THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Discipline ou spécialité :

INFORMATIQUE

Présentée et soutenue par :
Thanh Thanh LE THI

le : vendredi 30 septembre 2011

Titre :
Modélisation en UML/OCL
des Langages de Programmation et de leurs Propriétés
et Processus IDM

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
Institut de Recherche en Informatique de Toulouse - IRIT UMR 5505 CNRS

Directeur(s) de Thèse :

Pierre BAZEX Professeur émérite, Université Toulouse 3

Rapporteurs :
Henri BASSON Professeur, Université du Littoral Côte d'Opale
Sabine MOISAN Directeur de Recherche INRIA-Sophia Antipolis, HDR

Autre(s) membre(s) du jury

Jean-Paul BODEVEIX Professeur, Université Toulouse 3
Louis FERAUD Professeur, Université Toulouse 3
Akram IDANI Maître de conférences, Grenoble INP (ENSIMAG)

THESE

En vue de l'obtention du

Doctorat de l'Université de Toulouse

Délivré par

L'Université Toulouse 3 – Paul Sabatier

Discipline INFORMATIQUE

Présentée et soutenue par

Thanh-Thanh LE THI

Modélisation en UML/OCL des Langages de Programmation et de leurs Propriétés et Processus IDM

JURY

Henri BASSON	Professeur, Université du Littoral Côte d'Opale	*Rapporteur*
Pierre BAZEX	Professeur émérite, Université Toulouse 3	*Encadrant*
Jean-Paul BODEVEIX	Professeur, Université Toulouse 3	*Examinateur*
Louis FERAUD	Professeur, Université Toulouse 3	*Examinateur*
Akram IDANI	Maître de conférences, Grenoble INP (ENSIMAG)	*Examinateur*
Sabine MOISAN	Directeur de Recherche INRIA-Sophia Antipolis, HDR	*Rapporteur*

Ecole doctorale : Mathématique, Informatique et Télécommunications de Toulouse
Unité de recherche : Institut de Recherche en Informatique de Toulouse -- IRIT UMR 5505 CNRS
Equipe d'accueil : Modèles, Aspects, Composants pour des Architectures à Objets (MACAO)

Thanh-Thanh LE THI

Modélisation en UML/OCL
des Langages de Programmation et de leurs Propriétés,
Processus IDM

Directeur de thèse :
Pierre BAZEX, professeur des universités émérite – Toulouse 3- Paul Sabatier

Résumé :

Nous présentons dans ce document nos travaux de recherche axés principalement sur l'activité de génération de composants logiciels se situant en phase terminale des processus de développement de logiciels dirigés par les modèles. Il s'agit de montrer comment nos travaux de recherche ont contribué à obtenir plus de continuité entre les activités de modélisation des exigences et les activités de génération de codes.

Le domaine de recherche de cette étude recouvre certains aspects des technologies des modèles puisqu'un processus IDM peut être considéré comme une succession de transformations de modèles, ainsi que certains aspects des technologies des langages puisqu'il s'agit de produire les codes à partir des modèles. Dans une première partie, nous présentons donc les travaux de recherche déjà existants sur les modèles et les transformations de modèles, ainsi que sur la modélisation en UML/OCL des langages de programmation limitée, la plupart du temps, aux aspects syntaxiques.

Dans une deuxième partie, nous montrons comment nous modélisons en UML/OCL, les propriétés comportementales et axiomatiques des langages de programmation de style impératif. La modélisation des propriétés comportementales d'un langage permet d'exécuter des modèles de codes, si l'on dispose d'un éditeur UML intégrant un interprète du langage OCL et d'un langage d'actions. La modélisation des propriétés axiomatiques d'un langage nous amène à montrer comment on peut, à l'aide de triplets de Hoare, vérifier que des segments de modèles de programmes sont corrects. En fait, les assertions déduites des triplets de Hoare par application des propriétés axiomatiques du langage sont transmises à un Atelier B en vue d'étudier leurs éventuelles validités.

Dans une troisième partie, nous montrons comment on peut injecter au niveau du Méta-Modèle UML des propriétés comportementales et axiomatiques spécifiques au domaine d'applications considéré. Nous nous sommes limités au fragment du Méta-Modèle UML définissant les diagrammes d'activité que l'on peut retrouver généralement dans un processus IDM, en amont des modèles de codes, avant donc la génération proprement dite des codes. La cohérence entre les modèles et les codes peut donc se vérifier à l'aide de propriétés comportementales et axiomatiques en comparant les modèles issues des exigences et les modèles des codes.

Ces travaux ont été financés dans le cadre des projets de recherche de l'ANR : Domino (2008-2009) et MyCitizSpace (2010)

Institut de Recherche en Informatique de Toulouse – UMR 5505 CNRS
Université Toulouse 3 – Paul Sabatier, 118 route de Narbonne, F-31062 cedex 9

Thanh-Thanh LE THI

Modélisation en UML/OCL des Langages de Programmation et de leurs Propriétés, Processus IDM

Supervisor :
Pierre BAZEX, Emerit Professor at Toulouse 3 University - Paul Sabatier

Abstract :

Our work focuses on the software component generation phase that takes place at the last phase of a model driven development process. We are interested in how to obtain a better continuity between the requirement modeling and the code generation activity of a software development process.

Our work is related to either the modelware or the grammarware because the model driven process can be considered as a successive of model transformations whereas the code generation is a specific transformation from the model to a language grammar. In the first part, we resume some relative works in the domain of the models and of the models transformation; we also present the language modeling in UML which is generally restricted by the syntax modeling.

In the second part, we show how we model in UML/OCL the behavioral and axiomatic properties of imperative programming languages. The modeling of the behavioral properties helps to execute the code models if we dispose a right execution environment. In the other hand, the modeling of the axiomatic properties helps to demonstrate the correctness of the code model. In fact, the assertions obtained from the modeling of the axiomatic properties of the language will be transferred to a B atelier in order to have further validation.

In the third part, we show how we inject into the UML metamodel the considered domain behavioral and axiomatic properties. We focus on the activity diagram metamodel of the UML which defines the behavior part of a UML model. The coherence between the models and the codes can be then verified in comparing the behavioral and axiomatic properties of the models issued from the requirements and that of the codes.

Our work is financed by the ANR research projects: DOMINO (2008-2009) and MyCitizSpace (2010)

Institut de Recherche en Informatique de Toulouse – UMR 5505 CNRS
Université Toulouse 3 – Paul Sabatier, 118 route de Narbonne, F-31062 cedex 9

Remerciements :

Je tiens à remercier tous ceux qui, directement ou indirectement, m'ont aidée dans la réalisation de ce travail et de l'intérêt qu'ils ont témoigné à l'égard de mon travail.

Je remercie tout particulièrement Madame Moisan, directrice de recherche à l'INRIA à Sophia-Antipolis, et Monsieur Basson, professeur de l'Université du Littoral Côte d'Opale, qui m'ont fait l'honneur de rapporter sur mon mémoire. De part toutes les remarques qu'ils m'ont retournées, ils m'ont permis d'améliorer de façon signification la qualité de mon document, tant sur le fond que sur la forme. Je les remercie surtout d'avoir pu trouver du temps de lire et relire très attentivement mon document.

Je tiens à remercier Jean-Paul Bodeveix, Louis Féraud, professeurs à l'Université Paul Sabatier, ainsi que Akram Idani, maître de conférences à l'INP (ENSIMAG) à Grenoble, d'avoir accepté de participer à mon jury. Je suis très reconnaissante de la patience et de la disponibilité dont ils ont fait preuve pour discuter certains des aspects scientifiques et techniques du sujet, mais aussi pour résoudre les difficultés. L'approche formelle des solutions qu'ils m'ont suggérées m'a beaucoup aidée pour aborder mon travail de recherche avec plus de rigueur. J'ai particulièrement, apprécié tout au long de ce travail, le soutien qu'ils m'ont apporté, leurs encouragements et leurs conseils toujours pertinents.

Je tiens à remercier aussi tous les chercheurs de l'équipe de recherche MACAO de l'IRIT, dirigée par Christian Percebois, professeur à l'Université Paul Sabatier, puis par Bernard Coulette, professeur à l'Université du Mirail, pour toutes nos discussions et collaborations, enrichissantes tant sur le plan personnel que sur le plan professionnel. Merci à Thierry Millan, maître de conférences à l'IUT de l'Université Paul Sabatier, pour son accueil chaleureux et pour toute l'ambiance qu'il a su créer dans l'équipe.

Un éternel merci à Pierre Bazex, professeur émérite à l'Université Paul Sabatier. Merci de m'avoir accepté en thèse avec toute confiance. Je le remercie de toute sa disponibilité pour toutes nos discussions, pour tous ses conseils (scientifiques, en particulier) et pour tous ses encouragements pendant ces années de travail. Pour tout cela, je lui en suis infiniment redevable.

Je remercie l'ANR qui, au travers des projets Domino et MyCitizSpace, ont subventionné cette recherche. Au travers de projets impliquant une collaboration entre partenaires universitaires et industriels, j'ai pu ainsi découvrir ainsi la complémentarité que peut apporter une collaboration entre partenaires universitaires et industriels lors du développement de projets pouvant être soumis à de exigences très fortement critiques que ce soit pour des systèmes embarqués ou que ce soit pour des systèmes gérant des données personnelles, voire privées, destinés au grand public. Je tiens à remercier, tout spécialement, Olivier Nicolas, Chef de produit e-Citiz, et Mikaël Vera, Chef de plate-forme e-Citiz, pour leurs encouragements et leurs conseils avisés et pragmatiques. Ils m'ont aidée à découvrir le méta-modèle de Studio e-Citiz.

Je n'oublie non plus Dominique Rieu, professeur à l'Université de Grenoble 1, et Agnès Front, maitre de conférences de l'Université de Grenoble 1 qui m'ont accueillie dans le cadre de mon stage de recherche du M2 Informatique à l'Université de Grenoble et qui m'ont fait découvrir et apprécier le monde de la recherche.

Je termine enfin ces remerciements en dédiant cette thèse de doctorat à mon mari, mes parents et à mes amis que j'ai eus la chance d'avoir à mes côtés, qui m'ont soutenu tout au long de ces années de travail.

Table des matières

Introduction – L'activité de Génération de Codes des Processus de Développement Logiciels IDM

Chapitre I – Cadre de l'étude : Les Propriétés Syntaxiques et Sémantiques des Langages de Programmation

Chapitre II – Modélisation des langages et Environnements de Méta-Modélisation

Chapitre III – Modélisation en UML/OCL des Propriétés Syntaxiques des Langages de Programmation

Chapitre IV – Modélisation en UML/OCL des Propriétés Comportementales et Axiomatiques des Langages de Programmation

Chapitre V – Diagrammes d'Activité et exemple de Propriétés Syntaxiques et Sémantiques

Chapitre VI – Vers un Processus IDM incrémental pour l'activité de Génération de Codes à partir d'un Diagramme d'Activité

Bilan – Perspectives

Bibliographie

Introduction - L'activité de Génération de Codes des Processus de Développement Logiciels IDM

Nous présentons, dans cette introduction, le cadre de cette étude portant sur l'activité de génération de codes se situant en phase finale d'un processus de développement dirigé par les modèles.

Après avoir repositionné l'activité de génération de codes dans le cadre d'un processus de développement de logiciels, nous montrons que de nombreux travaux de recherche, académiques et industriels, sont consacrés aux aspects méthodologiques des processus en proposant des démarches amenant les Analystes/Concepteurs à modéliser par étapes successives les exigences des systèmes à développer. Il s'en suit ainsi, lors du processus de développement des logiciels, une discontinuité entre les activités de modélisation relativement abstraites de manière à faciliter la communication entre les différentes équipes de développement impliquées dans le processus et les activités de génération de codes prenant en compte les aspects techniques des plates-formes cibles. D'où une certaine discontinuité entre ces deux types d'activités, la première devant rester abstraite, la deuxième pouvant être très technique.

Nous rappelons, dans cette introduction, les aspects méthodologiques d'une démarche qui pourrait être utilisée dans le cadre d'un processus de développement dont l'activité de modélisation est réalisée à l'aide du langage UML, devenu un standard de fait. Nous montrons ensuite les difficultés qui peuvent se poser lors de l'activité de génération de codes puisqu'il s'agit de transformer des artéfacts de modélisation, relativement abstraits en artefacts de programmation reflétant bien les intentions des Concepteurs et devant respecter strictement les propriétés syntaxiques et sémantiques des langages de programmation.

De manière à obtenir une meilleure continuité entre les activités chargées de l'élaboration des modèles et l'activité de génération des codes, nous proposons de nous inspirer des technologies des langages de programmation et des traducteurs pour les appliquer au niveau des langages de modélisation. Comme pour les langages de programmation définis à l'aide de propriétés syntaxiques et sémantiques, les experts d'un domaine d'applications peuvent donc intégrer au niveau du langage UML des propriétés syntaxiques et sémantiques pouvant tenir compte des spécificités du domaine d'applications.

Ces propriétés permettent de vérifier tout d'abord que les modèles issus des exigences des applications ont bien les qualités que l'on peut en attendre. Les propriétés syntaxiques et sémantiques rajoutées au niveau du langage de modélisation UML et les propriétés syntaxiques et sémantiques des langages cibles permettent ensuite de vérifier la cohérence entre les modèles et les codes.

Nous décrivons alors les différentes parties de cette recherche dont l'objectif est de contribuer à obtenir plus de continuité tout au long du processus de développement.

Introduction - L'activité de Génération de Codes des Processus de Développement Logiciels IDM

Table des matières

I Démarche de Modélisation en UML et Processus de développement ... 4

II L'Ingénierie Dirigée par les Modèles ... 5

II.1 Processus de Développement des Logiciels et Modèles ... 5

II.1.a Activités de Conception et Génération de Codes ... 6
II.1.b Activité de Génération des Codes ... 6
II.1.c Cohérence entre les Modèles et les Codes de Composants Logiciels ? ... 7

II.2 De la Technologie des Langages à la Technologie des Modèles ... 7

II.2.a Langages de Programmation et Langages de Modélisation ... 7
II.2.b Modélisation en UML/OCL des Langages de Programmation et de leurs Propriétés ... 9
II.2.c Intégration d'un niveau de Modélisation des Codes dans un Processus IDM ... 10

III Plan de la thèse ... 11

III.1 Première partie : Propriétés des Langages de Programmation et Environnements de Modélisation et de Méta-Modélisation UML ... 12

III.1.a Chapitre I : Propriétés Syntaxiques et Sémantiques des Langages de Programmation ... 12
III.1.b Chapitre II : Etat de l'art : Modélisation des langages, Modélisation et Méta-Modélisation ... 12

III.2 Deuxième partie : Modélisation en UML et OCL des Propriétés des Langages de Programmation ... 12

III.2.a Chapitre III : Modélisation en UML /OCL de la Grammaire des langages de programmation et de leurs Propriétés de Typage ... 13
III.2.b Chapitre IV : Modélisation en UML/OCL des Propriétés opérationnelles et axiomatiques des Langages de Programmation ... 13

III.3 Troisième partie : Vers un Processus IDM pour l'activité de génération de codes ... 14

III.3.a Chapitre V : Diagrammes d'Activité, et exemple de Propriétés Syntaxiques et Sémantiques ... 14
III.3.b Chapitre VI : Vers un Processus IDM pour l'Activité de Génération de codes à partir d'un Diagramme d'Activité ... 15

III.4 Conclusion, bilan, perspectives des travaux de recherche ... 15

IV L'Environnement thématique de recherche ... 16

Tables des figures

Figure 1 : Chaine complète de la démarche de modélisation des exigences jusqu'aux codes *4*
Figure 2 : Les deux activités d'un Processus de Développement de logiciels Dirigés par les Modèles *6*
Figure 3 : Méta-Modèle UML et Grammaire d'un Langage de Programmation *7*
Figure 4 : Injection des Propriétés Syntaxiques et Sémantiques en OCL et en LA, applicables à des Modèles UML *8*
Figure 5 : Modélisation des Codes *9*
Figure 6 : Modélisation de la Grammaire d'un Langage et d'un Programme du Langage *9*
Figure 7 : Processus intégrant un niveau de Modélisation des Propriétés du ou des Langages cibles *10*

Introduction - Cadre de l'Etude : L'activité de Génération de Codes des Processus de Développement Logiciels IDM

I Démarche de Modélisation en UML et Processus de développement

Le langage de modélisation UML[1] a pour objectif de modéliser les exigences des applications à développer à l'aide de différents types de diagrammes, chacun représentant graphiquement sans sémantique bien définie, un point de vue particulier de l'application. Une chaîne complète d'une démarche de modélisation en UML peut guider les Analystes/Concepteurs lors du processus de modélisation d'une application. La figure suivante peut être un exemple d'une démarche de modélisation [Aud] décrivant les activités du processus de modélisation d'une application depuis l'expression de ses besoins jusqu'à la réalisation des codes des composants logiciels, en passant par l'identification des besoins, la spécification des fonctionnalités et par des phases d'analyse et de conception logicielle :

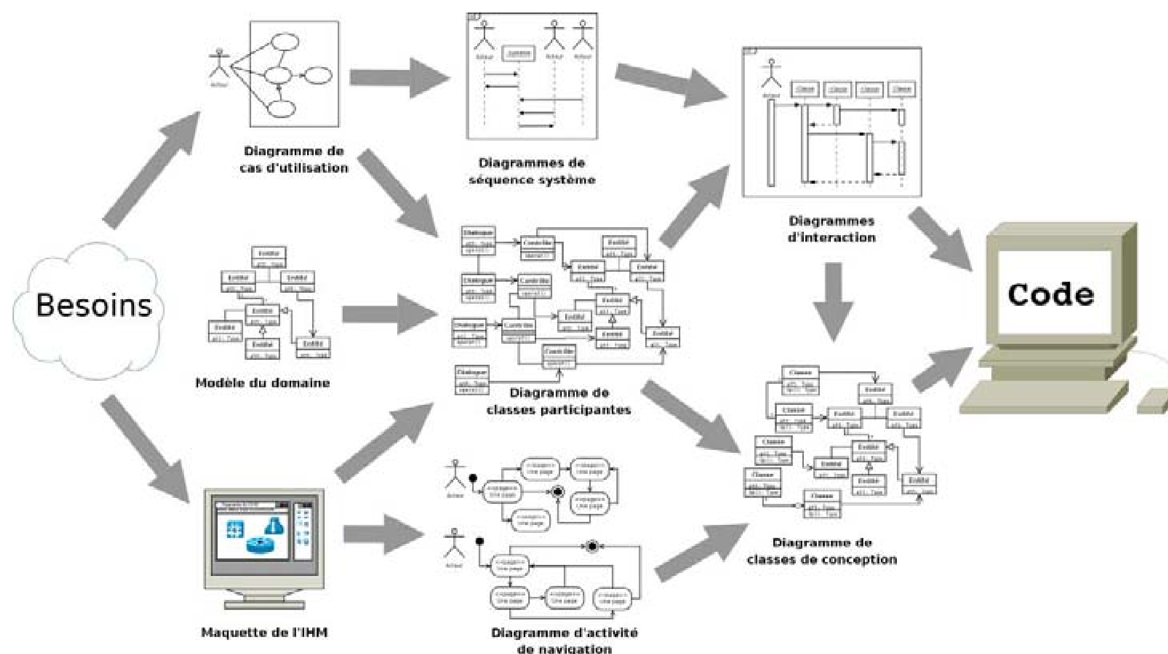


Figure 1 : ***Chaine complète de la démarche de modélisation des exigences jusqu'aux codes***

Sur cette figure, les différentes activités, phases ou étapes de la démarche préconisée sont représentées par des flèches, et pour chacune d'entre elles les diagrammes en entrée et en sortie de l'activité. Cette figure montre très schématiquement qu'il s'agit préalablement d'identifier les besoins des utilisateurs de l'application à développer et d'en spécifier ses fonctionnalités attendues à l'aide de diagrammes de cas d'utilisation. L'identification et la représentation des besoins se fait à l'aide de diagrammes de cas d'utilisation qui s'apparentent à une analyse fonctionnelle classique. Suit alors une spécification détaillée des besoins qui peut se faire sous la forme de diagrammes de séquences système. Une maquette de

[1] Unified Modeling Language (*http://www.uml.org/*)

l'application peut alors être initiée pour en montrer un aperçu et avoir un premier retour des utilisateurs concernés.

La phase d'analyse est centrée sur l'élaboration du diagramme de classes. Une analyse du domaine peut être réalisée afin d'identifier et de définir les classes et les liens entre elles, l'ensemble représentant les aspects métier du domaine de l'application. Le diagramme de classes participantes peut alors se déduire des cas d'utilisation, du modèle du domaine et de la maquette. Il a pour rôle d'assurer la transition entre ces diagrammes et la phase de conception logicielle modélisant les interactions et les classes. Cette phase d'analyse doit tenir compte de la maquette dont la modélisation devrait être réalisée lors de la spécification des fonctionnalités de manière à prendre en compte les interactions entre les futurs utilisateurs à l'aide des diagrammes d'activités de navigation.

La phase de conception, enfin, est réalisée à l'aide des diagrammes d'interaction issus des diagrammes de séquence système et des diagrammes de classes de conception. Ils serviront à l'implantation, en particulier lors de la production des codes des composants logiciels qui dépendent des plates-formes et des environnements cibles.

Cependant, UML reste un langage de modélisation, et cette démarche donnant un exemple d'utilisation et d'enchainement basique des différents diagrammes d'UML n'est ni une méthodologie, ni un processus de développement, comme le processus UP[2] ou RUP[3] par exemple, guidant l'équipe de développement tout au long du cycle de vie du projet, principalement lors des activités de gestion de projet.

II L'Ingénierie Dirigée par les Modèles

La démarche UML préconisée dans le paragraphe précédent, rentre dans un cadre plus large de processus de développement de logiciels où les modèles sont au centre du développement et où l'on cherche à déduire automatiquement les codes à partir des modèles. On parle d'une manière générale, d'Ingénierie Dirigée par les Modèles, dont on en rappelle dans ce paragraphe les idées principales.

II.1 Processus de Développement des Logiciels et Modèles

L'IDM[4] [MDA] est une méthode de développement des systèmes d'information dont l'objectif est la portabilité, l'interopérabilité et la réutilisabilité via une séparation architecturale des préoccupations. Elle a permis plusieurs améliorations significatives dans le développement en promouvant les notions d'abstraction, de réutilisation des savoir-faire et de séparation des préoccupations des Analystes/Concepteurs impliqués dans le processus de développement. L'IDM oriente le développement vers l'abstraction des systèmes par des modèles qui représentent une simplification des réponses de modélisation pour les exigences du système. Les modèles aident à encapsuler les différents aspects d'un système, et facilitent les échanges et les discussions entre différents participants au développement des systèmes d'informations.

[2] Unified Processus
[3] Rational Processus Unified
[4] Ingénierie Dirigée par les Modèles (MDE : Model Driven Engineering)

II.1.a Activités de Conception et Génération de Codes

D'où le découpage d'un processus de développement des logiciels en deux grandes activités, tel que le montre la figure suivante :

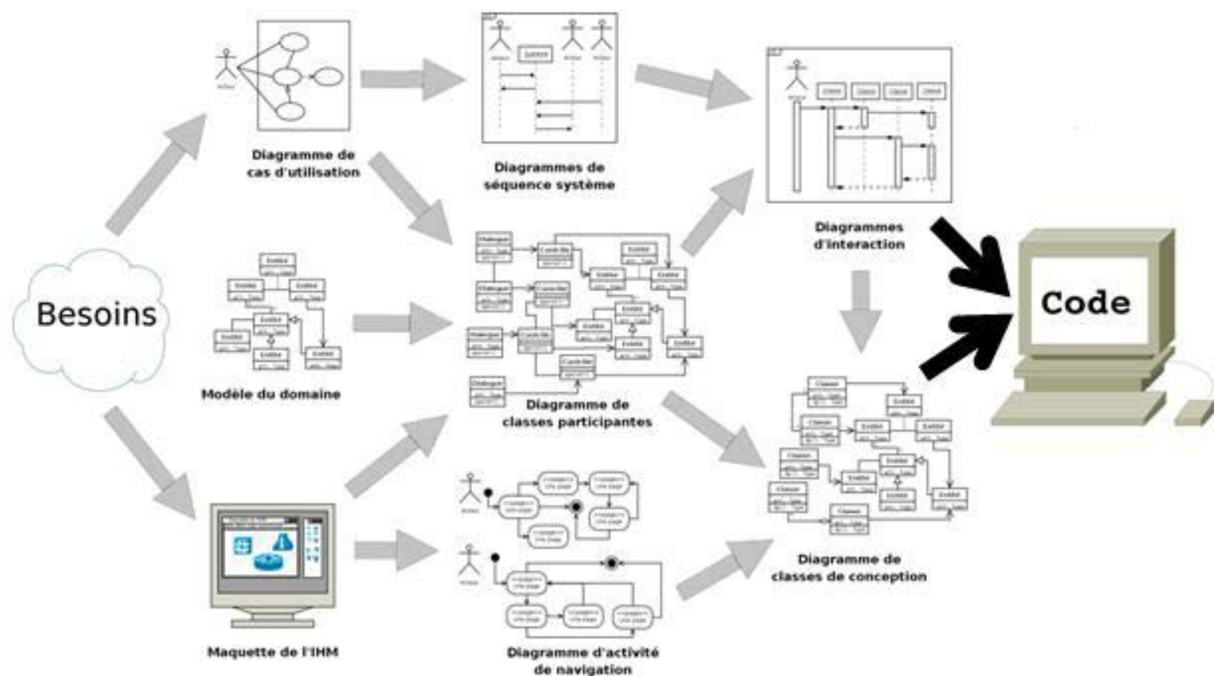


Figure 2 : ***Les deux activités d'un Processus de Développement de logiciels Dirigés par les Modèles***

Cette figure montre une première activité consacrée à la modélisation des exigences du système à développer pouvant être réalisée méthodologiquement par étapes successives en suivant une démarche particulière, celle dont on en a rappelé précédemment le principe général, et une seconde activité dédiée à la production des codes des composants logiciels.

Les activités de modélisation des exigences sont représentées par les flèches :

Les activités de génération des codes sont représentées par les flèches :

II.1.b Activité de Génération des Codes

Assuré d'une certaine qualité des modèles élaborés à l'aide d'environnements de modélisation et en appliquant méthodologiquement les démarches qui sont préconisées, la génération de codes a tendance devient une simple activité de transformation de modèles en codes de composants logiciels, réalisée directement à l'aide d'outils intégrés dans les éditeurs de modèles, assimilant ainsi les langages de modélisation à des langages de programmation de haut niveau.

Cependant cette phase d'élaboration des codes qui est réalisée à partir des modèles et que l'on souhaiterait être entièrement automatique, reste une activité des processus de développement des plus délicates. Il s'agit, en effet, de transformer des artéfacts de modélisation, basés sur le concept objet et relativement abstraits pour faciliter la communication entre les équipes impliquées dans le processus de développement, en artéfacts de programmation devant respecter strictement les propriétés syntaxiques et sémantiques des langages cibles. Pour des raisons pragmatiques évidentes, certains langages, en particulier les

DSL[5], spécifiques à des domaines d'applications bien ciblés, n'intègrent même pas le concept objet. Les codes doivent de plus répondre à des exigences non fonctionnelles destinées à faciliter les tests, à tracer les différentes évolutions des artefacts de modélisation au cours du processus. Ils doivent pouvoir être facilement maintenables de manière à pouvoir prendre en compte les différentes évolutions exigées par les utilisateurs. Ainsi, les codes doivent-ils répondre à des critères pouvant souvent être contradictoires avec des exigences non fonctionnelles, de performances par exemple, tout en suivant des règles de programmation et de documentation métier très strictes, spécifiques à chaque entreprise.

II.1.c Cohérence entre les Modèles et les Codes de Composants Logiciels ?

Si la modélisation des exigences d'une application réalisée dans le cadre d'une démarche bien définie reste donc l'activité essentielle des processus de développement, la génération de codes marque une coupure dans le processus entre les modèles contenant des artéfacts métier qui doivent rester abstraits et les codes qui sont écrits dans des langages dont les propriétés syntaxiques et sémantiques sont décrites formellement, voire selon une approche mathématique.

Comment dans ces conditions est-on sûr, ou peut-on être sûr, que les codes reflètent bien les intentions des concepteurs ?

Comment peut-on assurer la cohérence entre les modèles et les programmes correspondants, après maintes maintenances correctives et évolutives ?

II.2 De la Technologie des Langages à la Technologie des Modèles

Pour répondre à ces préoccupations, les travaux de recherche que nous présentons sont centrés sur l'activité de génération de code pouvant se situer en phase finale d'une démarche de modélisation UML.

II.2.a Langages de Programmation et Langages de Modélisation

UML est un langage de modélisation qui est défini par un Méta-Modèle de la même manière qu'un langage de programmation est défini préalablement par une grammaire. Tel que le montre la figure suivante, toute grammaire d'un langage de programmation est cependant complétée par un ensemble de propriétés syntaxiques et sémantiques donnant aux Analystes/Programmeurs une définition claire et nette, sans ambiguïté du langage de programmation :

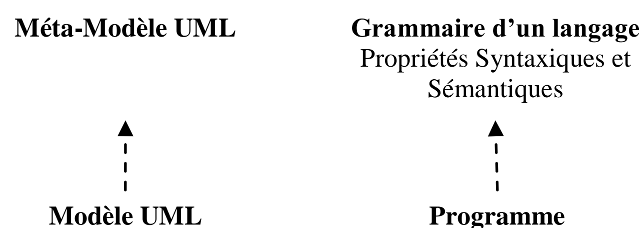


*Figure 3 : **Méta-Modèle UML et Grammaire d'un Langage de Programmation***

[5] Domain Specific Language

Cependant, UML est un langage graphique de modélisation, sans sémantique bien définie. L'idée de cette recherche est de rajouter au niveau du Méta-Modèle UML des propriétés syntaxiques et sémantiques à l'aide du langage de contraintes OCL[6], partie intégrante de la norme UML, et à l'aide d'un langage d'actions UML. La figure suivante montre donc le parallèle que l'on peut faire entre les technologies des langages de programmation et des langages de modélisation, en particulier le langage UML :

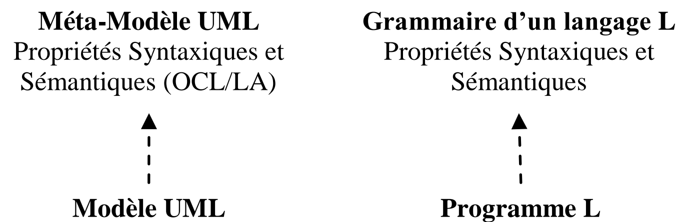


Figure 4 : ***Injection des Propriétés Syntaxiques et Sémantiques en OCL et en LA, applicables à des Modèles UML***

En rajoutant des propriétés syntaxiques et sémantiques au niveau du langage de modélisation UML pouvant dépendre du contexte applicatif, on peut offrir ainsi aux Analystes/Concepteurs une définition claire et nette, sans ambiguïté du langage de modélisation UML. Les modèles UML, issus des exigences du système à développer peuvent donc être vérifiés assurant qu'ils possèdent bien toutes les qualités requises. Les propriétés syntaxiques et sémantiques que l'on retrouve, dès lors au niveau conception et au niveau codes, permettent dès lors de vérifier la cohérence entre les modèles et les codes, assurant ainsi, lors de l'activité de génération des codes que les intentions des Analystes/Concepteurs ont été bien comprises et bien interprétées.

En cherchant à reporter et appliquer les technologies des langages de programmation au niveau du langage de modélisation UML, la recherche présentée dans ce document a pour objectif de contribuer à obtenir une meilleure continuité entre les modèles et les codes des composants logiciels.

Pour cela, nos travaux de recherche ont été réalisés en suivant une démarche en deux étapes :

- La première, à la fois plus technique et plus formelles, puisqu'elle a pour objectif de modéliser, en UML/OCL et dans un langage d'actions UML, les langages de programmation et leurs propriétés syntaxiques et sémantiques.
- La deuxième, plus méthodologique, puisqu'il s'agit d'insérer au niveau du processus de développement une activité de modélisation des codes des composants logiciels avant leur génération proprement dite de manière à vérifier, au niveau des modèles de codes, qu'ils ont bien les qualités requises.

[6] Object Constraint Language (*http://www.uml.org/*)

II.2.b Modélisation en UML/OCL des Langages de Programmation et de leurs Propriétés

Il s'agit, tout d'abord, de voir d'une manière générale comme on peut modéliser en UML un programme, tel que le montre la figure suivante :

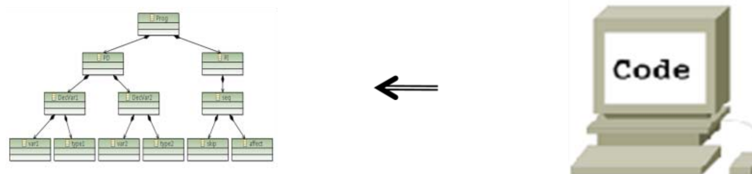


Figure 5 : ***Modélisation des Codes***

Le code étant écrit dans un langage qui est défini par une grammaire complétée par les propriétés syntaxiques et sémantiques, nous proposons, donc, de modéliser en UML et à l'aide du langage de contraintes OCL les propriétés syntaxiques et sémantiques des langages de programmation, tel que le montre la figure suivante :

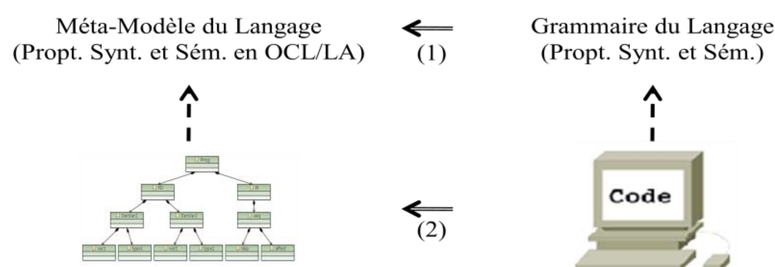


Figure 6 : ***Modélisation de la Grammaire d'un Langage et d'un Programme du Langage***

Il s'agit de modéliser en UML et en OCL, la grammaire d'un langage et les propriétés syntaxiques et sémantiques du langage (1). La modélisation de la grammaire et des propriétés d'un langage de programmation s'appelle le Méta-Modèle du langage. Tout programme du langage peut donc être modélisé (2) en tant d'instance du Méta-Modèle de la grammaire du langage et de ses propriétés. Il s'agit donc de proposer une méthode systématique et pragmatique de modélisation de la sémantique des langages de programmation fondée sur la description en UML et OCL [Male02], en se basant sur la précision mathématique de la sémantique dénotationnelle et sur l'accessibilité et le rôle prépondérant des définitions en UML/OCL.

Cette partie a donc pour objectif de transposer en UML/OCL le formalisme décrivant formellement les langages de programmation et leurs propriétés de manière à communiquer aux Analystes/Programmeurs une définition précise des langages de programmation. Basées sur la logique des prédicats du premier ordre, ces spécifications écrites en OCL et en LA[7] compenseront le peu de sémantique que peut offrir une représentation structurelle des

[7] Langage d'Actions UML

diagrammes UML. Les analystes/concepteurs peuvent ainsi s'assurer au niveau des modèles que les composants logiciels vérifient les propriétés des langages dans lesquels ils sont écrits.

Contrairement aux traducteurs qui sont considérés comme des boites noires, les modèles UML sont accessibles aux Analystes/Concepteurs. Il leur est aussi possible d'injecter de nouvelles propriétés, façonnant ainsi des environnements de modélisation en fonction de critères pouvant prendre en compte des spécificités métier du domaine d'applications.

II.2.c Intégration d'un niveau de Modélisation des Codes dans un Processus IDM

Afin d'obtenir plus de continuité entre les modèles qui doivent rester relativement abstraits, et les codes qui doivent respecter strictement les propriétés des langages dans lesquels ils sont écrits, nous proposons dans une deuxième étape, tel que le montre la figure suivante, d'intégrer préalablement à l'activité de génération proprement-dite des composants logiciels, un niveau de modélisation en UML/OCL du ou des langages cibles :

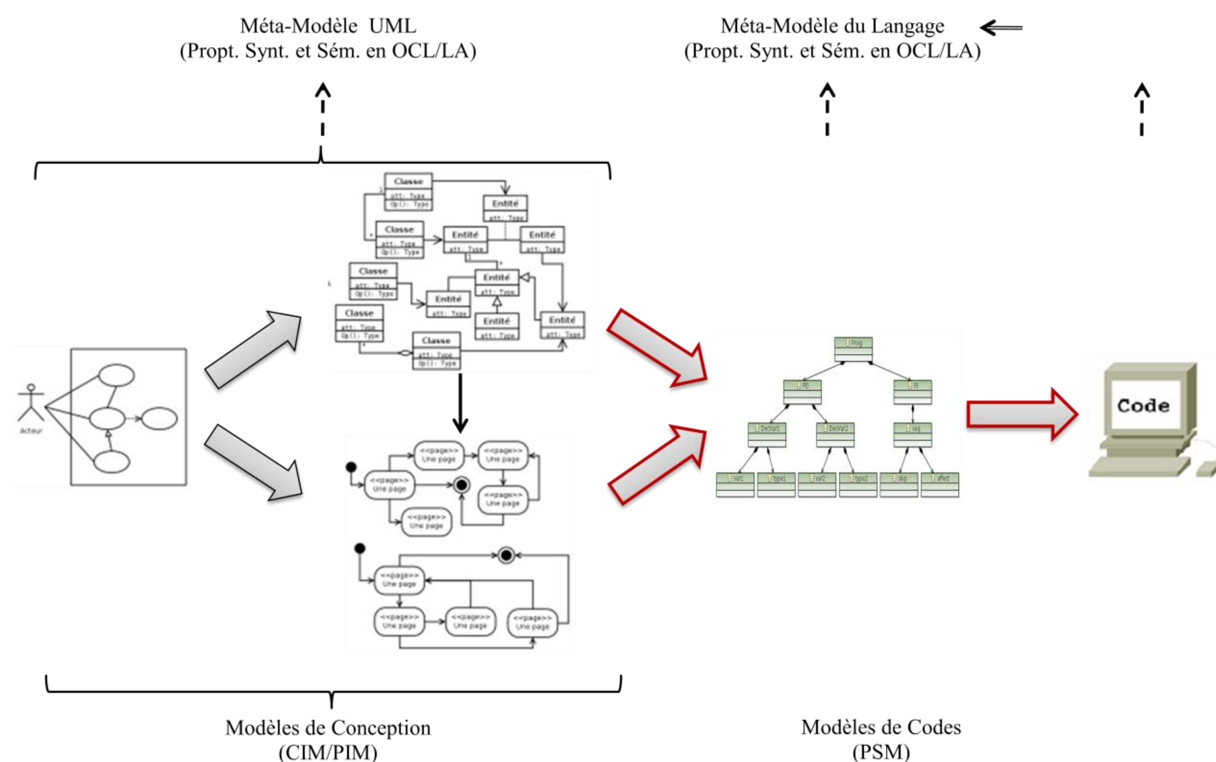


Figure 7 : ***Processus intégrant un niveau de Modélisation des Propriétés du ou des Langages cibles***

Le processus de développement dirigé par les modèles est ainsi prolongé jusqu'à ce niveau de modélisation des composants en faisant apparaître différents types de modèles : les *modèles de conception* intégrant au cours du processus les artéfacts déduits des exigences des applications, et les *modèles de composants* constitués d'artéfacts de programmation issus des artéfacts de modélisation. Les modèles de conception sont décrits syntaxiquement par le Méta-Modèle du langage UML (MM UML) ; les modèles de programmation par le Méta-Modèle du langage (MM L) déduit de la grammaire du langage cible et de ses propriétés.

Les modèles des composants se trouvant ainsi à la charnière entre les activités de conception des exigences et les activités de mise en œuvre des applications, il est ainsi possible aux Analystes/Concepteurs de vérifier la cohérence entre les modèles et les codes, et de s'assurer, en particulier, que les exigences des applications ont pu être prises en compte.

Un tel processus reprend tout naturellement l'architecture MDA[8], définie dans le cadre de l'IDM dont le niveau CIM[9] décrit d'une manière plus ou moins formelle les exigences des applications, et dont le niveau PIM[10], résultat de l'analyse fonctionnelle, doit rester, autant que l'on peut, indépendant des aspects techniques des plates-formes cibles. Ces deux niveaux ont été réunis dans la figure précédente. Le niveau PSM[11] est la prise en compte dans les modèles de conception des spécificités des plates-formes cibles, en l'occurrence les propriétés des langages de programmation, reconnaissables par leurs structures hiérarchisées.

Selon le schéma proposé, un processus IDM est donc constitué de trois grandes activités. L'activité classique de modélisation des exigences réalisée méthodologiquement par étapes successives d'intégration des artéfacts issus des exigences ; une activité de transformation des artéfacts de modélisation en artéfacts de programmation, réalisée en s'inspirant des techniques des traducteurs ; et une activité de génération de codes après avoir injecté dans les modèles de programmation les éléments de syntaxe concrète. L'intégration d'un niveau de modélisation des codes des composants logiciels avant leur génération assure d'une manière cohérente et uniforme avec démarche UML, la continuité entre les modèles et les codes. L'objectif est de pouvoir aider et assister les analystes/concepteurs à vérifier que les intentions des concepteurs ont bien été comprises et prises en compte lors de la production des codes des composants.

En appliquant les techniques de raffinage de codes au niveau des modèles, il est dès lors possible de réaliser cette activité de génération de codes par une succession de raffinages de modèles depuis les modèles de conception, les modèles les plus abstraits du processus IDM issus des exigences des applications, jusqu'aux modèles les plus proches des codes, contenant donc les artefacts de programmation les plus concrets.

Une telle activité de génération de codes s'intègre donc parfaitement dans un processus IDM considéré comme une succession de transformations de modèles et composé de deux grandes activités :

- L'élaboration des modèles de conception réalisée par étapes successives complétant et enrichissant les modèles à chaque prise en compte des exigences.
- La génération de codes, à partir des modèles de conception, réalisée par étapes de raffinage successives.

III Plan de la thèse

Nous présentons donc dans ce document nos travaux de recherche principalement axés sur l'activité de génération de composants logiciels se situant en phase terminale des processus de développement de logiciels dirigés par les modèles. Il s'agit de montrer comment nos travaux de recherche ont contribué à obtenir plus de continuité entre les activités de modélisation des exigences et de génération de codes.

[8] Model Driven Architecture
[9] Computation Independant Model
[10] Plate-form Independant Model
[11] Plate-form Specific Model

III.1 Première partie : Propriétés des Langages de Programmation et Environnements de Modélisation et de Méta-Modélisation UML

Cette première partie est consacrée à des rappels concernant les propriétés des langages de programmation, les travaux de recherche déjà existants dans le domaine de la modélisation de ces propriétés en UML/OCL, et concernant les principales caractéristiques des différents concepts de Modélisation et de Méta-Modélisation UML qui devront être respectées en vue d'être appliquées et re-utilisées lors d'un processus de développement IDM où sont impliquées différentes équipes d'Analystes/Concepteurs aux origines pouvant être différentes.

III.1.a Chapitre I : Propriétés Syntaxiques et Sémantiques des Langages de Programmation

Le **chapitre I** est consacré à l'étude des propriétés des langages de programmation. Il s'agit de rappeler qu'elles sont définies préalablement à l'aide d'une grammaire à partir de laquelle les propriétés syntaxiques et sémantiques sont spécifiées généralement sous la forme de règles d'inférence basées sur la logique qui en donnent une définition très précise. Nous en présentons les principes généraux qui seront repris lors de leur modélisation en UML, et lors de leurs applications par les Analystes/Concepteurs au cours d'un processus de développement de logiciels pour en vérifier le bon déroulement. Le langage de programmation pris comme exemple est un langage de type procédural, appelé langage « L ».

III.1.b Chapitre II : Etat de l'art : Modélisation des langages, Modélisation et Méta-Modélisation

Il existe déjà des travaux de recherche modélisant en UML des langages de programmation. Le **chapitre II** présente succinctement ces travaux qui ont servi de base à notre étude. Cependant, devant s'intégrer dans des processus de développement dirigés par les modèles, ce chapitre se poursuit par une présentation des environnements de Modélisation et de Méta-Modélisation définis par l'OMG. Ils ont servi de cadre aux maquettes que nous avons développées pour modéliser en UML et OCL les propriétés des langages de programmation. Une présentation des projets de recherche cadre dans lesquels cette étude a été réalisée termine ce chapitre.

