s.container = self))

context Constraint
-- a Constraint cannot be applied to itself.
inv: **not** self.constrainedElement→**includes**(self)

context Constraint
-- [1] A Constraint attached to a Stereotype must not conflict with Constraints on any inherited Stereotype, or associated with the baseClass.
-- cannot be specified with OCL
inv: **true**

context Constraint
-- [2] A Constraint attached to a stereotyped ModelElement must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the baseClass) of the ModelElement.
-- cannot be specified with OCL
inv: **true**

context Constraint
-- [3] A Constraint attached to a Stereotype will apply to all ModelElements classified by that Stereotype and must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the baseClass) of the ModelElement.
-- cannot be specified with OCL
inv: **true**

context CreateAction
-- [1] A CreateAction does not have a target expression.
inv: self.target→**isEmpty**

context DataType
-- a DataType can only contain Operations, which all must be queries.

inv: self . allFeatures→**forAll**(f |
f . **oclIsKindOf**(Operation) **and**
f . **oclAsType**(Operation).**isQuery**)

context DataType
-- a DataType cannot contain any other ModelElements.
inv: self . allContents→**isEmpty**

context DataValue
-- [1] A DataValue originates from exactly one Classifier, which is a DataType.
inv: (self . classifier →**size** = 1)
and self. classifier .**oclIsKindOf**(DataType)

context DataValue
-- [2] A DataValue has no AttributeLinks.
inv: self . slot →**isEmpty**

context DestroyAction
-- [1] A DestroyAction should not have arguments.
inv: self . actualArgument→**size** = 0

context Extend
-- [1] The referenced ExtensionPoints must be included in set of ExtensionPoint in the target UseCase.
inv: self . base.allExtensionPoints → **includesAll** (self. location)

context ExtensionPoint
-- [1] The name must not be the empty string.
inv: **not** (self . **name** = ’’)

context FinalState
-- [1] A final state cannot have any outgoing transitions.
inv: self . outgoing→**size** = 0

context GeneralizableElement
-- a root cannot have any Generalizations.
inv: self . isRoot **implies** self . generalization →**isEmpty**

context GeneralizableElement
-- no GeneralizableElement can have a parent Generalization to an element which is a Leaf.
inv: self . parent→**forAll**(s | **not** s. isLeaf)

context GeneralizableElement
-- circular inheritance is not allowed.
inv: **not** self . allParents→**includes**(self)

context GeneralizableElement
-- the parent must be included in the Namespace of the GeneralizableElement.
inv: self . generalization →**forAll**(g |
self . namespace.allContents→**includes**(g.parent))

context GeneralizableElement::parent:**Set**(GeneralizableElement)
-- the operation parent returns a set containing all direct parents
asl : self . generalization . parent
post: result = self . generalization . parent

context GeneralizableElement::allParents:**Set**(GeneralizableElement)
-- the operation allParents returns a set containing all the GeneralizableElements inherited by this GeneralizableElement (the transitive closure), excluding the GeneralizableElement itself.
asl : self . parent→**union**(self.parent.allParents→**asSet**)
post: result = self . parent→**union**(self.parent.allParents→**asSet**)

context Generalization
-- a GeneralizableElement may only be a child of a GeneralizableElement of the same kind.
inv: **true**

context Guard
-- [1] A guard should not have side effects.
inv: self . transition →stateMachine→**notEmpty implies**
(self . transition .stateMachine→**context** =
self . transition .stateMachine→**context@pre**)

context Instance
-- [1] The AttributeLinks match the declarations in the Classifiers.
inv: self . slot →**forAll** (al |
self . classifier →**exists** (c |
c. allAttributes →**includes** (al. attribute)))

context Instance
-- [2] The Links matches the declarations in the Classifiers.
inv: self . allLinks→**forAll** (l |
self . classifier →**exists** (c |
c. allAssociations →**includes** (l. association)))

context Instance
-- [3] If two Operations have the same signature they must be the same.
inv: self . classifier →**forAll** (c1 , c2 |
c1.allOperations→**forAll** (op1 |
c2.allOperations→**forAll** (op2 |
op1.hasSameSignature (op2) **implies** op1 = op2)))

context Instance
-- [4] There are no name conflicts between the AttributeLinks and opposite LinkEnds.
inv: self . slot →**forAll**(al |
not self . allOppositeLinkEnds→**exists**(le | le . **name** = al.**name**))
and self. allOppositeLinkEnds→**forAll**(le |
not self . slot →**exists**(al | le . **name** = al.**name**))

context Instance

-- [5] For each Association in which an Instance is involved, the number of opposite LinkEnds must match the multiplicity of the AssociationEnd.
inv: self . classifier .allOppositeAssociationEnds→**forAll** (ae |
ae. multiplicity .multiplicityRange→**exists** (mr |
self .selectedLinkEnds (ae)→**size** ≥ mr.lower **and**
(mr.upper = `'unlimited'` **or**
(mr.upper <>`'unlimited'` **and**
self .selectedLinkEnds (ae)→**size** ≤
mr.upper.**oclAsType** (**Integer**)))))

context Instance
-- [6] The number of associated AttributeLinks must match the multiplicity of the Attribute.
inv: self . classifier . allAttributes →**forAll** (a |
a. multiplicity .multiplicityRange→**exists** (mr |
self .selectedAttributeLinks(a)→**size** ≥ mr.lower **and**
(mr.upper = `'unlimited'` **or**
(mr.upper <>`'unlimited'` **and**
self .selectedLinkEnds (a)→**size** ≤
mr.upper.**oclAsType** (**Integer**)))))

context Instance::allLinks:**Set**(Link)
-- [1] The operation allLinks results in a set containing all Links of the Instance itself.
asl : self .linkEnd.link
post: result = self .linkEnd.link

context Instance::allOppositeLinkEnds:**Set**(LinkEnd)
-- [2] The operation allOppositeLinkEnds results in a set containing all LinkEnds of Links connected to the Instance with another LinkEnd.
post: result = self . allLinks . connection→**select** (le |
le . instance <> self)

context Instance::selectedLinkEnds(ae:AssociationEnd):**Set**(LinkEnd)
-- [3] The operation selectedLinkEnds results in a set containing all opposite LinkEnds corresponding to a given AssociationEnd.
post: result = self . allOppositeLinkEnds→**select** (le |
le . associationEnd = ae)