III.2 Deuxième partie : Modélisation en UML et OCL des Propriétés des Langages de Programmation

Les chapitres III et IV présentent les concepts et les techniques que nous avons appliqués pour modéliser en UML et OCL les langages de programmation et leurs propriétés. Ces propriétés sont spécifiées et mises en œuvre en tenant compte des environnements de modélisation UML intégrant un évaluateur d'expressions OCL et d'un langage d'actions UML. Il s'agit de permettre aux Analystes/Concepteurs d'exécuter des modèles ou des fragments de modèles ; les aidant, par exemple, à mettre en œuvre les premières maquettes proposées au cours de la démarche de modélisation, donnant ainsi un premier aperçu dès la conception de ce que pourra être l'application.

III.2.a Chapitre III : Modélisation en UML /OCL de la Grammaire des langages de programmation et de leurs Propriétés de Typage

A partir de la grammaire d'un langage, nous montrons, dans le **chapitre III**, comment nous en déduisons le modèle UML, appelé le Méta-Modèle du langage, prenant en compte les différentes constructions syntaxiques du langage complétées par un certain nombre de propriétés non exprimables structurellement. La modélisation des propriétés de typage est étudiée dans ce chapitre dans la mesure où, comme pour des programmes, elles permettent d'assimiler les modèles à des formules bien formées.

III.2.b Chapitre IV : Modélisation en UML/OCL des Propriétés opérationnelles et axiomatiques des Langages de Programmation

Le **chapitre IV** est consacré, tout d'abord, à la modélisation des propriétés comportementales pour l'animation de fragments de modèles ou de modèles plus complets si l'on dispose d'un environnement d'exécution. Il s'agit ensuite de modéliser les propriétés axiomatiques du langage permettant ainsi de vérifier que des fragments de modèles sont corrects par rapport à des propriétés rajoutées dans les modèles sous la forme de pré-conditions et de post-conditions. L'ensemble d'un fragment d'un modèle décoré de propriétés de type pré-conditions et post-conditions est appelé un triplet de Hoare. Vérifier qu'un triplet de Hoare est valide se fait à l'aide d'un Atelier B, spécialisé dans la preuve de programme. Les propriétés de typage, et les propriétés sémantiques des langages sont spécifiées en OCL, sous la forme de fonctions d'un langage d'action avec effet de bord, de manière à pouvoir manipuler des types comme des objets. Nous définissons une sémantique du concept de triplet de Hoare, ce qui autorise sa transformation en B, et ce qui donne aussi la possibilité d'exécuter au niveau modèle des triplets de Hoare, à partir de valeurs pouvant être judicieusement définis par les Concepteurs. Nous reprenons le langage « L » pour spécifier et décrire ses propriétés syntaxiques et sémantiques.

Contrairement aux traducteurs où ces techniques de modélisation de codes restent transparentes aux programmeurs, les Analystes/Concepteurs peuvent dès lors agir sur les modèles de programmes pour insérer, par exemple, des propriétés assurant, avant la génération de code, qu'ils ont bien les propriétés et les qualités requises.

Le passage d'une modélisation UML à une machine B se fait en deux étapes. Il s'agit tout d'abord de modéliser en B les constructions syntaxiques de la grammaire du langage, en l'occurrence le langage « L ». Cette étape peut se faire automatique soit à partir des règles de construction syntaxique de la grammaire du langage, soit à partir du Méta-Modèle UML. Il existe dans ce domaine de nombreux travaux de recherche. Il s'agit ensuite de prendre en compte les propriétés syntaxiques et sémantiques pour les exprimer en B. Ces propriétés étant écrites en OCL, c'est donc un problème de compilation. En fait, nous montrons comment nous pouvons rendre le plus transparent possible aux Analystes/Concepteurs le passage de modèles UML à des modèles de machines B, en particulier lorsque les types prédéfinis du langage cible et du langage B se correspondent. Dans ce cas, il n'est pas nécessaire de compiler en B les expressions OCL représentant les propriétés comportementales du langage de programmation.

III.3 Troisième partie : Vers un Processus IDM pour l'activité de génération de codes

Nous montrons comment les méthodes et les techniques décrites dans la partie précédente peuvent être appliquées dans le cadre d'une démarche UML vérifiant, à chacune de ses phases essentielles, que les modèles respectent les propriétés qui leurs sont imposées. Il s'agit de montrer, en particulier, comment on peut assurer que les transformations assurant le passage des modèles de conception à des modèles de programmation sont correctes et conservent les propriétés, vérifiant ainsi la cohérence entre les modèles de niveau conception et de niveau programmation.

Nous prendrons comme exemple le diagramme d'activité UML qui peut être utilisé pour élaborer des modèles de conception à partir desquels l'activité de génération de codes peut être effectuée. Le diagramme d'activité peut être utilisé à différents niveaux d'abstraction pour représenter les actions d'un système à développer et les différents enchaînements de ces actions. Il s'agit ensuite de montrer comment il est possible de transformer un diagramme d'activité en un modèle de programmation, dont le langage cible est le langage « L ». Nous nous limitons à la modélisation d'algorithmes séquentiels pour rester cohérent avec les deux premières parties du document.

Nous montrons, enfin, un exemple d'un modèle de conception réalisé à l'aide d'un diagramme d'activité intégrant les interactions entre les différents usagers (les swimlanes), les différentes actions et leurs enchainements d'un système à développer. Un tel modèle a été étudié dans le cadre du projet ANR MyCitizSpace.

Une description des environnements de Méta-Modélisation termine ce document. Parmi ces environnements, on peut citer, tout d'abord, la plate-forme USE, de l'Université de Brême, qui est un évaluateur d'expressions OCL et d'un langage d'actions UML qui a été utilisé pour modéliser en UML/OCL/LA les propriétés des langages de programmation et les propriétés que les experts d'un domaine métier d'applications souhaitent intégrer au niveau du langage de modélisation UML. Nous avons ensuite utilisé les plates-formes Kermeta et Topcased permettant l'échange de modèles UML. Le langage Java/EMF aurait pu être utilisé pour la mise en œuvre des algorithmes qui ont été présentés, mais, n'intégrant pas l'interprète OCL, posait des difficultés pour l'écriture de spécifications OCL.

III.3.a Chapitre V : Diagrammes d'Activité, et exemple de Propriétés Syntaxiques et Sémantiques

Le **chapitre V** est consacré à l'étude du diagramme d'activité qui a beaucoup évolué dans la version 2.0 de UML. Il peut être utilisé pour modéliser l'algorithme d'une méthode d'une opération définie dans une classe. L'algorithme décrit sous la forme d'un organigramme composé d'un ensemble d'activités élémentaires et de leurs différents enchainements. Un tel modèle qui peut être enrichi au fur et à mesure de la prise en compte des exigences, est assimilé à un modèle abstrait de plus haut niveau qu'un modèle de programmation relativement contraignant pour les Analystes/Concepteurs dans la mesure où il faut prévoir, dès le départ, l'ordonnancement hiérarchique des différentes actions à réaliser. Nous définissons ensuite les propriétés comportementales injectées au niveau du Méta-Modèle UML pour être applicables sur un diagramme d'activité. Cependant, ces propriétés dépendent des éléments de modélisation qui les contiennent, par exemple les variables. La

spécification des propriétés dépendent donc du niveau d'abstraction où se situe le diagramme d'activité.

D'où la nécessité de définir un Processus IDM où l'on définit deux niveaux d'abstraction du diagramme d'activité. Un premier niveau relativement abstrait permettant de modéliser l'ensemble des actions que le système doit réaliser et leurs enchainements, puis un deuxième niveau, plus contraignant pour les Concepteurs, puisqu'il s'agit de modéliser sous la forme d'une structure arborescente les actions à réaliser. Selon le niveau d'abstraction la spécification des propriétés peut s'avérer différentes. L'idée de définir un Processus IDM pour la génération de codes est d'aider et d'assister les Analystes/Concepteurs lors de cette activité de génération de codes, en particulier de vérifier qu'à chaque niveau de raffinage d'un modèle à un autre, la transformation est correcte.

III.3.b Chapitre VI : Vers un Processus IDM pour l'Activité de Génération de codes à partir d'un Diagramme d'Activité

Ce **chapitre VI** présente donc les principales caractéristiques d'un tel processus. Il s'agit, en particulier, de définir les spécifications des propriétés comportementales du diagramme d'activité à différents niveaux d'abstraction, et de vérifier que le passage d'un niveau de modélisation au suivant est correct.

Nous montrons dans ce chapitre un exemple de diagramme d'activité prenant en compte les interactions (les swimlanes) entre différents usagers et différentes actions et leurs enchainements d'un système à développer. Cette étude est réalisée dans le cadre du projet MyCitizSpace. Cet exemple a pour objectif de montrer comment il est possible de modéliser les droits d'accès des usagers à des documents pouvant évoluer dans le temps au fur et à mesure de leurs différents accès. Les propriétés définissant les autorisations d'accès, à l'aide de propriétés formelles, constituent l'une des caractéristiques importantes de cet exemple, dans la mesure où elles contribuent à montrer, voire démontrer, que les droits d'accès aux documents, qui peuvent contenir des informations personnelles et privées, sont bien respectés. Le projet MyCitizSpace qui vise à doter les administrations locales (Collectivités Locales et Territoriales) et centrales (Ministères, Organismes Nationaux) d'outils logiciels performants et agiles, garants, pour le citoyen, d'IHM de qualité, satisfaisant en particulier les exigences de sécurité, ubiquité et accessibilité. L'approche explorée dans le projet est une Ingénierie Dirigée par les Modèles (IDM). L'ambition est de produire un atelier IDM permettant aux différents acteurs (Maîtrise d'ouvrage, Maîtrise d'œuvre, Expert Métier, CNIL) de produire, de façon itérative et collaborative, des télé-procédures efficaces, capables de s'adapter aux dispositifs d'interaction de l'utilisateur (PC, téléphone, etc.) dans le respect des propriétés attendues (fonction et ergonomie). Ce deuxième exemple nous a servi d'application de l'étude présentée dans ce document.

III.4 Conclusion, bilan, perspectives des travaux de recherche

Ce travail a été initié dans le cadre du projet de l'ANR DOMINO[12] qui avait pour objectif de proposer une démarche basée sur la description d'un système par divers modèles exprimés dans des langages de modélisation dédiés différents, en exploitant l'Ingénierie Des Modèles (IDM ou en anglais MDA/MDE pour Model Driven Architecture/Engineering) pour

[12] DOMaINes et prOcessus méthodologique

fiabiliser tout processus d'ingénierie accompagnant le développement de logiciels. L'étude décrite dans ce document a fait l'objet du lot 4 portant sur la modélisation des propriétés syntaxiques et sémantiques des langages de programmation. Nous montrerons donc les retours d'expérience que nous avons pu en tirer.

Démontrer, au travers des triplets de Hoare, qu'un fragment de modèle est correct par rapport à des propriétés analogues à des pré-conditions et des post-conditions semble être l'une des parties la plus intéressante de l'étude puisqu'il s'agit de vérifier que les modèles ont les qualités requises et que les transformations de modèles sont correctes. Cependant, la mise en place de triplets reste relativement difficile puisque qu'il s'agit de bien maitriser la logique, et souvent d'aider l'Atelier B qui se charge de la preuve, exigeant donc, pour les Experts d'un d'un domaine métier de passer d'un environnement de modélisation à un autre. Cependant, un important travail est fait pour rappeler les principes généraux des triplets de Hoare et pour les appliquer au niveau des modèles. En fait, l'appel à l'Atelier B peut se faire très facilement dans le cas où les types prédéfinis des langages que l'on modélise correspondent aux types prédéfinis de B. En fait, sans aller jusqu'à la preuve qui se fait statiquement sur du code, donc dans notre cas sur des modèles, on peut utiliser l'évaluateur d'un langage d'actions UML pour vérifier, dynamiquement sur quelques données choisies en conséquence, qu'un triplet de Hoare est valide signifiant que le code correspondant est correcte par rapport à ses prè- et post-conditions. Ce qui peut satisfaire (partiellement) les Concepteurs.

Compte tenu de ce que nous avons réalisé, nous donnons les différentes pistes des recherches qui prolongeraient cette étude, en les positionnant par rapport aux travaux de recherche qui se font actuellement.

IV L'Environnement thématique de recherche

Nous présentons brièvement dans ce paragraphe l'environnement de recherche qui a influencé nos travaux de recherche. En particulier, un rapide tour d'horizon des travaux de recherche que nous re-détaillerons au fur et à mesure dans le document, permet de cibler les thèmes de recherche abordés dans ce document.

Les références bibliographiques des travaux présentés dans ce document se situent dans le cadre des activités du Groupement De Recherche CNRS de l'IDM et du Génie de la Programmation et du Logiciel (Actes des journées nationales de Toulouse en 2009 et de Pau en 2010), et dans le cadre de projets européens (Neptune), de projets du Ministère de l'Industrie (projet ANR DOMINO, en particulier), et de projets du pôle de compétitivité Aerospace (projet TopCased, en particulier).

En raison des objectifs que nous nous étions fixés, nous nous sommes documentés principalement dans le domaine des langages et des traducteurs, des modèles et des processus IDM, ainsi que dans le domaine des méthodes formelles pour la réalisation de logiciels corrects par construction. Les propriétés des langages sont spécifiées en OCL, pour rester dans la logique du langage de modélisation 'unifié' de UML. En particulier, nous avons cherché à rendre le plus transparent possible l'appel à la méthode B pour éviter aux analystes/concepteurs de passer d'un environnement de modélisation UML à un atelier B. Seules les obligations de preuves qu'il est nécessaire de prouver pour vérifier la cohérence des codes par rapport aux propriétés qui leur sont imposées sont calculées en OCL/LA, et traduites ensuite en B. Les analystes/concepteurs devront peut-être aider les ateliers B à les

valider, mais ces assertions B restent dans le formalisme décrit par les grammaires qui servent à décrire les propriétés syntaxiques et sémantiques des langages. Les articles de recherche qui nous ont les plus marqués sont, en particulier, dans les domaines suivants :

- La modélisation des langages de programmation et leurs propriétés,
- Les langages de description des propriétés appliquées aux langages de programmation et la prise en compte des propriétés axiomatiques,
- La modélisation des langages en UML/OCL,
- La vérification et preuves des transformations appliquées à des structures arborescentes,
- Les processus incrémentaux,
- L'ingénierie dirigée par les modèles, la méta-modélisation, les méta-modèles, fragments de méta-modèles, les transformations de modèles,
- Le couplage UML et B,
- …

Principaux mots-clés : Processus de développement de logiciels, Modèles, UML, OCL, Langage d'actions, Langage de programmation, Propriétés statiques et dynamiques des langages, DSL, DSML, Atelier B.

I – Cadre de l’étude : Les Propriétés Syntaxiques et Sémantiques des Langages de Programmation

Un langage de programmation est défini formellement par un ensemble de propriétés syntaxiques et sémantiques. L’objectif de ces propriétés est de fournir des méthodes pour raisonner sur les programmes en cherchant à prouver qu’ils vérifient toutes les propriétés que l’on peut exiger de leur part. La sémantique d’un langage de programmation repose donc sur une définition de propriétés formelles décrivant les aspects syntaxiques, opérationnels et axiomatiques de tout programme du langage.

Un langage de programmation est plus précisément défini à l’aide :

- d’une grammaire décrivant les symboles de base du langage et la structure des différentes constructions syntaxiques que doit respecter tout programme du langage, ainsi considéré comme un ensemble de formules bien formées.
- d’un ensemble de relations logiques décrivant le comportement attendu des différentes constructions syntaxiques du langage, sur des données dont on en connait a priori les caractéristiques opérationnelles.
- d’un ensemble de propriétés axiomatiques permettant de raisonner directement sur des fragments de codes contraints (‘décorés’) par des pré- et post-conditions.

Nous rappelons brièvement l’ensemble de ces notions que nous illustrerons sur un exemple classique de langage de style impératif, relativement simple, mais suffisamment significatif et complet pour montrer comment ces différentes notions sont appliquées. On appellera ce langage, le langage L.

Ces rappels sont extraits du cours [Moh] d’où le langage que nous avons pris en référence pour décrire la grammaire et les propriétés des langages est issu, et d’une manière plus générale de [Aho98] et principalement de [MyerB92] où est décrite de façon très détaillée la théorie sur les langages de programmation. Il existe aussi de nombreux travaux de recherche sur la génération de codes qui font donc référence aux grammaires des langages et de leurs propriétés syntaxiques [MullP05b, MullP06, Mull08,…]. Les travaux de recherche les plus proches dans ce domaine des propriétés des langages de programmation concerne aussi [Kerm], puisqu’ils traitent à la fois des grammaires et des modèles.

Ces notions constituent le socle de base de la modélisation en UML/OCL des langages de programmation puisqu’elles permettent de proposer aux Analystes/Concepteurs une démarche rigoureuse au niveau de la génération des codes. Ces notions seront en fait appliquées à tous les niveaux de la démarche, depuis la prise en compte des exigences. Il est essentiel, donc, d’en rappeler les principes fondamentaux de manière à justifier les techniques montrant comment les propriétés des langages de programmation seront, par la suite, modélisées en UML et OCL. D’autre part, ces propriétés, spécifiées en UML et OCL, sont accessibles aux utilisateurs, contrairement aux traducteurs qui sont considérés comme des boites noires. Elles peuvent donc être modifiées par les experts d’un domaine d’applications qui pourront les compléter, façonnant ainsi les langages en fonction d’exigences métier, par exemple, spécifiques à un domaine d’applications.

I – Cadre de l'étude : Les Propriétés Syntaxiques et Sémantiques des Langages de Programmation

Table des matières

I Grammaire d'un langage de programmation ... 22
I.1 Eléments de base d'une grammaire d'un langage de programmation ... 22
I.1.a Symboles et règles de constructions syntaxiques d'une grammaire ... 22
I.1.b Règles 'Et' et 'Ou' d'une grammaire ... 22
I.2 Exemple d'une grammaire d'un langage : le langage « L » ... 23
I.2.a Caractéristiques élémentaires du langage ... 23
I.2.b Grammaire du langage ... 24
I.2.c Exemple d'un programme ... 25
II Propriétés des langages de programmation ... 25
II.1 Propriétés de typage ... 25
II.1.a Propriétés de typage des expressions ... 26
II.1.b Propriétés de typage des instructions ... 27
II.2 Sémantique opérationnelle du langage ... 28
II.2.a L'environnement d'exécution d'un programme ... 28
II.2.b Propriétés comportementales des expressions ... 29
II.2.c Propriétés comportementales des instructions ... 30
II.3 Sémantique axiomatique ... 31
II.3.a Triplet de Hoare, code correct et Assertion ... 31
II.3.b Règles définissant les propriétés axiomatiques pour les instructions Skip et Test ... 34
II.3.c Opération de substitution et règle d'inférence pour l'instruction d'affectation ... 35
II.3.d Correction partielle, correction totale ... 36
III L'Environnement de Modélisation et de Méta-Modélisation de UML ... 38
III.1 L'environnement des données gérées par les traducteurs ... 39
III.1.a Programme source et représentation interne des données ... 39
III.1.b Syntaxe concrète et concrète ... 40
III.2 L'environnement des éditeurs de langage de modélisation ... 40
III.2.a Modèle et Méta-Modèle ... 40
III.2.b Abstraction et raffinage de modèles ... 41

I – Cadre de l'étude : Les Propriétés Syntaxiques et Sémantiques des Langages de Programmation

Table des figures

Figure 1 : Grammaire du Langage « L » ___ 24
Figure 2 : Exemple d'un programme écrit en langage « L », calculant la racine carrée entière approchée du nombre entier x ___ 25
Figure 3 : Règle définissant le typage d'une expression ___ 26
Figure 4 : Axiomes définissant le typage des constantes ___ 26
Figure 5 : Règles définissant le typage d'une variable ___ 26
Figure 6 : Règles définissant le typage des expressions ___ 26
Figure 7 : Règle définissant le typage d'une instruction ___ 27
Figure 8 : Règles définissant le typage des instructions ___ 27
Figure 9 : Propriétés de l'environnement d'exécution, et correspondances entre les symboles et l'environnement ___ 28
Figure 10 : Propriété définissant le comportement d'une expression ___ 29
Figure 11 : Règles définissant les propriétés comportementales des expressions ___ 29
Figure 12 : Propriété définissant le comportement d'une instruction ___ 30
Figure 13 : Règles définissant les propriétés comportementales des instructions du langage ___ 30
Figure 14 : Triplet de Hoare ___ 31
Figure 15 : Assertion associée à un triplet de Hoare ___ 32
Figure 16 : Extension de la grammaire du langage « L » pour la prise en compte des assertions du triplet de Hoare ___ 33
Figure 17 : Règle de conséquence ___ 34
Figure 18 : Règles d'inférence des propriétés axiomatiques pour la séquence et le test ___ 35
Figure 19 : Règle d'inférence définissant la sémantique axiomatique pour l'instruction d'affectation ___ 36
Figure 20 : Règle d'inférence décrivant la sémantique axiomatique pour la boucle à l'aide d'un invariant de boucle ___ 37
Figure 21 : Obligations de preuve pour une boucle ___ 37
Figure 22 : Invariant et variant de boucle du programme calculant la racine carrée entier d'un nombre entier 38
Figure 23 : Espace technologique des structures de données d'un traducteur de langage de programmation _ 40
Figure 24 : Environnement de Modélisation d'un éditeur ___ 41

I – Cadre de l'étude : Les Propriétés Syntaxiques et Sémantiques des Langages de Programmation

I Grammaire d'un langage de programmation

Un langage de programmation est formellement défini préalablement par une grammaire qui décrit les symboles de base du langage et les règles syntaxiques que devront respecter tout programme du langage. Nous rappelons, dans ce paragraphe, les principes généraux concernant les grammaires des langages de programmation que nous illustrerons sur un langage classique de type procédural. C'est à partir de la grammaire du langage que les propriétés du langage peuvent être explicitées.

I.1 Eléments de base d'une grammaire d'un langage de programmation

L'objectif étant de modéliser les codes des composants logiciels issus des modèles de conception, nous nous limitons à la description de la grammaire abstraite des langages ne contenant pas les détails de syntaxe concrète nécessaires lors de la conception des codes et lors du décodage des programmes par les traducteurs. Dans ces conditions, la description syntaxique des chaînes de caractères décrivant les nombres entiers, réels ou booléens est replacée par les types de données de base (datatype), appelés aussi types primitifs.

I.1.a Symboles et règles de constructions syntaxiques d'une grammaire

La grammaire d'un langage de programmation est constituée d'un ensemble des symboles (S) correspondant aux différents concepts du langage, et d'un ensemble de règles syntaxiques (RS) définissant l'ensemble des différentes constructions syntaxiques que tout programme du langage doit respecter. Ces règles, appelées règles de production, sont reprises directement par les traducteurs pour construire les structures de données arborescentes qui devront accueillir en interne les programmes à partir desquels les vérifications et les traductions seront effectuées.

On distingue deux types de symboles, les symboles non terminaux (SNT) pour lesquels il existe pour chacun d'entre eux, une règle qui en donne sa ou ses constructions syntaxiques élémentaires et les symboles terminaux (ST) qui correspondent aux types de base de l'environnement de modélisation, appelés types primitifs. Ces types primitifs peuvent être différents des types de base du langage, appelés types prédéfinis du langage, faisant donc l'objet de déclarations dans les règles de production de la grammaire du langage.

I.1.b Règles 'Et' et 'Ou' d'une grammaire

L'ensemble de ces règles permet de décrire la syntaxe des programmes représentables, en interne, selon une structure arborescente déployée à partir d'un symbole du langage appelé point d'entrée de la grammaire (PE).

Différentes manières, plus ou moins condensées, existent pour représenter la syntaxe des programmes d'un langage. Dans le cadre de cette étude, il est suffisant de supposer que l'on distingue deux types de règles, les règles 'Et' et les règles 'Ou' que l'on peut décrire, dans une première approche, de la manière suivante :

- Toute règle est composée de deux parties, une partie gauche représentée par le symbole dont la règle en définit la construction syntaxique, et une partie droite décrivant la construction syntaxique pour une règle ‘Et’, et une liste de symbole pour une règle ‘Ou’.
- toute règle ‘Et’ décrit l’ensemble des symboles entrant dans la composition du symbole que la règle définit, en précisant pour chacun d’entre eux son caractère de répétitivité, appelé aussi multiplicité.
- Plusieurs occurrences de même symbole peuvent apparaitre en partie droite d’une règle ‘Et’. Pour les distinguer, il est nécessaire de faire précéder chacune d’entre elles par un nom de rôle.
- Toute règle ‘Ou’ précise qu’un symbole a plusieurs définitions, chacune étant représentée par un symbole.

Ces règles de grammaires permettent de définir les constructions syntaxiques de base des programmes du langage. En fait, elles sont incomplètes, n’assurant pas en particulier qu’un programme doit être représenté par une structure de données arborescente.

I.2 Exemple d’une grammaire d’un langage : le langage « L »

Nous prenons dans ce paragraphe un exemple d’un langage simple, de type procédural où l’on retrouve les principaux concepts des langages, tels la désignation, l’emplacement de mémoire, les structures de contrôle et les expressions. Ce langage est donc suffisamment complet pour présenter des propriétés syntaxiques et sémantiques des langages. Nous appellerons ce langage, le langage « L ».

I.2.a Caractéristiques élémentaires du langage

Tout programme écrit en langage L est représenté par le symbole Prog_L, et est constitué d’une partie déclarative (PD), et d’une partie instruction (PI).

La partie déclarative est constituée d’un ensemble de déclarations de variables (DecVar), éventuellement vide, donnant pour chaque variable (Variable) son nom et le type prédéfini (TP_L) de la valeur que la variable représentera au cours de l’exécution du programme. On se limitera, ici, aux deux types prédéfinis des nombres entiers relatifs et booléens (Rel et Bool).

La partie instruction est constituée d’une instruction (Inst) qui peut être une instruction vide (Skip), une séquence d’instructions (Seq), une affectation (Affect), un test (Test) ou une boucle (Boucle).

Une instruction d’affectation est composée d’une variable et d’une expression (Exp).

Un test est composé d’une expression, d’une instruction qui sera exécutée si l’expression est vraie, et éventuellement d’une autre instruction qui sera exécutée dans le cas contraire.

Une boucle est constituée d’une expression et d’une instruction.

Une expression peut être une constante (Const), une variable, un test sur une valeur nulle (Null) ou une expression binaire (ExpBin).

Une Constante peut être un nombre entier relatif (C_Rel) ou une constante booléenne (C_Bool), cette dernière pouvant être la constante vrai (True) ou faux (False).

Une expression binaire peut être une addition, une multiplication, … , chacune ayant deux arguments qui sont des expressions.

I.2.b Grammaire du langage

La figure suivante montre la grammaire du langage « L ». Les symboles définis en partie gauche d'une règle 'Ou' apparaissent en italique, et les symboles terminaux définissant les types primitifs ont été appelés S pour les chaînes de caractères, Z pour les nombres relatifs et B pour les booléens :

Symboles non terminaux (SNT) : { Prog_L, PD, PI, DecVar, Inst, Variable, *TP_L*, Bool, Rel, *Inst*, Skip, Seq, Affect, Test, Boucle, *Exp*, *Const*, Null, ExpBin, C_Bool, C_Rel, True, False, Add, Sous, Mult, Div, Inf, … }

Symboles terminaux (Types primitifs) (ST) : { S, Z, B }
Point d'entrée (PE) : Prog_L

Règles de constructions syntaxiques du langage L (RCS) :

Prog_L →	PD,	PI
PD →	DecVar*	
PI →	Inst	

-- Variables typées

DecVar →	Variable,	TP_L
Variable →	S	

-- Types prédéfinis du langage L

TP_L →	Bool |	Rel
Bool →		
Rel →		

-- Constructions syntaxiques des instructions

Inst →	Skip |	Seq |	Affect |	Test |	Boucle
Skip →					
Seq →	Inst,	suite : Inst			
Affect →	Variable,	Exp			
Test →	Exp,	alors : Inst,	sinon : Inst?		
Boucle →	Exp,	Inst			

-- Constructions syntaxiques des expressions

Exp →	Variable |	Const |	Null |	ExpBin		
Const →	C_Bool |	C_Rel				
C_Bool →	True |	False				
True →						
False →						
C_Rel →	Z					
Null →	Exp					
ExpBin →	Add |	Sous |	Mult |	Div |	Xor |	Inf
Add →	exp1 : Exp,	exp2 : Exp				
Sous →	exp1 : Exp,	exp2 : Exp				
Mult →	exp1 : Exp,	exp2 : Exp				
Div →	exp1 : Exp,	exp2 : Exp				
Inf →	exp1 : Exp,	exp2 : Exp				
…						

Figure 1 : ***Grammaire du Langage « L »***

La règle définissant le symbole Prog_L est un exemple de règle ‘Et’ ; la règle définissant le symbole ‘Inst’ est un exemple de règles ‘Ou’. Les symboles S et Z correspondant aux types primitifs sont des symboles terminaux, n’apparaissent pas en partie gauche d’une règle.

I.2.c Exemple d’un programme

Un exemple de programme du langage « L », écrit dans une syntaxe concrète évidente, pourrait être :

```
programme p1
        x, y : Rel
        yc, y1c : Rel
        x := …
        y := 0
        yc := 0
        y1c := 1
        tant que  non( x < y1c )
                y := y + 1
                yc := y1c
                y1c := yc + 2 * y + 1
        -- resultat y
```

Figure 2 : ***Exemple d’un programme écrit en langage « L », calculant la racine carrée entière approchée du nombre entier x***

Ce programme calcule la racine carrée entière approchée de x. A partir de la valeur de y égale à 0, il suffit de déterminer, par itérations successives, une nouvelle valeur de y déduite de la valeur précédente par application de la formule suivante : $(y + 1)^2 = y^2 + 2 * y + 1$, jusqu’à ce que l’assertion suivant soit vérifiée :

$$y * y <= x < (y + 1) * (y + 1)$$

II Propriétés des langages de programmation

Tout programme d’un langage de programmation est représenté en mémoire sous la forme d’une structure de données arborescente. Les propriétés des langages de programmation sont traditionnellement spécifiées par des axiomes et des règles d’inférence spécifiées à l’aide de fonctions récursives dont les traitements sont bien adaptés à la manipulation de structures de données arborescentes. C’est pourquoi, dans ce chapitre, la spécification de ces propriétés se fera directement à l’aide d’une notation fonctionnelle qui rejoint en fait la notation généralement utilisée pour décrire la sémantique dénotationnelle des langages de programmation.

II.1 Propriétés de typage

Il s’agit, en particulier, de vérifier que les expressions des boucles et des tests apparaissant dans un programme écrit en langage « L » sont des expressions conditionnelles.

Les propriétés de typage permettent d'assimiler tout programme d'un langage à un ensemble de formules bien formées.

Les propriétés de typage indiquent le type du résultat des expressions, et indique si les éléments de programmation intervenant dans une instruction ont les types requis.

II.1.a Propriétés de typage des expressions

D'une manière générale, la propriété de typage d'une expression doit être définie à l'aide d'une fonction recevant en paramètre une expression, et retournant le type de l'expression :

$$\text{type}(\,Exp\,) : TP_L$$

Figure 3 : ***Règle définissant le typage d'une expression***

Les axiomes suivant définissent, a priori, les règles de typage des constantes :

$$\frac{}{\text{type}(\text{ True }) : \text{Bool}}$$

$$\frac{}{\text{type}(\text{ False }) : \text{Bool}}$$

$$\frac{}{\text{type}(\text{ C_Rel }) : \text{Rel}}$$

Figure 4 : ***Axiomes définissant le typage des constantes***

Ce qui signifie, en particulier, que la constante booléenne True est de type Bool.

Du fait que tout programme a une représentation arborescente, tout traitement se fait récursivement selon un parcours d'arbres en profondeur à partir de son point d'entrée. Définir le type d'une variable apparaissant dans la partie instruction du programme nécessite de remonter dans la partie déclarative, ce qui peut se faire de différentes manières qui peuvent dépendre de la manière dont les structures syntaxiques de la grammaire du langage ont été définies. La propriété de typage d'une variable est donc décrite, à ce niveau, de façon incomplète, par la règle d'inférence suivante :

$$\frac{}{\text{type}(\text{ Variable }) : \ldots}$$

Figure 5 : ***Règles définissant le typage d'une variable***

Les règles d'inférence suivantes décrivent les propriétés de typage des expressions Null et Add du langage « L » :

$$\frac{\text{type}(\,Exp\,) : \text{Rel}}{\text{type}(\text{ Null}(\,Exp\,)\,) : \text{Bool}}$$

$$\frac{\text{type}(\,Exp_i\,) : \text{Rel},\ \ \text{type}\big(\,Exp_j\,\big) : \text{Rel}}{\text{type}\big(\text{ Add}(\,Exp_i, Exp_j\,)\,\big) : \text{Rel}}$$

...

Figure 6 : Règles définissant le typage des expressions

Ces règles montrent, en particulier, que si une expression *Exp* est de type Rel, alors Null(*Exp*) est de type Bool. Si les expressions Exp_i et Exp_j sont de type Rel, alors Add(Exp_i, Exp_j) est de type Rel.

II.1.b Propriétés de typage des instructions

D'une manière générale, la propriété de typage d'une instruction doit être définie à l'aide d'une fonction recevant en paramètre une instruction, et indiquant en retour si le typage est correct :

$$\text{typage}(\, Inst \,) : \text{ok}$$

Figure 7 : Règle définissant le typage d'une instruction

Les règles d'inférence suivantes donnent les propriétés de typage des instructions du langage L :

$$\frac{}{\text{typage}(\,\text{Skip}\,) : \text{ok}}$$

$$\frac{\text{typage}(\, Inst_i \,) : \text{ok}, \quad \text{typage}(\, Inst_j \,) : \text{ok}}{\text{typage}\big(\,\text{Seq}\big(\, Inst_i,\ Inst_j \,\big)\big) : \text{ok}}$$

$$\frac{\text{type}(\,\text{Variable}\,) : \text{Rel}, \quad \text{type}(\, Exp \,) : \text{Rel}}{\text{typage}(\,\text{Affect}(\,\text{Variable}, Exp \,)\,) : \text{ok}}$$

$$\frac{\text{type}(\,\text{Variable}(\,\text{S}\,)\,) : \text{Bool}, \quad \text{type}(\, Exp \,) : \text{Bool}}{\text{typage}\big(\,\text{Affect}(\,\text{Variable}, Exp \,)\big) : \text{ok}}$$

$$\frac{\text{type}(\, Exp \,) : \text{Bool}, \quad \text{typage}(\, Inst \,) : \text{ok}}{\text{typage}(\,\text{Test}(\, Exp,\ Inst \,)\,) : \text{ok}}$$

$$\frac{\text{type}(\, Exp \,) : \text{Bool}, \quad \text{typage}(\, Inst_i \,) : \text{ok}, \quad \text{typage}(\, Inst_j \,) : \text{ok}}{\text{typage}\Big(\,\text{Test}\big(\, Exp,\ Inst_i,\ Inst_j \,\big)\Big) : \text{ok}}$$

$$\frac{\text{type}(\, Exp \,) : \text{Bool}, \quad \text{typage}(\, Inst \,) : \text{ok}}{\text{typage}\big(\,\text{Boucle}(\, Exp,\ Inst \,)\big) : \text{ok}}$$

Figure 8 : Règles définissant le typage des instructions

Ces règles montrent, par exemple, qu'une instruction d'affectation est correcte du point de vue typage si la variable cible Variable et l'expression *Exp* composant l'instruction d'affectation sont de même type. En ce qui concerne l'instruction de boucle, si l'expression *Exp* est de type Bool et si l'instruction *Inst* est correcte, alors l'instruction de boucle composée de *Exp* et de *Inst* est correcte du point de vue typage.

II.2 Sémantique opérationnelle du langage

La sémantique opérationnelle d'un langage se définit à l'aide d'un ensemble de valeurs de référence dont les caractéristiques, souvent représentées selon un formalisme mathématique, sont a priori connues. Il s'agit donc de décrire, sous forme de propriétés, le comportement de référence de chaque construction syntaxique du langage sur cet ensemble de données, appelé généralement mémoire du programme. Si l'on dispose donc d'un environnement capable d'exécuter directement les propriétés comportementales d'un programme sur cette mémoire du programme, on peut assimiler l'ensemble de ces propriétés à un interprète. Cette mémoire et ces propriétés constituent ce que l'on appelle généralement un environnement d'exécution des programmes.

Il s'agit donc, dans ce paragraphe, de définir les propriétés d'un tel environnement d'exécution, et les correspondances entre les symboles de la grammaire du langage et leurs propriétés au niveau de l'environnement d'exécution.

II.2.a L'environnement d'exécution d'un programme

L'environnement d'exécution d'un programme associe à chaque variable du programme la valeur qui sera calculée au fur et à mesure de l'exécution du programme. Il s'agit ensuite de définir :

- les principales opérations utiles sur l'environnement d'exécution,
- les correspondances entre les constantes du langage « L » et les constantes de l'environnement appelées valeurs sémantiques, et
- les correspondances entre les opérations du langage et de l'environnement d'exécution.

Ces opérations et ces correspondances sont supposées être les suivantes :

Opérations de l'Environnement d'Exécution : EnvExec	Environnement d'Exécution : EnvExec
mem(EnvExec, Variable) : v	Variable est associée à la valeur v
maj(EnvExec, Variable, v)	Variable devient associée à la valeur v

Constantes du langage « L »	Constantes dans l'Environnement d'Exécution
True	true
False	false
C_Rel	z

Opérations du langage « L »	Opérations de l'Environnement d'Exécution :
Add	add(n1, n2) : z
Sous	sous(n1, n2) : z
…	…

Figure 9 : ***Propriétés de l'environnement d'exécution, et correspondances entre les symboles et l'environnement***

Les propriétés comportementales des expressions et des instructions du langage « L » peuvent alors être décrites pour montrer l'effet qu'elles produisent sur un tel environnement d'exécution, en supposant que les codes vérifient déjà les propriétés de typage.

II.2.b Propriétés comportementales des expressions

D'une manière générale, l'évaluation d'une expression *Exp* sur l'environnement d'exécution EnvExec se spécifie de la manière suivante :

$$\text{eval}(\ \text{EnvExec}, Exp\ ) : \text{v}$$

Figure 10 : ***Propriété définissant le comportement d'une expression***

Ce qui signifie que l'évaluation de l'expression *Exp* dans l'environnement d'exécution EnvExec aura pour effet de retourner la valeur v.

Dans ces conditions, les axiomes et règles suivant définissent les propriétés comportementales des expressions du langage « L » :

$$\frac{}{\text{eval}(\ \text{EnvExec}, \text{True}\ ) = \text{true}}$$

$$\frac{}{\text{eval}(\ \text{EnvExec}, \text{False}\ ) =: \text{false}}$$

$$\frac{}{\text{eval}(\ \text{EnvExec}, \text{C_Rel}\ ) = \text{z}}$$

$$\frac{}{\text{eval}(\ \text{EnvExec}, \text{Variable}\ ) : \text{mem}(\ \text{EnvExec}, \text{Variable}\ )}$$

$$\frac{\text{eval}(\ \text{EnvExec},\ Exp\ ) = \text{z}}{\text{eval}\big(\ \text{EnvExec}, \text{Null}(\ Exp\ )\big) = \ \text{null}(\ \text{z}\ )}$$

$$\frac{\text{eval}(\ \text{EnvExec},\ Exp_i\ ) = \text{z}_\text{i},\ \text{eval}\big(\text{EnvExec},\ Exp_j\ \big) = \text{z}_\text{j}}{\text{eval}\big(\ \text{EnvExec}, \text{Add}\big(\ Exp_i,\ Exp_j\ \big)\big) = \text{add}(\ \text{z}_\text{i},\ \text{z}_\text{j}\ )}$$

...

$$\frac{\text{eval}(\ \text{EnvExec}, Exp_i\ ) =: \text{z}_\text{i},\ \text{eval}\big(\ \text{EnvExec},\ Exp_j\ \big) = \text{z}_\text{j}}{\text{eval}\big(\ \text{EnvExec}, \text{Inf}\big(\ Exp_i,\ Exp_j\ \big)\big) = \ \text{inf}(\ \text{z}_\text{i},\ \text{z}_\text{j}\ )}$$

...

Figure 11 : ***Règles définissant les propriétés comportementales des expressions***

Ces règles montrent, par exemple, que toute constante True de type Bool définie au niveau du langage L, est notée true au niveau de l'environnement d'exécution, et que toute constante de type Rel, est notée à l'aide de la lettre z au niveau de l'environnement d'exécution. D'autre part, si, compte tenu d'un environnement d'exécution se trouvant dans l'état EnvExec, l'évaluation de l'expression Exp_i retourne la valeur z_i, et si dans les mêmes conditions l'évaluation de l'expression Exp_j retourne la valeur z_j, alors l'évaluation de l'opération Add du langage L appliquée sur les deux expressions aura pour résultat l'évaluation de add(z_i, z_j).

II.2.c Propriétés comportementales des instructions

L'exécution d'une instruction modifie le contenu de l'environnement d'exécution. Ce passage d'un état à un autre de l'environnement d'exécution est noté de la manière suivante :

$$\mathrm{exec}(\ \mathrm{EnvExec}, \mathit{Inst}\ )\ \rhd\ \mathrm{EnvExec}$$

Figure 12 : ***Propriété définissant le comportement d'une instruction***

Les règles d'inférences suivantes définissent les propriétés comportementales des différentes constructions syntaxiques du langage L définies dans la partie instruction de la grammaire :

$$\frac{}{\mathrm{exec}(\ \mathrm{EnvExec_i}, \mathrm{Skip}\ ) \rhd \mathrm{EnvExec_i}}$$

$$\frac{\mathrm{exec}(\ \mathrm{EnvExec_i},\ Inst_i\ ) \rhd \mathrm{EnvExec_j},\ \ \mathrm{exec}\big(\ \mathrm{EnvExec_j},\ Inst_j\ \big) \rhd \mathrm{EnvExec_k}}{\mathrm{exec}(\ \mathrm{EnvExec_i}, \mathrm{Seq}\big(\ Inst_i,\ Inst_j\ \big)\ \rhd \mathrm{EnvExec_k}}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{v}}{\mathrm{exec}(\ \mathrm{EnvExec_i},\ \mathrm{Affect}(\ \mathrm{Variable}, Exp\ )\ )\ \rhd \mathrm{maj}(\ \mathrm{EnvExec_i}, \mathrm{Variable}, \mathrm{v}\ )}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{true},\ \ \mathrm{exec}(\ \mathrm{EnvExec_i},\ Inst\ ) \rhd \mathrm{EnvExec_j}}{\mathrm{exec}(\ \mathrm{EnvExec_i},\ \mathrm{Test}(\ Exp, Inst\ )\ )\ \rhd \mathrm{EnvExec_j}}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{false},\ \ \mathrm{exec}(\ \mathrm{EnvExec_i},\ Inst\ ) =: \mathrm{EnvExec_j}}{\mathrm{exec}(\ \mathrm{EnvExec_i},\ \mathrm{Test}(\ Exp, Inst\ )\ ) : \mathrm{EnvExec_i}}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{true}\ \ \ \mathrm{exec}\big(\ \mathrm{EnvExec_i},\ Inst_{j1}\ \big) \rhd \mathrm{EnvExec_{i1}},\ \ \mathrm{exec}\big(\ \mathrm{EnvExec_i}, Inst_{j2}\ \big)\ \rhd \mathrm{EnvExec_{j2}}}{\mathrm{exec}\big(\ \mathrm{EnvExec_i},\ \mathrm{Test}\big(\ Exp, Inst_{j1},\ Inst_{j2}\ \big)\big) : \mathrm{EnvExec_{j1}}}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{false}\ \ \ \mathrm{exec}\big(\ \mathrm{EnvExec_i},\ Inst_{j1}\ \big)\ \rhd \mathrm{EnvExec_{j1}},\ \ \mathrm{exec}\big(\ \mathrm{EnvExec_i},\ Inst_{j2}\ \big)\ \rhd \mathrm{EnvExec_{j2}}}{\mathrm{exec}\big(\ \mathrm{EnvExec_i},\ \mathrm{Test}\big(\ Exp, Inst_{j1},\ Inst_{j2}\ \big)\big) : \mathrm{EnvExec_{j2}}}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{true}\ \ \ \mathrm{exec}(\ \mathrm{EnvExec_i},\ Inst\ ) \rhd \mathrm{EnvExec_1},\ \ \mathrm{exec}(\ \mathrm{EnvExec_1},\ \mathrm{Boucle}(\ Exp,\ Inst\ )\ ) \rhd \mathrm{EnvExec_2}}{\mathrm{exec}(\ \mathrm{EnvExec},\ \mathrm{Boucle}(\ Exp, Inst\ )\ )\ \rhd \mathrm{EnvExec_2}}$$

$$\frac{\mathrm{eval}(\ \mathrm{EnvExec_i}, Exp\ ) = \mathrm{false}}{\mathrm{exec}(\ \mathrm{EnvExec_i},\ \mathrm{Boucle}(\ Exp, Inst\ )\ )\ \rhd \mathrm{EnvExec_i}}$$

Figure 13 : Règles définissant les propriétés comportementales des instructions du langage

Cette figure montre, par exemple, que si l'exécution de l'instruction $Inst_I$ fait passer l'environnement d'exécution de l'état $\mathrm{EnvExec_i}$ à l'état $\mathrm{EnvExec_j}$ et l'instruction $Inst_j$, de l'état $\mathrm{EnvExec_j}$ à l'état $\mathrm{EnvExec_k}$, alors l'exécution de la séquence composée des instructions $Inst_i$ et $Inst_j$ fait passer l'environnement d'exécution de l'état $\mathrm{EnvExec_i}$ à l'état $\mathrm{EnvExec_k}$.

II.3 Sémantique axiomatique

La sémantique axiomatique d'un langage permet de raisonner sur tout ou un fragment d'un programme par rapport à des propriétés définies sur les données du programme. La sémantique axiomatique se définit donc à l'aide d'une relation entre les données d'un programme et des propriétés de type pré- et post-condition applicables à un fragment de programme. Cette relation, donnant pour chaque construction syntaxique du langage les règles de transformation de ces assertions, se définit sous la forme d'un triplet, appelé triplet de Hoare, dont nous en rappelons, dans ce paragraphe, les principales caractéristiques.

II.3.a Triplet de Hoare, code correct et Assertion

Un triplet de Hoare est noté de la manière suivante :

{ P } *Inst* { Q }

Figure 14 : ***Triplet de Hoare***

où P est une pré-condition, Q une post-condition, et *Inst* une instruction.

Par définition, un triplet est valide, si pour tout état $EnvExec_i$ d'un environnement d'exécution EnvExec où l'assertion P est vraie, alors l'exécution de l'instruction *Inst*, fait passer cet environnement d'exécution à un état $EnvExec_j$ où la post-condition Q est vraie. Si tel est le cas, on dit alors que le code représenté par l'instruction est correct par rapport à ses pré- et post-conditions, ou bien ne les contredit pas.

Un tel triplet de Hoare peut se vérifier dynamiquement « au coup par coup », c'est-à-dire à chaque exécution du programme comme on le fait classiquement pour les pré- et post-conditions d'une opération. Cependant la logique de Hoare définit une propriété beaucoup plus exigeante, puisqu'il s'agit de voir, en analysant le code, si « pour tout état de l'environnement d'exécution » où la pré-condition P est vraie, l'exécution de l'instruction *Inst* amène un nouvel état de l'environnement où Q est vraie. Les propriétés relatives à la logique de Hoare, s'appliquant sur du code, rentre donc dans la catégorie des propriétés statiques d'un langage.

Par exemple, soit le triplet de Hoare suivant :

$$\{ x > 0 \} \ x := x + 10 \ \{ x > 8 \}$$

On peut voir que, pour une valeur de x égale à 1, la pré-condition P : $x > 0$ est vraie. L'exécution de l'instruction fait passer x de la valeur 0 à 11, rendant donc la post-condition Q : $x > 8$ vraie. En fait, ce triplet est valide si l'on peut démontrer que « pour toute valeur de x telle que P : $x > 0$ » est vraie, alors la post-condition Q : $x > 8$ sera vraie après exécution de l'instruction d'affectation.

Intuitivement, ce triplet de Hoare est valide puisque que pour toute valeur v positive associée à x dans l'environnement d'exécution avant l'exécution de l'instruction d'affectation, la pré-condition P est alors vraie, et dans ces conditions, l'exécution de l'instruction d'affectation amène à associer à x la valeur v + 10, rendant visiblement la post-condition Q

vraie. Dans ces conditions, l'instruction d'affectation est correcte, et ne contredit pas la pré- et post-condition.

En reprenant les propriétés fonctionnelles que nous avons définies précédemment pour décrire la sémantique opérationnelle des langages de programmation, on peut associer à tout triplet de Hoare :

$$\{ P \} \, Inst \, \{ Q \}$$

l'assertion de la logique du premier ordre suivante :

$$\forall \, EnvExec_i \in EnvExec, \, eval(EnvExec_i, P) = true \Rightarrow eval(\, exec(EnvExec_i, Inst), Q) = true$$

ou, d'une manière plus implicite :

$$\{ P \} \, Inst \, \{ Q \} \equiv eval(EnvExec, P) = true \Rightarrow eval(\, exec(EnvExec, Inst), Q) = true$$

Figure 15 : ***Assertion associée à un triplet de Hoare***

Ce qui signifie que pour montrer qu'un triplet de Hoare est valide, il faut démontrer que l'assertion de la logique du premier ordre qui lui est associée est valide.

La logique de Hoare a pour objectif de vérifier que des fragments de code sont corrects par rapport aux propriétés qui les encadrent, ou qu'ils décrivent des calculs satisfaisant ces propriétés. Ce passage de la logique de Hoare à la logique classique du premier ordre rappelle que tout triplet de Hoare sous-tend toute la partie comportementale du langage dans lequel il est décrit, en particulier, l'environnement d'exécution des programmes et les données manipulées dans cet environnement.

En reprenant le triplet de Hoare donné en exemple précédemment :

$$\{ x > 0 \} \; x := x + 10 \; \{ x > 8 \}$$

L'assertion qui lui est associée est la suivante :

$$\forall \, v \in Z, \; eval(\, EnvExec(\,(x, v)\,), Sup(x, 0)\,) = true \Rightarrow$$
$$eval(\, exec(\, EnvExec(\,(x, v)\,), Affect(x, x + 10)\,), \; Sup(x, 8)\,) = true$$

ou, plus simplement :

$$\forall \, v \in Z \, / \; eval(\, Sup(v, 0)\,) = true \Rightarrow eval(\, Affect(v, v + 10), \; Sup(v, 8)\,) = true$$

ou, encore : $$eval(\, Sup(v, 0)\,) = true \Rightarrow eval(\, Affect(v, v + 10), \; Sup(v, 8)\,) = true$$

Les assertions dérivant des triplets de Hoare peuvent contenir des opérateurs non définis dans le langage dans lesquels sont écrits les fragments de code à prouver. En particulier, les définitions précédentes font intervenir l'opérateur d'implication qui n'existe dans la grammaire du langage L qui doit donc être 'étendue' à l'ensemble de cet opérateur. Cependant cette grammaire étendue doit être construite de manière à ne pas dénaturer le langage L dont les programmes doivent pouvoir être pris en compte par les traducteurs déjà existants. La figure suivante donne les constructions syntaxiques des assertions qui pourront être utilisées pour 'décorer' des fragments de codes « L » à prouver, et pour en définir les règles de typage :

-- Fragment de la grammaire du langage L étendu à la prise en compte des propriétés axiomatiques

PI →	Inst				
Inst →	Skip |	Seq |	Affect |	Test |	Boucle
Seq →	**p : Assert ?,**	inst : Inst,	suite : Inst,	**q : Assert ?**	
Affect →	**p : Assert ?,**	Variable,	exp : Exp,	**q : Assert ?**	
Test →	**p : Assert ?,**	Exp,	alors : Inst,	sinon : Inst ?,	**q : Assert ?**
Boucle →	**p : Assert ?,**	Exp,	Inst,	**q : Assert ?**	
…					
Assert →	Not |	AssertBin |	Exp		
Not →	Assert				
AssertBin →	Imp |	And |	Eg		
Imp →	assert1 : Assert,	assert2 : Assert			
Eg →	assert1 : Assert,	assert2 : Assert			
…					

-- Règle définissant la relation de typage d'une assertion

typeAssert(*Assert*) : Bool

-- Règles d'inférence définissant les propriétés de typage des opérateurs portant sur les assertions

$$\frac{\text{type}(\ Exp_i\ ) : \text{bool}}{\text{typeAssert}(\ Exp_i\ ) : \text{Bool}}$$

$$\frac{\text{typeAssert}(\ Exp_i\ ) : \text{bool}}{\text{typeAssert}(\ \text{Not}(\ Exp_i\ )\ ) : \text{Bool}}$$

$$\frac{\text{typeAssert}(\ Assert_i\ ) : \text{bool} \quad \text{typeAssert}(\ Assert_j\ ) : \text{bool}}{\text{typeAssert}(\ Assert_i, Assert_j\ )}$$

…

typageHoare(*Inst*) : ok

$$\frac{\text{typeAssert}(\ Assert_j\ ) : \text{Bool} \quad \text{typage}(\ \text{Seq}(\ Inst_j\ , Inst_k\ )\ ) : \text{ok} \quad \text{typageAssert}(\ Assert_l\ ) : \text{Bool}}{\text{typageHoare}(\ Assert_j\ , \text{Seq}(\ Inst_j,\ \ Inst_k\ ), \text{Assert}_l\ )}$$

…

Figure 16 : ***Extension de la grammaire du langage « L » pour la prise en compte des assertions et les triplets de Hoare***

Cette figure montre, en particulier, qu'une expression *Exp* du langage L retournant une valeur de type Bool est une assertion de tout triplet de Hoare.

La sémantique opérationnelle des assertions d'un triplet de Hoare ne se définit pas, puisqu'il s'agit de chercher à valider les assertions dérivant d'un triplet de Hoare, au niveau du code, quelques soient les valeurs qui sont manipulées par le code.

Avant de rappeler les règles axiomatiques appliquées au langage L, nous donnons la règle de conséquence qui permet ‘d’élargir’ les pré-conditions et les post-conditions d’un triplet de Hoare :

$$\frac{\mathrm{P1} \Rightarrow \mathrm{P},\ \{\,\mathrm{P}\,\}\, Inst\, \{\,\mathrm{Q}\,\},\ \ \mathrm{Q} \Rightarrow \mathrm{Q1}}{\{\,\mathrm{P1}\,\} \qquad Inst \qquad \{\,\mathrm{Q1}\,\}}$$

Figure 17 : ***Règle de conséquence***

qui signifie que si le triplet de Hoare { P } *Inst* { Q } est valide, alors { P1 } *Inst* { Q1 } est aussi valide pour toute pré-condition P1 telle que $\mathrm{P1} \Rightarrow \mathrm{P}$, et pour toute post-condition Q1 telle que $\mathrm{Q} \Rightarrow \mathrm{Q1}$.

Sachant, par exemple, que le triplet de Hoare :

$$\{ x > 0 \}\ x := x + 10\ \{ x > 8 \}$$

est valide.
Puisque, $\forall\ x > 2$, on a $x > 0$, donc ($x > 2 \Rightarrow x > 0$) et de même ($x > 8 \Rightarrow x > 7$), on déduit que

$$\{ x > 2 \} \qquad x := x + 10 \qquad \{ x > 7 \}$$

est valide.

Démontrer qu’un triplet de Hoare est valide peut être difficile, voire impossible. On peut faire appel à cette règle pour démontrer qu’un triplet de Hoare est valide en tenant compte de pré-conditions et post-conditions plus ‘élargies’. La propriété de conséquence peut aussi être utilisée pour faire le lien entre les pré-conditions et post-conditions issues des exigences de l’application et définies par les analystes/concepteurs, et des pré-conditions et post-conditions rajoutées par les analystes/programmeurs pour vérifier qu’un fragment de code est bien construit techniquement, en particulier, et pour vérifier que les boucles se terminent.

Aux paragraphes suivants, nous rappelons les principes de base des règles axiomatiques, tout d’abord pour les différentes instructions Skip et Test, puis pour les instructions d’affectation et de boucle qui nécessitent des traitements particuliers. Ces principes de base seront illustrés sur les constructions syntaxiques du langage « L ».

II.3.b Règles définissant les propriétés axiomatiques pour les instructions Skip *et* Test

La figure suivante montre les règles d’inférence sur les triplets de Hoare pour les constructions syntaxiques Skip et Test du langage L :

$$\frac{}{\{\,\mathrm{P}\,\}\ \mathrm{Skip}\ \{\,\mathrm{P}\,\}}$$

$$\frac{\{\,\mathrm{P}\,\}\, Inst_i\, \{\,\mathrm{P1}\,\},\ \ \{\,\mathrm{P1}\,\}\, Inst_j\, \{\,\mathrm{Q}\,\}}{\{\,\mathrm{P}\,\}\ \mathrm{Seq}(\, Inst_i,\ Inst_j\,)\ \{\,\mathrm{Q}\,\}}$$

$$\frac{\{\,\mathrm{P} \wedge (\, Exp = \mathrm{True}\,)\,\}\, Inst\, \{\,\mathrm{Q}\,\},\quad \mathrm{P} \wedge (\, Exp = \mathrm{False}\,) \Rightarrow \mathrm{Q}}{\{\,\mathrm{P}\,\}\ \mathrm{Test}(\, Exp, Inst\,)\ \{\,\mathrm{Q}\,\}}$$

$$\frac{\{ P \wedge (Exp = \text{True}) \}\, Inst_i \,\{ Q \}, \quad \{ P \wedge (Exp = \text{False}) \}\, Inst_j \,\{ Q \}}{\{ P \}\, \text{Test}(Exp,\ Inst_i, Inst_j)\, \{ Q \}}$$

Figure 18 : ***Règles d'inférence des propriétés axiomatiques pour la séquence et le test***

Ces règles montrent, en particulier, d'une part que le triplet de Hoare { P } Skip { P } est valide, et d'autre part que si { P } $Inst_i$ { Q1 } est valide et si { Q1 } $Inst_j$ { Q2 } est valide, alors, le triplet de Hoare { P } Seq(Ins_i, $Inst_j$) { Q2 } est valide. Les deux autres règles d'inférence décrivent la sémantique axiomatique de la construction syntaxique du symbole Test.

II.3.c Opération de substitution et règle d'inférence pour l'instruction d'affectation

Le schéma d'axiome pour l'instruction d'affection se fait à l'aide de l'opération appelée substitution, définie à partir d'une assertion et de l'instruction d'affectation. Si Q est une assertion, cette opération est généralement notée :

Q[Variable := Exp]

Cette opération est purement textuelle. Elle a pour effet de retourner une assertion après avoir substituée dans l'assertion Q, chacune des occurrences de la variable Variable par l'expression apparaissant en partie droite de l'instruction d'affectation.

Par exemple : x > 8[x := x + 10] retourne le code : x + 10 > 8

Cette opération s'explique intuitivement dans le cadre du triplet de Hoare suivant :

{ Q(x)[x := f(x)] } x := f(x) { Q(x) }

ayant l'assertion Q(x) en post-condition et en pré-condition le résultat de la substitution de l'instruction d'affectation x := f(x) et de Q(x).
En effet, l'opération de substitution appliquée à la pré-condition { Q(x)[x := f(x)] } a pour résultat de rendre cette pré-condition fonction de f(x). Le triplet de Hoare devient donc :

{ Q(f(x) } x := f(x) { Q(x) }

Supposons que la variable x soit associée à la valeur v, dans l'environnement d'exécution, avant l'exécution de l'instruction d'affectation. L'exécution de ce triplet se fait selon le processus suivant :

- L'évaluation de la pré-condition se fait en considérant l'assertion Q(f(v)) ;
- L'exécution de l'instruction d'affectation a donc pour effet d'associer à la variable x la valeur f(v) ; et
- L'évaluation de la post-condition se fera en considérant l'assertion Q(f(v)).

Ce triplet de Hoare est donc valide, puisque pour toute valeur de v telle que la pré-condition Q(f(v)) est vraie, alors l'exécution de l'instruction d'affectation fait passer la valeur de x à la valeur f(v) rendant donc la post-condition Q(f(v)) vraie.

Cependant, on peut démontrer que ce triplet de Hoare est valide en vérifiant que l'assertion qui lui est associée, est valide. En effet, supposons qu'avant l'exécution de l'instruction d'affectation, x est associée à la valeur v, ce que l'on écrit sous la forme (x, v). Il s'agit donc de prouver que l'assertion suivante est valide :

$$\forall\ EnvExec_i \in EnvExec \ /\ eval(EnvExec_i, P) = true \Rightarrow eval(\ exec(EnvExec_i, \mathit{Inst}\ ), Q\ ) = true$$

soit : $\forall\ v \in R\ /\quad eval(EnvExec_i, Q(\ f(\ x\ )\ ) = true \Rightarrow eval(\ exec(EnvExec_i, x := f(\ x\ )\ ), Q(\ x\ )\ ) = true$

soit : $\forall\ v \in R\ /\quad eval(\ \mathbf{(\ x, v\ )},\ Q(\ f(\ x\ )\ ) = true \Rightarrow eval(\ exec(\ \mathbf{(\ x, v\ )},\ x := f(\ x\ )\ ), Q(\ x\ )\ ) = true$

soit : $\forall\ v \in R\ /\quad \mathbf{Q(\ f(\ v\ )}\ = true \Rightarrow eval(\ \mathbf{(\ x, f(\ v\ )\ )},\ Q(\ x\ )\ ) = true$

soit : $\forall\ v \in R\ /\quad \mathbf{Q(\ f(\ v\ )}\ = true \Rightarrow \mathbf{Q(\ f(\ v\ )\ )}\ = true$

Cette assertion est valide, donc le triplet de Hoare $\{\ Q(\ x\ )[\ x := f(\ x\ )\ ]\ \}\ x := f(\ x\ )\ \{\ Q(\ x\ )\ \}$ est valide.

En reprenant l'exemple précédent : $\{\ x > 8[\ x := x + 10\ ]\ \}\ x := x + 10\ \{\ x > 8\ \}$

on en déduit que le triplet de Hoare : $\{\ x > -2\ \}\quad x := x + 10\ \{\ x > 8\ \}$ est valide.

La sémantique axiomatique pour l'instruction d'affectation du langage L est donc la suivante :

$$\frac{}{\{\ Q[\ x := f(\ x\ )\ ]\ \}\ x := f(\ x\ )\ \{\ Q\ \}}$$

Figure 19 : ***Règle d'inférence définissant la sémantique axiomatique pour l'instruction d'affectation***

II.3.d Correction partielle, correction totale

D'une manière générale, la règle définissant la sémantique axiomatique pour la boucle est la suivante :

$$\frac{\{\ P \land (\ Exp = \text{true}\ )\ \}\ Inst\ \{\ P\ \}}{\{\ P\ \}\ \text{Boucle}(\ Exp, Inst\ )\ \{\ P \land (\ Exp = \text{false}\ )\ \}}$$

Cependant, on a l'habitude préalablement d'utiliser le triplet de Hoare pour contrôler le bon fonctionnement technique de la boucle par rapport à une expression booléenne, appelée invariant qui doit être vérifiée à chaque itération, et par rapport à une expression entière positive appelée variant qui décroit à chaque itération et qui assure donc la terminaison de la boucle. Si l'on trouve un invariant qui est vérifié à chaque itération, on dit que la boucle est correcte partiellement, et si de plus on démontre qu'il existe un variant qui décroit à chaque itération, on dit que la boucle est correcte totalement.

L'invariant à l'entrée de la boucle et à chaque itération doit donc assurer que le triplet de Hoare suivant est valide :

$$\{\text{Invariant} \wedge (Exp = \text{true})\}\ Inst\ \{\text{Invariant}\}$$

De plus, à la sortie de la boucle, l'invariant doit rester vrai, sachant que la condition de la boucle a la valeur faux. L'invariant doit donc aussi assurer que le triplet de Hoare suivant est valide :

$$\{\text{Invariant}\}\ \text{Boucle}(Exp, Inst)\ \{\text{Invariant} \wedge (Exp = \text{false})\}$$

On en déduit donc que la correction partielle de la boucle est définie par la règle d'inférence suivante :

$$\frac{\{\text{Invariant} \wedge (Exp = \text{true})\}\ Inst\ \{\text{Invariant}\}}{\{\text{Invariant}\}\ \text{Boucle}(Exp, Inst)\ \{\text{Invariant} \wedge (Exp = \text{false})\}}$$

Figure 20 : ***Règle d'inférence décrivant la sémantique axiomatique pour la boucle à l'aide d'un invariant de boucle***

Et que la correction totale de la boucle est définie par la règle d'inférence suivante :

$$\frac{\{\text{Invariant} \wedge (Exp = \text{true}) \wedge (\text{Variant} = \text{z})\}\ Inst\ \{\text{Invariant} \wedge (\text{Variant} < z)\},\quad \text{Invariant} \Rightarrow (\text{Variant} \geq 0)}{\{\text{Invariant}\}\ \text{Boucle}(Exp, Inst)\ \{\text{Invariant} \wedge (Exp = \text{false})\}}$$

L'expression : $\text{Invariant} \Rightarrow (\text{Variant} \geq 0)$, exprime le fait que le variant doit être positif.
Le triplet de Hoare suivant :

$$\{\text{Invariant} \wedge (Exp = \text{true}) \wedge (\text{Variant} = \text{z})\}\ Inst\ \{\text{Invariant} \wedge (\text{Variant} < z)\}$$

est valide si, à chaque itération, le variant appelé z diminue strictement.

Ainsi, d'une manière générale, pour valider le triplet de Hoare suivant :

$$\{\ P\ \}\ \text{Boucle}(Exp, Inst)\ \{\ Q\ \}$$

Il faut, tout d'abord, valider le triplet de Hoare suivant, après avoir défini un invariant et un variant de boucle :

$$\frac{\{\text{Invariant} \wedge (Exp = \text{true}) \wedge (\text{Variant} = \text{z})\}\ Inst\ \{\text{Invariant} \wedge (\text{Variant} < z)\},\quad \text{Invariant} \Rightarrow (\text{Variant} \geq 0)}{\{\text{Invariant}\}\ \text{Boucle}(Exp, Inst)\ \{\text{Invariant} \wedge (Exp = \text{false})\}}$$

et, en appliquant la règle d'inférence de la conséquence, il s'agit de démontrer que les assertions suivantes, appelées des obligations de preuve, sont valides :

$\text{Invariant} \wedge (Exp = \text{true}) \wedge (\text{Variant} = \text{z})$, qui signifie qu'à l'entrée de la boucle l'invariant doit être vérifié,
$\text{Invariant} \wedge (Exp = \text{false}) \Rightarrow Q$, qui signifie qu'en sortie de boucle la post-condition Q doit être vérifiée,
$\text{Invariant} \Rightarrow (\text{Variant} \geq 0)$, qui signifie que le variant doit rester positif.

Figure 21 : ***Obligations de preuve pour une boucle***

Si on reprend le programme donné en exemple en début du chapitre :

```
Programme p
        x, y :      Rel
        yc, y1c : Rel
        x := 20
        y := 0
        yc := 0
        y1c := 1
        -- P : { x > 0 }
        tant que  not ( x < y1c )
                y := y + 1
                yc := y1c
                y1c := yc + 2 * y + 1
        -- resultat y
        -- Q : { ( y * y <= x ) and ( x < ( y + 1 ) * ( y + 1 ) ) }
```

On peut vérifier que l'invariant et le variant de la boucle sont :

```
-- invariant :     ( y * y <= x ) and ( ( yc = y * y ) and ( y1c < ( y1 + 1 ) * ( y1 + 1 ) )
-- variant :       x – y1c }
```

Figure 22 : ***Invariant et variant de boucle du programme calculant la racine carrée entier d'un nombre entier***

Pour vérifier que la boucle est correcte par rapport à ses propriétés de pré-condition et post-condition, il faut démontrer que le triplet de Hoare suivant est valide :

```
{ x > 0 } tant que not x < y1c ) …   { ( y * y ) <= x and ( x < ( ( y + 1 ) * ( y + 1 ) ) }
```

avec les obligations de preuve qui en découlent.

Avant de développer les méthodes et les techniques de modélisation en UML/OCL des langages et de leurs propriétés rappelées dans ce chapitre, nous montrons les aspects communs que l'on peut retrouver entre les technologies des langages de programmation et celles des langages de modélisation, comme UML par exemple. Ce qui justifie le rapprochement entre ces deux espaces technologies qui sont cependant développés avec des objectifs très différents.

III L'Environnement de Modélisation et de Méta-Modélisation de UML

Les traducteurs et les environnements de modélisation sont amenés à gérer des structures de données qui leur permettent de représenter en interne les programmes des différents utilisateurs à traduire ou à interpréter pour les traducteurs, et les modèles prenant en compte les différentes exigences des applications, en ce qui concerne les environnements de modélisation,

Les traducteurs sont considérés comme des boites noires, masquant aux Analystes/Programmeurs ces différentes structures de données. Les langages de programmation ont donc des propriétés syntaxiques et sémantiques qui restent figées.

Par contre, les environnements de modélisation, comme UML par exemple, rendent possible l'accès aux structures de données représentant en interne les modèles. Il est ainsi possible aux experts des différents domaines d'applications, d'y apporter certaines modifications de manière à pouvoir prendre en compte au niveau du langage de modélisation des spécificités métier et de pouvoir y rajouter des propriétés compensant ainsi le peu de sémantique que peuvent exprimer les langages de modélisation basés sur des représentations graphiques. Cette remarque est importante dans la mesure où elle montre que les experts d'un domaine d'applications peuvent créer dynamiquement, sous certaines réserves, des langages de modélisation dont les modèles pourront jouer un rôle intermédiaire entre les exigences des applications à développer et les codes des applications.

Le rapprochement entre les technologies des langages de programmation et des langages de modélisation se situe donc à deux niveaux :

- Technique, puisqu'il s'agit d'appliquer les technologies des traducteurs au niveau des langages de modélisation et des modèles,
- Méthodologique, puisque les experts d'un domaine d'applications peuvent mettre en place dynamiquement des processus de développement de logiciels adaptés aux spécificités d'un domaine métier.

Nous montrons donc dans ce paragraphe, comment ces structures de données représentant en interne les codes des programmes, d'une part, et les modèles, d'autre part, permettront d'appliquer les technologies des langages de programmation au niveau des langages de modélisation.

III.1 L'environnement des données gérées par les traducteurs

Tout traducteur doit gérer, en interne les données représentant la grammaire et les propriétés du langage, ainsi que les programmes pris en charge en vue de leurs traduction ou de leurs interprétations. Tout traducteur déploie donc trois grandes activités lors de la prise en charge d'un programme : le chargement du programme et son décodage en fonction de la structure des données qui a été définie par les spécialistes dans l'environnement interne du traducteur du langage, la vérification des propriétés du langage et la traduction du programme dans un langage plus interne ou son interprétation, si toutes les propriétés du langage sont vérifiées.

III.1.a Programme source et représentation interne des données

De la grammaire d'un langage décrivant l'ensemble des constructions syntaxiques du langage, tel que le montre la figure suivante, le traducteur en déduit (1) les structures de données, appelées généralement métadonnées, permettent de représenter en interne (2) tout programme du langage sous la forme d'une structure arborescente :

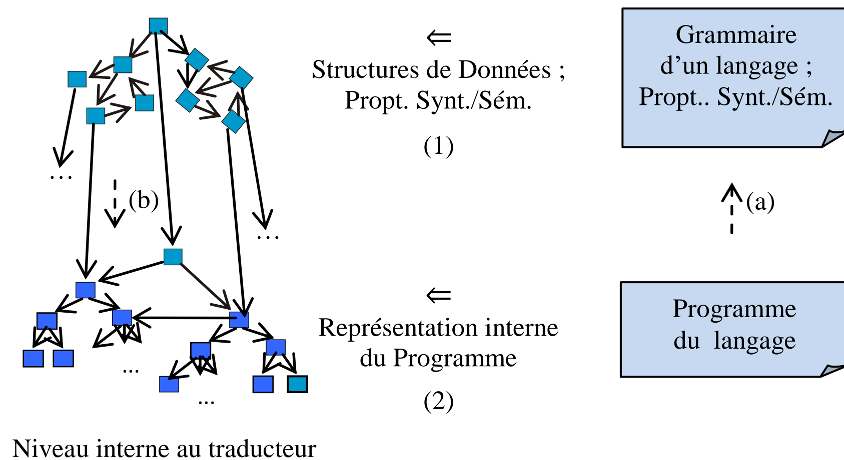


Figure 23 : ***Espace technologique des structures de données d'un traducteur de langage de programmation***

III.1.b Syntaxe concrète et abstraite

En général, on ne représente en interne que les concepts essentiels du programme, en faisant abstraction de tous les éléments syntaxiques nécessaires à tout programme source pour des raisons évidentes de lisibilité du code pour le programmeur, et pour le traducteur lors de la prise en charge. C'est la raison pour laquelle, on dit que l'on a en interne une représentation abstraire du programme. En général, on peut associer à une représentation abstraite d'un programme plusieurs représentations externes, chacune ayant une syntaxe bien définie.
On dira qu'un programme vérifie les propriétés du langage (a), si le traducteur a pu les vérifier (b) au niveau de sa représentation interne.

Le traducteur étant considéré comme une « boite noire », toutes les structures de données, mises en place et gérées par le traducteur restent inaccessibles aux programmeurs.

III.2 L'environnement des éditeurs de langage de modélisation

Au niveau modélisation, on retrouve, inévitablement, les mêmes niveaux de représentations externes et internes des données gérées par l'environnement de modélisation que pour un traducteur.

III.2.a Modèle et Méta-Modèle

Tout environnement de modélisation comprend entre autres, une interface interactive et ergonomique destinée aux Analystes/Concepteurs qui peuvent ainsi élaborer à l'écran les modèles sous forme graphique. L'ensemble des structures de données qui servent à accueillir en interne les données représentant un modèle est appelé le méta-modèle du langage de modélisation qui joue un double rôle, tel que le montre la figure suivante :

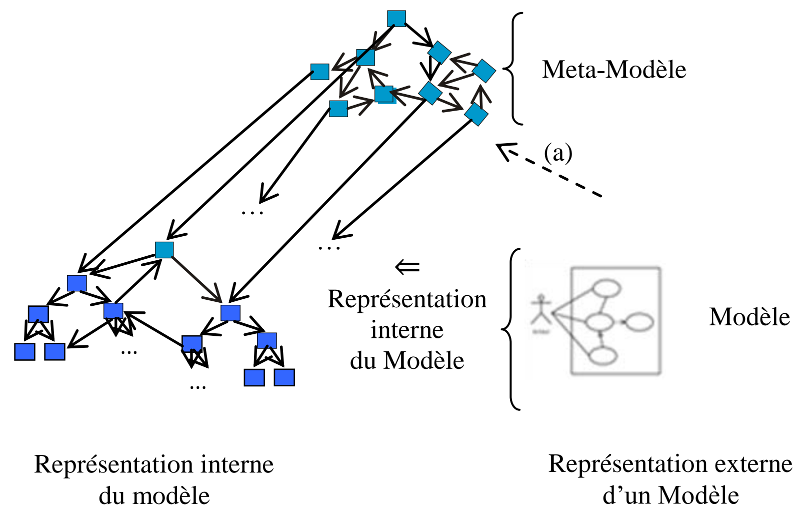


Figure 24 : ***Environnement de Modélisation d'un éditeur***

Le méta-modèle du langage de modélisation joue donc le rôle de grammaire du langage de modélisation.

Le méta-modèle du langage de modélisation est aussi l'ensemble des structures de données qui servent à accueillir, en interne, les données représentant tout modèle du langage. On dit que le modèle est conforme (a) à son méta-modèle si les données représentant en interne le modèle vérifient les propriétés définies dans le méta-modèle.

Le méta-modèle étant accessible aux concepteurs, les experts d'un domaine d'applications peuvent donc, sous certaines réserves, modifier le méta-modèle du langage et injecter dans le méta-modèle des spécificités métier du domaine d'applications.

III.2.b Abstraction et raffinage de modèles

Le Méta-Modèle d'un langage pouvant donc être modifié en fonction de certaines spécificités métier d'un domaine d'application, peut avoir un pouvoir d'abstraction plus important que ce que peut offrir un traducteur. Tout modèle d'une application peut donc jouer un rôle intermédiaire entre les exigences de l'application à développer et les programmes de l'application.

Les modifications que l'on peut apporter sur un Méta-Modèle ne peuvent être réalisées que par des experts dans la mesure où la modélisation des spécificités métier d'un domaine au niveau d'un Méta-Modèle doit rester cohérente par rapport aux propriétés déjà existantes, en l'occurrence celles qui ont fait l'objet de ce chapitre sur le langage « L ». Le Méta-Modèle « métier » peut prendre en compte, par exemple, des spécificités métier au niveau des métadonnées, ou des propriétés plus restrictives que celles qui sont définies au niveau du langage.

Il est cependant possible aux experts de définir plusieurs niveaux d'abstraction de manière à pouvoir effectuer le passage d'un modèle de conception en un modèle de programmation par raffinages successifs, de la même manière que dans un processus IDM, l'élaboration d'un modèle de conception devrait se réaliser par étapes successives en intégrant au fur et à mesure les différentes exigences des applications.

Avant d'aborder le chapitre portant sur les méthodes et les techniques de modélisation en UML et OCL des langages de programmation et de leurs propriétés syntaxiques et sémantiques, nous présentons, dans le chapitre suivant, un état de l'art des travaux de recherche les plus connexes par rapport à la modélisation des langages, ainsi qu'un rapide rappel sur les techniques de Méta-Modélisation en UML qui sont devenus des standards de fait et qui serviront à la mise en œuvre des logiciels que nous avons réalisés en vue de s'intégrer dans le cadre d'un processus IDM.

II – Modélisation des langages et Environnements de Méta-Modélisation

Ce chapitre est consacré à une étude préliminaire des travaux de recherche déjà existant dans le domaine de la modélisation en UML/OCL des langages de programmation.

Tout traducteur modélise les codes des programmes lors de leurs chargements en mémoire en vue d'y effectuer les traitements correspondants, en particulier vérification des propriétés, traduction dans des langages internes, interprétations, … . Les modèles de codes sont donc issus des codes. Cependant, cette étude, se situant dans un environnement de processus IDM,a une approche inverse dans la mesure où les codes sont issus des modèles.

Nous rappelons que pour vérifier que les codes reflètent bien les intentions des Analystes/Concepteurs, nous proposons de rajouter une activité de modélisation des codes : les modèles des codes se déduiront des modèles de conception (issus des exigences), et les codes se déduiront des modèles de codes.

On distingue donc dans un processus IDM, des modèles de conception représentant les exigences des applications à réaliser, appelés généralement les modèles de niveau PIM, et les modèles de programmation, les modèles PSM, à partir desquels les codes sont produits. S'assurer que les codes reflètent bien les intentions des Analystes/Concepteurs consiste donc à vérifier la cohérence entre les modèles de conception et les modèles de programmation.

Modéliser en UML/OCL les langages de programmation a donc pour objectif de rajouter les propriétés au niveau UML que devront respecter les modèles de codes correspondants. S'assurer que les codes reflètent bien les intentions des Analystes/Concepteurs consiste donc à vérifier la cohérence entre les propriétés exigées de la part des modèles de conception et de la part des modèles de programmation.

Par rapport à nos objectifs, les travaux déjà existants restent ponctuels. En effet, les modèles des codes sont définis à partir des modèles de grammaire des langages et ne s'intègrent pas dans un cadre plus général de processus de développement de logiciels dirigé par les modèles. Ces travaux se limitent essentiellement à la modélisation des propriétés syntaxiques. Il existe actuellement des environnements de modélisation du langage UML intégrant un évaluateur d'expressions OCL et d'opérations d'un langage d'actions. Il devient donc possible de pouvoir intégrer au niveau du langage de modélisation UML des propriétés syntaxiques et sémantiques, spécifiques à des domaines d'applications.

Les travaux les plus proches par rapport à nos objectifs restent bien sûr ceux que nous avons repris au niveau du premier chapitre traitant de la sémantique des langages de programmation [Moh] où la modélisation des aspects structurels des grammaires et l'implémentation de leurs propriétés syntaxiques et sémantiques ont été réalisées en langage Coq de manière à pouvoir prendre en compte les aspects preuve de programmes.

Après donc avoir rappelé ces premiers travaux de recherche qui décrivent le cadre de cette étude, nous décrivons dans ce chapitre les principes généraux de la modélisation et de la méta-modélisation qui constituent la base de notre travail. Nous terminerons ce chapitre par une présentation succincte des projets de recherche dans lesquels nous nous sommes impliqués et qui nous ont accompagnés tout au long de cette étude.

II – Modélisation des langages et Environnements de Méta-Modélisation

Table des matières

I Modélisation des langages de programmation : Etat de l'art ... 46

I.1 Modélisation de la grammaire d'un langage de programmation ... 47

I.2 Sémantique des langages de programmation avec Alloy ... 49

I.3 Syntaxe concrète et syntaxe abstraite ... 49

I.4 Gallina ... 50

II Modélisation et Méta-Modélisation : la démarche MDA ... 51

II.1 La démarche MDA ... 52

II.1.a Conception des applications portables ... 52

II.1.b Mise en œuvre de la démarche ... 52

II.1.c Les niveaux de modélisation PIM et PSM de la démarche ... 53

II.1.d Modélisation et Méta-Modélisation ... 53

II.1.e Langage de Méta-Modélisation Objet MOF ... 54

II.2 Environnement de Modélisation et de Méta-Modélisation du langage UML ... 55

II.2.a Le langage de Modélisation UML et la hiérarchie des différents niveaux de Modélisation et Méta-Modélisation ... 55

II.2.b Modélisation UML, et Environnement d'exécution ... 56

II.2.c Exemple d'un modèle UML ... 58

II.2.d Propriétés sur un modèle ... 58

II.3 Méta-Modèle UML (MM UML) ... 59

II.3.a Exemple d'un fragment de MM UML ... 59

II.3.b Fragment du MM UML définissant les éléments de modélisation du diagramme de classes ... 60

II.3.c Représentation interne du modèle de l'application, à l'aide d'un diagramme d'objets ... 62

II.3.d Mise en œuvre de la méta-propriété ... 64

II.4 Profil UML ... 64

II.4.a Fragment du Méta-Modèle définissant les éléments de modélisation pour les profils ... 65

II.5 XML Métadata Interchange (XMI) ... 67

II.5.a Principe ... 67

II.5.b Exemple ... 67

III Les Projets de Recherche et les Environnements de développements qui ont accompagné cette étude ... 70

III.1 Les Projets de recherche ... 70

III.1.a Le projet DOMINO ... 71

III.1.b Le projet TopCased ... 71

III.1.c Le projet MyCitizSpace ... 72

III.2 Les Environnements de développement ... 73

III.2.a L'environnement de spécification USE basé sur UML/OCL ... 73

III.2.b L'Environnement TopCased et Le langage de méta-modélisation exécutable KerMeta ... 74

III.2.c QVT, ATL ... 75

III.2.d Java/EMF ... 75

II – Modélisation des langages et Environnements de Méta-Modélisation

Table des figures

Figure 1 :Définition de la grammaire d'un DSL avec xText 48
Figure 2 : Editor produit par xText pour le DSL précédent 48
Figure 3 : Unification du processus d'analyse et de synthèse [MullP08] 50
Figure 4 : Les étapes d'une démarche MDA 52
Figure 5 : Hiérarchie des niveaux de Modélisation et de Méta-Modélisation (OMG) 54
Figure 6 : Hiérarchie des différents niveaux de Méta-Modélisationà 4 niveaux (OMG) 55
Figure 7 : Les différents éléments de modélisation de UML resitués dans la hiérarchie des niveaux de Modélisation et de Méta-Modélisation 56
Figure 8 : L'environnement simplifié d'un éditeur UML 57
Figure 9 : Exemple d'un diagramme de classes d'un modèle UML d'une application 58
Figure 10 : Propriétés sur les données d'une application 58
Figure 11 : Fragment du méta-Modèle des deux grandes familles d'éléments qui forment le contenu des Modèles 60
Figure 12 : Fragment du MM UML définissant un modèle UML comme un package 61
Figure 13 : Fragment simplifié du MM UML relatif aux éléments de modélisation du diagramme de classes 62
Figure 14 : Représentation interne du modèle UML, instance du MM UML 63
Figure 15 : Exemple de mise en œuvre d'une propriété définie au niveau du MM UML, s'appliquant sur le modèle 64
Figure 16 : Profil UML pour un domaine d'applications 65
Figure 17 : Fragment du Méta-Modèle UML concernant les éléments de modélisation des profils 65
Figure 18 : Fragment du Méta-Modèle UML concernant les éléments de modélisation des profils (Suite) 66
Figure 19 : Exemple d'un profil 66
Figure 20 : Fragment du document XMI de l'exemple UML, définissant les classes et leurs propriétés structurelles 68
Figure 21 :Fragment du document XMI de l'exemple UML, définissant l'association et ses propriétés structurelles 69
Figure 22 :Fragment du document XMI de l'exemple UML, décrivant les profils où sont définis les types primitifs 70
Figure 23 : Diagramme montrant les principaux concepts de USE 73

II – Modélisation des langages et Environnements de Méta-Modélisation

I Modélisation des langages de programmation : Etat de l'art

Dans ce paragraphe, nous décrivons les travaux qui ont contribué au rapprochement des technologies des grammaires et des modèles. Il s'agit d'intégrer lors d'un processus de développement IDM d'une application, un niveau de modélisation en UML/OCL des codes d'une application, afin de pouvoir vérifier la cohérence entre les codes et les modèles prenant en compte les exigences de l'application.

Le langage de modélisation UML est le résultat d'une très longue expérience depuis les travaux de Booch, Jacobson et Rumbaugh qui ont proposé une méthode de conception unifiée en 1995 basée sur le concept objet. Adoptée par l'OMG en 1997, UML est devenu un standard de fait. Depuis, suite à de nombreux retours du monde de l'entreprise, des mises à jour et des transformations d'UML ne cessent d'être effectuées pour supprimer les incohérences, apporter les améliorations et ajouter de nouveaux concepts. La récente version d'UML (Version 2.0) est le résultat de travaux qui ont été réalisés en particulier sur l'identification et la définition de la sémantique des concepts fondamentaux formant les briques de base de la modélisation objet. Ces concepts constituent les artefacts du développement que les intervenants d'un projet doivent s'échanger. De manière à réaliser et faciliter ces échanges, le Méta-Modèle de UML (MM UML) est décrit formellement en classant les concepts par niveau d'abstraction (Architecture à quatre niveaux de modélisation), de complexité (Vues statiques et dynamiques des différents diagrammes) et de domaine d'applications (Notion de profil basé sur des techniques de généricité). Ce MM sert de description de référence pour la construction d'outils et le partage de modèles entre les différents outils. Le langage OCL, basé sur la logique des prédicats, permet de donner une définition syntaxique et sémantique des différents artefacts de développement apportant la précision qui manque à une notation graphique. Enfin un langage d'actions intégrant OCL peut rendre les modèles exécutables si l'on dispose d'un environnement d'exécution laissant percevoir aux Analystes/Concepteurs une première idée de ce que pourra être leurs applications, à différents niveaux d'abstraction. Ce langage d'actions (LA) est spécifié comme on le fait traditionnellement pour les langages de programmation.

Connexe aux technologies des langages de programmation, et des langages de modélisation, cette thématique portant sur la modélisation en UML/OCL des langages de programmation et de leurs propriétés fait l'objet d'une intense activité de recherche. L'approche est cependant différente dans la mesure où les traducteurs de langages de programmation, considérés comme des boites noires, masquent aux Analystes/Programmeurs les structures de données codant en interne les grammaires des langages et leurs programmes, alors que l'objectif du langage de modélisation UML est de permettre aux Analystes/Concepteurs d'agir sur le Méta-Modèle de manière à ce que les modèles puissent prendre en compte les spécificités métier facilitant ainsi le passage entre les exigences des applications d'une part et les codes des applications d'autre part.