context Instance::selectedAttributeLinks(ae:Attribute):**Set**(AttributeLink)
-- [4] The operation selectedAttributeLinks results in a set containing all AttributeLinks corresponding to a given Attribute.
post: result = self . slot →**select** (s |
s . attribute = ae)
context Interaction
--[1] All Signals being sent must be included in the namespace owning the Collaboration in which the Interaction is defined.
inv: self .message→**forAll** (m |
m.action.**oclIsKindOf**(SendAction) **implies**
self .**context**.namespace.allContents→**includes** (
m.action→**oclAsType** (SendAction).signal))

context Interface

```
    -- an Interface can only contain Operations.
    inv: self.allFeatures→forAll(f |
                f.oclIsKindOf(Operation) or f.oclIsKindOf(Reception))

context Interface
    -- an Interface cannot contain any ModelElements.
    inv: self.allContents→isEmpty

context Interface
    -- all Features defined in Interface are public.
    inv: self.allFeatures→forAll(f | f.visibility = #public)

context Link
-- [1] The set of LinkEnds must match the set of AssociationEnds of the Association.
inv: Sequence {1..self.connection→size}→forAll ( i |
        self.connection→at (i).associationEnd =
        self.association.connection→at (i) )

context Link
-- [2] There are not two Links of the same Association which connects the same set of Instances
in the same way.
inv: self.association.link→forAll ( l |
       Sequence {1..self.connection→size}→forAll ( i |
           self.connection→at (i).instance =
           l.connection→at (i).instance )
           implies self = l )

context LinkEnd
-- [1] The type of the Instance must match the type of the AssociationEnd.
inv: self.instance.classifier →union (
           self.instance.classifier.allParents)→includes (
           self.associationEnd.type)

context LinkObject
-- [1] One of the Classifiers must be the same as the Association.
inv: self.classifier →includes(self.association)

context LinkObject
-- [2] The Association must be a kind of AssociationClass.
inv: self.association.oclIsKindOf(AssociationClass)

context Message
-- [1] The sender and the receiver must participate in the Collaboration which defines the context
of the Interaction.
inv: self.interaction.context.ownedElement→includes (self.sender)
    and
    self.interaction.context.ownedElement→includes (self.receiver)

context Message
-- [2] The predecessors and the activator must be contained in the same Interaction.
```

inv: self . predecessor→**forAll** (p | p. interaction = self . interaction)
and
self . activator . interaction = self . interaction

context Message
-- [3] The predecessors must have the same activator as the Message.
inv: self . allPredecessors→**forAll** (p | p. activator = self . activator)

context Message
-- [4] A Message cannot be the predecessor of itself.
inv: **not** self . allPredecessors→**includes** (self)

context Message
-- [5] The communicationLink of the Message must be an AssociationRole in the context of the Message Interaction
inv: self . interaction .**context**.ownedElement→**includes** (
self . communicationConnection)

context Message
-- [6] The sender and the receiver roles must be connected by the AssociationRole which acts as the communication connection.
inv: self . communicationConnection→**size** > 0 **implies**
self . communicationConnection.connection→**exists** (ar |
ar . type = self . sender)
and
self . communicationConnection.connection→**exists** (ar |
ar . type = self . receiver)

context Message::allPredecessors:**Set**(Message)
-- [1] The operation allPredecessors results in the set of all Messages that precede the current one.
post: result = self . predecessor→**union**(self.predecessor . allPredecessors)

context Method
-- if the realized Operation is a query, then so is the Method.
inv: self . specification →**isQuery implies** self.**isQuery**

context Method
-- the signature of the Method should be the same as the signature of the realized Operation.
inv: self . hasSameSignature(self. specification)

context Method
-- the visibility of the Method should be the same as for the realized Operation.
inv: self . visibility = self . specification . visibility

context Method
-- the realized Operation must be a feature (possibly inherited) of the same Classifier as the Method.
inv: self . owner.allOperations→**includes**(self. specification)

context Method

-- if the realized Operation has been overriden one or more times in the ancestors of the owner of the Method, then the Method must realize the latest overriding (that is, all other Operations with the same signature must be owned by ancestors of the owner of the realized Operation).

inv: self . specification .owner.allOperations→**includesAll**(
(self .owner.allOperations→**select**(op |
self .hasSameSignature(op))))

context ModelElement::supplier:**Set**(ModelElement)

-- the operation supplier results in a Set containing all direct suppliers of the ModelElement.

asl : self .clientDependency.supplier

post: result = self .clientDependency.supplier

context ModelElement::allSuppliers:**Set**(ModelElement)

-- the operation allSuppliers results in a Set containing all the ModelElements that are suppliers of this ModelElement, including the suppliers of these ModelElements. This is the transitive closure.

asl : self . supplier→**union**(self.supplier . allSuppliers)

post: result = self . supplier→**union**(self.supplier . allSuppliers)

context ModelElement::model:**Set**(Model)

-- the operation model results in the set of Models to which the ModelElement belongs.

asl : self .namespace.allSurroundingNamespaces→**including**(self.namespace)→**select**(ns |
ns.**oclIsKindOf**(Model))

post: result = self .namespace.allSurroundingNamespaces→**including**(self.namespace)
→**select**(ns | ns.**oclIsKindOf**(Model))

context ModelElement::isTemplate:**Boolean**

-- a ModelElement is a template when it has parameters.

asl : self .templateParameter→**notEmpty**

post: result = self .templateParameter→**notEmpty**

context ModelElement::isInstantiated:**Boolean**

-- a ModelElement is an instantiated template when it is related to a template by a Binding relationship.

asl : self .clientDependency→**select**(**oclIsKindOf**(Binding))→**notEmpty**

post: result = self .clientDependency→**select**(**oclIsKindOf**(Binding))→**notEmpty**

context ModelElement::templateArguments:**Set**(ModelElement)

-- the templateArguments are the arguments of an instantiated template, which substitute for template parameters.

asl : self .clientDependency→**select**(d:Dependency | d.**oclIsKindOf**(Binding))
→**collect**(b | b.**oclAsType**(Binding).argument)→**asSet**

post: result = self .clientDependency→**select**(d:Dependency | d.**oclIsKindOf**(Binding))
→**collect**(b | b.**oclAsType**(Binding).argument)→**asSet**

context ModelElement

-- Tags associated with a ModelElement (directly via a property list or indirectly via a Stereotype) must not clash with any metaattributes associated with the Model Element.