Au chapitre précédent, nous avons rappelé les travaux sur les langages de programmation portant sur les approches théoriques des propriétés opérationnelles et axiomatiques des langages [MyerB92] et sur la spécification de ces propriétés décrites dans le langage Coq spécialisé dans la preuve de programme. Outre ces travaux, on trouve des travaux de recherche relativement ponctuels sur la modélisation en UML des langages de programmation, plus particulièrement sur la représentation en UML de la syntaxe concrète et abstraite des langages [Joua06b, MullP06, MullP08,…], ou plus récemment sur la modélisation en UML/OCL des langages de programmation [Male02] allant jusqu'à la spécification en OCL des propriétés opérationnelles des langages, à l'aide d'invariants et de pré- et pos-conditions. Cependant, l'implémentation des propriétés comportementales dans un langage d'actions n'est pas proposée, et en fait peu de travaux de recherche ont été réalisés dans ce domaine [Duch]. En fait, on trouve des travaux sur les processus de développement de logiciels dirigé par les modèles, plus proches donc de nos préoccupations, en particulier sur les processus itératifs et incrémentaux [Fomb07] assurant le bon déroulement du processus à chacune de ses phases essentielles.

Nous rappelons les travaux les plus en rapport et les plus significatifs avec cette étude, en particulier ceux qui ont été développés autour du framework xText, développé dans l'environnement Eclipse, ou dans le cadre du langage Alloy. Nous terminons ce paragraphe en citant les travaux réalisés autour de Syntax faisant le lien entre la syntaxe abstraite et concrète et certains travaux relativement plus anciens mais qui restent toujours d'actualité. Il existe, cependant, d'autres travaux de recherche connexes à ces domaines de recherche, en particulier : TCS & Sintak [MullP06, MullP08], HUTN [MullP05b], les travaux de [Alan03] réalisant le rapprochement entre les technologies des grammaires (Grammarware) et des modèles (Modelware) [Wimm06]. Par contre, pour des références à des travaux de recherche plus importants, nous les évoquerons au fur et à mesure puisqu'ils nécessitent d'être replacés dans leur contexte de recherche, comme par exemple les travaux sur les langages de description ou les travaux sur UML et B.

I.1 Modélisation de la grammaire d'un langage de programmation

xText[xText] est un framework de développement des langages de programmation et des langages spécifiques à des domaines d'applications (DSL[1]). L'outil, à partir la grammaire du langage décrite en suivant la grammaire EBNF[2] simplifiée, produit un parseur, un éditeur du langage ainsi que le méta-modèle de l'arbre de syntaxe abstraite. Le framework xText intègre tous les technologies d'Eclipse – EMF[3], GMF[4], M2T[5] et une partie de EMFT[6] et fournit un environnement de développement des langages.

Les deux figures suivantes montrent un exemple d'utilisation de xText au travers de son interface interactive : la première figure montre un écran où la grammaire d'un DSL a été introduite sous une forme relative EBNF ; la deuxième figure montre l'interface de l'éditeur de ce DSL produit par xText.

[1] Domain Specific Language
[2] Extended Backus-Naur Form
[3] Eclipse Modeling Framework (http://www.eclipse.org/emf/)
[4] Eclipse Graphic Modeling Framework (http://www.eclipse.org/modeling/gmf/)
[5] Model-to-Text (http://www.eclipse.org/modeling/gmf/)
[6] Eclipse Modeling Framework Technology (http://www.eclipse.org/emft)

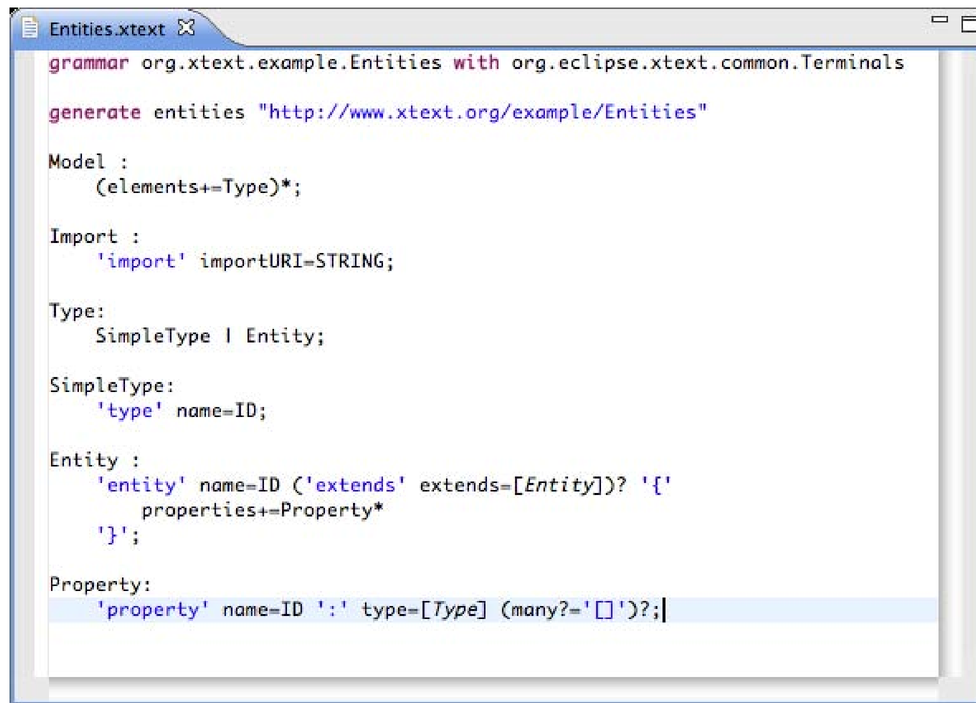


*Figure 1 :**Définition de la grammaire d'un DSL avec xText***

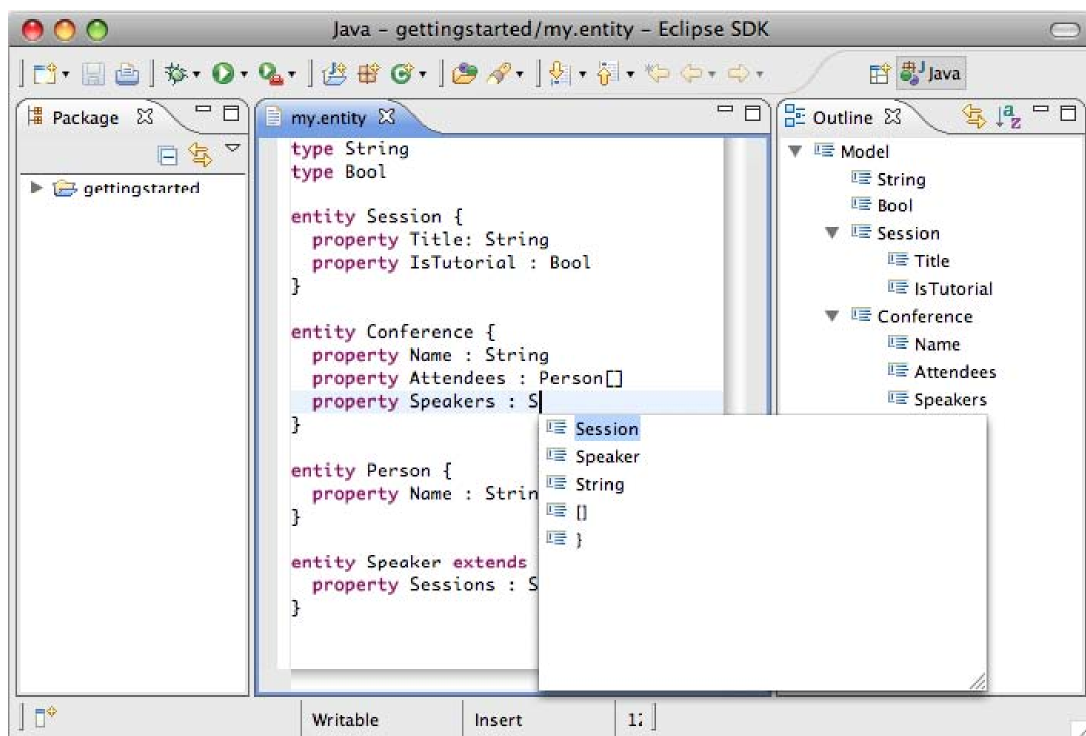


Figure 2 ***: Editor produit par xText pour le DSL précédent***

xText mets aussi à notre disposition la possibilité d'obtenir un projet de base pour le générateur du codes du DSL. Il suffit alors de compléter ce squelette, en utilisant le langage spécifique M2T Xpand, afin de générer à partir d'un modèle les codes du DSL correspondant. [xText].

xText facilite ainsi l'ingénierie des DSLs en nous aidant à construire rapidement des outils autour du DSL. Il favorise les changements potentiels des concepts du DSLs (correction de la grammaire et re-génération de l'éditeur). xText est développé sous Eclipse/EMF, donc permettant de profiter tous les avantages de cet environnement, en particulier lors des traitements effectués sur des modèles du DSL.

I.2 Sémantique des langages de programmation avec Alloy

Alloy [Kels08] permet de spécifier des ensembles de prédicats et d'assertions définissant les propriétés d'un système. C'est un langage et un outil de vérification et de validation de modèles formels. Il a été développé au MIT. Alloy permet de modéliser les systèmes afin de simuler, vérifier et valider des propriétés. Basé sur le langage Z, il permet de présenter une vue simplifiée des systèmes en mettant l'accent sur les propriétés et les contraintes.

Alloy est un langage déclaratif qui permet de modéliser des structures complexes auxquelles on peut rajouter des entités intégrant des propriétés et des contraintes. C'est un analyseur chargé de vérifier les structures et leurs propriétés ainsi définies. Alloy utilise deux notions de base qui sont, d'une part les atomes, entités élémentaires servant à modéliser les aspects du monde réel, et d'autre part les relations servant à définir des corrélations entre les atomes. Les spécifications de logiciels peuvent ainsi être vérifiées.

Le langage Alloy peut être utilisé pour décrire la syntaxe abstraite des langages et leurs sémantiques statiques et dynamiques. L'analyseur autorise donc une élaboration incrémentale de la modélisation des langages et de leurs propriétés, permettant en particulier de vérifier la cohérence entre les propriétés des langages et les propriétés issues des exigences de l'application, et donnant la possibilité aux Concepteurs d'intervenir dès qu'une erreur est signalée.

Les modèles [Kels08] sont élaborés en utilisant un ensemble de types, appelés signature, qui peuvent posséder plusieurs champs (*Fields*). Les *fact*s servent à décrire les contraintes qui sont des propriétés additionnelles que l'on peut ajouter dans le modèle. Les *prédicat*s sont les contraintes paramétrées que l'on peut utiliser pour décrire les opérations. Les *function*s sont les expressions qui renvoient un résultat. Les assertions sont les hypothèses sur le modèle qui peuvent être vérifiées par l'analyseur d'Alloy. Un exemple d'utilisation de Alloy peut être trouvé dans [Mank] décrivant un langage de description des politiques de contrôle d'accès et de vérification des accès.

Le langage Alloy, fournissant un formalisme unique et uniforme pour toute description, permet donc de chercher dans un modèle une instance répondant à un prédicat, ou un contre-exemple d'une hypothèse.

Avec cette approche de P.Kelsen et Q. Ma, nous avons une description formelle et uniforme d'un langage avec ses sémantiques et les moyens de vérifier la cohérence de la spécification elle-même.

I.3 Syntaxe concrète et syntaxe abstraite

Le pont entre la syntaxe concrète et abstraite fait l'objet des différents travaux de recherches dont TCS et Sintaks[MullP06, MullP08] en font partie. L'idée est de construire des outils qui facilitent la conversion entre ces deux types de syntaxe du langage. Généralement, ces outils automatisent le processus de synthèse et d'analyse des langages, en basant sur le Méta-Modèle abstrait du langage.

Ces deux travaux mettent l'accent sur la différence entre les niveaux de méta-modélisation des langages et de leurs propriétés, et de modélisation des codes, représentés selon des structures arborescentes, issues et déduites de la syntaxe concrète des langages. Cependant, chaque langage peut avoir plusieurs syntaxes concrètes. Déduire l'arbre abstrait à partir d'une syntaxe concrète amène la dépendance de l'arbre avec cette spécifique syntaxe concrète. C'est la raison pour laquelle dans ces deux études, il est proposé de construire le Méta-Modèle quelque soit la grammaire du langage. De cette manière, le Méta-Modèle peut prendre en compte toutes les abstractions du langage indépendamment de la syntaxe concrète.

On peut encore citer ces deux travaux de recherche :

- L'outil TCSSL, développé par CEA-LIST, propose un « pretty printer » pour produire les textes structurés correspondants aux modèles et pour générer la spécification de compilateurs de compilateurs construisant les modèles à partir des textes.
- L'outil Sintak (équipe Triskell – Rennes) [Sintak], développé dans le contexte du projet Kermeta, est un métalangage qui peut être utilisé pour exprimer la syntaxe abstraite et aussi la sémantique opérationnelle du langage.
-

L'approche proposée par ces deux travaux est montrée dans la figure suivante :

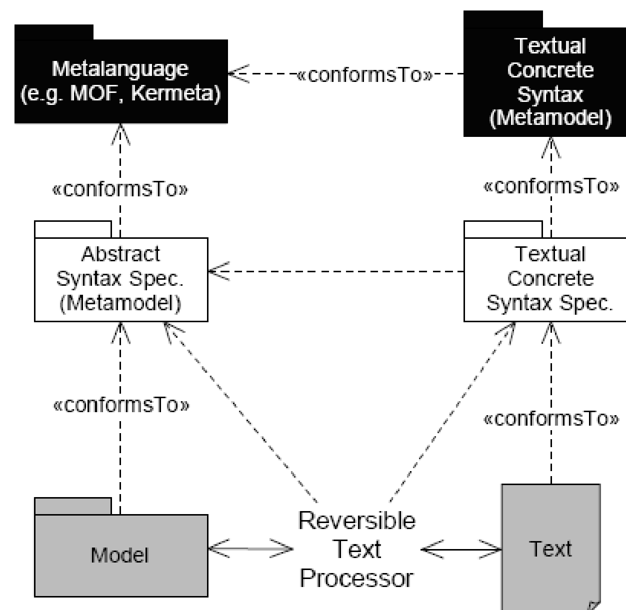


Figure 3 : ***Unification du processus d'analyse et de synthèse [MullP08]***

I.4 Gallina

Les rappels sur les langages de programmation et leurs propriétés effectués dans le chapitre précédent, ont été très fortement inspirés de [MyerB92] pour les concepts généraux. Cependant, les travaux réalisés autour de [Moh] ont été déterminants dans la mesure où nous en avons repris tout le formalisme que nous avons adapté à une notation fonctionnelle anticipant une modélisation en OCL et en langage d'actions UML, permettant en particulier, de manipuler des types comme des objets. Cependant, les travaux de recherche autour de [Moh] principalement axés sur la construction effective de preuves formelles et de leur vérification sur ordinateur sont réalisés en Coq. En ce qui nous concerne, la preuve de

programme nécessitant une interaction avec l'utilisateur, nous avons préféré faire appel à un Atelier B dont les approches de modélisation sont relativement proches de UML. Par contre, comme on le verra par la suite, le formalisme de OCL étendu au langage d'actions de UML se prête bien à une implémentation d'un formalisme mathématique basé sur la logique des prédicats pouvant s'intégrer dans un processus de développement logiciel dirigé par les modèles où l'un des objectifs est d'assurer un certaine cohérence entre la spécification de logiciels et leurs implémentations.

L'ensemble de ces travaux permet déjà d'entrevoir les solutions que nous proposerons pour la modélisation des langages de programmation et de leurs propriétés. Cependant, l'on sera amené à proposer des modèles qui devront s'intégrer dans un processus de développement logiciels où s'impliquent différentes équipes d'Analystes/Concepteurs à des niveaux différents de modélisation mais dont les modèles devront s'enchainer tout au long d'un processus de développement. Pour cela, nous nous basons sur la démarche MDA[7] (IDM en français)où les modèles sont au centre du développement. La démarche MDA, proposée par l'OMG[8], a tendance à devenir un standard de fait.

II Modélisation et Méta-Modélisation : la démarche MDA

La démarche MDA est basée sur la séparation entre les spécifications fonctionnelles d'un système et les spécifications de son implémentation sur une plate-forme donnée. Cette démarche permet de réaliser le même modèle sur plusieurs plates-formes grâce à des projections standardisées. Elle permet aux applications d'inter-opérer en reliant leurs modèles et facilite l'évolution des plates-formes et des techniques. La mise en œuvre de la démarche MDA est entièrement basée sur les modèles et leurs transformations.

Nous rappelons, dans ce paragraphe, les principales caractéristiques de cette démarche qui apporte des solutions, à la fois méthodologiques en ce qui concerne les modèles prenant en compte les exigences des applications et leurs évolutions tout au long du processus, et technologiques en ce qui concerne les environnements de Méta-Modélisation prenant en charge et gérant les modèles. Au cœur de la démarche MDA se trouvent plusieurs standards définis par l'OMG, en particulier le langage de modélisation UML[9], le langage de Méta-Modélisation MOF[10], ou le langage XMI[11], développé pour échanger des instances du MOF . Ces standards définissent l'infrastructure du MDA. Forts de cette standardisation, certains outils et certaines plates-formes sont développés autour de ces standards, parmi lesquels on peut citer les logiciels qui ont été réutilisés dans cette étude : la plate-forme de modélisation et de génération de codes EMF[12] plugée directement à Eclipse, les éditeurs UML, la plate-forme Kermeta pouvant échanger des modèles spécifiés en XMI, ou l'évaluateur dynamique d'un langage d'actions UML intégrant des expressions OCL : la plate-forme USE[13].

[7] Model Driven Architecture (*http://www.omg.org/mda/specs.html*)
[8] Object Management Group (*http://www.omg.org*)
[9] Unified Model Language (*http://www.uml.org*)
[10] Meta Object Faciliy
[11] XML Metadata Interchange (*http://omg/technology/documents/formal/xmi.html*)
[12] Eclipse Modeling Framework (*http://www.eclipse/org/emf*)
[13]UML_based Specification Environment (*http://www.db.informtik.univ-bremen.de*)

II.1 La démarche MDA

Basée sur une architecture de spécification structurée en modèles indépendants des plates-formes et en modèles spécifiques aux plates-formes cibles, la démarche MDA a été développée afin de résoudre les problèmes d'interopérabilité et de portabilité dès le niveau modélisation.

II.1.a Conception des applications portables

MDA se veut indépendante de toute plate-forme et de tout système. Elle se place donc dans un contexte de conception d'applications portables au niveau des langages de programmation, des systèmes d'exploitation mais aussi des middlewares. Cette indépendance totale vise à changer d'infrastructure sans perdre ce qui a déjà été conçu. Elle assure ainsi une certaine capitalisation du travail effectué pendant les phases d'analyse et de conception.

MDA se présente en différentes couches de spécification : au cœur se trouvent les techniques (UML, MOF, CWM), autour quelques unes des plates-formes supportées, en surface les services systèmes et enfin à l'extérieur les domaines pour lesquels les composants métiers doivent être définis (Domain Facilities). Ces services (aussi bien systèmes que métiers) doivent être disponibles dès les premières phases de modélisation, c'est pourquoi ils doivent faire partie des spécifications du MDA.

II.1.b Mise en œuvre de la démarche

La démarche MDA supporte les différentes activités du développement et standardise le passage d'une activité à une autre. Elle peut se découper en quatre types d'activités, tel que le montre la figure suivante où les activités (2) et (4) peuvent être répétées un nombre indéterminé de fois :

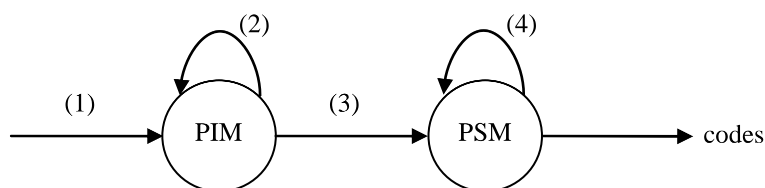


Figure 4 : ***Les étapes d'une démarche MDA***

- La première activité (1) initie le processus de développement en élaborant à partir d'exigences du système à développer un modèle indépendant de toute plate-forme cible, et dans une deuxième étape, de l'enrichir (2) en prenant en compte au fur et à mesure toutes les exigences. On obtient ainsi un modèle de niveau PIM[14].

- Les deux dernières activités qui dépendent de la plate-formecorrespondent donc au passage des modèles de niveau PIM (3) vers des modèles de niveau PSM[15] dépendant des plates-formes cibles, et leurs raffinements (4) en intégrant au fur et à mesure les caractéristiques et les spécificités de ces plates-formes cibles et en prenant compte des exigences pouvant dépendre de ces plates-formes.

[14]Plateform Independent Model
[15]Plateform Specific Model

Selon une démarche MDA où le code se déduit des modèles de niveau PSM, tout est considéré comme modèle, aussi bien les schémas que les codes source ou binaires. Les deux types de modèles identifiés sont donc des PIM et des PSM. Chaque activité correspond à la transformation d'une modèle vers un autre (du même type ou non).

II.1.c Les niveaux de modélisation PIM et PSM de la démarche

Les modèles de niveau PIM représentent uniquement les capacités fonctionnelles métiers et le comportement du système, sans considération technologique. La clarté de ce niveau de modélisation doit permettre à des experts d'un domaine d'application de comprendre la prise en compte des exigences. L'objectif est de leur permettre ainsi de vérifier qu'à un certain niveau de modélisation, le modèle est complet et correct. Ce niveau d'abstraction des modèles facilite la communication entre les différentes équipes impliquées dans le processus de développement et facilite ainsi le report au niveau des modèles PIM, des mises à jour qu'une maintenance évolutive pourrait exiger.

Les modèles de niveau PSM tiennent compte des caractéristiques d'exécution et des informations de configuration des plates-formes cibles. L'idée générale serait d'assimiler les modèles de niveau PIM à des modèles génériques dont les modèles de niveau PSM ne seraient que des instanciations des modèles PIM. Il reste cependant difficile d'associer l'activité de transformation d'une modélisation de niveau PIM en une modélisation de niveau PSMà une opération d'instanciation pouvant se réaliser d'une manière automatique.

Dans le cadre du processus 2TUP[16] [Roqu], appelé aussi cycle en Y, il est proposé de modéliser d'une part, les aspects fonctionnels des applications donnant les modèles de niveau PIM et d'autre part les aspects techniques, correspondant par exemple à la capture des besoins techniques, à la définition de l'architecture logicielle et applicative, et au framework technique. L'activité de conception débouchant donc sur des modèles de niveau PSM est alors assimilée à des opérations de jointure de modèles obtenus après avoir pris en compte les aspects fonctionnels et les aspects techniques. Cette opération de jointure peut se faire par étapes successives de raffinement jusqu'aux codes des applications.

II.1.d Modélisation et Méta-Modélisation

Un processus de développement de logiciels d'une application dirigé par les modèles peut être vu comme une succession de transformations de modèles depuis la prise en compte des exigences de l'application jusqu'à la génération des codes. Les transformations de modèles, automatiques ou partiellement automatiques, ont pour rôle d'assister et d'aider les Analystes/Concepteurs tout au long du processus en les guidant éventuellement selon les différentes activités du processus. Il s'agit de vérifier que les modèles ont bien les qualités que l'on peut attendre de leur part, assurant ainsi le bon déroulement du processus à chacune de ses phases importantes.

La figure suivante montre la hiérarchie définissant les niveaux de modélisation et de méta-modélisation standardisés par l'OMG :

[16] Two Track Unified Process

M2 : Méta-Modélisation	Méta-Modèle
M1 : Modélisation	Modèle d'une application
M0 : Données, Objets	Instances d'une l'application

Figure 5 : ***Hiérarchie des niveaux de Modélisation et de Méta-Modélisation (OMG)***

Le niveau M0 correspond aux données (ou les objets) des applications.
Le niveau M1 correspond aux modèles des applications.
Enfin, le niveau M2 où est décrit le langage de modélisation. A ce niveau, on trouve donc la description dela syntaxe des diagrammes que les Analystes/Concepteurs doivent suivre lors de l'élaboration des différents modèles de leurs applications. Compte tenu du double rôle joué par le Méta-Modèle, c'est aussi la description des structures de données utilisées par les éditeurs de modèles pour enregistrer les objets représentant en interne les modèles.

II.1.e Langage de Méta-Modélisation Objet MOF

L'OMG a défini au sommet de la hiérarchie, c'est-à-dire au niveau M3, le standard MOF[17]. Ce niveau correspond au langage de Méta-Modélisation assurant ainsi l'échange de constructions utilisées par le MDA. Les autres modèles standards de l'OMG, comme UML et CWM, sont définis par des constructions MOF, ce qui permet de les relier entre elles assurant le mécanisme par lequel les modèles sont sérialisés en XMI. Le MOF est un exemple de méta-méta-modèles, ou de modèle du méta-modèle. Il définit ainsi la syntaxe et la structure des méta-modèles, utilisés pour construire des modèles orientés objet. La spécification MOF fournit les points suivants :

- Un modèle abstrait d'objets MOF génériques et leurs associations.
- Un ensemble de règles pour exprimer en méta-Modèle MOF à l'aide d'interface IDL, et une implantation de ces interfaces pour un méta-modèle donné peut être utilisée pour manipuler une instance de celui-ci (un modèle).
- Un ensemble de règles sur le cycle de vie, la composition et la fermeture sémantique des éléments d'un méta-modèle MOF.
- Une hiérarchie d'interfaces réflexives permettant de découvrir et manipuler des modèles basés sur des méta-modèles MOF dont on ne connait pas les interfaces.

Un des intérêts du MOF est de pouvoir échanger des méta-modèles différents. Une application MOF peut manipuler un modèle à l'aide d'opérations génériques sans connaissance du domaine.

La figure suivante montre la hiérarchie de modélisation et de méta-modélisation à niveaux définie par l'OMG :

[17] Meta-Object Facility (*http://www.omg.org/mof*)

M3 : Méta-Méta-Modélisation	Méta-Méta-Modèle (MOF)
M2 : Méta-Modélisation	Méta-Modèle
M1 : Modélisation	Modèle d'une application
M0 : Données, Objets	Instances d'une l'application

Figure 6 : ***Hiérarchie des différents niveaux de Méta-Modélisationà 4 niveaux (OMG)***

Cette figure montre qu'au niveau M3, le MOF est un langage de Méta-Modélisation, à partir duquel on devrait pouvoir décrire la syntaxe graphique de tout langage de Méta-Modélisation.

II.2 Environnement de Modélisation et de Méta-Modélisation du langage UML

Dans ce paragraphe, nous rappelons les aspects importants de UML. On peut trouver de très nombreux documents sur le langage de modélisation UML, en particulier la norme officielle de UML [UML], et un certain nombre de documents, en particulier [MULL] où l'on trouve une présentation détaillée du langage OCL, des cours sur Internet, un résumé très complet dans le numéro de la revue des Techniques de l'Ingénieur [GIRO] consacré sur UML, ou bien sûr des cours [AUDI].

II.2.a Le langage de Modélisation UML et la hiérarchie des différents niveaux de Modélisation et Méta-Modélisation

Le langage UML a été défini par l'**OMG** (*Object Management Group*), dans le but de mettre au point des standards garantissant la compatibilité entre des applications programmées à l'aide de langages objet et fonctionnant sur des réseaux hétérogènes (de différents types). Il s'agissait de définir une méthode objet qui permette de modéliser les exigences d'une application de manière rigoureuse et unique afin de lever les ambigüités. De nombreuses méthodes objet ont été définies, mais aucune n'a pu s'imposer en raison du manque de standardisation. A partir de 1997, UML est devenue une norme de l'OMG, ce qui lui a permis de s'imposer en tant que méthode de développement objet et être reconnue et utilisée par de nombreuses entreprises.

La figure suivante montre l'architecture des différents niveaux de modélisation définie par l'OMG, appliqués au langage de modélisation UML :

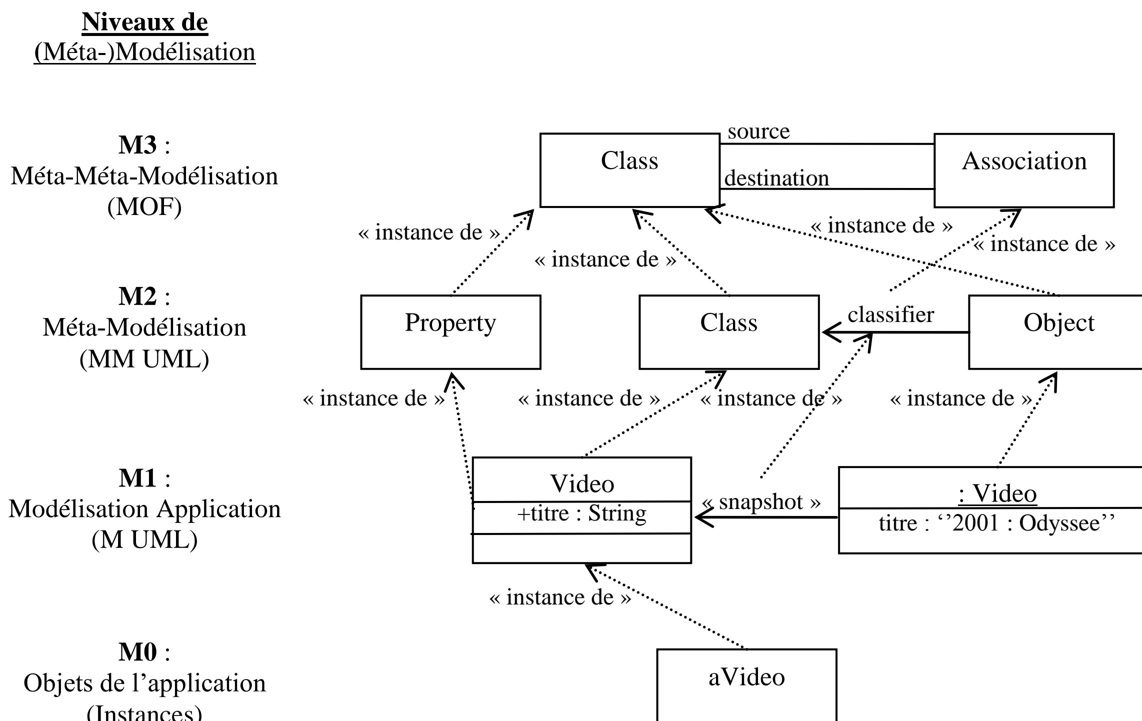


Figure 7 : ***Les différents éléments de modélisation de UML resitués dans la hiérarchie des niveaux de Modélisation et de Méta-Modélisation***

Cette figure montre que les objets d'une application se situent au niveau M0. Au niveau M1, on trouve les différents diagrammes constituant les Modèles des applications, en l'occurrence la classe Video. Ces diagrammes suivent une syntaxe qui est décrite au niveau Méta-Modélisation, précisant par exemple qu'un modèle peut être constitué de classes (Meta-Class Class), chacune pouvant avoir un certains nombre de propriétés (Méta-Class Property). Le Méta-Modèle UML (MM UML) est décrit par une syntaxe se situant au niveau Méta-Méta-Modélisation où se situe le langage MOF, appelé EMF (ECORE) dans l'environnement Eclipse.

II.2.b Modélisation UML, et Environnement d'exécution

UML n'est pas une méthode. C'est un langage de modélisation graphique pour représenter les divers aspects d'un système d'information, facilitant ainsi la communication entre les différentes équipes impliqués dans le processus de développement des logiciels d'une application. Aux graphiques, peuvent bien sûr être associés des textes qui expliquent leur contenu. UML est donc un métalangage qui fournit les éléments permettant de construire le modèle qui, lui, sera le langage du projet. S'il est impossible de donner une représentation graphique complète d'un logiciel, ou de tout autre système complexe, il est possible d'en donner des vues partielles dont l'ensemble pourra en donner une idée complète.

UML 2.0 est basé sur la notion de diagrammes représentant autant de vues distinctes pour représenter des concepts particuliers du système. Ils se répartissent en deux grands groupes : Les diagrammes modélisant d'une part les aspects structurels ou statiques (*UML Structure*) du système à modéliser et à développer, et d'autre part, les aspects dynamiques ou comportementaux (UML Behavior). Ces diagrammes, d'une utilité variable selon les cas, ne

sont pas nécessairement tous produits à l'occasion d'une modélisation. Les plus couramment utilisés sont les diagrammes d'activité, de cas d'utilisation, de classes, d'objets, de séquence et d'états-transitions.

Le langage de contraintes OCL, basé sur la logique des prédicats, est partie intégrante de UML. Il permet de prendre en compte, sous forme de propriétés, les exigences des applications à développer qui ne peuvent s'exprimer graphiquement à l'aide des diagrammes UML apportant ainsi le complément indispensable à une représentation graphique des modèles. Basé sur la logique des prédicats du premier ordre, et reprenant le typage de UML, le langage OCL peut être d'un abord difficile. C'est un langage fonctionnel qui, comme le langage de bases de données relationnel SQL (StructuredQueryLanguage), permet de naviguer au sein d'un modèle de manière à pouvoir exprimer des propriétés sous la forme d'invariants, de pré- et post-conditions, qui pourront ensuite être vérifiées sur les objets. Un langage d'actions complète le langage OCL. Il permet de spécifier les opérations des méthodes des classes dont les spécifications peuvent être complétées par des pré et post-conditions.

UML permet de modéliser un système indépendamment de toute démarche ou plate-forme, c'est donc tout naturellement qu'il est utilisé pour décrire les modèles de niveau PIM mais aussi pour décrire la plupart des modèles de niveau PSM. En particulier, nous l'utilisons pour modéliser la grammaire et les propriétés syntaxiques et sémantiques des langages de programmation. Les spécificités de chaque plate-forme peuvent être modélisées, grâce aux mécanismes d'extension d'UML (stéréotypes) complétés par des propriétés. On peut définir pour chaque système un profil qui regroupe les éléments nécessaires à ses caractéristiques.

Nous rappelons la figure donnée au chapitre précédent montrant le double rôle du Méta-Modèle d'un langage de modélisation qui est appliqué à UML, vis-à-vis d'une part de la représentation des différents modèles d'une application et de la représentation interne des métadonnées apparaissant dans les modèles :

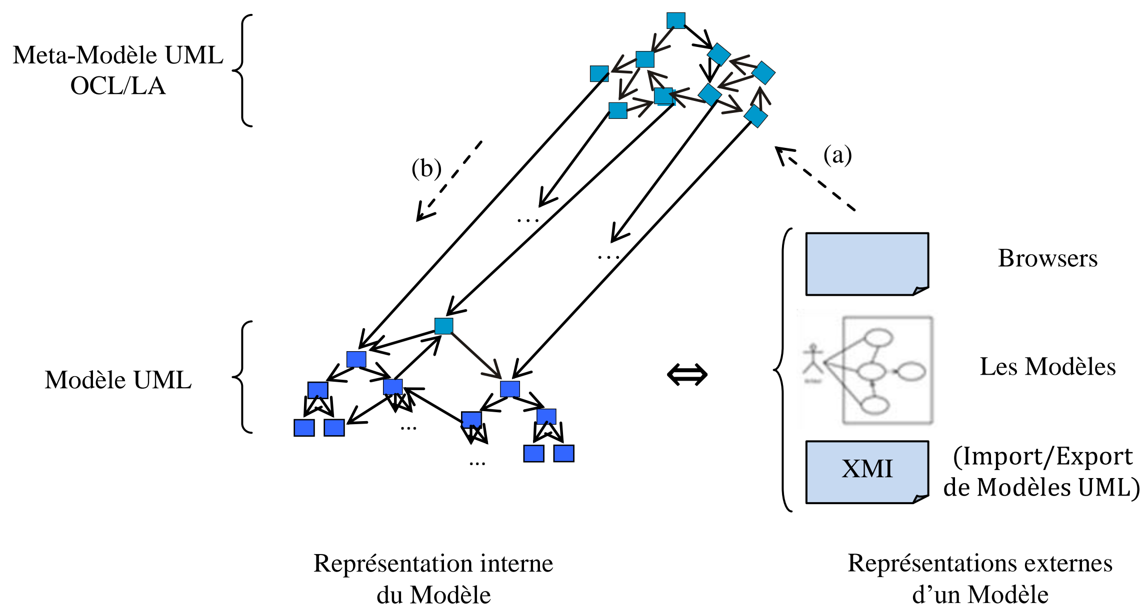


*Figure 8 : **L'environnement simplifié d'un éditeur UML***

Cette figure montre que tout modèle UML peut être représenté graphiquement et peut se mettre sous la forme d'un document interchangeable XMI. L'interface graphique de l'éditeur UML permet aux Analystes/Concepteurs de visualiser les différents fragments des modèles

sous la forme d'une structure arborescente compatible à celles que l'on retrouve dans le document XMI.

II.2.c Exemple d'un modèle UML

Nous prenons en exemple le modèle d'une application réalisé à l'aide d'un diagramme de classes.

Le diagramme de classes exprime de manière générale la structure statique d'un système en termes de classes et de relations entre ces classes. Il est composé d'un ensemble de classes et d'associations entre ces classes. Toute classe est définie par des propriétés structurelles, les attributs dont les valeurs sont typées, et par des propriétés comportementales, les opérations pouvant avoir un certain nombre de paramètres. Les invariants OCL permettent de rajouter à toute classe des propriétés non exprimables à l'aide d'un graphique. Toute association est définie par des propriétés structurelles indiquant, en particulier, les classes qu'elles relient.

La figure suivante montre un exemple simple de diagramme de classes pouvant faire partie d'un modèle plus complet :

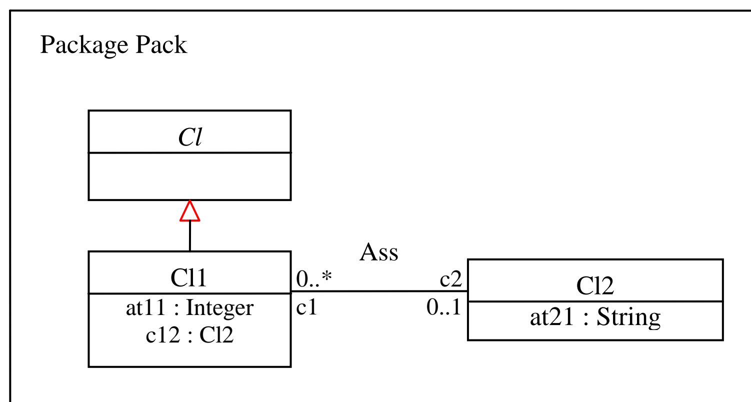


Figure 9 : ***Exemple d'un diagramme de classes d'un modèle UML d'une application***

Ce diagramme montre que l'on a la classe abstraite Cl dont la classe Cl1 définie par les attributs at11 et cl12, hérite, et la classe Cl2. Les classes Cl1 et Cl2 sont reliées par l'association de nom Ass. Ces éléments de modélisation sont regroupés dans le package Pack.

II.2.d Propriétés sur un modèle

Sur ce modèle de données, on pourrait rajouter des propriétés. Par exemple, on pourrait imaginer l'attribut at21 de la classe Cl2 est une clé, au sens où il ne peut y avoir deux objets différents de Cl2 qui ont la même valeur de clé. Cette propriété pourrait s'écrire en OCL, à l'aide de l'invariant suivant :

```
context Cl2
        inv at21Unique : Cl2.allInstances->forAll( cl2x | self <> cl2y implies cl2x.at21 <> cl2y.at21 )
```

Figure 10 : ***Propriétés sur les données d'une application***

Cette propriété étant définie au niveau de ce diagramme de classes s'appliquera à toute instance de ce modèle, c'est-à-dire au niveau de modélisation M0.

On pourrait imaginer des contraintes sur le modèle lui-même, c'est-à-dire sur les métadonnées. Par exemple, on pourrait exiger que les noms des classes et des associations soient différents deux à deux. Une telle contrainte portant sur le diagramme de classes lui-même doit donc être définie au niveau MM UML de manière à s'appliquer sur tout diagramme d'un modèle.

D'une manière générale, pour avoir un langage de modélisation qui puisse prendre en compte les spécificités métier d'un domaine d'applications, les Analystes/Concepteurs doivent, à l'aide d'invariants OCL, ou de pré- et post-conditions OCL, intégrer au niveau du MM UML les propriétés correspondantes. La connaissance du MM UML doit donc être connue de tout expert d'un domaine d'applications. Nous en donnons dans ce paragraphe ses principales caractéristiques, pour l'ensemble du MM, et pour les diagrammes que nous rencontrerons dans cette étude. Dans le paragraphe suivant, nous détaillons ce fragment de manière à pouvoir mettre en œuvre, en OCL, de telles propriétés.

II.3 Méta-Modèle UML (MM UML)

Nous rappelons, dans ce paragraphe, les principes généraux sur le MM UML, sur lequel les propriétés pourront être spécifiées.

II.3.a Exemple d'un fragment de MM UML

La figure suivante montre le fragment du MM UML décrivant, à un très haut niveau d'abstraction, la syntaxe d'un modèle UML montrant le lien entre les éléments de modélisation d'un modèle et leur représentation graphique :

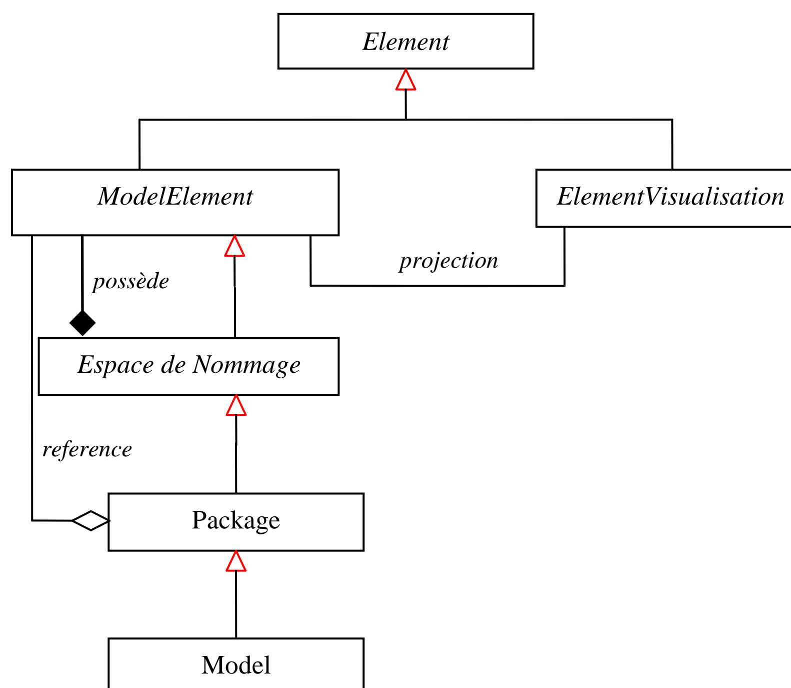


Figure 11 : ***Fragment du méta-Modèle des deux grandes familles d'éléments qui forment le contenu des Modèles***

Ce fragment de MM montre, en particulier, que tout élément d'un modèle peut être un élément de modélisation ou un élément de visualisation. A chaque élément de modélisation correspond son élément de visualisation. Un élément de modélisation peut être un package qui peut être composé de plusieurs éléments de modélisation.

II.3.b Fragment du MM UML définissant les éléments de modélisation du diagramme de classes

Une modélisation UML, comme on l'a rappelé précédemment, est constituée de plusieurs types de diagrammes, chacun ayant son propre langage défini par une grammaire, l'ensemble étant défini par une grammaire générale appelée le MM UML unifiant ces différents langages, chacun des diagrammes étant défini donc par un fragment du Méta-Modèle. Dans ce paragraphe, on décrit les principaux fragments du MM UML, chacun décrivant plus ou moins partiellement la structure des méta-données d'un diagramme UML. On montrera ensuite les techniques mises en place pour modifier le MM UML, et les règles à respecter par les différents Analystes/Concepteurs impliqués dans un processus pour éviter les interférences.