--not specified in OCL

inv: **true**

context ModelElement
-- A model element must have at most one tagged value with a given tag name.
inv: self.taggedValue→**forAll**(t1, t2 : TaggedValue |
t1.tag = t2.tag **implies** t1 = t2)

context ModelElement
-- (Required tags because of stereotypes) If T in modelElement.stereotype.require Tag.such that T.value = unspecified, then the modelElement must have a tagged value with name = T.name.
inv: self.stereotype.requiredTag→**forAll**(tag |
tag.value→**isEmpty implies**
self.taggedValue→**exists**(t |
t.tag = tag.tag))

context Namespace
-- if a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace.
inv: self.allContents→**forAll**(me1, me2 : ModelElement |
(**not** me1.**oclIsKindOf**(Association) **and not** me2.**oclIsKindOf**(Association)
and
me2.**name** = me1.**name**) **implies**
me1 = me2)

context Namespace
-- all Associations must have a unique combination of name and associated Classifiers in the Namespace.
inv: self.allContents→**select**(e | e.**oclIsKindOf**(Association))→
forAll(a1, a2 |
a1.**name** = a2.**name**
and a1.**oclAsType**(Association).connection.type
= a2.**oclAsType**(Association).connection.type
implies a1 = a2)

context Namespace::contents:**Set**(ModelElement)
-- the operation contents results in a Set containing all ModelElements contained by the Namespace.
asl : self.ownedElement→**union**(self.namespace.contents)
post: result = self.ownedElement→**union**(self.namespace.contents)

context Namespace::allContents:**Set**(ModelElement)
-- the operation allContents results in a Set containing all ModelElements contained by the Namespace.
asl : self.contents
post: result = self.contents

context Namespace::allVisibleElements:**Set**(ModelElement)
-- the operation allVisibleElements results in a Set containing all ModelElements visible outside of the Namespace.
asl : self.allContents→**select**(e |
e.elementOwnership.visibility = #public)
post: result = self.allContents→**select**(e |

```
            e.elementOwnership.visibility = #public)

context Namespace::allSurroundingNamespaces:Set(Namespace)
   -- the operation allSurroundingNamespaces results in a Set containing all surrounding Na-
mespaces.
   asl : self.namespace.allSurroundingNamespaces→including(self.namespace)
   post: result = self.namespace.allSurroundingNamespaces→including(self.namespace)

context NodeInstance
-- [1] A NodeInstance must have only one Classifier as its origin, and it must be a Node.
inv: self. classifier →forAll ( c | c.oclIsKindOf(Node))
                  and
                   self. classifier →size = 1

context NodeInstance
-- [2] Each ComponentInstance that resides on a NodeInstance must be an instance of a Com-
ponent that resides on the corresponding Node.
inv: self.resident→forAll(n |
          self. classifier .oclAsType(Node).resident→includes(n.classifier))

context Object
-- [1] Each of the Classifiers must be a kind of Class.
inv: self. classifier →forAll ( c | c.oclIsKindOf(Class))

context ObjectFlowState
-- [1] Parameters of an object flow state must have a type and direction compatible with classifier
or classifier-in-state of the object flow state.
inv: let osftype : Classifier =
   (if self.type.oclIsKindOf(ClassifierInState)
   then self.type.type else self.type) in
   self.parameter.forAll(
      type = osftype
      or (parameter.kind = #in
      and osftype.allSupertypes→includes(type))
      or ((parameter.kind = #out or parameter.kind = #return)
      and type.allSupertypes→includes(osftype))
      or (parameter.kind = #inout
      and ( osftype.allSupertypes→includes(type)
      or type.allSupertypes→includes(osftype))))

context ObjectFlowState
-- [2] Downstream states have entry actions that accept input conforming to the type of the
classifier or classifier-in-state. The entry actions use the input parameters of the object flow
state. Valid downstream states are calculated by traversing outgoing transitions transitively,
skipping pseudo states, and entering and exiting subactivitystates, looking for regular states. If
the object flow state has no parameters, then the target of downstream actions must conform to
the type of the classifier or classifier-in-state.
inv: self. allnextleafstates .size > 0 and
       self. allnextleafstates .forAll(self.isinputaction(entry))

context ObjectFlowState
```

-- [3] Upstream states have entry actions that provide output or return values conforming to the type of the classifier or classifier-in-state. The entry actions use the output or return parameters of the object flow state. Valid upstream states are calculated by traversing incoming transitions transitively, skipping pseudo states, entering and exiting subactivity states, looking for regular states.
inv: self . allnextleafstates .**forAll**(self . allpreviousleaf
states .**size** > 0 **and**
self . allpreviousleafstates .**forAll**(self .isoutputaction(entry))

context Package
-- [1] A Package may only own or reference Packages, Classifiers, Associations, Generalizations, Dependencies, Constraints, Collaborations, StateMachines, and Stereotypes.
inv: self .contents→**forAll** (c |
c.**oclIsKindOf**(Package) **or**
c.**oclIsKindOf**(Classifier) **or**
c.**oclIsKindOf**(Association) **or**
c.**oclIsKindOf**(Generalization) **or**
c.**oclIsKindOf**(Dependency) **or**
c.**oclIsKindOf**(Constraint) **or**
c.**oclIsKindOf**(Collaboration) **or**
c.**oclIsKindOf**(StateMachine) **or**
c.**oclIsKindOf**(Stereotype))

context Package
-- [2] No imported element (excluding Association) may have the same name or alias as any element owned by the Package or one of its supertypes.
inv: self .allImportedElements→**reject**(re |
re.**oclIsKindOf**(Association))→**forAll**(re |
(re.elementImport.alias <> '' **implies**
not (self .allContents − self .allImportedElements)→
reject(ve |
ve.**oclIsKindOf** (Association))→**exists** (ve |
ve.**name** = re.elementImport.alias))
and
(re.elementImport.alias = '' **implies**
not (self .allContents − self .allImportedElements)→
reject (ve |
ve.**oclIsKindOf** (Association))→**exists** (ve |
ve.**name** = re.**name**)))