Le MM UML décrit la syntaxe des différents diagrammes représentant le modèle d'une application, instance donc du MM UML. Ce MM se présente, en fait, sous la forme d'un diagramme de classes UML, décrivant la représentation interne des différents diagrammes représentant le modèle. Ce MM définit donc, aussi, la structure des données de cette représentation interne du modèle. La figure suivante montre le fragment simplifié du MM UML décrivant les éléments de modélisation du diagramme de classes :

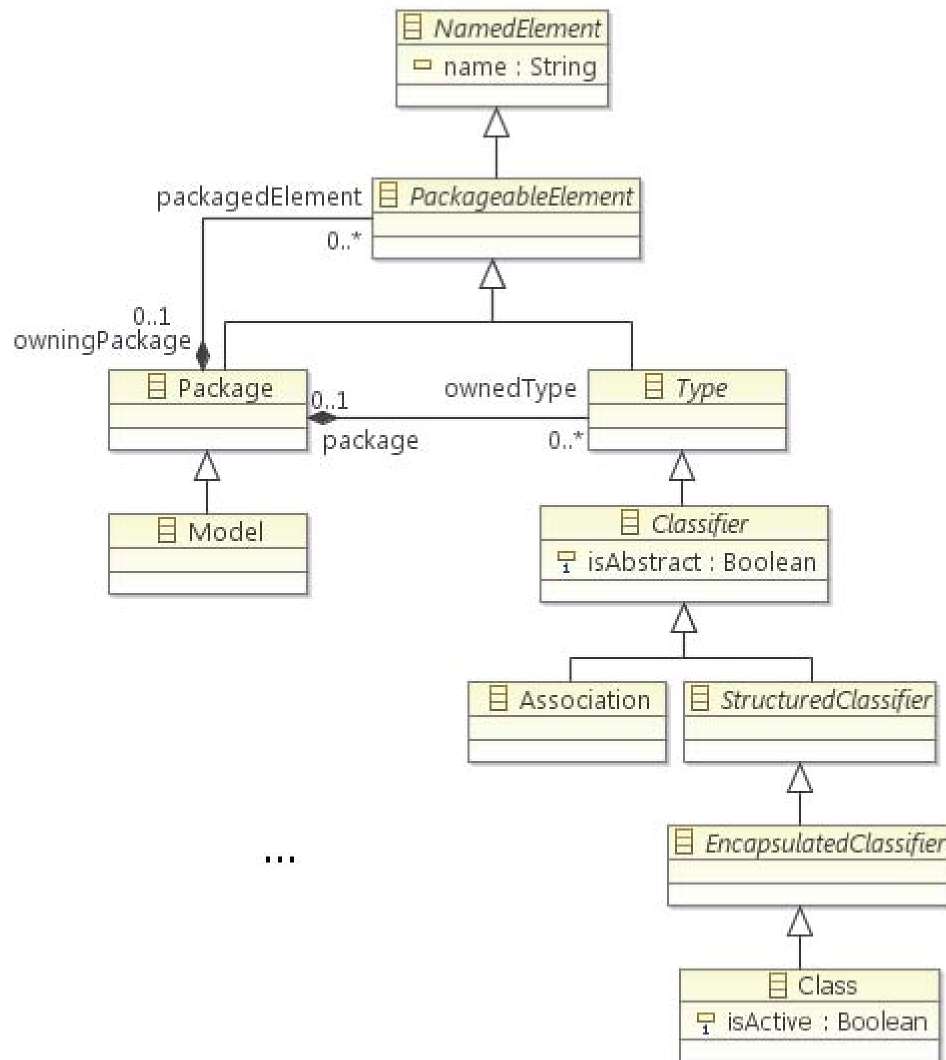


Figure 12 : ***Fragment du MM UML définissant un modèle UML comme un package***

Cette figure montre qu'un modèle est un package qui peut être composé d'un ensemble d'éléments empaquetés (Méta-Classe abstraite PackageableElement) typés, qui peuvent être, en particulier des classes ou des associations.

La figure suivante montre un fragment du MM UML faisant apparaître la description des propriétés et des méthodes les classes :

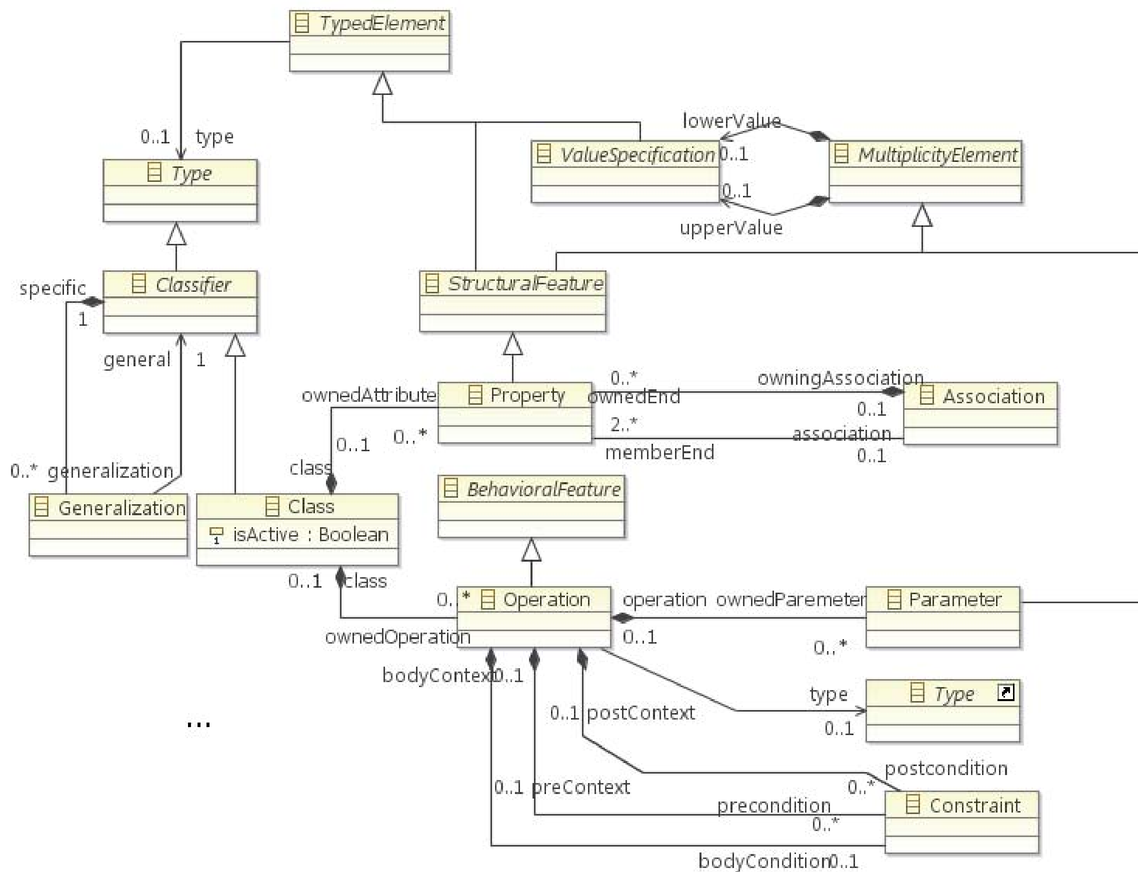


Figure 13 : ***Fragment simplifié du MM UML relatif aux éléments de modélisation du diagramme de classes***

Ce fragment du MM UML montre que toute classe (méta-classe Class) peut être définie par un ensemble d'attributs (méta-classe Property) typés (lien avec la méta-classe Type) et un ensemble d'opérations (méta-classe Operation), où l'on peut préciser pour chacune d'entre elle, ses paramètres, le type de la valeur retournée, des pré- et post-conditions.

II.3.c Représentation interne du modèle de l'application, à l'aide d'un diagramme d'objets

La figure suivante montre la représentation interne du modèle dont la représentation graphique a été donnée plus haut.

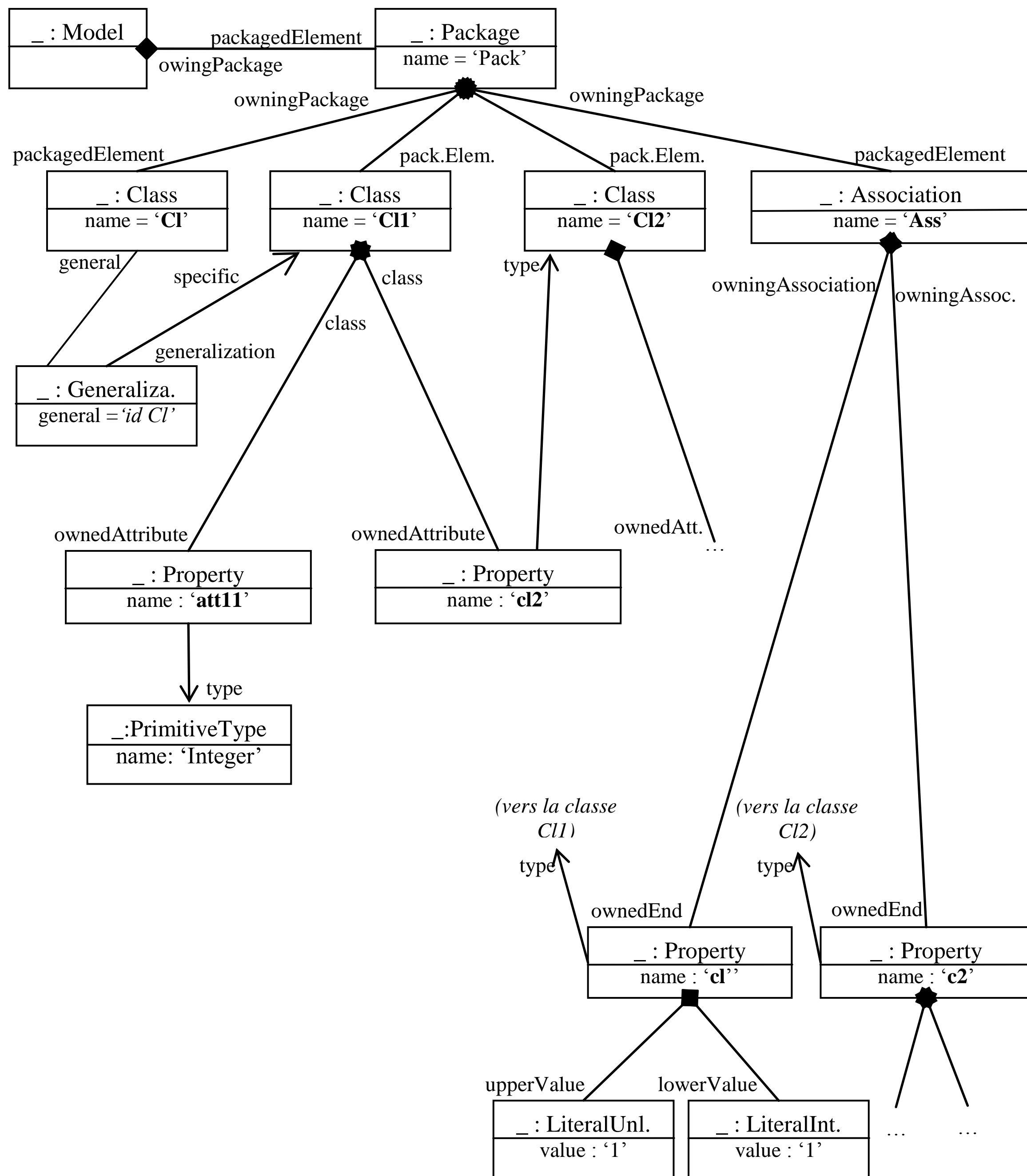


Figure 14 : ***Représentation interne du modèle UML, instance du MM UML***

Cette représentation donne la définition des méta-données du modèle et des liens entre ces méta-données, sous la forme d'un diagramme d'objets, conformément à ce qui est défini dans le MM UML. L'ensemble des éléments empaquetés apparaissent en haut de la figure, puis la définition de l'héritage, suivi de l'ensemble des attributs des deux classes Cl1 et Cl2. Les artéfacts de modélisation de l'association de nom Ass se trouve en bas du diagramme d'objets.

Les fragments de MM UML décrits dans ce paragraphe sont issus du document décrivant la norme UML : Superstructure Specification de l'OMG.

II.3.d Mise en œuvre de la méta-propriété

La méta-propriété, spécifiée en OCL sur le fragment du MM OCL, portant sur les noms des classes et des associations qui devraient être différents deux à deux, est donc la suivante :

```
context Model
invnomClassesAssocUnique :
        self.packagedElement.packageElement->select(c |    c.oclIsTypeOf( Class ) or
                                                            c.oclIsTypeOf( Association )

                                      )->forAll(c_a1, c_a2 |    c_a1 <> ca2      implies
                                                                 c_a1.name <> c_a2.name
                                                            )
```

Figure 15 : ***Exemple de mise en œuvre d'une propriété définie au niveau du MM UML, s'appliquant sur le modèle***

II.4 Profil UML

La modification et le rajout de propriétés au niveau du Méta-Modèle UML ne sont pas souhaitables dans la mesure où l'aspect standard du Méta-Modèle perdrait tout son sens.

C'est la raison pour laquelle il existe en UML le concept de profil qui permet de protéger les éléments de modélisation du Méta-Modèle UML en donnant la possibilité d'étendre certains de ses éléments de modélisation en vue de prendre en compte les spécificités d'un domaine d'applications sur ces éléments. La figure suivante schématise la structure d'un profil UML où l'on peut voir que les spécificités métiers d'un domaine d'applications se définissent tout d'abord à un niveau structurel en 'étendant' et 'profilant' les méta-classes du Méta-Modèle en fonction des spécificités métier du domaine d'applications considéré, et en rajoutant les propriétés sur les extensions du profil :

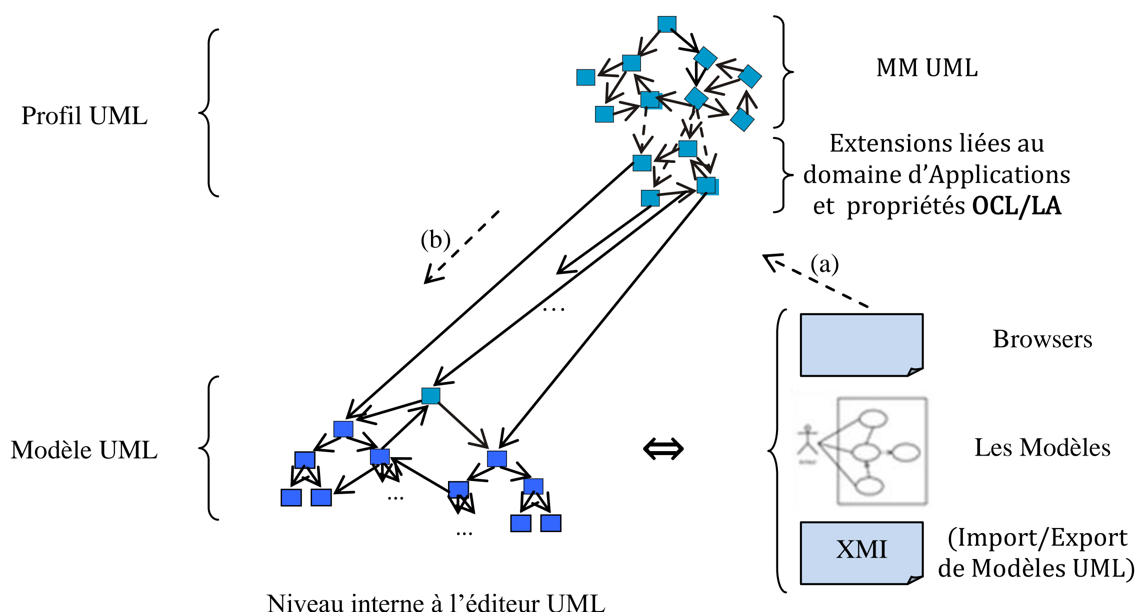


*Figure 16 : **Profil UML pour un domaine d'applications***

II.4.a Fragment du Méta-Modèle définissant les éléments de modélisation pour les profils

Techniquement, un profil n'est qu'une extension d'un ensemble de Méta-Classes, appelées stéréotypes, permettant d'établir une correspondance entre les concepts UML et les concepts d'un domaine d'applications. Comme pour des Méta-Classes du MM UML on peut définir sur les stéréotypes des propriétés structurelles et comportementales, dans la mesure où elles n'entrent pas en contradiction avec les propriétés des méta-classes qui ont servi à définir le profil.

La figure suivante montre le fragment du Méta-Modèle UML à prendre en compte lors de la définition d'un profil UML :

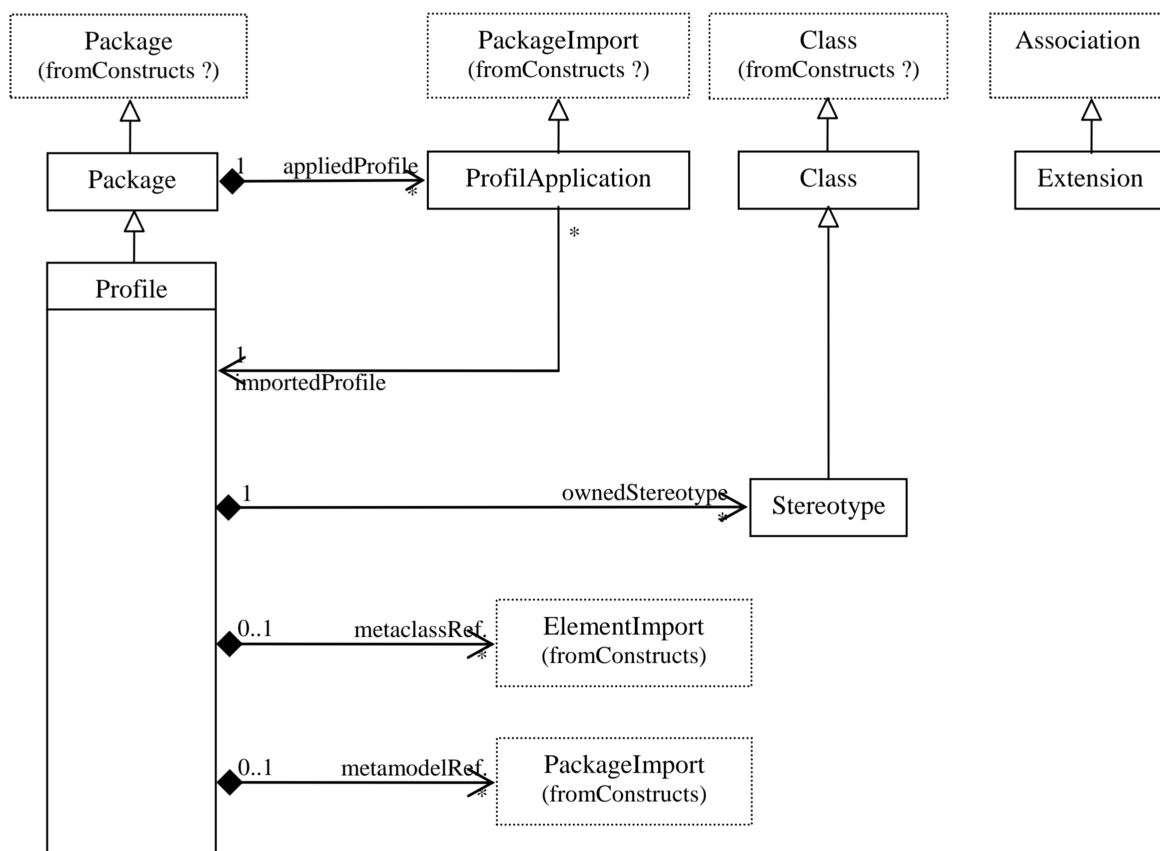


*Figure 17 : **Fragment du Méta-Modèle UML concernant les éléments de modélisation des profils***

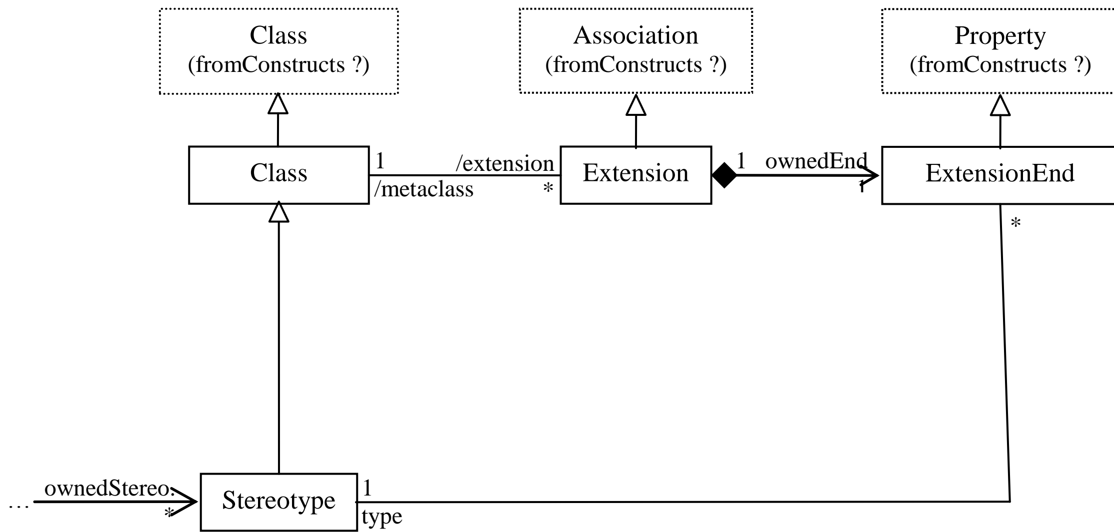


*Figure 18 : **Fragment du Méta-Modèle UML concernant les éléments de modélisation des profils (Suite)***

Un exemple d'application des profils est donné à la figure suivante :

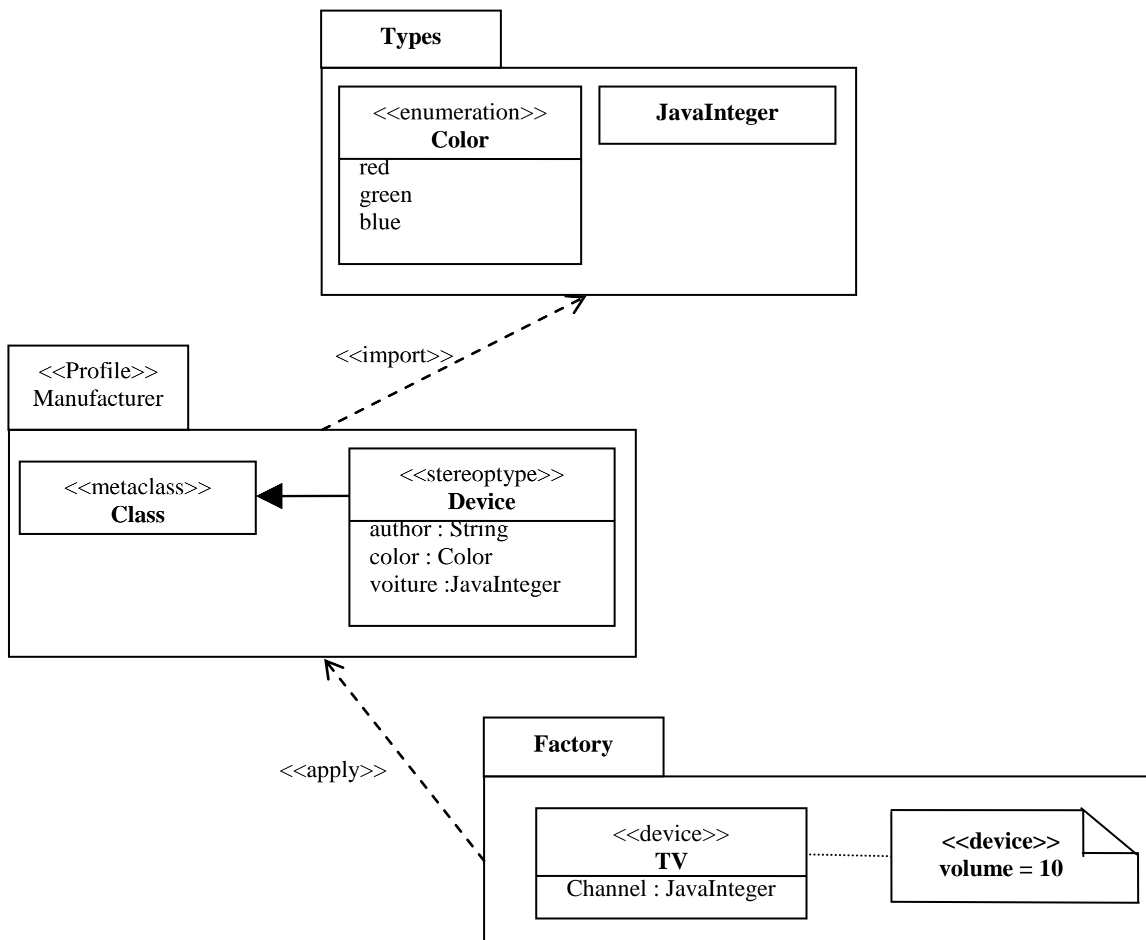


*Figure 19 : **Exemple d'un profil***

II.5 XML Métadata Interchange (XMI)

Le langage XMI permet de décrire une instance du MOF sous forme textuelle, grâce au langage XML[18] du W3C. XMI définit comment utiliser les balises XML pour représenter un modèle MOF en XML. Mes Méta-Modèles MOF sont décrits par des DTDs[19]et les modèles traduits dans des documents XML conformes à leur DTD correspondante.

XMI résout beaucoup de problèmes rencontrés lorsque l'on veut représenter des objets et leurs associations sous forme textuelle. De plus, puisque XMI est basé sur XML, les méta-données (tags) et les instances (elements) sont regroupées dans le même document, ce qui permet à une application de comprendre les instances grâce à leurs méta-données. XMI est le format d'échange standard entre les différents outils MDA.

II.5.a Principe

Dans les différentes modifications apportées à la nouvelle version d'UML 2.0, on peut noter un rapprochement assez important entre la représentation interne d'un Méta-Modèle UML et le document XMI correspondant. Il s'agit de faciliter le passage de UML à XMI, en retrouvant les mêmestypes de navigation, en OCL et en XSLT qui est un langage de scripts pour transformer des documents XML.

En particulier, à une instance de la Méta-Classe PackageableElement correspond une balise, ayant son propre identifiant. Un document XMI structure les données qu'il contient selon une structure hiérarchisée en suivant les liens de composition définis dans le MM UML. Le nom des balises est repris du nom de rôles des liens de composition. Enfin, toute association, qui représente donc un lien transverse dans la hiérarchie des compositions est représentée en XMI à l'aide d'un attribut, de même nom que le nom de rôle, et de même type que la Méta-Classe que cette association référence.

On montre, dans le paragraphe suivant, comment est représenté, en XMI, le modèle donné en exemple d'une modélisation UML.

II.5.b Exemple

Une copie du document XMI se trouve en annexe en fin du chapitre. Afin de mieux faire apparaître le parallèle entre la représentation interne du Méta-Modèle de l'exemple donné (voir la figure 13) et le document XMI, ce dernier est représenté sous la forme d'une structure arborescente, de balises, telles que le montrent les 3 figures suivantes, chacune représentant un fragment du document.

La figure suivante correspond à la description en XMI du fragment définissant les classes du Méta-Modèles et leurs propriétés structurelles :

[18] Extensible Markup Language (http://www.w3c.org/xml/

[19] XML Document Type Definitions

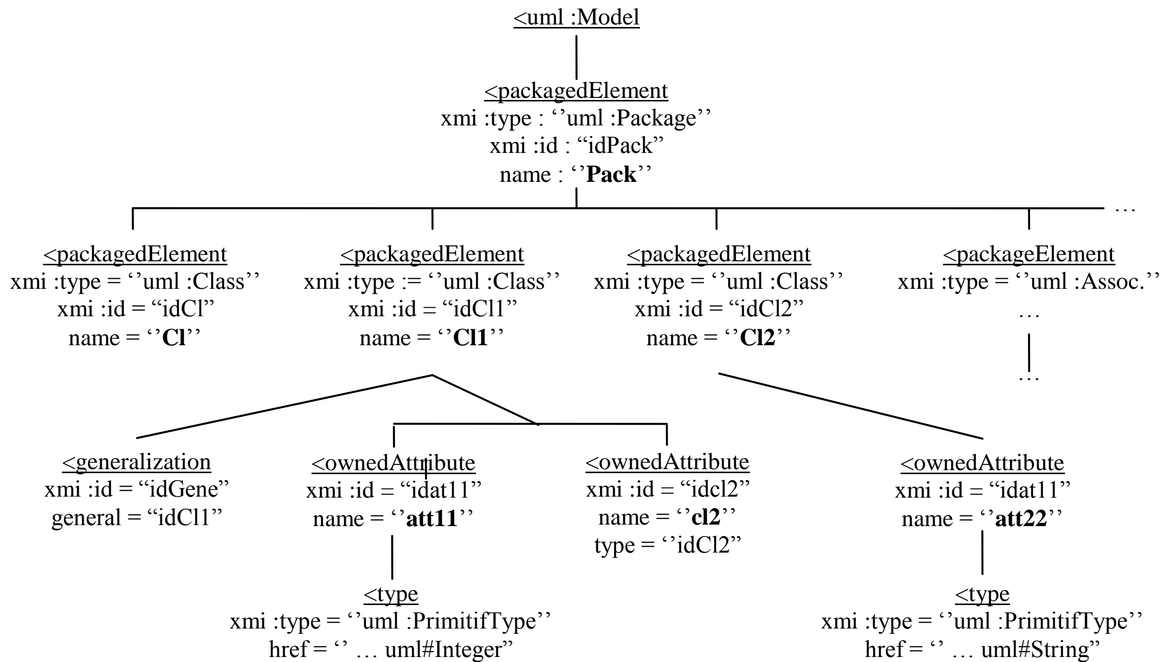


Figure 20 : ***Fragment du document XMI de l’exemple UML, définissant les classes et leurs propriétés structurelles***

En se reportant à la figure décrivant la structure interne du Modèle donné en exemple au paragraphe sur UML, on peut reconnaitre les noms des balises qui sont soulignées et précédées du caractère ‘<’. La balise dont le nom est <uml :Model est composée de la balise de nom <packageElement définie par les attributs type et id qui sont des éléments de modélisation UML. En comparant avec la représentation interne du Méta-Modèle de l’exemple, on peut constater que l’on retrouve le lien de composition liant l’instance de Model à l’instance de Package, en navigant à l’aide du nom de rôle packagedElement. Cet objet packageElement est un package qui contient le modèle. En continuant, on peut voir que dans ce paquetage on trouve d’autres éléments empaquetés représentant les classes du modèle, en particulier. Sur cette figure, on peut voir que l’attribut de nom att11 est un attribut de la classe Cl1, et que le type correspond au type primitif UML Integer. Par contre, l’attribut cl2 a pour type la classe Cl2 qui est une manière de représenter en XMI l’association UML reliant cette caractéristique structurelle de la classe à l’aide de l’identifiant de la classe Cl2.

La figure suivante montre le fragment du Méta-Modèle décrivant l’association de nom Ass, avec ses propriétés indiquant les classes que cette association relie :

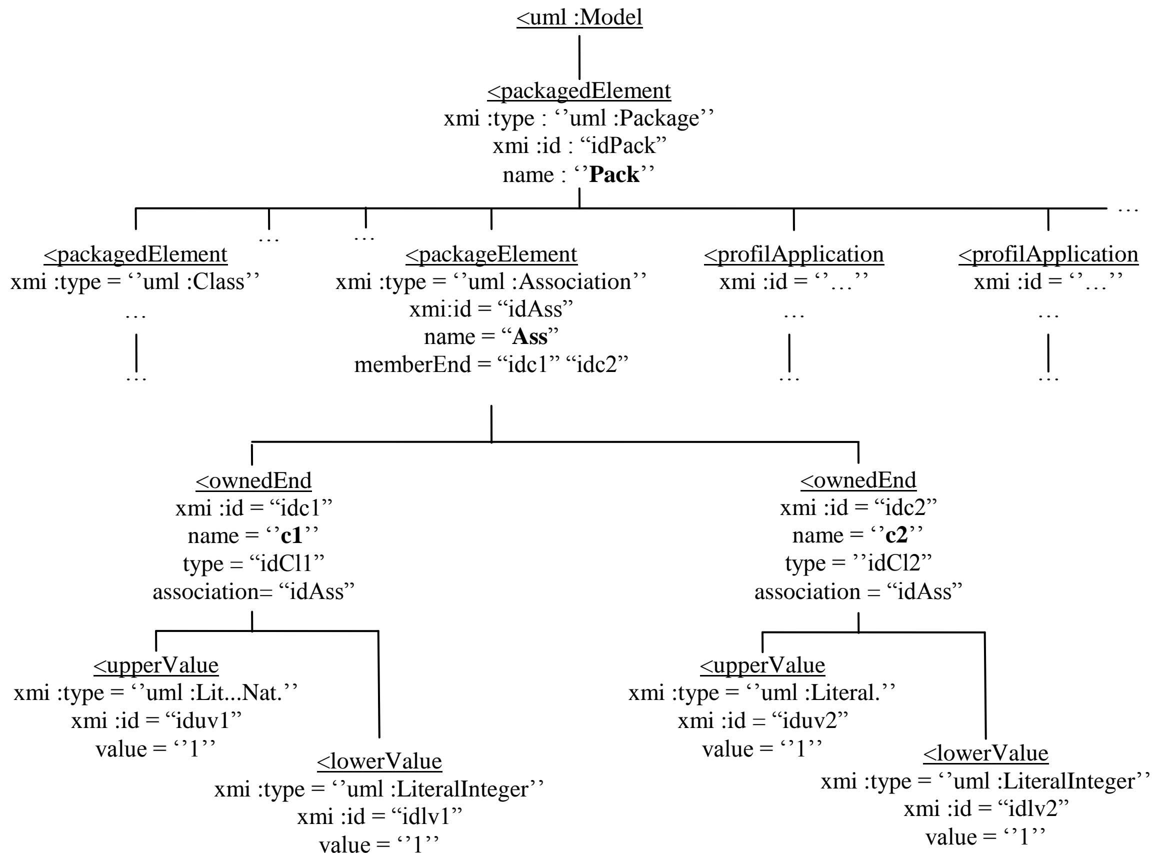


*Figure 21 :**Fragment du document XMI de l'exemple UML, définissant l'association et ses propriétés structurelles***

Ce fragment de document XMI (représenté sous la forme d'une structure arborescente) montre un autre exemple de navigation transversale qui doit pouvoir se faire dans un sens ou dans un autre : il peut être nécessaire, par exemple, de rechercher les classes qui sont reliées à une classe dont on connait le nom, ou inversement, de rechercher les classes qu'une association relie. Au niveau UML, la composition reliant la Méta-Classe Association et Propertyétant bi-navigable, permettra de répondre facilement en OCL et dans un langage d'actions UML à ces besoins. Par contre, en XML, il n'en est pas de même compte tenu de la structure arborescente. Pour répondre plus facilement à cette nécessité de bi-navigabilité, il est rajouté au niveau XMI des attributs memberEnd et association référençant mutuellement les Méta-Classes correspondantes. D'où, dans le MM UML, la présence de l'association entre Property et Association qui parait redondante avec le lien de composition qui existe déjà entre ces deux Méta-Classes, mais qui assure la cohérence entre le MM UML et le document XMI.

De cette manière, que ce soit en UML ou en XMI, on peut naviguer simplement d'une association vers les classes qu'elle relie à l'aide des rôles ownedEnd et type, et d'une classe vers les classes qui lui sont associées à l'aide des rôles ownedAttribute, owingAssociation, memberEnd et type.

Cet exemple de redondance de l'information apparaissant dans le MM UML amène à faire deux remarques :

- Cette redondance reste transparente aux Analystes/Concepteurs qui élaborent le méta-modèle de leurs applications à l'aide d'un éditeur graphique UML qui en tiendra compte, en interne, pour gérer cette redondance et pour assurer le passage entre le Méta-Modèle et XMI, et vice-versa.

- Cette redondance d'information semble contredire la démarche MDA qui préconise de rendre indépendant de toute considération technique les modèles de niveau PIM. On peut cependant remarquer que c'est un exemple typique d'exigences d'une application que l'on a simplement 'remonté' au niveau conception, de manière à anticiper des solutions satisfaisantes qui peuvent être préparées au niveau des modèles PSM. Nous reviendrons souvent sur ce genre de remarque par la suite.

La figure suivante montre le fragment du document XMI montrant les liens entre les attributs des classes dont les types sont des types prédéfinis UML héritant de la Méta-Classe ValueSpecification, et définis à l'aide de profil UML :

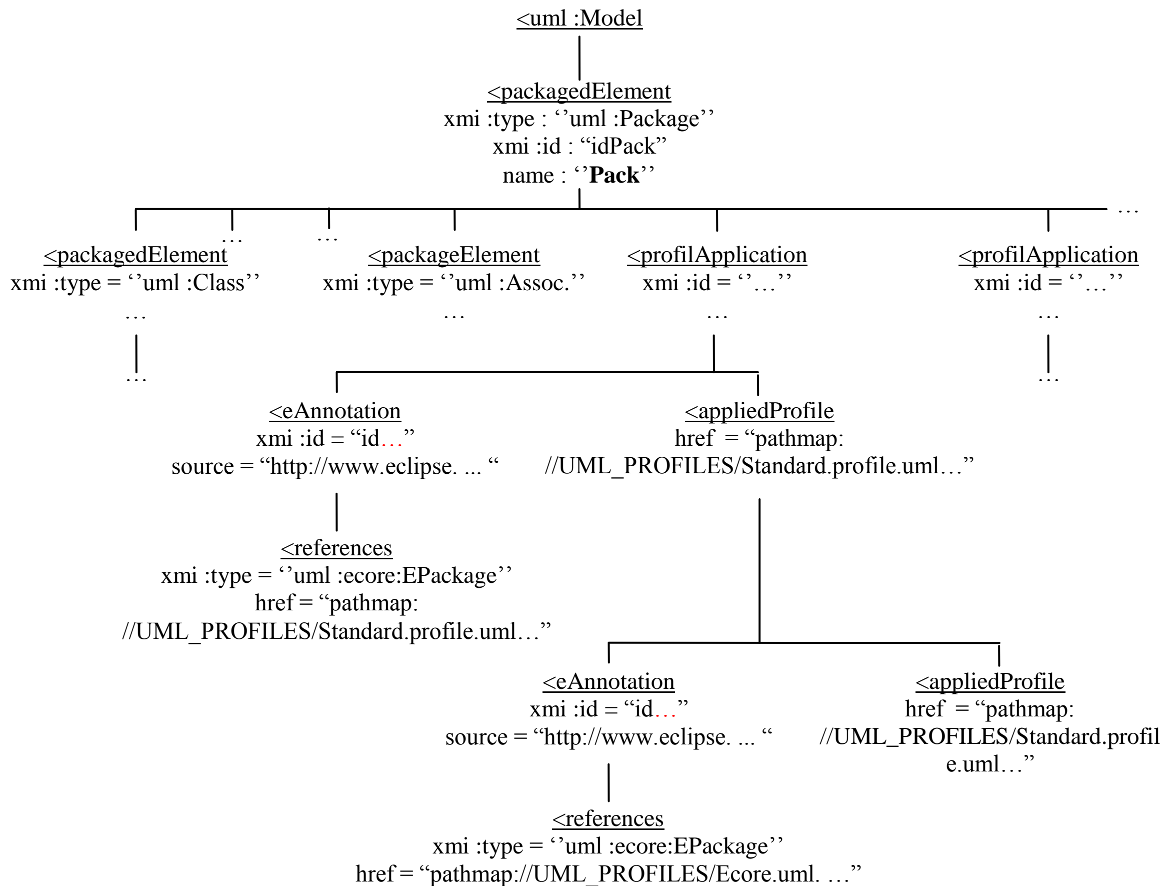


*Figure 22 :**Fragment du document XMI de l'exemple UML, décrivant les profils où sont définis les types primitifs***

III Les Projets de Recherche et les Environnements de développements qui ont accompagné cette étude

III.1 Les Projets de recherche

Nous présentons brièvement dans ce paragraphe les projets de recherche dans lesquels nous avons été impliqués et qui nous ont permis de nous situer par rapport aux autres équipes de recherche et par rapport aux besoins des industriels ayant participé à ces projets.

Nos recherches se situent dans la continuité du projet européen Neptune portant sur la vérification statique de modèles et la génération de documentation, et le projet de l'ANRT TRAMs (2001-2003) [LeDe] portant sur les processus de migration générique.

Le projet européen Neptune[20] (2000-2003), initié par l'équipe de recherche MACAO de l'IRIT, est orienté sur la vérification statiquement des modèles UML. Il avait pour objectif de guider les Analystes/Concepteurs tout au long du cycle de développement des logiciels d'une application. Un ensemble de règles OCL a pu être identifié pour la mise en conformité d'un ensemble de points de vue intra- et inter-modèles. Une plate-forme, basée sur les standards d'échange XMI et de transformation XSL, a été réalisée pour valider ces règles. Elle rend un diagnostic indiquant les éléments de modélisation non conformes aux règles.

Le projet de l'ANRT TRAMs (2001-2003) orienté sur la migration de systèmes d'information dirigée par les modèles, proposait un cadre méthodologique basé sur la rétro-conception du système à migrer et sa re-génération à partir des modèles après mises à jour et vérifications par les Analystes/Concepteurs devant introduire les nouvelles fonctionnalités exigées lors de la migration du système. L'équipe MACAO s'est impliquée plus particulièrement sur la Méta-Modélisation et les transformations de Modèles.

Par contre, cette étude de recherche a été réalisée dans le cadre des projets DOMINO, TopCased et MyCitizSpace.

III.1.a Le projet DOMINO

Cette recherche a été réalisée dans le cadre du lot 4 portant sur la modélisation des langages de programmation du projet ANR DOMINO[21] (2007-2009), initié par l'équipe MACAO. L'objectif du projet a été de proposer une démarche basée sur la description d'un système par divers modèles exprimés dans des langages de modélisation dédiés différents, en exploitant l'Ingénierie Des Modèles (IDM ou en anglais MDA/MDE pour Model Driven Architecture/Engineering) pour fiabiliser tout processus d'ingénierie accompagnant le développement de logiciels. Le terme de processus fiable, fondamental dans le contexte d'une modélisation multi-formalisme, doit être compris comme l'ensemble des techniques permettant de concevoir et fiabiliser des logiciels ayant des contraintes importantes de continuité de service en opérationnel. Fiabiliser un processus de développement est la première étape essentielle pour délivrer un service de confiance justifié. Cette recherche a été appliquée, plus particulièrement, sur un projet de mise en œuvre de procédures opérationnelles destinées à la synchronisation de deux unités de calcul de commandes de vol.

III.1.b Le projet TopCased

Le projet TopCased[22] (2004-) concerne les environnements de développement logiciel de systèmes logiciels et matériels critiques embarqués. Ce projet vise la construction d'un atelier de développement libre «Open Source» avec les objectifs suivants :

---

[20]Nice Environment with a Process and Tools Using Norms and Example (*http://neptune.irit.fr*)
[21]DOMaINes et prOcessus méthodologique (http://domino-rntl.org)
[22]Toolkit in OPen-source for Critical Application SystEms Development (http://www.topcased.org)

- pérenniser les méthodes et outils pour le développement de systèmes embarqués critiques allant de la spécification des systèmes jusqu'à la réalisation logicielle et matérielle en passant par la définition des équipements ;
- minimiser les coûts de possession ;
- assurer l'indépendance par rapport aux plates-formes de développement ;
- intégrer aussi tôt que possible les derniers résultats issus des laboratoires de recherche ainsi que les changements méthodologiques ;
- offrir une capacité d'adaptation des outils au processus et non pas du processus aux outils comme c'est le cas aujourd'hui ;
- prendre en compte les contraintes de qualification.

Pour répondre à ces objectifs, TopCased s'appuie, d'une part sur les dernières technologies en Ingénierie Dirigée par les Modèles dans le cadre de la plate-forme Eclipse et sur des approches de vérification formelles, et d'autre part sur la définition d'un processus commun basé sur les modèles dérivés de l'EIA 632.
Nous nous sommes plus particulièrement impliqués dans les actions sur les transformations de modèles, et dans les actions du lot 4 et des cas d'études du projet DOMINO. En particulier, nous avons participé à la rédaction d'un guide méthodologique sur les transformations de modèles.

Nous nous sommes plus impliqués au niveau des activités du groupe de travail WP5 portant sur la transformation de modèles.

III.1.c Le projet MyCitizSpace

Le projet MyCitizSpace[23] (2008-) vise à doter les administrations locales (Collectivités Locales et Territoriales) et centrales (Ministères, Organismes Nationaux) d'outils logiciels performants et agiles, garants, pour le citoyen, d'IHM de qualité, satisfaisant en particulier les exigences de sécurité, ubiquité et accessibilité. L'approche explorée dans le projet est une Ingénierie Dirigée par les Modèles (IDM). L'ambition est de produire un atelier IDM permettant aux différents acteurs (Maîtrise d'ouvrage, Maîtrise d'œuvre, Expert Métier, CNIL) de produire, de façon itérative et collaborative, des télé-procédures efficaces, capables de s'adapter aux dispositifs d'interaction de l'utilisateur (PC, téléphone, etc.) dans le respect des propriétés attendues (fonction et ergonomie). Dans le cadre de ce projet, nous nous sommes impliqués plus particulièrement au niveau de la gestion des droits attribués aux différents acteurs limitant l'accès aux données qui les concernent exclusivement. Généralement, les autorisations d'accès aux données accordées aux différents acteurs, ainsi que la vérification des accès sont des fonctionnalités des systèmes de gestion de bases de données. Cependant, la gestion des droits d'accès des applications devant manipuler et échanger des données personnelles voire privées, est une exigence très critique qu'il est important de prendre en compte au niveau des télé-procédures de manière à s'assurer que les traitements qui y sont décrits respectent bien les droits d'accès des utilisateurs. C'est à ce niveau que nous nous sommes impliqués dans ce projet, en intégrant dans des langages des télé-procédures les propriétés définissant les autorisations d'accès aux données des différents acteurs. D'une manière analogue à la prise en compte et à la vérification des propriétés de typage des programmes, il est ainsi possible de « remonter » au niveau du code des télé-procédures tout ce qui concerne la vérification statique des droits d'accès. En particulier, l'approche formelle de la définition des propriétés des langages permet de justifier et même prouver que les différents traitements décrits dans les télé-procédures respectent les droits d'accès.

[23] http://genibeans.com/cgi-bin/twiki/view/MyCitizSpace/PresentationDuProjet

III.2 Les Environnements de développement

La plate-forme USE qui est un interprète d'expressions OCL (requêtes, invariants, pré-post-conditions) et d'un langage d'actions UML pour l'animation de modèles UML, dispose d'une interface graphique gérant dynamiquement des diagrammes d'objets. Cette plate-forme nous a servi pour implanter directement les spécifications des propriétés syntaxiques et sémantiques des langages de programmation. Ces spécifications ont été développées ensuite en KerMeta, implantée sur Eclipse, donc compatible avec tout l'environnement Eclipse/EMF[24]. Nous avons utilisé aussi la plate-forme TopCased disposant de nombreuses API, en particulier les éditeurs de modèles et de méta-modèles. Le formalisme qui a été choisi facilite la mise en œuvre des spécifications en Java/EMF.

III.2.a L'environnement de spécification USE basé sur UML/OCL

La plateforme USE[25] est un système permettant de spécifier des systèmes d'informations. USE est basée sur un sous-système du langage de modélisation UML. Une spécification USE contient une description textuelle d'un diagramme de classes (classes, associations, …). Des expressions OCL peuvent être utilisées pour spécifier des contraintes d'intégrité sur les modèles. Les modèles peuvent être animés en vue de valider des spécifications représentant des exigences non exprimables structurellement. Des états des données peuvent être affichés durant l'animation des modèles. A chaque état, les propriétés OCL sont automatiquement calculées. Les informations concernant l'état d'un système peut être représentées graphiquement. Le schéma suivant, issu de la documentation de USE, donne un aperçu général des principaux concepts véhiculés dans USE et les différents rapports entre eux :

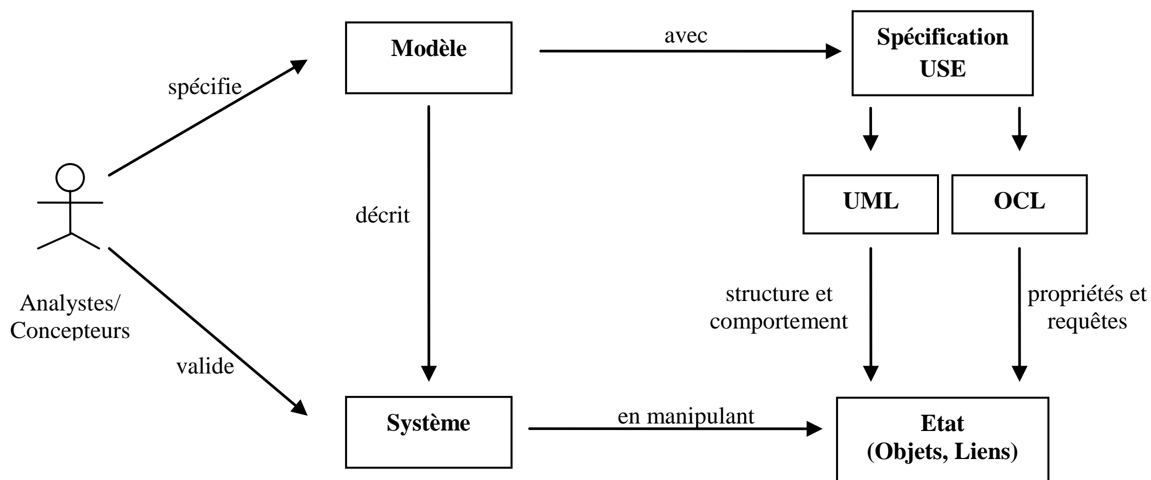


Figure 23 : ***Diagramme montrant les principaux concepts de USE***

[24] Eclipse Modeling Framework
[25] *http://www.db.informatik.uni-bremen.de/projects/USE/*

Le langage d'action que nous avons pris comme référence est celui de la plate-forme USE qui permet de créer des objets, d'éliminer et de modifier des objets. Ce langage est impératif et procédural. Il permet d'autre part de mettre en œuvre les opérations définies dans les classes, en particulier sous la forme de fonctions. Cependant, les propriétés que nous avons rappelées dans le chapitre précédent ont été spécifiées à l'aide de fonctions et de règles d'inférence. Il s'agit donc de spécifier les propriétés des langages de programmation en reprenant ces aspects fonctionnels offrant ainsi une définition précise des modèles de programmes s'intégrant dans un processus IDM de manière cohérente et uniforme avec une démarche UML. La gestion des objets et des associations se fait à l'aide des instructions suivantes :

- Création d'un objet : !createnom_objet : nom_Classe
- Mise à jour d'un attribut d'un objet :
  !set nom_objet.nom_attribut := exp_OCL
- Elimination d'un objet : !destroy nom_objet
- Création d'une association : !insert (exp_OCL1, exp_OCL2) intonom_Association
- Destruction d'une association : !delete (exp_OCL1, exp_OCL2) fromnom_Association
- …

USE dispose d'une interface permettant de mettre à jour dynamiquement un diagramme d'objets, instance d'un diagramme de classes. Cette interface permet de vérifier que le diagramme d'objets vérifie toutes les propriétés définies au niveau du diagramme de classes. Des commandes permettent d'exécuter des opérations après en avoir vérifiées les pré et post-conditions.

III.2.b L'Environnement TopCased et Le langage de méta-modélisation exécutable KerMeta

L'objectif de Topcased est de couvrir l'ensemble des besoins de développement logiciel et système (la branche descendante du cycle en V), ainsi que les besoins transverses comme la gestion de configuration, la gestion des changements ou l'ingénierie des exigences. La plate-forme suit une approche de type MDE (Model Driven Engineering, ou Ingénierie dirigée par les modèles) au niveau de ses fonctionnalités et de ses méthodes de développement.

KerMeta[26] est un langage de méta-modélisation exécutable. Il dispose d'un environnement de développement de méta-modèles basé sur EMOF dans un environnement Eclipse. Il permet non seulement de décrire la structure des méta-modèles, mais aussi leur comportement. Il permet ainsi de définir et d'outiller de nouveaux DSL en améliorant la manière de spécifier, simuler et tester la sémantique opérationnelle des méta-modèles. Il permet en outre d'appliquer plus facilement les techniques de l'Ingénierie Des Modèles aux outils IDM eux-mêmes.

KerMeta workbench se compose d'outils s'interfaçant facilement avec les outils existants dans la communauté Open-source d'Eclipse.

L'environnement de KerMeta (KerMeta workbench) propose :

[26] http://www.irisa.fr/triskell/softwares-fr/kermeta/

- un éditeur textuel supportant la coloration syntaxique et une aide à l'écriture (complétion)
- une vue synthétique (outline)
- un interpréteur, un dévermineur (debugger)
- des fonctions de conversion (depuis/vers ecore)
- ...

Certains environnements de modélisation UML, tels que Kermeta, permettent d'introduire les opérations dans un profil UML ou dans un Méta-Modèle UML selon les techniques de programmation par aspect, assurant une gestion dynamique des opérations d'un Méta-Modèle ou d'un profil UML, et donnant ainsi la possibilité de définir plusieurs applications sur le même Méta-Modèle ou profil UML.

III.2.c QVT, ATL

QVT[27]est un langage d'actions destiné à la manipulation et à la transformation de modèles. Il existe un certain nombre de travaux de recherche sur la spécification d'un tel langage.

Le langage ATL est un langage développé à l'INRIA, en réponse à l'appel à proposition de l'OMG pour définir un langage de transformations de modèles. Basé sur une approche logique, ATL combine les aspects déclaratifs et impératifs des langages de programmation. ATL permet de décrire les transformations de modèles à l'aide de règles basées sur des propriétés syntaxiques et sémantiques, reprises par l'interprète en vue d'effectuer les transformations correspondantes.

En fait, ces langages sont réalisés dans la continuité des travaux de recherche sur les langages de bases de données objet définis comme une extension du langage déclaratif SQL. Le SGBDOO[28] O2, de l'INRIA, en a été pratiquement à l'origine, dans les années 1980/1990. Il était, en particulier, possible d'afficher à l'écran les objets d'une base et les liens entre ces objets. Une interface graphique permettait d'activer une méthode, en cliquant sur la méthode souhaitée d'un objet affiché à l'écran.

III.2.d Java/EMF

EMF[29] est une plate-forme destinée à la construction d'outils et d'applications basés sur les modèles. A partir d'une spécification décrite en XMI[30], EMF fournit un environnement d'édition et d'exécution pour produire des modèles contenant des classes Java. EMF offre un certain nombre de services pour manipuler des objets (SDO[31]), tels que XML

[27]Query-View-Transformation
[28] Système de Gestion de bases de Données Orientés Objet
[29] Eclipse Modeling Framework
[30] XML Metadata Interchange
[31] Service Data objects

SchemaDefinition (XSD[32]) en vue de créer et de manipuler des documents structurés à l'aide de schémas XML définissant des modèles d'objets de type DOM[33]. Cet environnement EMF offre d'autres services relativement variés pour naviguer dans des modèles, pour gérer des services transactionnels.

[32] XML Schema Definition
[33] Document Object Model

III - Modélisation en UML/OCL des Propriétés Syntaxiques des Langages de Programmation

Après avoir décrit les principaux travaux de recherche portant sur la modélisation des langages et rappelé les principales caractéristiques des processus de développement de logiciels dirigés par les modèles, nous abordons au cours des chapitres III et IV les principes et les techniques de modélisation en UML et OCL des langages de programmation et de leurs propriétés. Il s'agit de reprendre les définitions et les propriétés rappelées au premier chapitre, et de montrer comment elles peuvent se spécifier en UML et en OCL et à l'aide d'un langage d'actions. En disposant d'un environnement d'exécution qui permet d'exécuter des expressions OCL et des opérations écrites dans un langage d'actions UML, les Analystes/Concepteurs peuvent ainsi vérifier si les modèles possèdent toutes les qualités requises.

Au cours de ce chapitre III, nous montrons comment la grammaire d'un langage peut être modélisée en UML. On obtient ainsi le Méta-Modèle du langage que nous complétons par un ensemble d'invariants OCL prenant en compte les propriétés non structurelles, non exprimables directement en termes de classes et d'associations entre ces classes. La mise en œuvre de ces propriétés, non explicitement décrites au niveau de la grammaire, et au niveau du typage, est nécessaire si l'on souhaite exécuter tout ou partie des modèles de codes.

Nous montrerons ensuite comment nous pouvons prendre en compte, au niveau des Méta-Modèles des langages les propriétés de typage des langages assimilant les programmes à des formules bien formées.

La modélisation des propriétés opérationnelles et axiomatiques fera l'objet du chapitre IV. La troisième partie de ce document sera consacrée à l'intégration de propriétés syntaxiques et sémantiques au niveau du Méta-Modèle UML. Ces propriétés s'appliquant donc à des modèles de conception, ne sont pas liées à des aspects techniques des langages, mais sont définies par les experts d'un domaine d'applications en fonction des spécificités du domaine.

Le langage « L » est pris comme exemple. La plate-forme USE a été utilisée pour valider l'ensemble de ces spécifications qui ont été portées sur la plate-forme Kermeta de manière à montrer que les travaux présentés dans ce domaine peuvent être appliqués dans le cadre d'un processus et d'une démarche MDA. Nous monterons que ces spécifications peuvent être écrites aussi en Java/EMF.

III - Modélisation en UML/OCL des Propriétés Syntaxiques des Langages de Programmation

Table des matières

I Modélisation en UML/OCL de la grammaire d'un langage ... 80
I.1 Méta-Modèle de la grammaire d'un langage de programmation ... 81
I.1.a Modélisation des règles 'Et' et 'Ou' ... 81
I.1.b Classe abstraite Symbole et point d'entrée de la grammaire ... 84
I.1.c Méta-Modèle du langage « L » ... 85
I.2 Exemple de modélisation d'un programme du langage « L » ... 87
I.2.a Modélisation d'un programme du langage ... 88
I.2.b Propriétés définies au niveau du Méta-Modèle du langage ... 89

II Modélisation en UML/OCL de la BNF et de ses propriétés ... 90
II.1 La BNF et Modélisation de la BNF ... 91
II.1.a Modélisation de la BNF ... 91
II.1.b Méta-Modèle de la BNF ... 91
II.1.c Modèle et Méta-Modèle de la Grammaire d'un langage de programmation ... 93
II.2 Exemple d'une instance du Méta-Modèle de la BNF ... 95
II.2.a Le modèle de la grammaire du langage « L » ... 95
II.2.b Les propriétés définies au niveau MM BNF portant sur les symboles de la grammaire du langage « L » 95
II.3 Génération du Méta-Modèle de la BNF et du Méta-Modèle d'une grammaire ... 98
II.3.a Environnement de modélisation de la plate-forme USE ... 98
II.3.b Environnement de développement sous Eclipse/EMF ... 99

III Modélisation des Propriétés de typage des langages ... 99
III.1 Calcul du type des expressions du langage « L » ... 100
III.1.a Constructeur retournant un type (Langage d'actions de la plate-forme USE) ... 100
III.1.b Fonctions retournant le type des opérations unaires et binaires ... 100
III.1.c Fonctions retournant le type d'une constante ... 101
III.1.d Fonction retournant le type d'une variable ... 101
III.2 Vérification du typage des instructions du langage « L » ... 102
III.2.a Fonction vérifiant le typage d'une instruction ... 102
III.2.b Fonctions vérifiant le typage des différentes instructions du langage ... 102

IV Gestion dynamique des propriétés des langages de programmation ... 103

III - Modélisation en UML/OCL
des Propriétés Syntaxiques des Langages de Programmation

Table des figures

Figure 1 : Positionnement des grammaires et des programmes par rapport aux niveaux de Méta-Modélisation 80
Figure 2 : Modélisation de UML/OCL de règles 'Et' d'une grammaire d'un langage de programmation 81
Figure 3 : Modélisation en UML/OCL d'une règle 'Et' définissant un symbole à partir d'un type primitif 82
Figure 4 : Modélisation, en UML/OCL, de règles 'Et', où les liens sont représentés par des attributs 83
Figure 5 : Modélisation, en UML, des règles de grammaire 'Ou' 83
Figure 6 : Classe abstraite Symbole et classe MM_L dans le MM d'un langage (Version association) 84
Figure 7 : Classe abstraite Symbole et classe MM_L dans le MM d'un langage (Version attribut) 85
Figure 8 : Fragment des parties déclaratives et Instruction du Méta-Modèle du langage « L » 86
Figure 9 : Fragment des expressions du Méta-Modèle du langage « L » 86
Figure 10 : Méta-modèle simplifié de la grammaire du langage « L » 87
Figure 11 : Fragment du modèle d'un programme, instance du Méta-Modèle de la grammaire du langage « L » 88
Figure 12 : Quelques propriétés portant sur les symboles terminaux de la grammaire du langage « L » 90
Figure 13 : Grammaire de la BNF simplifiée 91
Figure 14 : Méta-Modèle de la BNF (MM BNF) 92
Figure 15 : Représentation du Méta-Modèle de la BNF simplifié 93
Figure 16 : Grammaire d'un langage et son Méta-Modèle 93
Figure 17 : Processus de Modélisation de la grammaire d'un langage en passant par la BNF 94
Figure 18 : Modélisation des codes des composants logiciels dans le cadre d'un processus de développement IDM 94
Figure 19 : Modèle de la grammaire du langage L, en tant qu'instance du méta-modèle de la BNF 95
Figure 20 : Exemples d'invariants OCL spécifiant quelques Méta-Règles 97
Figure 21 : Processus de création du Méta-Modèle de la grammaire d'un langage sous selon l'environnement de développement de la plate-forme USE 98
Figure 22 : Processus de création du Méta-Modèle de la grammaire d'un langage sous selon l'environnement de Méta-Modélisation Eclipse/EMF 99
Figure 23 : Constructeurs retournant les objets correspondant aux types prédéfinis du langage « L » 100
Figure 24 : Spécification de la fonction retournant le type d'une expression 100
Figure 25 : Fonctions OCL retournant le type des opérations unaires et binaires d'une expression 101
Figure 26 : Fonctions OCL retournant le type d'une constante 101
Figure 27 : Fonctions OCL retournant le type d'une variable 101
Figure 28 : Spécification de la fonction validant le typage d'une instruction 102
Figure 29 : Fonctions vérifiant le typage des différentes instructions du langage « L » 103

L'objectif de ce chapitre est de modéliser en UML/OCL les propriétés syntaxiques d'un langage de programmation qui sont définies par la grammaire du langage ainsi que par les propriétés de typage du langage assimilant les programmes à des formules bien formées. Ces propriétés ont été rappelées au chapitre I. Le chapitre suivant sera consacré à la modélisation des propriétés comportementales et axiomatiques.

I Modélisation en UML/OCL de la grammaire d'un langage

Nous montrons préalablement le positionnement des grammaires et des programmes par rapport à l'architecture des différents niveaux de Méta-Modélisation définie par l'OMG. Ce positionnement est montré à la figure suivante :

	Niveaux de méta-modélisation de l'OMG	Espace technologique des Grammaires
M3	Méta-Méta-Modélisation	Grammaire BNF
M2	Méta-Modélisation	Grammaire d'un langage
M1	Modèle Application	Programme du langage
M0	Objets de l'application	Données du programme

*Figure 1 : **Positionnement des grammaires et des programmes par rapport aux niveaux de Méta-Modélisation***

Au niveau de modélisation M0, on trouve les données manipulées par les programmes se situant au niveau M1. Tout programme doit être conforme à la grammaire du langage correspondant. La grammaire d'un langage se situe donc au niveau M2. La BNF étant un langage pouvant définir la syntaxe des grammaires se situe au niveau Méta-Méta-Modélisation (M3).

La modélisation des propriétés d'un langage de programmation se situe donc au niveau Méta-Modèle défini par l'OMG. Il s'agit donc de prendre en compte et représenter en UML/OCL les différents symboles de la grammaire et ses constructions syntaxiques à l'aide d'un profil UML étendant le MM UML sous la forme de stéréotypes, d'attributs et de contraintes, prenant en compte ces spécificités du langage. Cependant, puisque les grammaires définissent un ensemble de concepts structurels (les symboles ou les mots du langage) et de liens entre ces concepts (les constructions syntaxiques du langage), le Méta-Modèle de la grammaire d'un langage peut être réalisé à l'aide d'un diagramme de classes prenant en compte les aspects structurels.

Dans ce paragraphe, nous montrons les principes généraux, repris en partie des différents articles de recherche rappelés au chapitre précédent, modélisant en UML/OCL une grammaire à l'aide d'un diagramme de classes. Nous verrons plus tard comment étendre le MM UML de manière à positionner le Méta-Modèle des grammaires au niveau M2, respectant ainsi la hiérarchie des différents niveaux de Méta-Modélisation définie par l'OMG.

I.1 Méta-Modèle de la grammaire d'un langage de programmation

La grammaire du langage pris en référence au chapitre précédent est composée d'un ensemble de règles 'Et' et 'Ou' décrivant à l'aide de symboles, les constructions syntaxiques élémentaires du langage, et d'un point d'entrée qui est le symbole du langage qui se trouvera à la racine de l'arbre représentant le modèle de tout programme du langage.

La modélisation de la grammaire d'un langage se fait donc à l'aide d'un diagramme de classes où les classes représentent les symboles du langage, et où les relations entre les classes représentent les constructions syntaxiques du langage définies par les règles.

Dans ce paragraphe nous montrons les principes permettant de modéliser d'une manière générale les règles 'Et' et 'Ou' que nous appliquerons ensuite sur la grammaire du langage « L ».

I.1.a Modélisation des règles 'Et' et 'Ou'

Toute règle 'Et' d'une grammaire définit une relation de composition entre le symbole de gauche de la règle et tout symbole apparaissant dans la partie droite de la règle. On en déduit donc un lien de composition entre la classe représentant le symbole de gauche d'une règle 'Et' et la classe représentant tout symbole de droite de cette règle.

La figure suivante montre deux exemples de règles 'Et', et le fragment de modèle UML que l'on peut en déduire, en supposant que S n'est pas un symbole terminal (type primitif) :

Règles 'Et', et leur représentation graphique simplifiée :

…
X → R, s1 : **S**, s2 : **S+, T ?**
…
Y →T, U

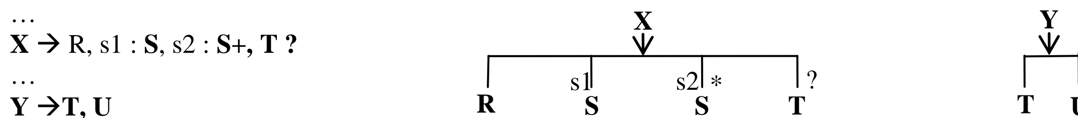
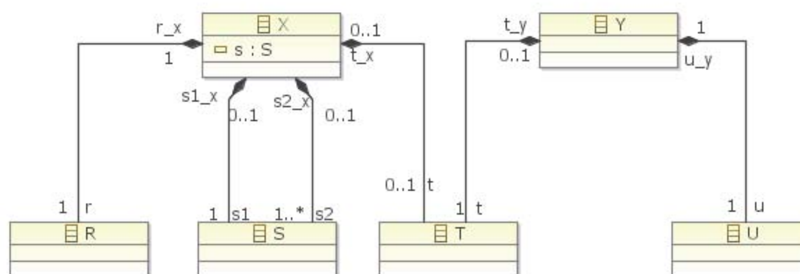


Diagramme de classes :



Invariants :

T ::t_x__t_y	self.t_x <> oclUndefined(X)	xor	self.t_y <> oclUndefined(Y)
S ::s1_x__s2_x	self.s1_x <> oclUndefined(X)	xor	self.s2_x <> oclUndefined(X)

Figure 2 : ***Modélisation de UML/OCL de règles 'Et' d'une grammaire d'un langage de programmation***

Le nom des liens de compositions n'apparaissent pas dans le diagramme de classes. Tout nom se déduit du nom des classes entrant dans le lien de composition, et du sens de la composition : X_R, X_s1, X_s2, X_T, Y_T et Y_U.

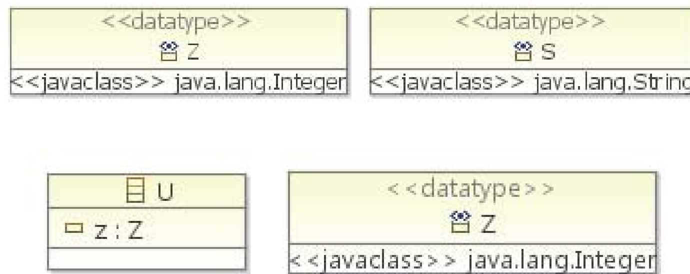
Les noms des attributs deviennent des noms de rôles.

Les multiplicités dépendent des caractères de répétitivité apparaissant au niveau des symboles apparaissant en partie droite de la règle : 0..1, 1..1, 0..* et 1..* ; et du fait qu'un symbole peut apparaitre dans une seule partie droite d'une règle ou dans plusieurs. Par exemple, le symbole T apparait dans la règle définissant la construction syntaxique de X et de Y. Ce qui signifie que toute instance de T devra être rattachée à une instance de X ou (exclusif) une instance de Y. D'où la multiplicité 0..1 aux extrémités des classes X et Y. Il en est de même pour une instance de S, par rapport à X. Les deux invariants OCL rajoutés à la fin de la figure expriment cette contrainte non exprimable graphiquement, et définie de manière implicite dans la grammaire.

La figure suivante montre une règle 'Et' définissant un symbole à partir d'un type primitif Z, et sa modélisation en UML :

Règle 'Et', et représentation graphique :

Diagramme de classes :

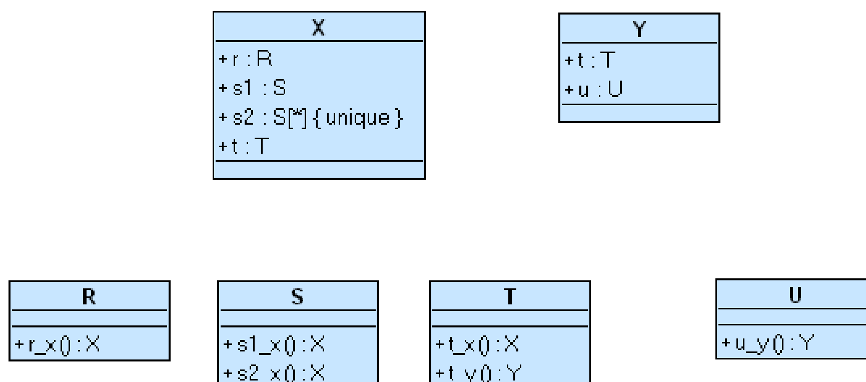


Invariant : U::z self.z <> oclUndefined(Z)

Figure 3 : ***Modélisation en UML/OCL d'une règle 'Et' définissant un symbole à partir d'un type primitif***

Remarque : La figure suivante donne une autre solution de la modélisation des règles 'Et', où les liens de composition sont représentés à l'aide d'attributs intégrés dans les classes représentant les symboles apparaissant en partie gauche des règles :

Diagramme de classes :



Invariants :

```
X ::t       self.t <> oclUndefined( R )                Y::t    self.t <> oclUndefined( T )
X ::s1      self.s1 <> oclUndefined( S )               Y::u    self.u <> oclUndefined( Y )
X ::s2      self.s2->size() > 0

R ::r_x             self.r_x()  <> oclUndefined( X )
S ::s1_x__s2_x      self.s1_x()  <> oclUndefined( X )  xor     self.s2_x() <> oclUndefined( X )
T ::t_x__t_y        self.t_x()  <> oclUndefined( X )   xor     self.t_y() <> oclUndefined( Y )
U ::u_v             self.u_v()  <> oclUndefined( Y )
```

Figure 4 : ***Modélisation, en UML/OCL, de règles 'Et', où les liens sont représentés par des attributs***

On retrouve les mêmes invariants que pour la première approche. Il s'agit en plus de prendre en compte les contraintes sur les multiplicités 1..*.

Cette figure montre que les liens entre les classes sont représentés par des attributs, ce qui en principe est suffisant pour parcourir des structures de données arborescentes représentant des instances de programmes. Nous avons fait apparaitre la spécification des opérations permettant de 'remontrer' les arbres de programmes qui pourraient être implantés si les Analystes/Concepteurs le jugeaient utiles. En général, c'est cette solution qui est adoptée lors que la grammaire d'un langage est modélisée en langage Java.

Les deux approches présentées pour la modélisation des règles 'Et' d'une grammaire sont équivalentes.

D'une manière générale, toute règle 'Ou' indique qu'un symbole peut avoir plusieurs définitions. On peut donc modéliser une règle 'Ou' à l'aide du concept d'héritage, faisant correspondre une classe abstraite à chaque symbole de gauche de la règle et définissant un lien d'héritage entre cette classe abstraite et tout symbole apparaissant en partie droite de la règle. La figure suivante montre un exemple de règle 'Ou', la représentation simplifiée de la règle et la modélisation en UML. La classe représentant le symbole de gauche est abstraite :

Diagramme de classes :

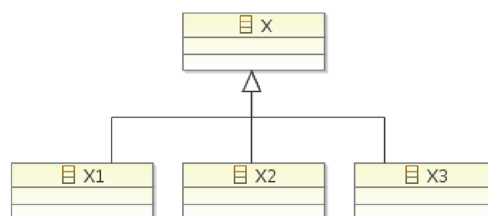


Figure 5 : ***Modélisation, en UML, des règles de grammaire 'Ou'***

Les règles, rappelées dans ce paragraphe, sont suffisantes pour déduire d'une grammaire d'un langage de programmation, son Méta-Modèle UML. Cependant, en vue de

faciliter la spécification des propriétés du langage et leur implémentation dans un langage d'actions, il est nécessaire de compléter le Méta-Modèle. Différentes solutions sont possibles qui peuvent dépendre de l'exploitation que l'on souhaite faire du modèle. Dans le paragraphe suivant, nous décrivons une solution générale qui reste dans l'esprit du Méta-Modèle UML.

I.1.b Classe abstraite Symbole et point d'entrée de la grammaire

Le fait d'avoir rappelé, dans le chapitre précédent, les propriétés des langages permet de compléter le Méta-Modèle en fonction des traitements que l'on doit mettre en œuvre. Cependant, rajouter des associations, souvent appelées « raccourcies » puisqu'elles permettront de naviguer plus ou moins facilement dans un modèle de programme qui a une structure d'arbre, ne paraît pas être une bonne solution dans la mesure où elles ne font que surcharger le Méta-Modèle d'un langage qui devient déjà vite complexe pour des langages de programmation classiques. Il s'agit donc de trouver des solutions qui ne polluent pas le Méta-Modèle, et restent générales et classiques.

Nous avons pu remarquer que le Méta-Modèle UML contient la méta-classe abstraite de nom *ModelElement* dont toutes les méta-classes héritent, et que la méta-classe Model permet de référencer tous les objets d'un diagramme UML. Nous avons repris ces deux méta-classes que nous avons appelées *Symbole* et MM_L. Ainsi, toutes les classes (méta-classes) déduites des symboles des règles héritent de cette méta-classe *Symbole*, et la méta-classe MM_L joue le même rôle que la classe Model dans le méta-modèle d'un langage. Une association entre les classes MM_L et Symbole permet de relier toutes les méta-classes représentant les symboles de la grammaire, et l'association MM_L__peSymb permet d'accéder directement à la méta-classe représentant le point d'entrée de la grammaire. Ces deux méta-classes et ces deux associations permettent de naviguer entre les méta-classes, en reprenant, en particulier, les techniques qui ont été expliquées plus haut pour 'remonter' dans un modèle de programme. Ces deux associations ne polluent pas le Méta-Modèle du langage puisqu'elles peuvent apparaître sans interférer avec les liens de composition déduits de la grammaire.

Les deux figures suivantes montrent le fragment du Méta-Modèle d'un langage faisant apparaître la règle 'Et' : S → S1, S2, et dont le symbole S est le point d'entrée de la grammaire :

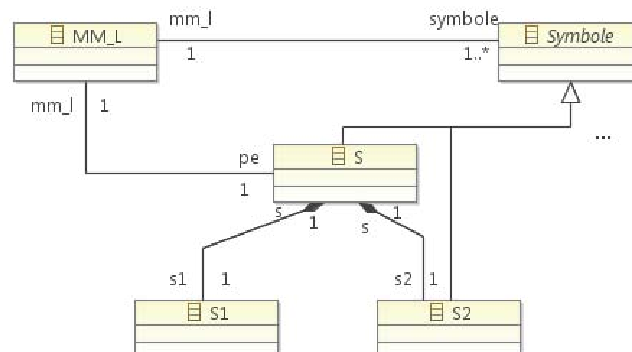


Figure 6 : ***Classe abstraite Symbole et classe MM_L dans le MM d'un langage (Version association)***

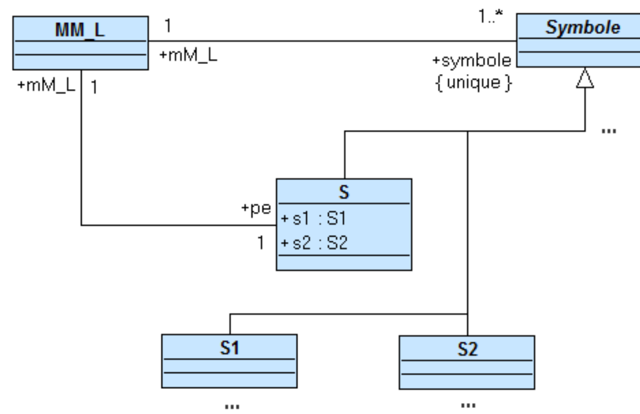


Figure 7 : ***Classe abstraite Symbole et classe MM_L dans le MM d'un langage (Version attribut)***

La méta-classe *Symbole* peut être utilisée pour y intégrer des opérations qui doivent être accessibles de toutes les méta-classes, en particulier comme on le verra plus tard, les constructeurs.

La grammaire d'un langage fait préalablement référence à l'ensemble des symboles terminaux et non terminaux avant la description des différentes règles décrivant les constructions syntaxiques élémentaires du langage. Cette énumération préalable des symboles permet d'effectuer des contrôles sur les symboles apparaissant dans les différentes règles. Ces deux classes MM_L et Symbole modélisent, en fait, cette énumération des symboles dans le méta-modèle UML.

Cette classe *Symbole* permet, par exemple, de réaliser en OCL l'opération de la classe S1 qui retourne l'objet s de S dont dépend l'objet s1 :

```
S1 ::s1_s() : S = self.mm_l.symbole->select( s | s.oclIsTypeOf( S )  and ( s <> oclUndefined( S ) ) and
                                                     s.oclAsType(S).s1 == self )->asSequence()->first().oclAsType( S )
```

I.1.c Méta-Modèle du langage « L »

Les deux figures suivantes montrent les deux fragments du méta-modèle du langage « L » :

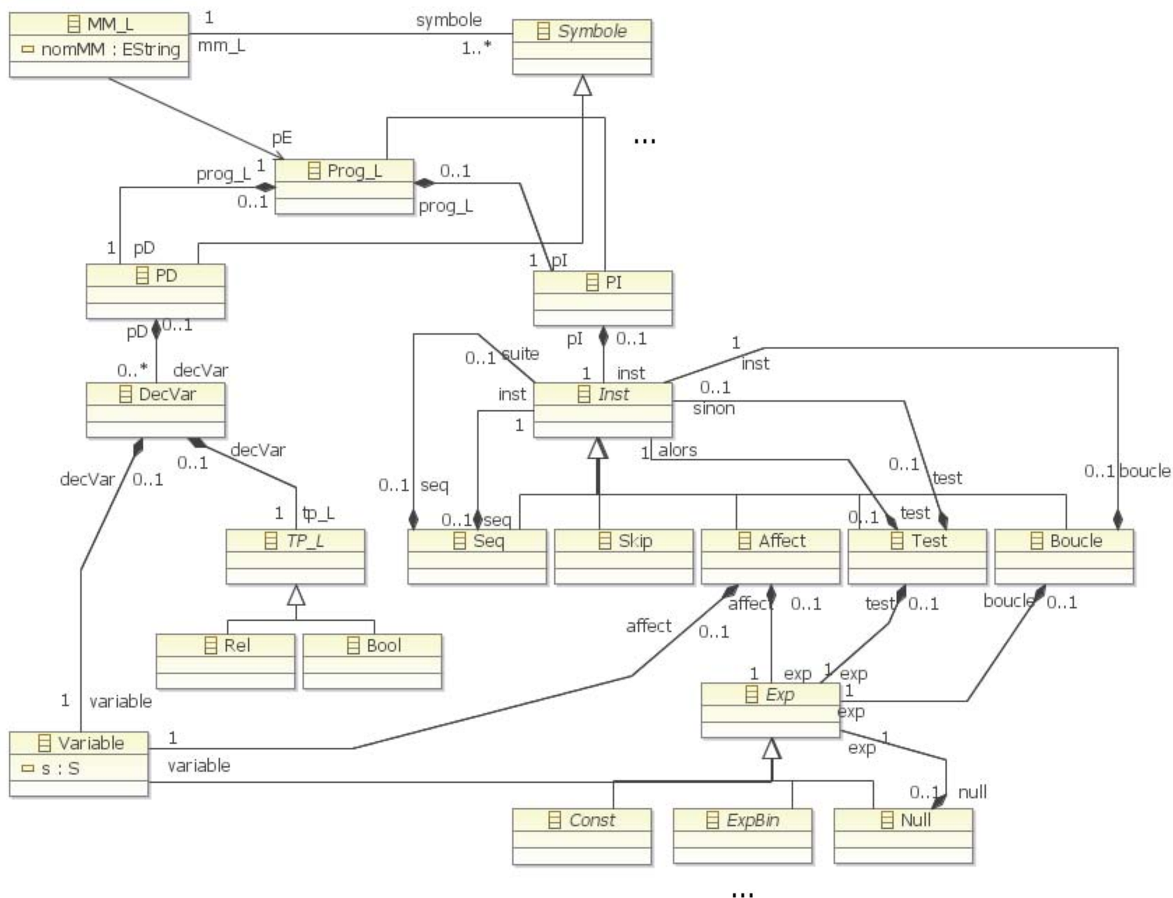


*Figure 8 : **Fragment des parties déclaratives et Instruction du Méta-Modèle du langage « L »***

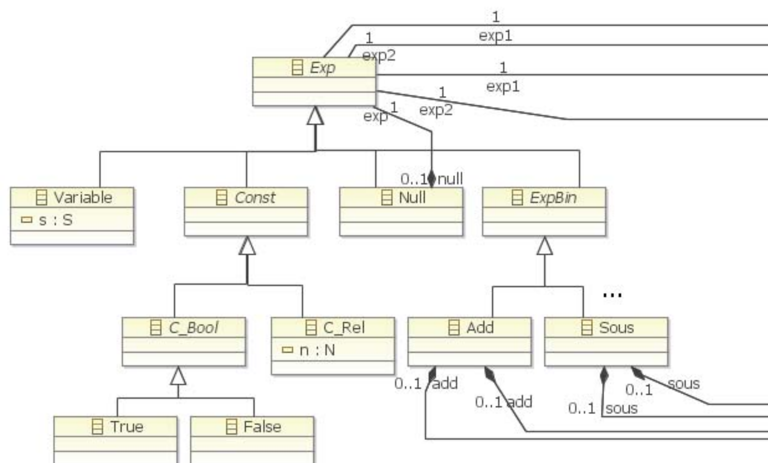


*Figure 9 : **Fragment des expressions du Méta-Modèle du langage « L »***

Pour en faciliter la représentation graphique, il n'est pas représenté dans ce Méta-Modèle les noms des rôles lorsqu'ils se déduisent directement des noms des symboles et les multiplicités quand elles sont égales à 1..1.

Il serait nécessaire de rajouter les invariants, tels qu'ils ont été décrits dans les paragraphes précédents.

Une version du Méta-Modèle où les attributs remplaçant les associations entre les symboles et les noms des rôles correspondants pourrait être donnée, complétée par les invariants qui en découlent. Ces deux versions sont identiques, la même expression OCL peut se retrouver dans l'une ou l'autre version. Cependant, le Méta-Modèle défini à l'aide des associations est plus lisible.

Le méta-modèle d'un langage de programmation peut être important et donc difficile à lire. La figure suivante montre que l'on peut représenter le méta-modèle d'une manière simplifiée, en dupliquant les méta-classes rentrant dans la composition de plusieurs méta-classes, telles que *Variable* ou Exp, évitant ainsi le croisement des associations :

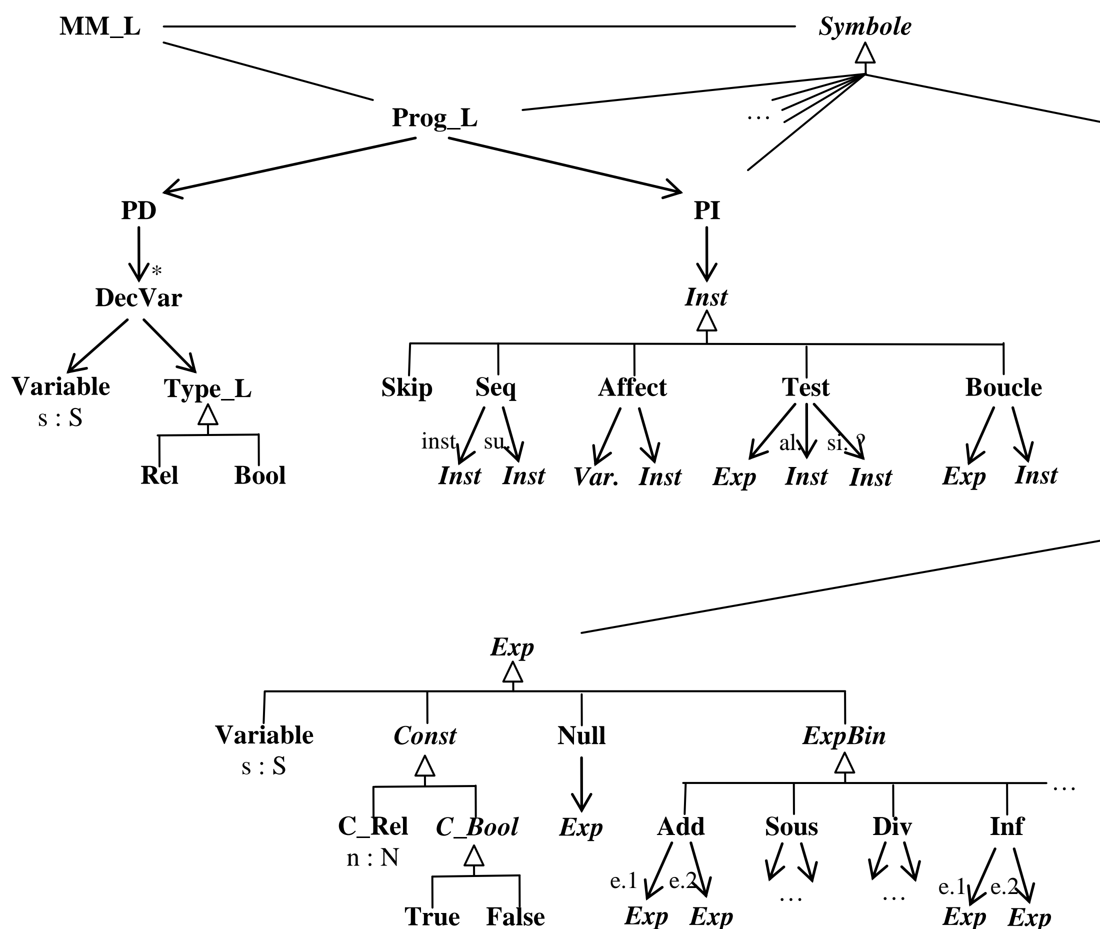


Figure 10 : ***Méta-modèle simplifié de la grammaire du langage « L »***

I.2 Exemple de modélisation d'un programme du langage « L »

Dans ce paragraphe, nous montrons un exemple de modèle UML représentant un programme « L », instance du Méta-Modèle de la grammaire du langage « L ». On donnera ensuite quelques exemples significatifs de propriétés en OCL, définies au niveau du Méta-Modèle de la grammaire et s'appliquant sur les feuilles des modèles de programmes « L ».