context Package
-- [3] Imported elements (excluding Association) may not have the same name or alias.
inv: self .allImportedElements→**reject**(re |
not re.**oclIsKindOf** (Association))→**forAll**(r1, r2 |
(r1.elementImport.alias <> '' **and**
r2.elementImport.alias <> '' **and**
r1.elementImport.alias = r2.elementImport.alias
implies r1 = r2)
and
(r1.elementImport.alias = '' **and**
r2.elementImport.alias = '' **and**

```
        r1.name = r2.name implies r1 = r2)
            and
        (r1.elementImport.alias <> '' and
        r2.elementImport.alias = '' implies
        r1.elementImport.alias <>r2.name))

context Package
-- [4] No imported element (Association) may have the same name or alias combined with
the same set of associated Classifiers as any Association owned by the Package or one of its
supertypes.
inv: self.allImportedElements→select( re |
        re.oclIsKindOf(Association) )→forAll( re |
        (re.elementImport.alias <> '' implies
        not (self.allContents − self.allImportedElements)→
        select( ve |
            ve.oclIsKindOf(Association) )→exists(
            me : ModelElement | let ve : Association = me.oclAsType(Association) in
            ve.name = re.elementImport.alias
            and
        ve.connection→size = re.connection→size and
        Sequence {1..re.connection→size}→forAll( i |
            re.connection→at(i).type =
            ve.connection→at(i).type ) ) )
            and
            (re.elementImport.alias = '' implies
            not (self.allContents − self.allImportedElements)→
            select( ve |
            not ve.oclIsKindOf(Association) )→exists( ve :
            Association |
            ve.name = re.name
            and
            ve.connection→size = re.connection→size and
            Sequence {1..re.connection→size}→forAll( i |
                re.connection→at(i).type =
                ve.connection→at(i).type ) ) ) )

context Package
-- [5] Imported elements (Association) may not have the same name or alias combined with the
same set of associated Classifiers.
inv: self.allImportedElements→select ( re |
    re.oclIsKindOf (Association) )→forAll ( m1, m2 : ModelElement |
    let r1 = m1.oclAsType(Association) in
    let r2 = m2.oclAsType(Association) in
    (r1.connection→size = r2.connection→size and
    Sequence {1..r1.connection→size}→forAll ( i |
    r1.connection→at (i).type =
    r2.connection→at (i).type and
    r1.elementImport.alias <> '' and
    r2.elementImport.alias <> '' and
    r1.elementImport.alias = r2.elementImport.alias
```

```
   implies r1 = r2))
      and
   (r1.connection→size = r2.connection→size and
   Sequence {1..r1.connection→size}→forAll ( i |
   r1.connection→at (i).type =
   r2.connection→at (i).type and
   r1.elementImport.alias = '' and
   r2.elementImport.alias = '' and
   r1.name = r2.name
   implies r1 = r2))
      and
   (r1.connection→size = r2.connection→size and
   Sequence {1..r1.connection→size}→forAll ( i |
   r1.connection→at (i).type =
   r2.connection→at (i).type and
   r1.elementImport.alias <>'' and
   r2.elementImport.alias = ''
   implies r1.elementImport.alias <>r2.name)))
```

context Package::contents:**Set**(ModelElement)
-- [1] The operation contents results in a Set containing the ModelElements owned by or imported by the Package.
post: result = self.ownedElement→**union**(self.importedElement)

context Package::allImportedElements:**Set**(ModelElement)
-- [2] The operation allImportedElements results in a Set containing the Model Elements imported by the Package or one of its supertypes.

```
post: result = self.importedElement→union(
    self.parent→collect(g | g.oclAsType(Package).allImportedElements→select(re |
       re.elementImport.visibility = #public or
       re.elementImport.visibility = #protected)))
```

context Package::allContents:**Set**(ModelElement)
-- [3] The operation allContents results in a Set containing the ModelElements owned by or imported by the Package or one of its ancestors.

```
post: result = self.contents→union(
    self.parent→collect(g | g.oclAsType(Package).allContents→select(e |
       e.elementOwnership.visibility = #public or
       e.elementOwnership.visibility = #protected)))
```

context Pseudostate
-- [1] An initial vertex can have at most one outgoing transition and no incoming transitions.

```
inv: ( self.kind = #initial) implies
   (( self.outgoing→size ≤ 1) and (self.incoming→isEmpty))
```

context Pseudostate
-- [2] History vertices can have at most one outgoing transition.

```
inv: (( self.kind = #deepHistory) or (self.kind = #shallowHistory))
   implies
   ( self.outgoing→size ≤ 1)
```

context Pseudostate
-- [3] A join vertex must have at least two incoming transitions and exactly one outgoing transition.
inv: (self .kind = #join) **implies**
((self .outgoing→**size** = 1) **and** (self.incoming→**size** ≥ 2))

context Pseudostate
-- [4] A fork vertex must have at least two outgoing transitions and exactly one incoming transition.
inv: (self .kind = #fork) **implies**
((self .incoming→**size** = 1) **and** (self.outgoing→**size** ≥ 2))

context Pseudostate
-- [5] A junction vertex must have at least one incoming and one outgoing transition.
inv: (self .kind = #junction) **implies**
((self .incoming→**size** ≥ 1) **and** (self.outgoing→**size** ≥ 1))

context Pseudostate
-- [6] A choice vertex must have at least one incoming and one outgoing transition.
inv: (self .kind = #choice) **implies**
((self .incoming→**size** ≥ 1) **and** (self.outgoing→**size** ≥ 1))

context Pseudostate
-- [1] In activity graphs, transitions incoming to (and outgoing from) join and fork pseudostates have as sources (targets) any state vertex. That is, joins and forks are syntactically not restricted to be used in combination with composite states, as is the case in state machines.
inv: self .stateMachine.**oclIsTypeOf**(ActivityGraph) **implies**
((self .kind = #join **or** self.kind = #fork) **implies**
(self .incoming→**forAll**(source.**oclIsKindOf**(State) **or**
source.**oclIsTypeOf**(Pseudostate)) **and**
(self .outgoing→**forAll**(source.**oclIsKindOf**(State) **or**
source.**oclIsTypeOf**(Pseudostate)))))

context Pseudostate
-- [2] All of the paths leaving a fork must eventually merge in a subsequent join in the model. Furthermore, multiple layers of forks and joins must be well nested, with the exception of forks and joins leading to or from synch state. Therefore the concurrency structure of an activity graph is in fact equally restrictive as that of an ordinary state machine, even though the composite states need not be explicit.
inv: **true**

context Reception
-- [1] A Reception can not be a query.
inv: **not** self.**isQuery**

context SendAction
-- [1] The number of arguments is the same as the number of parameters of the Signal.
inv: self .actualArgument→**size**=self.signal.allAttributes→**size**

context SendAction
-- [2] A Signal is always asynchronous.