I.2.a Modélisation d'un programme du langage

Nous rappelons le programme suivant, écrit selon une syntaxe concrète évidente, et pris en exemple au chapitre précédent :

```
Le programme p
        x : Rel, y : Rel, yc : Rel, y1c : Rel
        x := 20
        y := 0
        yc := 0
        y1c := 1
        tant que non( x < y1c )
                y := y + 1
                yc := y2c
                y2c := yc + 2 * y + 1
        -- resultat y
```

Le modèle de ce programme, peut se représenter, d'une manière simplifiée, à l'aide d'un diagramme d'objets, instance du Méta-Modèle de la grammaire de « L », tel que le montre la figure suivante :

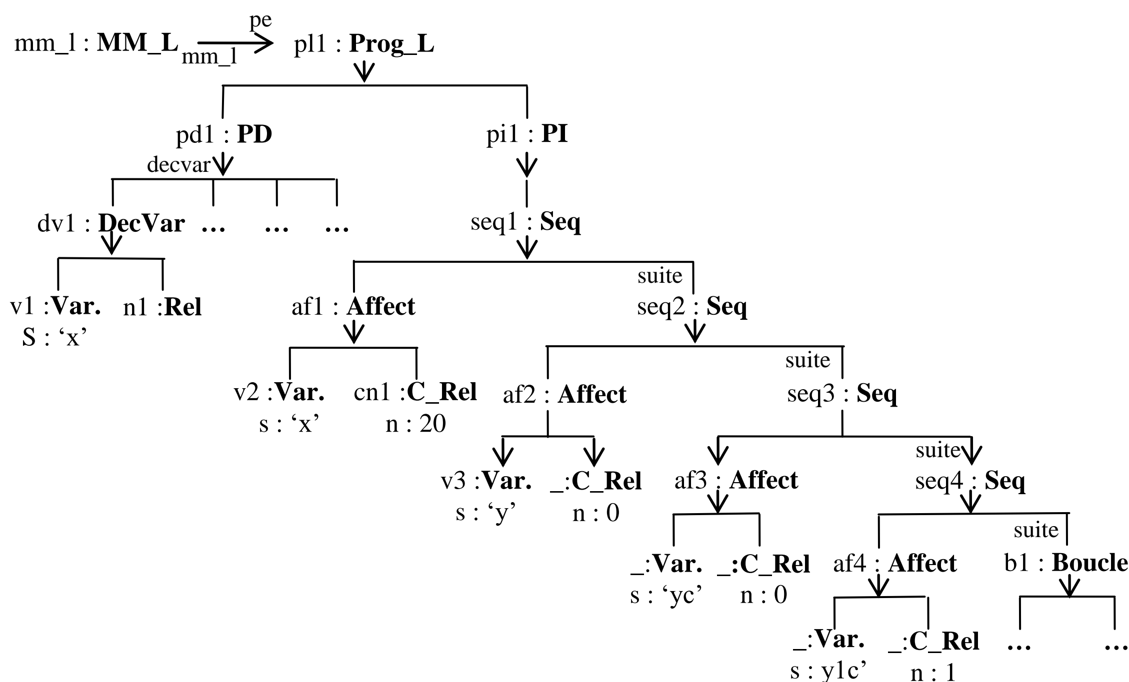


Figure 11 : ***Fragment du modèle d'un programme, instance du Méta-Modèle de la grammaire du langage « L »***

Pour des raisons de lisibilité du diagramme d'objets, nous n'avons pas fait apparaître les liens reliant l'objet, instance de MM_L, à la racine de l'arbre, aux nœuds et aux feuilles de l'arbre.

La figure précédente fait apparaître le modèle du programme comme une structure arborescente dont la racine est une instance de la Méta-Classe correspondant au point d'entrée de la grammaire. Chaque feuille de l'arbre dans la partie instruction porte, selon le type qu'elle référence, le nom d'une variable, ou une constante d'un type primitif.

Dans le modèle du programme on aurait pu regrouper certains objets, comme par exemple les objets représentant la même variable, ou les objets représentant le même type prédéfini. Pour cela, il aurait fallu le prévoir dans le méta-modèle du langage en prenant en compte, par exemple, qu'une instance d'une variable peut être en lien avec un ou plusieurs objets.

La grammaire d'un langage de programmation définit la structure syntaxique abstraite de tout programme du langage. Cependant, il peut apparaître des propriétés complétant les aspects structurels du langage mais non exprimables structurellement. Le langage OCL permet donc d'exprimer ces propriétés à l'aide d'invariants.

Cependant, ces propriétés peuvent porter soit sur les feuilles des structures arborescentes des programmes, soit sur les nœuds de ces structures arborescentes. Dans le premier cas, les propriétés, exprimées au niveau Méta-Modèle de la grammaire, portent sur le nom des variables et les constantes des programmes qui sont de types primitifs. Dans le deuxième cas, les propriétés portent sur les nœuds des structures arborescentes représentant les modèles de programmes, par exemple sur les symboles de la grammaire. De telles propriétés doivent donc être exprimées à un niveau de modélisation, au dessus des grammaires, c'est-à-dire à un niveau M3.

Au paragraphe suivant, nous donnons quelques exemples de propriétés définies au niveau Méta-Modèle de la grammaire. Les propriétés devant être exprimées au niveau M3 feront l'objet d'un autre paragraphe.

I.2.b Propriétés définies au niveau du Méta-Modèle du langage

Les propriétés suivantes donnent quelques exemples significatifs de contraintes portant sur le nom des variables ou sur les constantes pouvant apparaître dans un modèle de programme :

```
constraints

context Prog_L


-- Prog_L_1 : Tout nom de variable doit être défini, toute constante réelle doit être définie

        -- Règle de la grammaire du langage qui est concernée :     Variable → S
        --                                                          C_Rel → Z
        -- Invariant OCL :

inv prog _l_11 : self.mm_L.symbole->forAll( v | v.oclIsTypeOf( Variable )   implies
                                        v.s <> oclUndefined( String )        )
inv prog _l _l_12 : self.mm_L.symbole->forAll( z | z.oclIsTypeOf( C_Rel )   implies
                                        z.s <> oclUndefined( Integer )        )



-- Prog _L_2 : Les noms des variables apparaissant dans la partie déclarative sont différents deux à deux

        -- Règles de la grammaire du langage qui sont concernées :   Prog_L → PD
                                                                      PD → DecVar
                                                                      DecVar → Variable, TP_L
```

Variable → S

-- Invariant OCL :

```
inv prog _l_2 : self.decVar->forAll( dv1, dv2 | dv1 <> dv2 implies dv1.s <> dv2.s )
```

-- Prog _L_3 : Toute variable de la partie instruction doit avoir fait l'objet d'une déclaration dans la partie déclarative

-- Règles de la grammaire du langage qui sont concernées :

Prog_L → PD, PI
PD → decVar
DecVar → Variable, TP_L
Variable → S
PI → Inst
Inst → Affect | Exp | …
Affect → Variable, Exp
Exp → Variable | Const | …

-- Invariant OCL :

```
inv prog _l_3 : let setVarS_PI : Set( Variable ) =
                    self.mm_L.symbole->select( v | v.oclIsTypeOf( Variable )   and
                                                   v.decVar <> oclIsTypeOf( DecVar )  )->asSet()
          in setVarS_PI->forAll( v | self.pd.decvar.variable.s->includes( v.s )
```

…

*Figure 12 : **Quelques propriétés portant sur les symboles terminaux de la grammaire du langage « L »***

On peut remarquer que ces propriétés auraient pu s'écrire directement à partir de la grammaire du langage. Le Méta-Modèle du langage qui a été défini n'a apporté en fait aucune information supplémentaire. Ce qui signifie que le langage OCL pourrait être utilisé pour spécifier des propriétés que l'on pourrait rajouter à toute grammaire d'un langage et qui sont donc applicables à tout programme du langage.

II Modélisation en UML/OCL de la BNF et de ses propriétés

La vérification des propriétés portant sur les symboles de la grammaire se situe à un niveau Méta par rapport aux grammaires des langages de programmation. Il s'agit donc de reprendre le concept de BNF[1] qui est une grammaire donnant la syntaxe des règles de grammaire des langages de programmation. Considérée comme un langage de programmation, la BNF peut donc être modélisée à l'aide d'un diagramme de classes, appelé le Méta-Modèle de la BNF. Ce paragraphe est consacré à la BNF, à sa modélisation en UML/OCL, et aux propriétés qui porteront donc sur les symboles des grammaires des langages de programmation.

La BNF peut être considérée comme un DSL puisque c'est un langage 'dédié' au domaine d'applications concernant la description et la spécification des propriétés des langages de programmation.

[1] Backus-Naur Form

II.1 La BNF et Modélisation de la BNF

Les règles de grammaires décrivant des langages de programmation peuvent avoir une syntaxe plus ou moins élaborées. Nous avons déjà simplifié la syntaxe des règles de grammaire du langage L, pris comme exemple de référence d'un langage de programmation. En début de ce chapitre, nous avons décrit intuitivement la syntaxe des règles d'une grammaire, que nous reprenons ici pour décrire la grammaire de la BNF. Les notions présentées dans ce paragraphe peuvent être utiles à des experts d'un domaine d'applications pour définir formellement une famille de langages devant intégrer des propriétés métier par exemple, spécifiques à leurs domaines d'applications.

II.1.a Modélisation de la BNF

La grammaire de la BNF peut être la suivante :

SNT : { BNF, Règle, Symb, RegleET, RegleOU, Decl, Role, Mult, Un, PointInterro, Etoile, Plus }
ST : { S }
PE : BNF
RCS :
-- BNF
BNF → Symb +, Regle +, pe : SymbNT

-- les Symb
Symb → SymbNT | SymbT
SymbNT → S
SymbT → S

-- Les Regles
Regle → RegleET | RegleOU

-- Regles 'ET'
RegleET → SymbNT, Decl*
Decl → Role ?, Symb, Mult ?
Role → S
Mult → Un | PointInterro | Etoile | Plus
Un →
PointInterro →
Etoile →
Plus →

-- Règles 'OU'
RegleOU → SymbNT, ensSymbNT : SymbNT+

Figure 13 : ***Grammaire de la BNF simplifiée***

II.1.b Méta-Modèle de la BNF

Le diagramme de classes de la figure suivante montre le Méta-Modèle de la BNF, déduite de la BNF sous forme textuelle, après avoir repris les mêmes règles décrites précédemment pour passer d'une grammaire d'un langage à sa modélisation en UML :

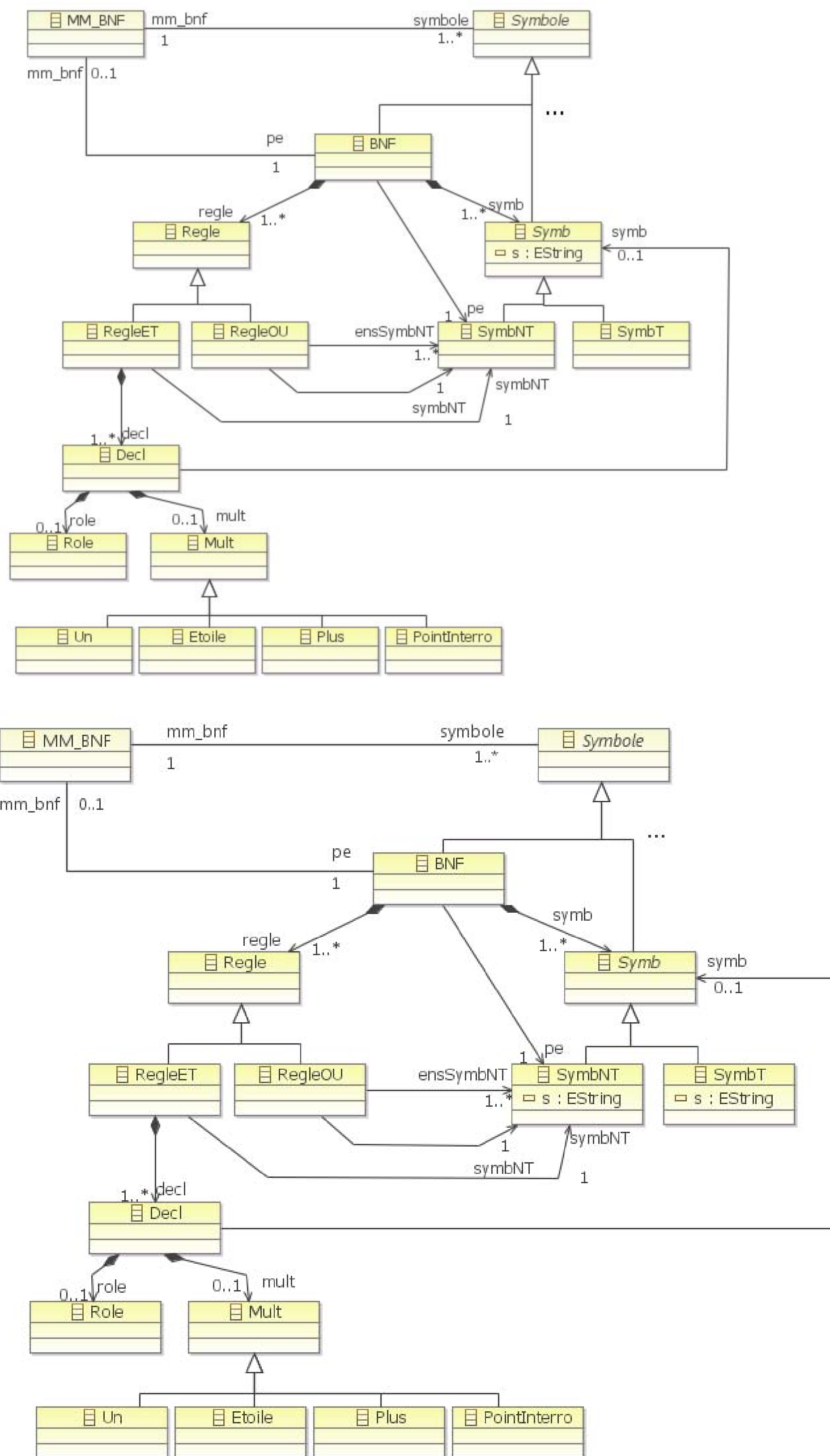


Figure 14 : ***Méta-Modèle de la BNF (MM BNF)***

La figure suivante montre une représentation arborescente du Méta-Modèle de la BNF :

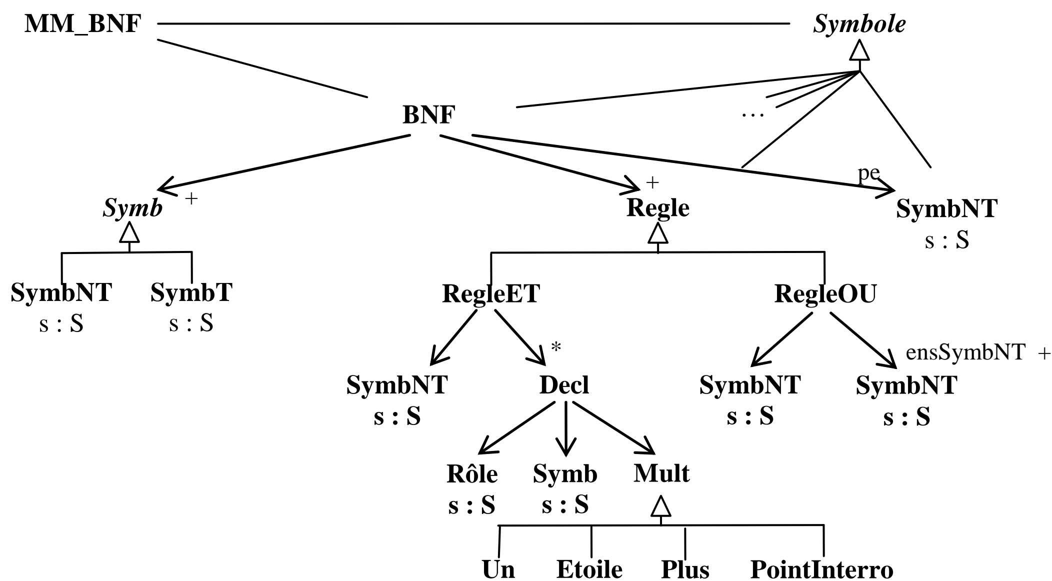


Figure 15 : ***Représentation du Méta-Modèle de la BNF simplifié***

II.1.c Modèle et Méta-Modèle de la Grammaire d'un langage de programmation

La figure suivante rappelle que le Méta-Modèle de la grammaire d'un langage de programmation était déduit directement de la grammaire du langage, à l'aide d'un diagramme de classes :

Niveaux de méta-modélisation de l'OMG

M3

M2 **MM de la Grammaire** (+OCL/LA) ← Grammaire d'un Langage

M1 Modèle du Programme ← Programme du Langage

Figure 16 : ***Grammaire d'un langage et son Méta-Modèle***

De manière à pouvoir exprimer des propriétés au niveau du Méta-Modèle de la BNF, et applicable sur la grammaire d'un langage, il s'agit préalablement de modéliser la grammaire d'un langage sous la forme d'une instance de la BNF, tel que le montre la figure suivante :

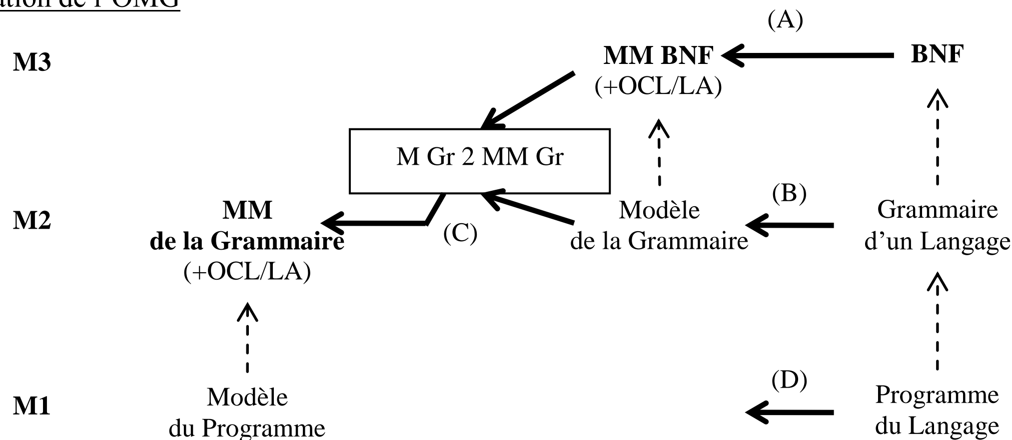


Figure 17 : ***Processus de Modélisation de la grammaire d'un langage en passant par la BNF***

Cette figure montre que de la BNF, pris sous une forme textuelle, on en déduit (A) le Méta-Modèle de la BNF, sous la forme d'un diagramme de classes UML, comme on l'a fait précédemment pour la grammaire du langage de programmation « L ».

Afin de pouvoir vérifier si la grammaire d'un langage de programmation vérifie toutes les propriétés, il est nécessaire de modéliser (B) cette grammaire en tant qu'instance du Méta-Modèle de la BNF. Si cette instance vérifie les propriétés établies au niveau du Méta-Modèle de la BNF, alors il est nécessaire de transformer (C) le Modèle de la grammaire, instance de la BNF, en un Méta-Modèle. On obtient ainsi le Méta-Modèle du langage sur lequel on pourra rajouter toutes les propriétés syntaxiques et sémantiques que devra respecter tout programme du langage. La vérification de ces propriétés se fait après avoir obtenu (D) le modèle d'un programme du langage, instance du Méta-Modèle du langage.

Ce processus de création du méta-modèle d'un langage de programmation reste, en principe, transparent aux Analystes/Concepteurs dont l'objectif est de transformer un modèle de conception, tel que le montre la figure suivante, en un modèle de codes de composants logiciels exécutables (a), et de produire les codes correspondants (b) :

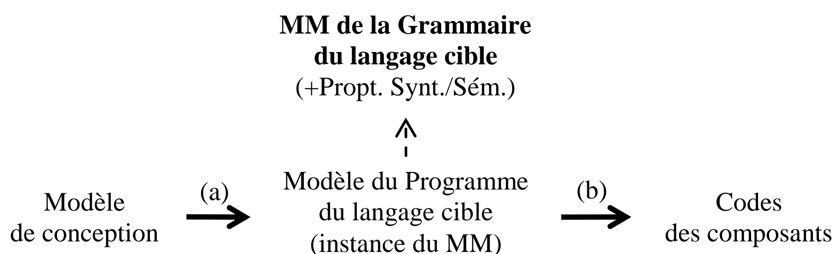


Figure 18 : ***Modélisation des codes des composants logiciels dans le cadre d'un processus de développement IDM***

II.2 Exemple d'une instance du Méta-Modèle de la BNF

II.2.a Le modèle de la grammaire du langage « L »

La figure suivante montre un exemple d'instance du Méta-Modèle de la BNF qui est le modèle de la grammaire du langage « L » :

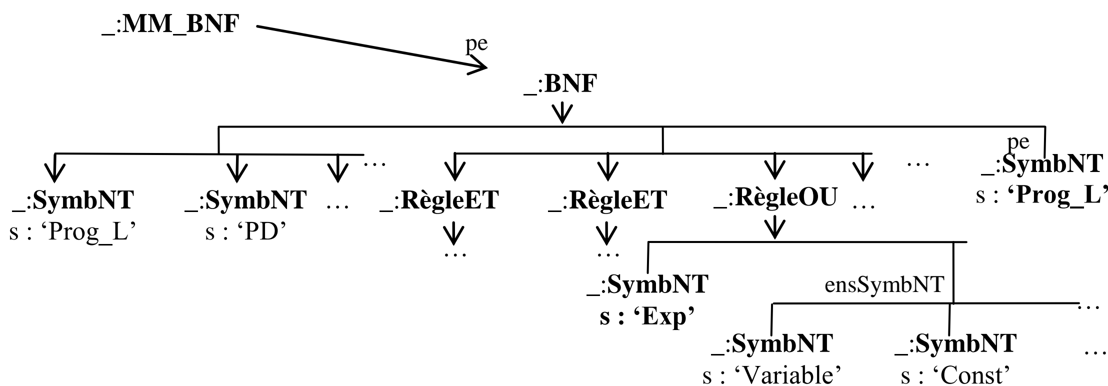


Figure 19 : ***Modèle de la grammaire du langage L, en tant qu'instance du méta-modèle de la BNF***

Pour plus de lisibilité de ce diagramme d'objets, on ne fait pas mention des liens entre les objets de la classe MM_BNF et les objets représentant les symboles.

Comme pour un modèle de programme, on aurait pu regrouper les objets représentant le même symbole, mais le diagramme d'objets devient alors illisible. Cependant, les multiplicités exprimant le rattachement éventuel d'un objet à plusieurs objets, fait qu'un méta-modèle de la BNF plus dans l'esprit de UML doit être proposé.

II.2.b Les propriétés définies au niveau MM BNF portant sur les symboles de la grammaire du langage « L »

Compte tenu du modèle simplifié des règles de production de la grammaire que nous avons choisie, les méta-propriétés sont les suivantes :

constraints

context BNF

-- BNF_1 : Le nom du symbole dans toute instance de la classe Symb doit être défini

-- Règle de la grammaire du langage qui est concernée : SymbNT → S
-- SymbT → S
-- Invariant OCL :

inv bnf_11 : self.mm_bnf.symbole->select(sNT | sNT.oclIsTypeOf(SymbNT)

```
                                      ).oclAsType( SymbNT )->forAll( sy | sy.s <> oclUndefined( String ) )
Inv bnf_11 : self.mm_bnf.symbole ->select( sT |    sT.oclIsTypeOf(    SymbT    )
                                      ).oclAsType( SymbT    )->forAll( sy | sy.s <> oclUndefined( String ) )
```

-- BNF_2 : Le nom du point d'entrée, les noms des symboles des parties gauches et des parties droites des règles sont des noms qui doivent apparaitre dans la liste des noms de symboles

-- BNF_3 : Le point d'entrée de la grammaire est un symbole non terminal

-- Règle de la grammaire du langage qui est concernée : BNF → Symb +, Regle +,
pe : SymbNT

-- Invariant OCL :

```
inv bnf_3 : let enssNT : self.mm_bnf.symbole ->select( sy | sy.oclIsTypeOf( SymbNT )
                                  ).clAsType( SymbNT )->asSet()
        in  enssNT.s->includes( self.pe.s )
```

-- BNF_4 : Le nom des symboles apparaissant en partie gauche de toute règle sont des symboles non terminaux

-- Règle de la grammaire du langage qui est concernée : Regle → regleET | RegleOU
RegleET → SymbNT ; Decl *
RegleOU → SymbNT ;
ensSymbNT SymbNT

-- Invariant OCL :

```
inv bnf_4 : let reET : ensRegleET = self.regle->select( rET | rET.oclIsTypeOf( RegleET )
                                          ).oclAsType( RegleET )
          in let syNT : ensSymbNT = self.mm_bnf.symbole ->select( syNT | syNT.oclIsTypeOf(
SymbNT )
                                          ).oclAsType( SymbNT )
            in syNT.s->includesAll( reET.symbNT.s )
```

-- Commentaire : reET est l ensemble des regles ET et syNT est l ensemble des symboles non terminaux

…

-- BNF_5 : Le même nom de symbole ne peut pas apparaître en partie gauche que dans une seule règle

-- BNF_6 : Le nom des symboles de droite d'une règle OU sont tous différents

-- …

-- BNF_7 : Pour tout symbole différent du point d'entrée, il doit exister un chemin d'extrémité initiale le point d'entrée et d'extrémité terminale le symbole

```
BNF ::cFC_iter_suiv( ensNomSymb_pE : Set( String ) ) : Set(String ) =
        let ensReglesET :          Set( Symb ) =
              self.regle→select( r | r.oclIsTypeOf( RegleET ) ).oclAsType( ReglesET )->asSet()
      in let ensReglesET_Suiv : Set( Symb ) =
              ensReglesET->select( r |       ensNomSymb_pE->includes( r.symbNT.s ) and
```

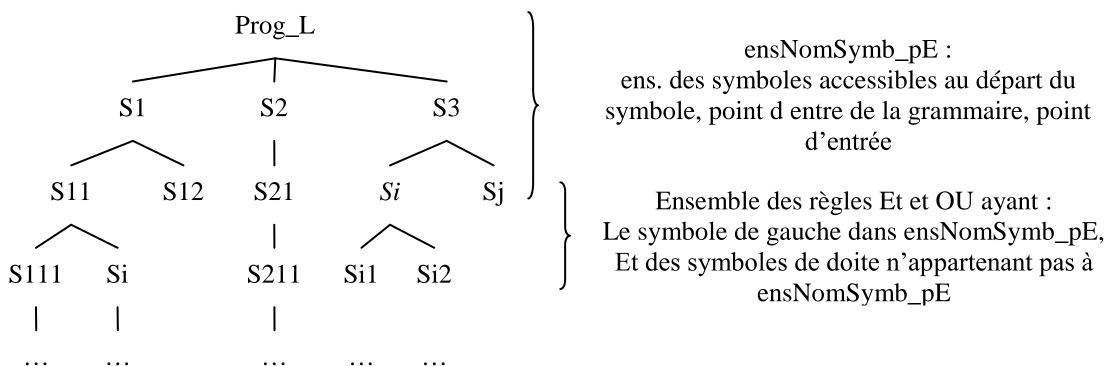
```
                                  ( not ensNomSymb_pE.s->includesAll( r.decl.symb.s )
                              )->asSet()
          in let ensReglesOU : …
             in  let ensRgelesOU_suiv : …

                in ensNomSymb_pe->union( ensReglesET.ensSymbNT.s )->asSet()
                                     )->union( ensReglesOU.symbNT.s )->asSet()

BNF::cFC( ensNomSymb_pE : set( String ) ) : Set( String ) =
        if self.cFC_iter( ft_ensNomSymb_pE )->size() = 0
        then ensNomSymb_pE
        else  self.cFC( ensNomSymb_pE->union( self.cFC_iter_suiv( ensNomSymb_pE ) )->asSet()
        endif

inv bnf_7 : self.cFC( Set{ self.pe.s } ) =
        self.mm_bnf.symbole->select( syNT | syNT.oclIsTypeOf( SymbNT )
                          ).oclAsType( SymbNT
                                      ).s->asSet()->union( select( syT |  syT.oclIsTypeOf( SymbT )
                                                              ).oclasType( SymbT ).s )->asSet()
```

-- Commentaires : Calcul de la composante fortement connexe d'un graphe



inv bnf_8 : Les liens d'héritage peuvent se représenter sous la forme d'une structure arborescente dont la racine est la classe abstraite Symbole
…

Figure 20 : ***Exemples d'invariants OCL spécifiant quelques Méta-Règles***

A ce niveau de modélisation UML, on aurait pu s'attendre à avoir des expressions OCL moins compliquées, surtout qu'elles n'apparaissent pas dans les documents de référence que nous avons repris sur les langages dans le premier chapitre ! Ces propriétés sont apparues nécessaires dans la mesure où l'on souhaitait exécuter les modèles des codes.

Ces expressions OCL auraient-elles été plus simples sur une BNF moins élaborée ? Le langage OCL répond-il aux objectifs que nous nous sommes fixés ? Est-ce la raison pour laquelle il reste encore peu utilisé et appliqué en milieu industriel ?

En particulier, nous avons supposé que les instances des Méta-Modèles sont des structures arborescentes. On aurait pu, dans certains cas, regrouper certains objets, comme par exemple, les instances de la Méta-Classe Symbole qui ont le même nom dans le Méta-Modèle de la BNF, ou les instances de la Méta-Classe Variable qui ont le même nom dans le Méta-Modèle du langage « L ».

En fait, ces expressions OCL peuvent être difficiles à élaborer et à comprendre car OCL est un langage typé qui s'applique pleinement sur les concepts d'héritage et de navigation des modèles UML, où les interprètes OCL peuvent être très exigeants sur la vérification du typage des expressions. Le langage OCL a été utilisé dans les cas d'étude du projet DOMINO de l'ANR, parce que c'est le complément indispensable pour apporter les précisions nécessaires à des représentations graphiques de modèles où la sémantique est absente. L'expérience a montré que lorsque les expressions OCL représentant les invariants et, comme on le verra plus tard, les expressions OCL apparaissant dans des opérations, ont été définies et testées dans un environnement d'exécution, la génération des codes ne pose plus de problème dans la mesure où tout a pu être exprimé précisément. N'est-ce pas l'objectif des modèles et des précisions que l'on peut apporter sur les modèles pour que les Analystes/Programmeurs puissent comprendre sans ambigüité les intentions des Analystes/Concepteurs ?

II.3 Génération du Méta-Modèle de la BNF et du Méta-Modèle d'une grammaire

II.3.a Environnement de modélisation de la plate-forme USE

La figure suivante décrit le processus de création du Méta-Modèle d'une grammaire d'un langage, sous l'environnement de la plate-forme USE :

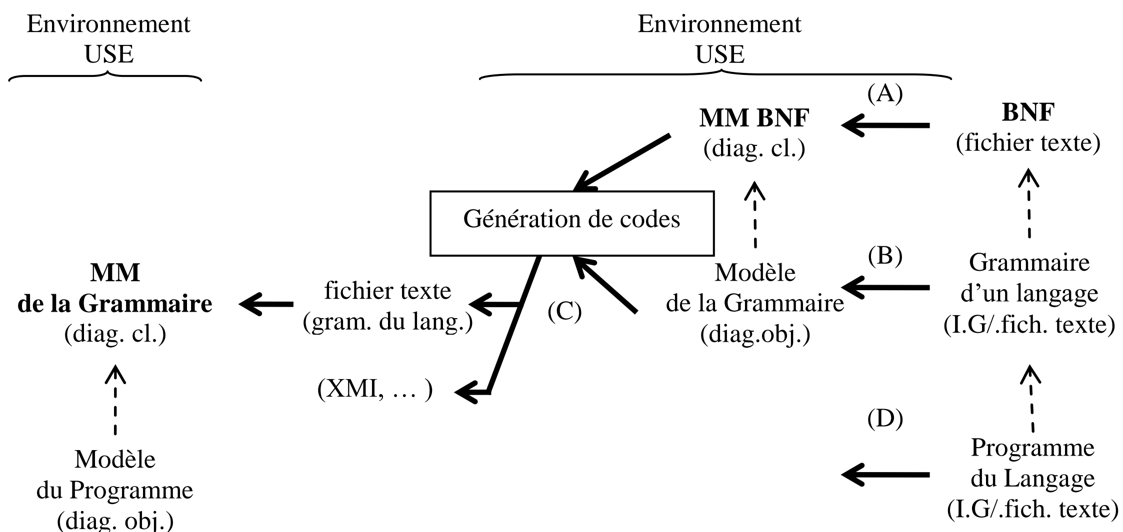


Figure 21 : ***Processus de création du Méta-Modèle de la grammaire d'un langage sous selon l'environnement de développement de la plate-forme USE***

Dans le cadre de cet environnement USE, il existe un langage de commandes pour créer un diagramme de classes (fichier .use) à partir d'un fichier texte décrivant les classes et les associations. Ce langage de commande permet de charger (A) le Méta-Modèle de la BNF sous la forme d'un diagramme de classes. A partir de ce diagramme de classes, on peut gérer

(B) dynamiquement à l'aide d'une interface graphique, ou statiquement à l'aide d'un fichier texte (.cmd) un diagramme d'objets qui, dans notre cas, représente la grammaire d'un langage de programmation. Nous avons, ensuite, à l'aide du langage d'actions USE écrit un programme générant le texte source permettant de créer (C) le Méta-Modèle du langage sous la forme d'un diagramme de classes. On peut alors créer (D) un Modèle de programme du langage selon le même principe, mais dans notre étude sur les processus IDM, le modèle d'un programme viendra directement d'une transformation de diagrammes UML.

II.3.b Environnement de développement sous Eclipse/EMF

La figure suivante décrit le processus de création du Méta-Modèle d'une grammaire d'un langage, sous l'environnement de Méta-programmation Eclipse/EMF :

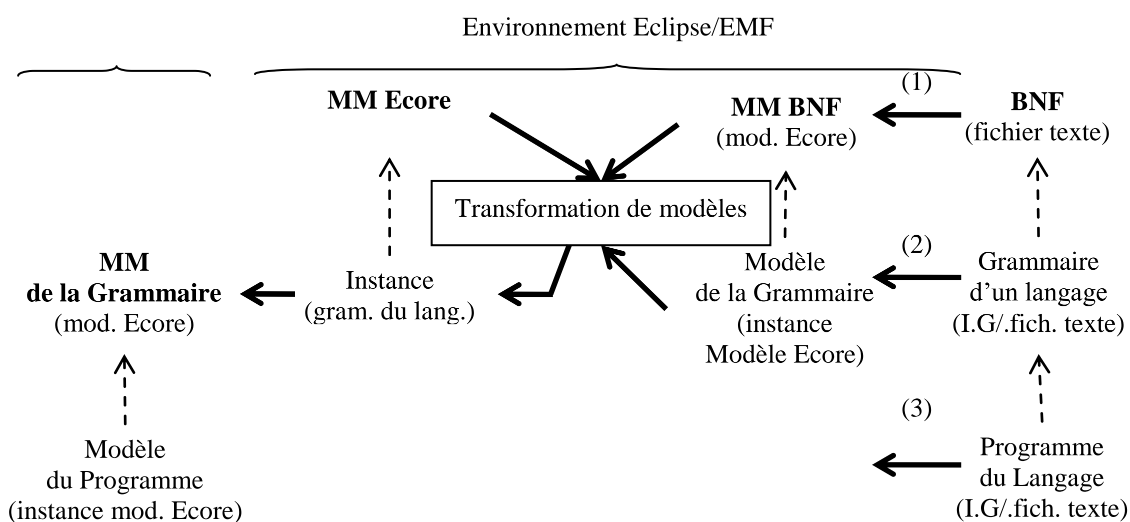


Figure 22 : ***Processus de création du Méta-Modèle de la grammaire d'un langage sous selon l'environnement de Méta-Modélisation Eclipse/EMF***

Dans l'environnement Eclipse/EMF, l'éditeur de méta-modèle Ecore a été utilisé pour construire directement le méta-modèle de la BNF. Les invariants décrits précédemment sont mis en œuvre en Kermeta. Il est alors possible d'introduire, à l'aide de l'éditeur de modèles de EMF, la grammaire d'un langage en tant qu'instance du méta-modèle de la BNF. Un programme Kermeta a alors été écrit pour transformer cette instance en un méta-modèle du langage.

III Modélisation des Propriétés de typage des langages

Dans ce paragraphe, nous reprenons les règles décrivant les propriétés de typage du langage « L », rappelées au premier chapitre, et montrons comment elles peuvent être spécifiées en OCL.

Il s'agit, tout d'abord, de déterminer le type de la valeur retournée de chaque expression. La vérification du typage des instructions peut alors être vérifiée. Par exemple, il faut s'assurer que la partie gauche et la partie droite d'une instruction d'affectation sont de même type.

Ces propriétés qui se définissent au niveau du Méta-Modèle du langage, se vérifient sur des modèles de codes d'une manière statique.

III.1 Calcul du type des expressions du langage « L »

Les types doivent être manipulés comme des objets indiquant le type de chaque expression. On définit préalablement les constructeurs des types prédéfinis du langage « L » qui sont écrits dans le langage d'actions de USE.

III.1.a Constructeur retournant un type (Langage d'actions de la plate-forme USE)

Les constructeurs retournant les objets définissant les types des expressions sont les suivants :

```
Symbole ::Bool() : Bool
begin
   var          b :          Bool
   create       b := new Bool
   set result := b
end
```

```
Symbole::Rel() : Rel
begin
   var          z :          Rel
   create       z := new Rel
   set result := z
end
```

Figure 23 : ***Constructeurs retournant les objets correspondant aux types prédéfinis du langage « L »***

D'une manière générale, la fonction retournant le type d'une expression est la suivante :

Exp ::type() : TP_L

Figure 24 : ***Spécification de la fonction retournant le type d'une expression***

Le type de l'expression retournée lors de son exécution dépend du type des éléments qui composent l'expression, qui peut être une opération binaire, une constante ou une variable. Les liens entre une expression et son type ont été spécifiés au chapitre précédent. Le type d'une expression unaire ou binaire dépend du type de son (ou ses) argument(s) ; le type d'une constante est défini a priori comme un axiome. Par contre le type d'une variable dépend de sa déclaration qui en a été faite dans la partie déclarative.

III.1.b Fonctions retournant le type des opérations unaires et binaires

Les fonctions retournant le type du résultat des opérations unaires et binaires, se déduisent directement des spécifications qui ont été rappelées au chapitre précédent, tel que le montre la figure suivante :

```
Null::type() :     TP_L =  if      self.exp.type().oclIsTypeOf( Rel )
                           then                   Bool()
                           else    oclUndefined( Bool   )
                           endif

Add ::type() :     TP_L =  if      self.exp1.type().oclIsTypeOf( Rel ) and
                                   self.exp2.type().oclIsTypeOf( Rel )
```

```
                              then                  Rel()
                              else      oclUndefined( Rel   )
                                        endif
…
Inf ::type() :     TP_L =     if        self.exp1.type().oclIsTypeOf( Rel ) and
                                        self.exp2.type().oclIsTypeOf( Rel )
                              then                  Bool()
                              else      oclUndefined( Bool   )
                              endif
…
```

Figure 25 : ***Fonctions OCL retournant le type des opérations unaires et binaires d'une expression***

Il nous parait important de bien remarquer que ces fonctions se spécifient en OCL, compte tenu des constructeurs écrits en langage d'actions UML, en l'occurrence celui de la plate-forme USE. Ces constructeurs permettent de retourner des objets qui ne sont pas des objets du modèle de codes où les fonctions seront appliquées. C'est pourquoi, on peut dire que ces fonctions OCL sont sans effet de bord sur les modèles de codes, mais cependant nécessitent l'exécution d'un constructeur faisant partie du langage d'action.

III.1.c Fonctions retournant le type d'une constante

La correspondance entre une constante du langage « L » est définie, bien sûr, a priori : Une constante de type C_Rel, est de type Rel, et une constante de type C_Bool est de type Bool. Les fonctions retournant le type d'une constante sont donc les suivantes :

```
Const::type() :    TP_L
C_Bool::type() :   TP_L =  Bool()
C_Rel::type() :    TP_L =  Rel()
```

Figure 26 : ***Fonctions OCL retournant le type d'une constante***

III.1.d Fonction retournant le type d'une variable

Retourner le type d'une variable peut se réaliser selon des approches différentes.

Une première solution consiste à chercher dans la partie déclarative le type associé à la variable et à retourner une copie de l'objet type associé, comme pour les fonctions OCL retournant le type d'une constante :

```
Variable ::type() : TP_L =  self.mm_l.pe.pd.decvar->select( dv| dv.variable.s = self.s )->first().tp_l.copy()
```

Figure 27 : ***Fonctions OCL retournant le type d'une variable***

La recherche de la variable dans la partie déclarative se fait en repartant de la classe MM_L pour sélectionner parmi tous les objets, instances de la classe abstraite *Symbole*, l'objet de DecVar où la variable dont on cherche le type est déclaré.

Une deuxième solution qui pourrait être appliquée à des langages plus élaborés consiste à passer en paramètre la liste des déclarations de variables et de leur type :

```
Exp ::type( ensVarTyp : Sequence( DecVar ) ) : TP_L
```

Ainsi la recherche du type associé à une variable est plus directe, puisqu'il suffit de parcourir la séquence de déclarations passée en paramètre :

```
Variable ::type( ensVarTyp : Sequence( DecVar ) ) : TP_L
        = ensVarTyp.decvar->select( dv| dv.variable.s = self.s )->first().tp_l.copy()
```

Pour cette solution, il faut donc que chacune des fonctions retournant le type d'une expression passe en paramètre la séquence des déclarations. Ce paramètre doit donc être initialisé avant d'évaluer le type d'une expression.

III.2 Vérification du typage des instructions du langage « L »

La vérification du typage des instructions se fait à partir des spécifications qui ont été rappelées dans le chapitre précédent.

III.2.a Fonction vérifiant le typage d'une instruction

D'une manière générale, la spécification de la fonction vérifiant le typage d'une instruction est la suivante :

Inst ::ok() : Boolean = true

Figure 28 : ***Spécification de la fonction validant le typage d'une instruction***

III.2.b Fonctions vérifiant le typage des différentes instructions du langage

Les fonctions validant le typage des différentes instructions du langage « L » sont les suivantes :

```
Skip ::ok() :     Boolean =     true

Seq ::ok() :      Boolean =     self.inst.ok() and self.suite.ok()

Affect::ok() :    Boolean =     ( self.variable.type().oclIsTypeOf( Rel  )   and
                                  self.exp.type().oclIsTypeOf(      Rel  )        ) or
                                ( self.variable.type().oclIsKindOf( Bool )   and
                                  self.exp.type().oclIsKindOf(      Bool )        )

Test::ok() :      Boolean =     ( self.exp.type().oclIsKindOf(      Bool )   and
                                  self.alors.ok()                                  ) and
                                ( self.sinon <> oclUndefined( Inst )         implies
                                  self.sinon.ok()                                  )

Boucle::ok() :    Boolean =       self.exp.type().oclIsKindOf(      Bool )   and
                                  self.inst.ok()
```

Figure 29 : ***Fonctions vérifiant le typage des différentes instructions du langage « L »***

Dans le cas où la séquence associant chaque variable à son type est passée en paramètre, ces fonctions sont les suivantes :

Inst ::ok(ensVarTyp : Sequence(DecVar)) : Boolean = true

En particulier on devra écrire :

```
Boucle::ok(ensVarTyp : Sequence( DecVar ) ) : Boolean =
                    self.exp.type( ensVarTyp ).oclIsKindOf( Bool )   and
                    self.inst.ok()
```

L'avantage de cette solution vient du fait que l'on pourrait prendre en compte des déclarations de variables dans des blocs d'instructions, en particulier dans le langage L, on pourrait rajouter au niveau du symbole Seq une partie déclarative :

Seq → **DecVar**, inst : **Inst**, suite : **Inst**

Dans ce cas, la fonction validant le type de cette instruction devra être :

```
Seq ::ok(ensVarTyp : Sequence( DecVar ) ) : Boolean =
                      let evt : Sequence( DecVar ) = ensVarTyp->union( self.decvar )
                    in self.inst.ok( evt ) and self.suite.ok( evt )
```

De telles fonctions récursives manipulant des structures de données arborescentes gèrent de fait l'imbrication des déclarations de variables et leurs visibilités dans les séquences (sous-arbres) de modèles correspondantes.

Les fonctions validant le typage des instructions sont des expressions dont la syntaxe est OCL, mais elles font appel à des constructeurs écrits en langage d'actions. Elles restent sans effet de bord sur les modèles des programmes.