inv: self .isAsynchronous

context StateMachine
-- [1] A StateMachine is aggregated within either a classifier or a behavioral feature.
inv: self .**context**.**oclIsKindOf**(BehavioralFeature) **or**
self .**context**.**oclIsKindOf**(Classifier)

context StateMachine
-- [2] A top state is always a composite.
inv: self .top.**oclIsTypeOf**(CompositeState)

context StateMachine
-- [3] A top state cannot have any containing states.
inv: self .top.container→**isEmpty**

context StateMachine
-- [4] The top state cannot be the source of a transition.
inv: (self .top.outgoing→**isEmpty**)

context StateMachine
-- [5] If a StateMachine describes a behavioral feature, it contains no triggers of type CallEvent, apart from the trigger on the initial transition (see OCL for Transition [8]).
inv: self .**context**.**oclIsKindOf**(BehavioralFeature) **implies**
self . transitions →**reject**(t |
t .source.**oclIsKindOf**(Pseudostate) **and**
t .source.**oclAsType**(Pseudostate).kind= #initial).trigger→**isEmpty**

context Stereotype
-- Stereotype names must not clash with any baseClass names.
inv: Stereotype.**allInstances**→**forAll**(st |
st .baseClass <>self.**name**)
context Stereotype
-- Stereotype names must not clash with the names of any inherited Stereotype.
inv: self . allParents→**forAll**(st |
st .**name** <>self.**name**)

context Stereotype
-- Stereotype names must not clash in the (M2) meta-class namespace, nor with the names of any inherited Stereotype, nor with any baseClass names.
--M2 level not accessible
inv: **true**

context Stereotype
-- The baseClass name must be provided ; icon is optional and is specified in an implementation specific way.
inv: self .baseClass <>''

context Stereotype
-- [5] Tag names attached to a Stereotype must not clash with M2 meta-attribute namespace of the appropriate baseClass element, nor with Tag names of any inherited Stereotype.

-- M2 level not accessible
inv: **true**

context Stimulus
-- [1] The number of arguments must match the number of Arguments of the Action.
inv: self.dispatchAction.actualArgument→**size** = self.argument→**size**

-- [2] The Action must be a SendAction, a CallAction, a CreateAction, or a DestroyAction.
inv: self.dispatchAction.**oclIsKindOf** (SendAction)
or self.dispatchAction.**oclIsKindOf** (CallAction)
or self.dispatchAction.**oclIsKindOf** (CreateAction)
or self.dispatchAction.**oclIsKindOf** (DestroyAction)

context SubactivityState
-- [1] A subactivity state is a submachine state that is linked to an activity graph.
inv: self.submachine.**oclIsKindOf**(ActivityGraph)

context SubmachineState
-- [1] Only stub states allowed as substates of a submachine state.
inv: self.subvertex→**forAll** (s | s.**oclIsTypeOf**(StubState))

context SubmachineState
-- [2] Submachine states are never concurrent.
inv: self.isConcurrent = **false**

context Subsystem
-- [1] For each Operation in an Interface offered by a Subsystem, the Subsystem itself or at least one contained specification element must have a matching Operation.
inv: self. specification .allOperations→**forAll**(interOp |
self.allOperations→**union**
(self. allSpecificationElements →**select**(specEl|
specEl.**oclIsKindOf**(Classifier))→**forAll**(c|
c.**oclAsType**(Classifier).allOperations))→**exists**
(op | op.hasSameSignature(interOp)))

context Subsystem
-- [2] The Features of a Subsystem may only be Operations or Receptions.
inv: self.feature→**forAll**(f | f.**oclIsKindOf**(Operation) **or**
f.**oclIsKindOf**(Reception))

context Subsystem::allSpecificationElements:**Set**(ModelElement)
-- [1] The operation allSpecificationElements results in a Set containing the Model Elements specifying the behavior of the Subsystem.
post: result = self.allContents→**select**(c |
c.elementOwnership.isSpecification)

context Subsystem
-- [2] The operation contents results in a Set containing the ModelElements owned by or imported by the Subsystem.
post: result = self.ownedElement→**union**(self.importedElement)

context SynchState
-- [1] The value of the bound attribute must be a positive integer, or unlimited.
inv: (self .bound > 0) **or** (self.bound = unlimited)

context SynchState
-- [2] All incoming transitions to a SynchState must come from the same region and all outgoing transitions from a SynchState must go to the same region.
inv: **true**

context TerminateAction
-- [1] A TerminateAction has no arguments.
inv: self .actualArgument→**size** = 0

context TerminateAction
-- [2] A TerminateAction has no target expression.
inv: self .target→**isEmpty**

context Transition
-- [1] A fork segment should not have guards or triggers.
inv: self .source.**oclIsKindOf**(Pseudostate) **implies**
((self .source.**oclAsType**(Pseudostate).kind = #fork) **implies**
((self .guard→**isEmpty**) **and** (self.trigger→**isEmpty**)))

context Transition
-- [2] A join segment should not have guards or triggers.
inv: self .target .**oclIsKindOf**(Pseudostate) **implies**
((self .target .**oclAsType**(Pseudostate).kind = #join) **implies**
((self .guard→**isEmpty**) **and** (self.trigger→**isEmpty**)))

context Transition
-- [3] A fork segment should always target a state.
inv: (self .stateMachine→**notEmpty**) **implies**
self .source.**oclIsKindOf**(Pseudostate) **implies**
((self .source.**oclAsType**(Pseudostate).kind = #fork) **implies**
(self .target .**oclIsKindOf**(State)))

context Transition
-- [4] A join segment should always originate from a state.
inv: (self .stateMachine→**notEmpty**) **implies**
self .target .**oclIsKindOf**(Pseudostate) **implies**
((self .target .**oclAsType**(Pseudostate).kind = #join) **implies**
(self .source.**oclIsKindOf**(State)))

context Transition
-- [5] Transitions outgoing pseudostates may not have a trigger.
inv: self .source.**oclIsKindOf**(Pseudostate)
implies (self . trigger →**isEmpty**))

context Transition

```
-- [6] Join segments should originate from orthogonal states.
inv: self.target.oclIsKindOf(Pseudostate) implies
    ((self.target.oclAsType(Pseudostate).kind = #join) implies
    (self.source.container.isConcurrent))

context Transition
-- [7] Fork segments should target orthogonal states.
inv: self.source.oclIsKindOf(Pseudostate) implies
    ((self.source.oclAsType(Pseudostate).kind = #fork) implies
        (self.target.container.isConcurrent))

context Transition
-- [8] An initial transition at the topmost level may have a trigger with the stereotype "create."
An initial transition of a StateMachine modeling a behavioral feature has a CallEvent trigger
associated with that BehavioralFeature. Apart from these cases, an initial transition never has
a trigger.
inv: self.source.oclIsKindOf(Pseudostate) implies
    ((self.source.oclAsType(Pseudostate).kind = #initial) implies
    (self.trigger→isEmpty or
    ((self.source.container = self.stateMachine.top) and
    (self.trigger.stereotype.name = 'create')) or
    (self.stateMachine.context.oclIsKindOf(BehavioralFeature)
    and
    self.trigger.oclIsKindOf(CallEvent) and
    (self.trigger.oclAsType(CallEvent).operation =
    self.stateMachine.context))
    ))

context Transition
-- see [8]
inv: self.source.oclIsKindOf(Pseudostate) implies
    ((self.source.oclAsType(Pseudostate).kind = #initial) implies
    (self.trigger→isEmpty or
    ((self.source.container = self.stateMachine.top) and
    (self.trigger.stereotype.name = 'create')) or
    (self.StateMachine.context.oclIsKindOf(BehaviouralFeature)
        and
    self.trigger.oclIsKindOf(CallEvent) and
    (self.trigger.operation =
    self.stateMachine.context))
    ))

context UseCase
-- [1] UseCases can only have binary Associations.
inv: self.associations→forAll(a | a.connection→size = 2)

context UseCase
-- [2] UseCases can not have Associations to UseCases specifying the same entity.
inv: self.associations→forAll(a |
        a.allConnections→forAll(s, o|
```

```
    let s_uc : UseCase = s.type.oclAsType(UseCase) in
    let o_uc : UseCase = o.type.oclAsType(UseCase) in
    (s_uc.specificationPath→isEmpty and
    o_uc.specificationPath→isEmpty )
       or
    (not s_uc.specificationPath→includesAll(
    o_uc.specificationPath ) and
    not o_uc.specificationPath→includesAll(
    s_uc.specificationPath ))
    ))

context UseCase
-- [3] A UseCase cannot contain any Classifiers.
inv: self.contents→isEmpty

context UseCase
-- [4] The names of the ExtensionPoints must be unique within the UseCase.
inv: self.allExtensionPoints → forAll (x, y |
            x.name = y.name implies x = y )

context UseCase::specificationPath:Set(Namespace)
-- [1] The operation specificationPath results in a set containing all surrounding Namespaces
that are not instances of Package.
post: result = self.allSurroundingNamespaces→select(n |
    n.oclIsKindOf(Subsystem) or n.oclIsKindOf(Class))

context UseCase::allExtensionPoints:Set(ExtensionPoint)
-- [2] The operation allExtensionPoints results in a set containing all ExtensionPoints of the
UseCase.
post: result = self.allParents→collect(ep | ep.oclAsType(ExtensionPoint).extensionPoint)
    → union (self.extensionPoint)

context UseCaseInstance
-- [1] The Classifier of a UseCaseInstance must be a UseCase.
inv: self. classifier →forAll ( c | c.oclIsKindOf (UseCase) )
```

Bibliographie

[1] Convergence project. Technical report, 2001.

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

[3] William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi. Transformation in intentional programming. In P. Devanbu and J. Poulin, editors, *Proceedings : Fifth International Conference on Software Reuse*, pages 114–123. IEEE Computer Society Press, 1998.

[4] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. *Lecture Notes in Computer Science*, 791 :152– ? ?, 1994.

[5] Alcatel, ILogix, Kennedy-Carter, Kabira, Project Technology, Rational Software, and Telelogic. Action semantics for the UML final submission. Technical report, Object Management Group, 2001.

[6] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *Proceedings of UML'2001*, 2001.

[7] Matthew H. Austern. *Generic Programming and the STL : using and extending the C++ Standard Template Library*. Addison-Wesley professional computing series. Addison-Wesley Longman Publ. Co., 1998.

[8] Thomas Baar and Reiner Hähnle. An integrated metamodel for OCL types. In Robert France, Bernhard Rumpe, Jean-Michel Bruel, Ana Moreira, John Whittle, and Ileana Ober, editors, *Proc. OOPSLA 2000, Workshop Refactoring the UML : In Search of the Core, Minneapolis, Minnesota, USA, 2000.*, 2000.

[9] Franck Barbier and Brian Henderson-Sellers. Object metamodelling of the whole-part relationship. In Christine Mingins, editor, *Proceedings of TOOLS Pacific 1999*. IEEE Computer Society, 1999.

[10] Franck Barbier, Brian Henderson-Sellers, Andreas L. Opdahl, and Martin Gogolla. The whole-part relationship in the Unified Modeling Language : A new approach. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language : Systems Analysis, Design and Development Issues*, chapter 12, pages 185–209. Idea Publishing Group, 2001.

[11] Richard Barker. *Case Method : Entity Relationship Modelling*. Addison Wesley, 1990.

[12] John Barnes, Ben Brosgol, et al. Ada 95 rationale. the language. the standard libraries. Technical report, Intermetrics Inc., 733 Concord Av, Cambridge, MA 02138, January 1995.

[13] David Bellin, Susan Suchman Simone, and Grady Booch. *The CRC Card Book*. Addison Wesley, 1997.

[14] Jean Bézivin. Modèles et méta-modèles dans le cycle de développement de logiciel. In *EJC'2001*, 2001.

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

[16] Richard S Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[17] G.S. Blair, G. Coulson, F. Costa, and H.A. Duran. On the design of reflective middleware platforms. In *Proceedings of the Reflective Middleware Workshop, RM 2000*, 2000.

[18] D.G. Bobrow and M. Stefik. *The Loops Manual*. Xerox PARC, Palo Alto CA, USA, December 1983.

[19] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, 1993.

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

[21] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A visualization of ocl using collaborations. In *Proceedings of UML'2001*, 2001.

[22] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. In *Technique et Science Informatique*, volume 20. Hermes, 2001.

[23] Jean-Pierre Briot and Pierre Cointe. The OBJVLISP Model : Definition of a Uniform, Reflexive and Extensible Object Oriented Language. In *Proceedings of ECAI'86*, pages 225–232, 1986.

[24] Jean-Pierre Briot and Pierre Cointe. A Uniform Model for Object-Oriented Languages Using the Class Abstraction. In *Proceedings of IJCAI'87*, pages 40–43, 1987.

[25] Walter Cazzola, Shigeru Chiba, and Thomas Ledoux. Reflection and meta-level architectures : State of the art and future trends. In *ECOOP Workshops*, pages 1–15, 2000.

[26] María Victoria Cengarle and Alexander Knapp. A formal semantics for OCL 1.4. In *Proceedings of UML'2001*, 2001.

[27] Shigeru Chiba. A metaobject protocol for C++. pages 285–299.

[28] Tony Clark, Andy Evans, and Stuart Kent. The specification of a reference implementation for the unified modeling language. *L'OBJET*, 7(1), 2001.

[29] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, and Steve Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. Technical report, The pUML Group, 2000.

[30] P. Cointe. Metaclasses are first class : the objvlisp model. In *Proceedings of OOPSLA'87*, pages 156–167. ACM, 1987.

[31] Dan Connolly, editor. *XML : Principles, Tools, and Techniques*, volume 2(4) of *The World Wide Web Journal*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel : +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel : +1 617 354 5800, Winter 1997.

[32] Stefan Conrad and Klaus Turowski. Temporal OCL : Meeting specification demands for business components. In Keng Siau and Terry Halpin, editors, *Unified Modeling Language : Systems Analysis, Design and Development Issues*, chapter 10, pages 151–166. Idea Publishing Group, 2001.

[33] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, , and Alan Wills. The amsterdam manifesto on ocl. Technical Report TUM-I9925, Technische Univerität München, 1999.

[34] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, Reading, Mass., 1999.

[35] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. http ://www.extreme.indiana.edu/ tveldhui/papers/dagstuhl/.

[36] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming : methods, tools and applications*. Addison-Wesley, 2000.

[37] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a temporal logic for object-based systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.

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

[39] Thierry Duval and François Pennaneac'h. Using the pac-amodeus model and design patterns to make interactive an existing object-oriented kernel. In *Technology of Object-Oriented Languages and Systems (TOOLS Europe)*, volume 33, pages 407–418. IEEE Computer Society, 2000.

[40] E. Ernst. Separation of concerns and then what, 2000.

[41] H. Fleischhack and B. Grahlmann. A compositional Petri net semantics for SDL. *Lecture Notes in Computer Science*, 1420 :144–??, 1998.

[42] R. France, JM. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 247–260, Canterbury, UK, 1997. Chapman and Hall, London.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[44] U. Glässer, R. Gotzhein, and A. Prinz. Towards a New Formal SDL Semantics Based on Abstract State Machines. In *SDL'99 - The Next Millenium, Proceedings of the 9th SDL Forum*. Elsevier Science B.V., 1999.

[45] Martin Gogolla and Francesco Parisi-Presicce. State diagrams in UML : A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

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

[47] M. Große-Rhode. Using a formal reference model for consistency checking and integration of UML diagrams. In Peter A. Ng, editor, *Proc. Fifth International Conference on Integrated Design and Process Technology (IDPT'2000), June 2000, Dallas, Texas*, 2000.

[48] Alain Le Guennec. Génie logiciel et méthodes formelles avec UML : Spécification, validation et génération de tests, 2001.

[49] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *Proceedings of UML 2000*, volume 1939 of *LNCS*, pages 482–496. Springer Verlag, 2000.

[50] Ali Hamie, John Howse, and Stuart Kent. Navigation expressions in object-oriented modelling. In Egidio Astesiano, editor, *Proceedings Fundamental Approaches to Software Engineering, 1st Internationsl Conference, FASE'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998*, volume 1382, pages 123– ? ? Springer, 1998.

[51] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, October 1993.

[52] Brian Henderson-Sellers and Franck Barbier. Black and white diamonds. 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 550–565. Springer, 1999.

[53] Brian Henderson-Sellers and Bhuvan Unhelkar. *OPEN Modeling with UML*. Addison-Wesley, 2000.

[54] Wai-Ming Ho. Contribution à la réification d'un processus de conception.

[55] W.M. Ho, F. Pennaneac'h, and N. Plouzeau. Umlaut : A framework for weaving UML-based aspect-oriented designs. In *Technology of object-oriented languages and systems (TOOLS Europe)*, volume 33, pages 324–334. IEEE Computer Society, June 2000.

[56] Stan-Ober Ileana. Harmonisation des langages de modélisation avec des extensions orientées-objet et une sémantique exécutable, 2001.

[57] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, revised printing edition, 1993.

[58] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[59] C. Jard, J.-M. Jézéquel, and F. Pennaneac'h. Vers l'utilisation d'outils de validation de protocoles dans UML. *Technique et Science Informatique*, 17(9), September 1998.

[60] 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.

[61] Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Validating distributed software modeled with the Unified Modeling Language. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 365–377. Springer, 1999.

[62] 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.

[63] Jean-Marc Jézéquel and François Pennaneac'h. Preliminary ideas for validating distributed OO software. In *Workshop on Models, Formalisms and Methods for Object-Oriented Distributed Computing (ECOOP'97 Workshop #6)*, Finland, June 1997.

[64] Kennedy-Carter. Executable UML (xuml), http ://www.kc.com/html/xuml.html.

[65] S. Kent, A. Hamie, J. Howse, F. Civello, and R. Mitchell. Semantics through pictures, 1998.

[66] Stuart Kent and John Howse. Mixing visual and textual constraint languages. 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 384–398. Springer, 1999.

[67] Ismaïl Khriss, Mohammed Elkoutbi, and Rudolf K. Keller. Automating the synthesis of UML StateChart diagrams from multiple collaboration diagrams. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 132–147. Springer, 1999.

[68] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[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, pages 220–242. Springer-Verlag, New York, NY, 1997.

[70] Anneke Kleppe and Jos Warmer. Extending OCL to include actions. 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 440–450. Springer, 2000.

[71] Anneke Kleppe, Jos Warmer, and Steve Cook. Informal formality ? the Object Constraint Language and its application in the UML metamodel. 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 127–136, 1998.

[72] Reto Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA*. IEEE CS Press, Los Alamitos, 1998.

[73] Philippe Kruchten. *Rational Unified Process : an Introduction*. Addison-Wesley, Reading/MA, 1998.

[74] Régine Laleau and Fiona Polack. Coming and going from uml to b : a proposal to support traceability in rigorous is development.

[75] K. Lano, J. Bicarregui, and A. Evans. Structured axiomatic semantics for UML models, 2000.

[76] C. Lopes and G. Kiczales. Aspect-oriented programming with aspectj.

[77] José Àlvarez, Andy Evans, and Paul Sammut. Mml and the metamodel architecture. Technical report, The pUML Group, 2001.

[78] David Maley. Supporting design by contract in C++. *Journal of Object-Oriented Programming*, august 2001.

[79] Luis Mandel and María Victoria Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999.

[80] Rafael Marcano and Nicole Levy. Transformations d'annotations OCL en expressions B. In *AFADL'2001*, 2001.

[81] J. McAffer. Meta-level programming with coda. In *Proceedings of ECOOP'95*, number 952 in Lecture Notes in Computer Science, pages 190–214. AITO, Springer-Verlag, 1995.

[82] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, October 1992.

[83] Bertrand Meyer. *Eiffel : The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[84] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[85] Hafedh Mili, Francois Pachet, Ilham Benyahia, and Fred Eddy. Metamodeling in OO : OOPSLA'95 workshop summary. pages 105–110.

[86] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and associates, 2000.

[87] Thomas J. Mowbray and Ron Zahavi. *Essential CORBA*. John Wiley and Sons, 1995.

[88] Pierre-Alain Muller. *Modélisation objet avec UML*. Editions Eyrolles, 1997. ISBN : 2-212-08966-x.

[89] D. Muthiayen and V. S. Alagar. Formalizing UML for rigorous software development. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why ? How ?*, 1998.

[90] K. Nygaard and O.-J. Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages*. acm press, 1981.

[91] H. Okamura, Y. Ishikawa, and M. Tokoro. Al-1/d : A distributed programming system with multi-model reflection framework. In A. Yonezawa and B.C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architectures, Reflection and Metalevel Architectures*, pages 36–47. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.

[92] Anders Olsen, Ove Faergemand, R. Reed, J.R.W. Smith, and B. Mller-Pedersen. *Systems Engineering Using Sdl-92*. 1997.

[93] OMG. Action semantics for the UML RFP. Technical report, Object Management Group, 1998.

[94] OMG. Meta object facility (mof), version 1.3. Technical report, Object Management Group, 2000.

[95] OMG. Common warehouse metamodel specification. Technical report, Object Management Group, 2001.

[96] OMG. Model driven architecture, a technical perspective. Technical report, Object Management Group, 2001.

[97] OMG. UML 2.0 OCL RFP. Technical report, Object Management Group, 2001.

[98] OMG. Updated final submission of the software process engineering metamodel v 2.1. Technical report, Object Management Group, 2001.

[99] Gunnar Övergaard. A formal approach to relationships in the Unified Modeling Language. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[100] François Pennaneac'h, Jean-Marc Jézéquel, Jacques Malenfant, and Gerson Sunyé. UML Reflections. In *Reflections 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[101] François Pennaneac'h and Gerson Sunyé. Towards an execution engine for the UML. *UML'2k Conference, workshop on dynamic behaviour in UML models : semantic questions*, October 2000.

[102] Projtech-Technology. Executable UML, http ://www.projtech.com/pubs/xuml.html.

[103] R. Alur and T.A. Henzinger. Logics and Models of Real-Time : A Survey. In *Real Time : Theory in Practice*, volume 600, pages 74–106. Springer-Verlag, 1991.

[104] 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.

[105] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects : The OORam Software Engineering Method*. Prentice-Hall, 1995.

[106] G. Reggio and E. Astesiano. A proposal of a dynamic core for uml metamodelling with mml. april 2001.

[107] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th International Conference on Conceptual Modeling (ER)*, volume 1507, pages 449–464. Springer-Verlag, 1998.

[108] Mark Richters and Martin Gogolla. A metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd International Conference on the Unified Modeling Language (UML)*, volume 1723 of *LNCS*, pages 156–171. Springer-Verlag, 1999.

[109] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[110] UML RTF. *OMG Unified Modeling Language Specification, Version 1.4 draft*. OMG, February 2001.

[111] E. Rudolph, J. Grabowski, and P. Graubmann. Towards a harmonization of UML-Sequence Diagrams and MSC. In R. Dsoulli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 : The Next Millennium, Proceedings of the 9th SDL Forum*, Montreal, Canada, June 1999. Elsevier Science Publishers.

[112] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objekt-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[113] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[114] H. A. Sahraoui. The METAGEN system. *Lecture Notes in Computer Science*, 936 :590– ? ?, 1995.

[115] John Anil Saldhana. Uml diagrams to object Petri net models : An approach for modeling and analysis.

[116] Shane Sendall and Alfred Strohmeier. Enhancing OCL for specifying pre- and postconditions. In *UML 2.0 - The Future of the UML Object Constraint Language (OCL). Workshop of UML 2000 - The Unified Modeling Language : Advancing the Standard, Third International Conference, York, UK, October 2 - 6, 2000.*, 2000. Also available as Technical Report EPFL-DI No 00/349.

[117] Anthony Simons. On the compositional properties of UML statechart diagrams. In *Rigorous Object-Oriented Methods*, 2000.

[118] Anthony J. H. Simons and Ian Graham. 30 things that go wrong in object modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 221–242. Kluwer, Dordrecht, 1999.

[119] John McGregor Sita Ramakrishnan. Modelling and testing OO distributed systems with temporal logic formalisms.

[120] J. M. Spivey. *The Z Notation : A Reference Manual*. International Series in Computer Sciences. Prentice-Hall, London, second edition, 1992.

[121] Bjarne Stroustrup. *The C++ Programming Language : Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[122] S.M. Tan, D.K. Raila, and R.H. Campbell. An Object-Oriented Nano-Kernel for Operating System Hardware Support. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems, IWOOOS'95*. IEEE, Coputer Society Press, 1995.

[123] P. Tarr, H. Ossher, W. Harrison, J. Stanley, and M. Sutton. degrees of separation : Multi-dimensional separation of concerns, 1999.

[124] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the Object Constraint Language of UML, December 1999. Response to Object Management Group's Request for Information on UML 2.0.

[125] Jos Warmer and al. Response to the UML 2.0 RFP for OCL, 2001.

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

[127] Roel Wieringa and Jan Broersen. A minimal transition system semantics for lightweight class- and behavior diagrams. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[128] Ho W.M., Jézéquel J.-M., Le Guennec A., and Pennaneac'h F. UMLAUT : an extendible UML transformation framework. (RR-3775), Oct 1999. http ://www.inria.fr/RRRT/RR-3775.html.

[129] Øystein Haugen. From msc-2000 to uml-2000 - the future of sequence diagrams. 2001.