IV Gestion dynamique des propriétés des langages de programmation

Au cours de ce chapitre, nous avons montré les techniques permettant de déduire de la grammaire d'un langage de programmation le méta-modèle du langage. Un expert peut alors injecter au niveau du Méta-Modèle les propriétés syntaxiques et les propriétés de typage en OCL et en LA. Cependant, il ne s'agit pas de refaire à un niveau de modélisation tout ce que fera le traducteur lorsqu'il prendra en charge un programme du langage. Il s'agit principalement de bien définir ce que l'on souhaite contrôler au niveau des modèles des programmes. On peut, par exemple, mettre en place des propriétés métier obligeant aux Analystes/Concepteurs de suivre les règles de programmation et de modélisation en usage dans l'entreprise.

C'est cet aspect d'accès aux Méta-Modèles des langages qui nous parait important par rapport aux traducteurs qui sont assimilés à des boites noires. Le rôle des experts d'un domaine d'applications, vu sous cet angle, parait donc beaucoup plus difficile qu'on aurait pu le penser a priori.

IV - Modélisation en UML/OCL des Propriétés Comportementales et Axiomatiques des Langages de Programmation

Le chapitre précédent, consacré à la modélisation des symboles et des règles de constructions syntaxiques d'une grammaire d'un langage de programmation, a permis de montrer comment le langage OCL étendu aux seuls constructeurs d'un langage d'actions (LA) de UML (LA), peut être utilisé pour mettre en œuvre les propriétés syntaxiques, assimilant tout programme à un ensemble de formules bien formées.

Dans ce chapitre, nous montrons comment le langage OCL et les constructeurs d'un langage d'actions peut être utilisés pour spécifier les propriétés comportementales et axiomatiques d'un langage de programmation.

Les propriétés comportementales sont spécifiées par rapport à un comportement de référence des structures syntaxiques de la grammaire. Il s'agit donc de modéliser en UML/OCL ce comportement de référence qui permettra de simuler l'exécution de tout modèle ou tout fragment de modèle d'un programme du langage à l'aide de l'environnement d'exécution. Ces propriétés comportementales expriment une sémantique dynamique par rapport, par exemple, aux propriétés de typage qui sont vérifiées directement sur le code.

Les propriétés axiomatiques d'un langage de programmation permettent de vérifier si un fragment de code du langage est correct par rapport à ses propriétés de type pré- et post-conditions rajoutées sur le code, l'ensemble constituant le triplet de Hoare. Appliquées au niveau UML, elles permettent donc de vérifier si un modèle de code ou un fragment de modèle de code est correct par rapport à ces propriétés. Leur vérification statique amène à étudier la validité des assertions déduites directement du code et des pré- et post-conditions. Dans ce chapitre, nous montrons qu'il est nécessaire de faire appel à un prouveur devant gérer une bibliothèque de règles d'assertions à partir desquelles les preuves pourront être faites. Cependant, apporter la preuve qu'une assertion est valide (si elle l'est) n'est pas forcément évident dans la mesure où il peut être nécessaire d'aider le prouver à trouver « bonnes règles » et la « bonne stratégie » pour la démonstration. Nous avons utilisé un atelier B qui permet de modéliser les assertions d'une manière très proche à UML. Ce qui permet donc aux Analystes/Concepteurs de rester dans un environnement de processus dirigé par les modèles, et de passer d'un formalisme UML à un formalisme de machine B, sans difficultés majeures.

Nous montrons ensuite comment on peut appliquer les propriétés comportementales et axiomatiques lors de la transformation de modèles de programmes, comme le fait tout traducteur pour simplifier ou optimiser des programmes. Ces exemples nous servirons d'introduction à la troisième partie de notre travail où de telles propriétés peuvent être appliquées sur des diagrammes d'activité. Ces propriétés peuvent jouer un double rôle. On peut les voir comme une abstraction des propriétés des langages cibles à un niveau de modélisation du processus en amont des modèles de codes. On peut les voir aussi comme un raffinement des propriétés issues des exigences des applications anticipant les modèles de code. Le diagramme d'activité peut donc servir de charnière dans un processus IDM entre les activités de modélisation des exigences et les activités de génération des codes.

IV - Modélisation en UML/OCL des Propriétés Comportementales et Axiomatiques des Langages de Programmation

Table des matières

I Modélisation en UML/OCL des Propriétés Comportementales ... *108*
I.1 Environnement d'exécution d'un programme ... **108**
I.1.a Définition de l'environnement ... 108
I.1.b Exécution d'un modèle de programme ... 110
I.2 Mise en œuvre des propriétés comportementales du langage « L » ... **111**
I.2.a Mise en œuvre des propriétés comportementales des expressions ... 111
I.2.b Mise en œuvre des propriétés comportementales des instructions ... 111
I.2.c Spécification et mise en œuvre des propriétés comportementales de la partie déclarative ... 112
I.3 Réalisation sous la plate-forme USE et Kermeta ... **112**
II Modélisation en UML/OCL des Propriétés Axiomatiques ... *113*
II.1 Les environnements nécessaire à la prise en compte des propriétés axiomatiques ... **113**
II.1.a Grammaire du langage « L » intégrant la définition des assertions ... 114
II.1.b Propriétés de typage des assertions et des triplets de Hoare ... 116
II.1.c Exemple d'un fragment de modèle de programme du langage « L » ... 117
II.1.d Machine B et raffinement prenant en compte un triplet de Hoare ... 118
II.2 Algorithme de la Plus Faible Pré-condition d'un triplet de Hoare ... **122**
II.2.a Algorithme de la Plus Faible Pré-condition (pfp) et les obligations de preuve ... 122
II.2.b Règles de calcul de la plus faible pré-condition pour les instructions du langage « L » ... 124
II.2.c Obligations de preuve pour les instructions du langage « L » ... 124
II.3 Modélisation en UML/OCL des propriétés axiomatiques ... **125**
II.3.a Fonctions OCL de substitution des instructions d'affectation du langage « L » ... 125
II.3.b Fonctions OCL calculant les pfp des instructions du langages « L » ... 126
II.3.c Fonction OCL retournant les obligations de preuve des instructions du langage « L » ... 126
II.3.d Synthèse des techniques présentées dans ce paragraphe ... 127
III Quelques exemples d'application ... *129*
III.1 Exemple d'un Modèle de code du langage « L » ... **130**
III.1.a Modèle de code ... 130
III.1.b Arbres déduits du triplet de Hoare et du calcul de la pfp ... 131
III.2 Retour au calcul de la racine carrée approchée d'un nombre entier ... **132**
III.2.a Programme « L » ... 132
III.2.b Invariant, Variant, Triplet de Hoare et Obligations de preuve ... 132
III.2.c Modèle du programme « L » ... 132
III.2.d Calcul de l'assertion associée au triplet de Hoare ... 133
III.3 Factorisation d'une expression ... **135**
III.3.a Fragment de programme ... 135
III.3.b Triplet de Hoare ... 136
III.3.c Calcul de l'obligation de preuve associée au triplet de Hoare ... 136
III.3.c.1 Modélisation en B de l'assertion définissant le triplet de Hoare ... 137
III.3.c.2 Modélisation en B de l'assertion du triplet de Hoare après le calcul de la pfp ... 138
III.3.c.3 Cas particulier : Les types prédéfinis sont comparables ... 139
III.3.d Modélisation en UML/OCL ... 139
III.3.e Calcul de l'obligation de preuve ... 140
III.4 Vers la vérification de transformations de modèles ... **141**

IV - Modélisation en UML/OCL de la Sémantique des Langages de Programmation

Table des figures

Figure 1 : Grammaire (1) et Méta-Modèle(2) définissant l'environnement d'exécution du langage « L » 109
Figure 2 : Correspondance entre les types prédéfinis et les constantes du langage L et le type des valeurs sémantiques 109
Figure 3 : Constructeurs et gestion de l'environnement d'exécution 110
Figure 4 : Correspondance entre les opérateurs des expressions du langage « L » et les opérations sur les valeurs sémantiques 110
Figure 5 : Schéma montrant le modèle d'un programme et son environnement d'exécution 110
Figure 6 : Spécification de la propriété comportementale d'une expression 111
Figure 7 : Mise en œuvre des propriétés comportementales des différents types d'expressions 111
Figure 8 : Spécification de la propriété comportementale d'une instruction 111
Figure 9 : Mise en œuvre des propriétés comportementales des instructions du langage « L » 112
Figure 10 : Exemple d'extension de la grammaire du langage « L » prenant en compte la définition des assertions du triplet de Hoare, et leurs opérations 114
Figure 11 : Méta-Modèle du langage « L » prenant en compte la modélisation simplifiée des assertions 115
Figure 12 : Propriétés de typage des assertions 116
Figure 13 : Propriétés comportementales des assertions et des triplets de Hoare 116
Figure 14 : Exemple d'un modèle de programme du langage « L » contenant un triplet de Hoare 118
Figure 15 : Atelier B prenant en charge la validation des obligations de preuve issues des triplets de Hoare 119
Figure 16 : Règles de calcul de la plus faible pré-condition pour les instructions du langage « L » 124
Figure 17 : Tableau récapitulatif des obligations de preuve pour les instructions du langage « L » 125
Figure 18 : Fonction de substitution OCL/LA, dans une expression, d'une variable par une expression 126
Figure 19 : Fonction OCL/LA retournant le pfp d'une instruction et d'une post-condition 126
Figure 20 : Exemple simple d'un Modèle de code du langage « L » 130
Figure 21 : Arbre de l'Obligation de Preuve après le calcul de la pfp 131
Figure 22 : Correspondance entre les types et les opérateurs du Langage « L » et de B 131
Figure 23 : Fragment du modèle du programme faisant apparaître l'instruction de boucle 133
Figure 24 : Calcul de l'assertion associée au triplet de Hoare pour une instruction de boucle 134
Figure 25 : Structure de données relative à l'obligation de preuve pour l'instruction de boucle 134
Figure 26 : Fragment du Modèle de programme concernant le triplet de Hoare 140
Figure 27 : Modèle de la post-condition du triplet de Hoare 140
Figure 28 : Modèle de l'assertion associée au triplet de Hoare 141
Figure 29 : Modèle Source, Modèle cible ; programme de transformation 141

Au chapitre précédent, nous avons décrit un exemple de méta-modèle de la grammaire d'un langage et de ses propriétés.

Un premier paragraphe est consacré à la sémantique opérationnelle qui permet d'étudier le comportement des modèles ou des fragments de modèles lors de leur élaboration ou de leur maintenance. Un deuxième paragraphe est consacré à la sémantique axiomatique permettant de raisonner sur des modèles et des fragments de modèles. Nous terminerons le chapitre, en présentant des exemples simples. Ils nous permettront de justifier le report des techniques des traducteurs au niveau des modèles de programmes et surtout au niveau des modèles de conception réalisés à l'aide de diagramme d'activité. Il s'agit de voir l'esprit dans lequel les propriétés des langages peuvent être appliquées au cours d'un processus de développement de logiciels basé sur les modèles.

I Modélisation en UML/OCL des Propriétés Comportementales

Il s'agit en fait de construire un interprète exécutant les modèles de programmes du langage. Dans ce paragraphe nous décrivons préalablement un environnement d'exécution associant à chaque variable du programme la donnée qui sera utilisée par l'interprète lors de l'exécution du programme. Nous pourrons ensuite définir les fonctions mettant en œuvre le comportement des expressions et des instructions dans leur environnement d'exécution.

I.1 Environnement d'exécution d'un programme

Il s'agit, dans ce paragraphe, de définir l'environnement d'exécution, et les propriétés qui s'y rattachent.

I.1.a Définition de l'environnement

La figure suivante donne la grammaire (1) et le modèle UML (2) de l'environnement d'exécution, en rappelant que le triplet, associant à chaque variable, son type, et la valeur sémantique manipulée par l'interprète, est appelé la mémoire du programme :

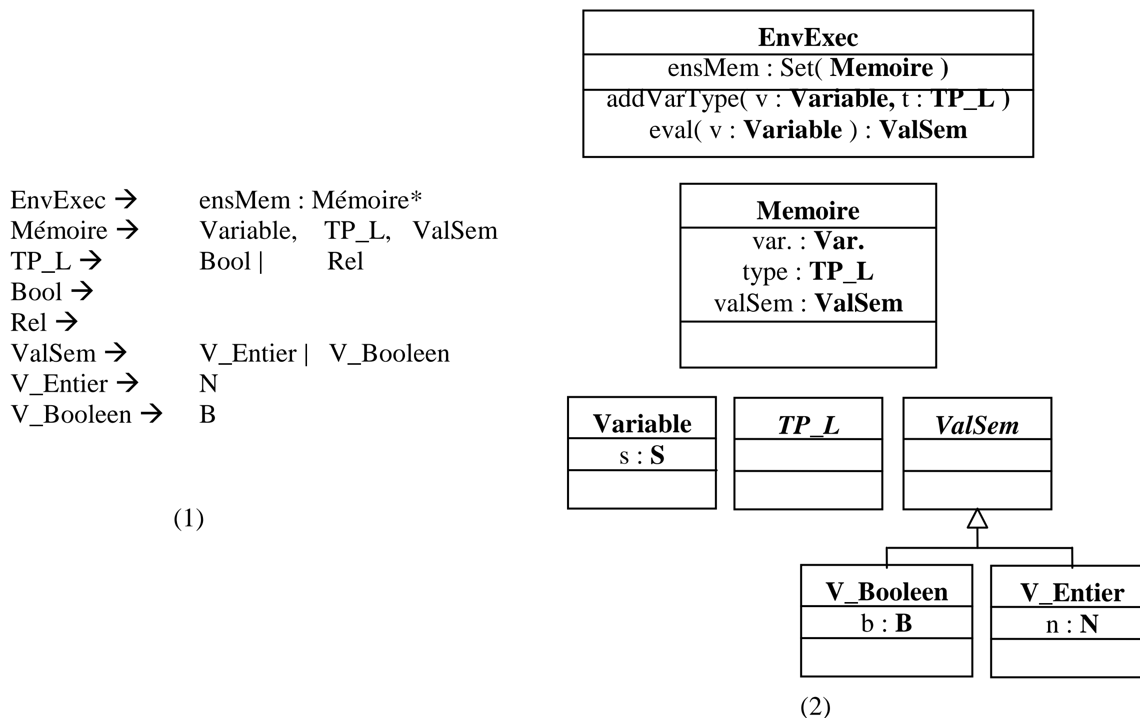


Figure 1 : ***Grammaire (1) et Méta-Modèle(2) définissant l'environnement d'exécution du langage « L »***

On peut ainsi définir la correspondance entre les types prédéfinis du langage L, les constantes du programme et les valeurs sémantiques manipulées par l'interprète, qui est donnée à la figure suivante :

Types prédéfinis du langage « L »	Type des valeurs sémantiques
Bool	V_Booleen(b : B)
Rel	V_Entier(n : N)

Constantes du langage « L »	Valeurs sémantiques
True	V_Booleen(true)
False	V_Booleen(false)
C_Rel	V_Entier(n)

Figure 2 : ***Correspondance entre les types prédéfinis et les constantes du langage L et le type des valeurs sémantiques***

Les opérations gérant l'environnement d'exécution d'un programme sont les suivantes :

-- les constructeurs, dont le code dépend du langage d'actions :

```
Mémoire( v : Variable, t : Type, vc : ValSem ) : Mémoire
EnvExec() : EnvExec
V_Booleen( b : Boolean ) : V_Booleen
V_Entier( n : Integer ) : V_Entier
```

-- La gestion de l'environnement :

```
EnvExec ::addVar( v : Variable, t : TP_L ) : EnvExec =
        EnvExec( self.memoire->including( Memoire( v, t, oclUndefined( ValSem ) ) )
```

```
EnvExec::valVar( v : Variable ) : ValSem =
                    self.ensMem->select( m | m.variable.s = v.s )->asSequence()->first().valSem.copy()
EnvExec::majVar( v : Variable, valSem ) : EnvExec =
                    EnvExec( self.ensMem->excluding( m | m.variable.s = v.v ) )->including( Memoire(
                                                                                    v, t, oclUndefined( ValSem ) ) )
```

Figure 3 ***: Constructeurs et gestion de l'environnement d'exécution***

-- Correspondance entre les opérations définies dans les expressions du langage et les opérations définies dans l'environnement d'exécution :

Les opérateurs du langage L	Operations sur les valeurs sémantiques
Null	V_Entier ::null()
Add	V_Entier ::add(n : V_Entier)
Sous	V_Entier ::sous(n : V_Entier)
…	…

Figure 4 ***: Correspondance entre les opérateurs des expressions du langage « L » et les opérations sur les valeurs sémantiques***

I.1.b Exécution d'un modèle de programme

L'exécution d'un modèle de programme s'apparente à une transformation de modèles, tel que le montre la figure suivante :

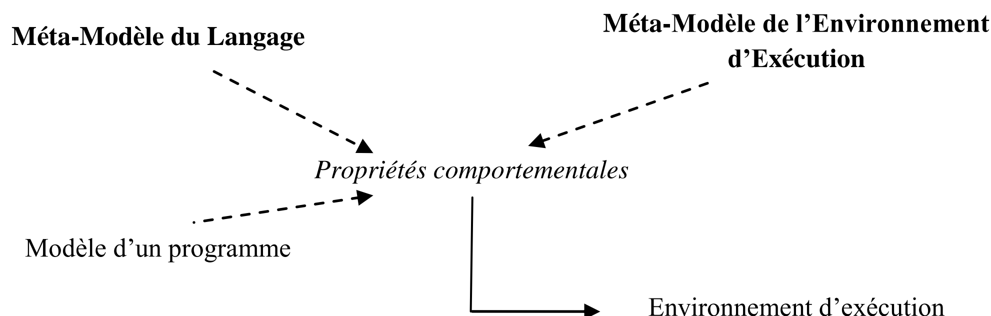


Figure 5 ***: Schéma montrant le modèle d'un programme et son environnement d'exécution***

Le modèle d'un programme, instance du Méta-Modèle du Langage, correspond au modèle source de la transformation. L'environnement d'exécution, utilisé par l'interprète pour calculer les valeurs associées aux différentes variables du programme durant une exécution du modèle de programme correspond au modèle cible.

Les propriétés comportementales agissent sur l'environnement d'exécution du modèle d'un programme. Comme pour les propriétés de typage de langage, elles doivent rester sans effet de bord sur le modèle du programme.

Les opérations qui seront décrites dans ce paragraphe s'appuient sur un formalisme qui s'apparente à celui de la sémantique dénotationnelle, puisqu'elles prennent en entrée un environnement d'exécution et qu'elles retournent, selon le cas, une donnée ou un environnement d'exécution.

Il est à noter aussi que l'environnement d'exécution ne référence aucun objet du modèle d'un programme.

I.2 Mise en œuvre des propriétés comportementales du langage « L »

I.2.a Mise en œuvre des propriétés comportementales des expressions

D'une manière générale, la spécification de la propriété comportementale d'une expression est la suivante :

```
Exp::eval( env : EnvExec ) :          ValSem
```

Figure 6 : ***Spécification de la propriété comportementale d'une expression***

La mise en œuvre des propriétés des différents types d'expressions du langage « L » est donnée par les fonctions suivantes:

```
Variable ::eval( env : EnvExec ) :  ValSem = env.valVar( self )

Const::eval( env : EnvExec ) :      ValSem

C_Bool ::eval( env : EnvExec ) :    ValSem

True::eval( env : EnvExec ) :       ValSem = self.V_Booleen( true )
False::eval( env : EnvExec ) :      ValSem = self.V_Booleen( false )

C_Rel::eval( env : EnvExec ) :      ValSem = self.V_Entier( self.n )

Null::eval( env : EnvExec ) :       ValSem = self.exp.eval( env ).oclAsType( V_Entier ).null()

ExpBin ::eval( env : EnvExec ) :    ValSem

Add ::eval( env : EnvExec ) :       ValSem = self.exp1.eval( env ).oclAsType( V_Entier ).add(
                                                 self.exp2.eval( env ).oclAsType( V_Entier )      )
…
Inf ::eval( env : EnvExec ) :       ValSem = self.exp1.eval( env ).oclAsType( V_Entier ).inf(
                                                 self.exp2.eval( env ).oclAsType( V_Entier )      )
```

Figure 7 : ***Mise en œuvre des propriétés comportementales des différents types d'expressions***

I.2.b Mise en œuvre des propriétés comportementales des instructions

D'une manière générale, la spécification de la propriété comportementale d'une instruction est la suivante :

```
Inst ::exec( env : EnvExec ) : EnvExec
```

Figure 8 : ***Spécification de la propriété comportementale d'une instruction***

Les fonctions OCL suivantes exécutent les différents types d'instructions du langage « L » :

```
Skip ::exec( env : EnvExec ) :      EnvExec = env
```

```
Seq ::exec( env : EnvExec ) :        EnvExec = self.suite.exec(  self.inst.exec( env ) )

Affect ::exec( env : EnvExec ) :     EnvExec = env.majVar(   self.variable.copy(),
                                                             self.exp.eval( env )  )

Test ::exec( env : EnvExec ) :       EnvExec = if  self.exp.eval( env ).oclAsType( V_Booleen ).b
                                               then              self.alors.exec( env )
                                               else    if   self.sinon <> oclUndefined( Inst )
                                                       then      self.sinon.exec( env )
                                                       else                      env
                                                       endif
                                               endif

Boucle ::exec( env : EnvExec ) :     EnvExec = if  self.exp.eval( env ).oclAsType( V_Boolean ).b
                                               then    self.exec( self.inst.exec( env )
                                               else                              env
                                               endif
```

Figure 9 : ***Mise en œuvre des propriétés comportementales des instructions du langage « L »***

I.2.c Spécification et mise en œuvre des propriétés comportementales de la partie déclarative

En parcourant la partie déclarative du modèle de programme, ces opérations ont pour effet de construire l'environnement d'exécution, associant pour chaque variable du programme la valeur qui sera calculée lors de l'évaluation de la partie instruction :

```
MM_L ::exec( env : EnvExec ) : EnvExec = EnvExec( oclEmpty( Set( Memoire ) ) )
Prog_L::exec( env : EnvExec ) : EnvExec = self.pi.exec( self.pd.exec( env ) )
PD ::exec( env : EnvExec ) : EnvExec =
        self.decvar->iterate( dc ;  env : EnvExec = oclEmpty( Sequence( Memoire ) |
                                      env.addVar( dc.variable, dc.type )
                          )
PI ::exec( env : EnvExec ) : EnvExec = self.pi.exec( env )
DecVar ::exec( env : EnvExec  ) : EnvExec = env.addVar( self.variable, self.type )
```

I.3 Réalisation sous la plate-forme USE et Kermeta

La plate-forme USE dispose d'une interface graphique qui, après avoir chargé un diagramme de classes, offre un certain nombre de commandes pour gérer dynamiquement un diagramme d'objets, instance du diagramme de classes.

Dans notre cas, le Méta-Modèle du langage est représenté par le diagramme de classes, et le modèle d'un programme est représenté par le diagramme d'objets. A chaque mise à jour du modèle d'un programme, le système vérifie toutes les propriétés (les invariants) complétant le diagramme de classes. Après avoir vérifié les propriétés de typage sur le modèle du programme, les Analystes/Concepteurs peuvent exécuter le modèle de manière à en vérifier son comportement. Cette interface permet d'exécuter un fragment de modèle pour étudier son comportement après une mise à jour conséquente de l'environnement d'exécution.

La plate-forme KerMéta permet d'implanter le Méta-Modèle du langage ainsi que les propriétés syntaxiques et comportementales. L'intérêt, par rapport à la plate-forme USE, est son intégration dans Eclipse, son interconnexion avec TopCased intégrant un éditeur UML, avec accès donc au Méta-Modèle UML standard, défini par l'OMG. De plus, de part sa programmation par aspect, le chargement dynamiques des méthodes des classes permet sur un même Méta-Modèle de programmer plusieurs applications indépendantes. En particulier, on peut charger tout d'abord les propriétés syntaxiques et de typage pour vérifier que les modèles des programmes sont bien construits. On peut ensuite charger les propriétés comportementales pour exécuter ou animer les modèles de programmes. Il n'est pas nécessaire de charger en même temps les deux applications.

II Modélisation en UML/OCL des Propriétés Axiomatiques

Pour pouvoir prendre en compte les triplets de Hoare, il s'agit de définir les environnements de données qui permettront de 'décorer' l'instruction d'un modèle de langage de programmation à l'aide d'assertions de type pré-condition et post-condition. Il s'agit ensuite de montrer comment on peut valider un modèle de triplet de Hoare.

Démontrer qu'une assertion est valide fait appel à des techniques de preuve de programmes pour lesquelles il existe des travaux de recherche décrivant le rapprochement entre des modèles UML et des Machines B [Snoo92][MyerE01] [Lale00] déjà appliqués dans des projets académiques et industriels, ainsi que des assistants de preuve spécialisés dans ce domaine, en l'occurrence des Ateliers B [At.B].

A partir d'un ensemble de règles de déduction correctement appliquées, ces prouveurs peuvent démontrer, par déduction logique, la validité d'une assertion. Toute assertion peut ne pas être valide, et une non réponse du prouveur n'implique pas forcément que l'assertion qu'il cherche à valider ne soit pas valide. Un atelier B dispose donc d'une interface qui peut aider le prouveur dans la gestion de son raisonnement. Face à ces difficultés, nous avons choisi de faire appel à un Atelier B pour chercher à démontrer qu'un fragment de modèle est correct par rapport à ses propriétés.

Il s'agit, cependant, d'éviter aux Analystes/Concepteurs impliqués dans le processus de développement IDM une surcharge supplémentaire en leur imposant un changement de formalismes et de notations des logiciels lors de l'appel à l'Atelier B. C'est pourquoi, nous montrons comment il est possible de réduire au strict minimum l'appel à un assistant de preuve, en effectuant au niveau UML/OCL un prétraitement sur les triplets de Hoare de manière à n'adresser à l'assistant de preuve que l'environnement de modélisation portant sur les assertions à valider. Dans une deuxième étape, nous montrons comment à l'aide de l'algorithme de la plus faible pré-condition appliqué à un triplet de Hoare, on peut rendre pratiquement transparent aux Analystes/Concepteurs le passage d'un environnement UML à un formalisme B.

II.1 Les environnements nécessaire à la prise en compte des propriétés axiomatiques

Nous décrivons dans ce paragraphe le principe général nécessaire à la prise en compte des propriétés axiomatiques d'un langage appliqué au niveau modélisation UML. Il s'agit tout d'abord de modéliser en UML les triplets de Hoare et les assertions qui leur sont associées. Le langage « L » est pris comme exemple de référence. Le calcul validant ces assertions fait ensuite l'objet d'un appel à un Atelier B.

II.1.a Grammaire du langage « L » intégrant la définition des assertions

La figure suivante reprend l'exemple d'extension de la grammaire du langage « L » prenant en compte la définition des assertions devant apparaitre dans un triplet de Hoare, tel que nous avions pu la définir au premier chapitre :

-- Exemple de fragment de la grammaire du langage L étendue à la prise en compte des propriétés axiomatiques

PI →	Inst				
Inst →	Skip |	Seq |	Affect |	Test |	Boucle
Seq →	**p : Assert ?,**	inst : Inst,	suite : Inst,	**q : Assert ?**	
Affect →	**p : Assert ?,**	Variable,	exp : Exp,	**q : Assert ?**	
Test →	**p : Assert ?,**	Exp,	alors : Inst,	sinon : Inst ?,	**q : Assert ?,**
Boucle →	**p : Assert ?,**	Exp,	Inst,	**q : Assert ?**	
				invariant : Assert ?,	
				variant : Assert ?	
…					
Assert →	Not |	AssertBin |	Exp		
Not →	Assert				
AssertBin →	Imp |	And |	Eg		
Imp →	assert1 : Assert,	assert2 : Assert			
Eg →	assert1 : Assert,	assert2 : Assert			
…					

Figure 10 : ***Exemple d'extension de la grammaire du langage « L » prenant en compte la définition des assertions du triplet de Hoare, et leurs opérations***

Cependant, la règle de la construction syntaxique du symbole Test a été modifiée de manière à pouvoir prendre en compte les invariants et les variants de boucle.

Cet exemple a été décrit au cours du premier chapitre.

On supposera, en particulier, que ces assertions ne contiennent que des variables apparaissant dans la partie déclarative du modèle de programme correspondant, et que pour être en accord avec la définition du triplet de Hoare, le quantificateur pour tout, est implicite sur chacune des variables de la pré-condition. La figure suivante montre le Méta-Modèle de la grammaire du langage « L », étendu à la définition à la pré-condition et à la post-condition du triplet de Hoare :

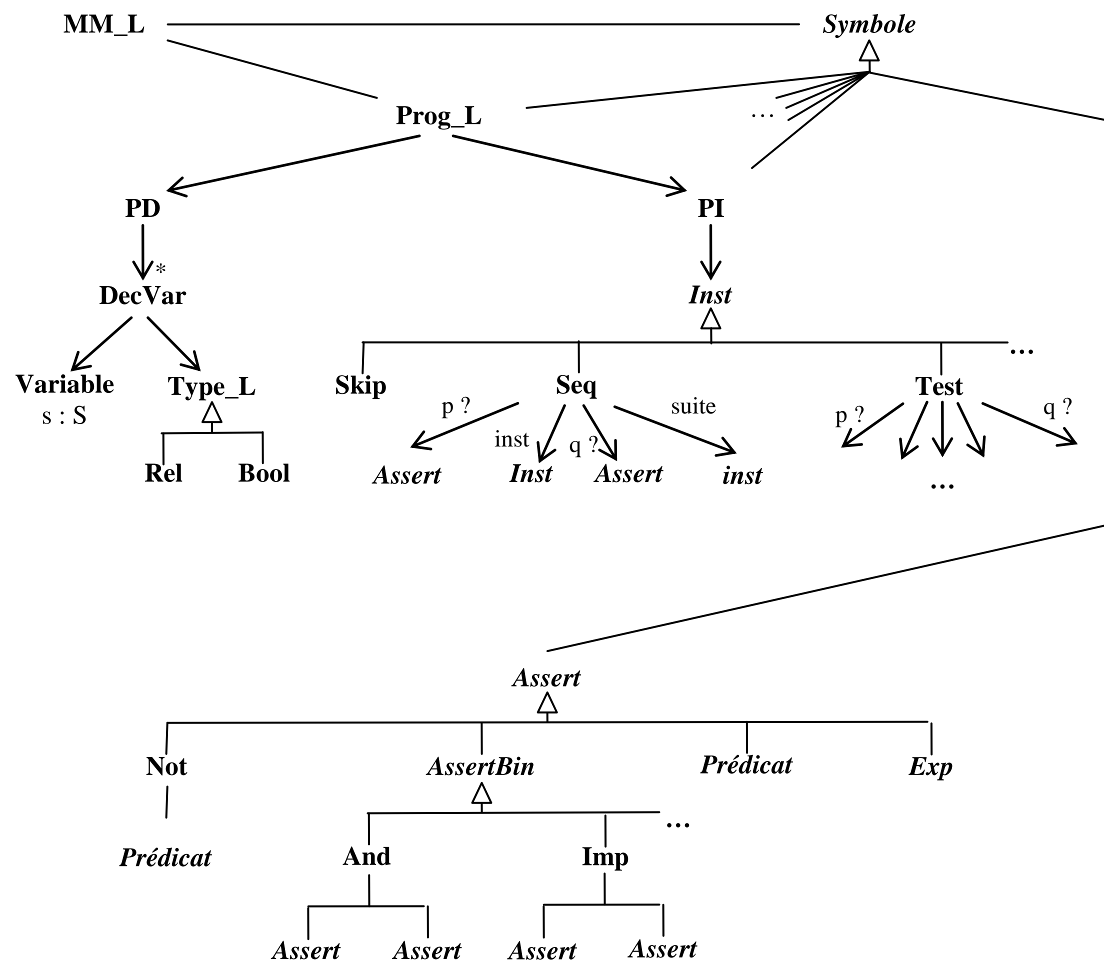


Figure 11 : ***Méta-Modèle du langage « L » prenant en compte la modélisation simplifiée des assertions***

Cette figure montre que les constructions syntaxiques des expressions et des instructions du langage « L » ne sont pas modifiées.

Il est donc nécessaire de rajouter les fonctions modélisant en UML/OCL prenant en compte les propriétés de typage et les propriétés comportementales des assertions et des triplets de Hoare. Le tableau suivant en montre la correspondance avec les propriétés définies pour le langage L :

Langage L	Assertions et Triplets de Hoare
Exp ::type() : TP_L	*Assert* ::typeAssert() : TP_L
Inst ::typage() : Boolean	*Inst* ::typageHoare() : Boolean
Exp ::eval(env : EnvExec) : ValSem	*Assert* ::evalAssert(env : EnvExec) : ValSem
Inst ::exec(env : EnvExec) : EnvExec	

II.1.b Propriétés de typage des assertions et des triplets de Hoare

Les propriétés de typage des assertions et des triplets de Hoare sont donc :

Soit, en OCL :

```
Assert::typeAssert() : TP_L
Not ::typeAssert() : TP_L =
        if self.assert.typeAssert().oclIsTypeOf( Bool ) and self.assert.typeAssert() <> oclUndefined( Bool )
        then              Bool()
        else oclUndefined( Bool   )
        endif
…
Inst ::typageHoare() : Boolean
Seq ::typageHoare() : Boolean =   self.p.typeAssert().oclIsTypeOf( Bool )     and
                                  ( self.inst.typage()                        and
                                   ( self.suite.typage()                      and
                                     self.q.typeAssert().oclIsTypeOf( Bool )        ) )
…
```

Figure 12 : ***Propriétés de typage des assertions***

Comme pour les expressions du langage « L », il faut définir les propriétés comportementales des assertions permettant de décrire les pré-conditions et les post-conditions du triplet de Hoare. Les propriétés se définissent de la même manière que pour les expressions du langage « L ». Soit, en OCL :

```
Assert ::evalAssert( env : EnvExec ) : ValSem
```

Et, selon les opérateurs de définissant les assertions :

```
Imp ::evalAssert( env : EnvExec ) : ValSem =        self.assert1.evalAssert( env ).imp
                                                  ( self.assert2.evalAssert( env )      )
…
```

Figure 13 : ***Propriétés comportementales des assertions et des triplets de Hoare***

Une interprétation comportementale du triplet de Hoare : { P } Inst { Q } pourrait donc être définie de la manière suivante, pour un environnement d'exécution passé en paramètre :

Inst ::evalTdeH(env : EnvExec) : ValSem =

```
self.p.evalAssert(        env    ).oclAsType(V_Booleen).imp(
self.q.evalAssert( self.exec( env ) ).oclAsType( V_Booleen )
                                                             )
```

Dans la mesure où l'opérateur d'implication imp est défini au niveau de l'environnement d'exécution.

Pour vérifier que le triplet de Hoare { P } Inst { Q } est valide, il faut démontrer que :

Inst ::evalTdeH(env) est vrai pour tout état de env ∈ EnvExec

Cette propriété inductive ne peut être vérifiée que par un assistant de preuve comme Coq prenant en compte le raisonnement par récurrence tel que nous le montrons dans l'exemple suivant.

II.1.c Exemple d'un fragment de modèle de programme du langage « L »

On suppose, par exemple que l'on a le programme suivant :

```
x : Rel
x := -2
x := x + 10
```

On suppose que l'on voudrait vérifier que l'instruction d'affectation est correcte par rapport aux propriétés suivantes :

p = { x > 0 }
q = { x > 8 }

Il s'agit donc de vérifier si le triplet de Hoare suivant :

{ x > 0 } x := x + 10 { x > 8 }

est valide.

La figure suivante montre le modèle du programme, puisque l'on est censé étudier des propriétés sur des modèles de codes :

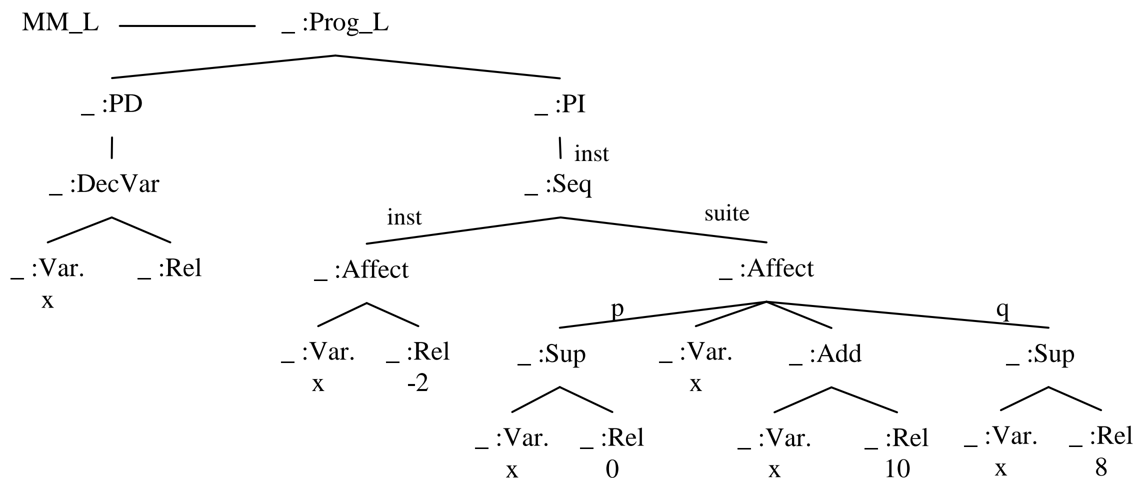


Figure 14 : ***Exemple d'un modèle de programme du langage « L » contenant un triplet de Hoare***

En appliquant la définition d'un triplet de Hoare, on en déduit que le triplet suivant est valide :

$$\{ x > 0 \}\ x := x + 10\ \{ x > 8 \}$$

si l'assertion suivante est valide :

$$\forall x \in Rel,\ eval(\ EnvExec,\ x > 0\ ) = True \Longrightarrow eval(\ exec(\ EnvExec,\ x := x + 10\ ),\ x > 8\ ) = True \text{ valide ?}$$

II.1.d Machine B et raffinement prenant en compte un triplet de Hoare

Pour qu'un Atelier B puisse démontrer qu'un triplet de Hoare est valide, une solution consiste à modéliser en B le Méta-Modèle de la grammaire et ses propriétés, et le modèle de programme contenant le triplet de Hoare. La figure suivante montre un schéma général montrant la correspondance entre les éléments de modélisation en UML du langage « L » et ceux de B :

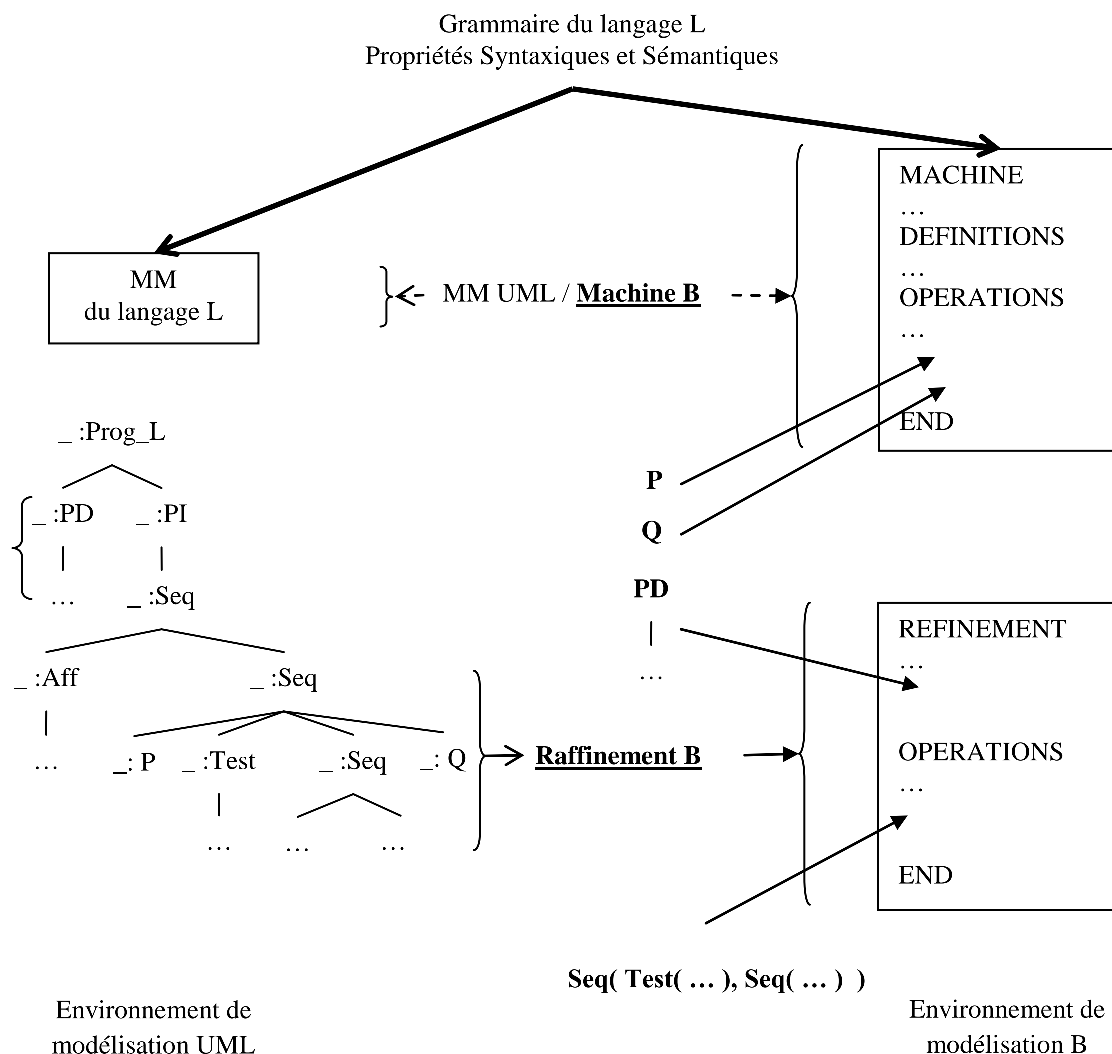


Figure 15 : ***Atelier B prenant en charge la validation des obligations de preuve issues des triplets de Hoare***

Ce schéma montre que le Méta-Modèle du langage, la pré-condition et la post-condition du triplet de Hoare constituent les éléments de modélisation d'une Machine B, après avoir été traduit, bien sûr, en B. Après traduction en B, l'instruction du triplet de Hoare constitue l'élément essentiel du raffinement de la machine B.

On retrouve la même architecture que nous avions définie pour la modélisation des propriétés comportementales d'un langage. On peut donc assimiler la Machine B et son raffinement à l'environnement d'exécution. De la même manière que le Méta-Modèle du langage est déduit de la grammaire du langage, la Machine B peut se déduire de la grammaire du langage. Du Modèle du programme on peut alors en déduire le raffinement de la Machine B. L'environnement de modélisation B étant défini, l'ensemble peut donc être transmis à l'Atelier B qui se chargera de voir si le triplet de Hoare est valide.

La figure suivante montre la spécification et le raffinement de la Machine B correspondant à l'exemple précédent, ce qui montre très schématiquement les correspondances entre les concepts UML et B :

```
MACHINE
   MachineAffect
SETS
   Expression;    Instruction
ABSTRACT_VARIABLES
   Exp, OB , Add , Mult, ConstE,    exp1 , exp2 , ctNat, VARI ,    Inst, Affect, pgAff, pdAff

DEFINITIONS
   inv == (
   Inst <: Instruction &    Affect <: Inst &       Exp <: Expression  &    OB <: Exp &
   Add <: Expression &    Mult <: Expression &    OB <: Expression &
   ConstE <: Expression &    VARI <: Expression &
   Add <: OB &    Mult <: OB &
   ConstE <: Exp &    VARI <: Exp &
   Add /\ Mult = {} &    Add \/ Mult = OB &    OB /\ ConstE = {} &
   OB /\ VARI = {} &    ConstE /\ VARI = {} &    OB \/ ConstE \/ VARI = Exp &
   exp1 : Expression +-> Exp &    exp2 : Expression +-> Exp &
   exp1 : OB --> Exp &    exp2 : OB --> Exp &
   ctNat : ConstE --> NAT &

   pgAff : Instruction +-> VARI &    pdAff : Instruction +-> Exp &
   pgAff : Affect --> VARI &    pdAff : Affect +-> Exp
);

          isEval(eval) == (
   eval: (VARI --> NAT) * Exp --> NAT
    &   !(env, add).(       env: VARI --> NAT
                  &    add:Add => eval(env,add) = eval(env, exp1(add)) + eval(env, exp2(add))
               )
   &   !(env, mul).(       env: VARI --> NAT
                 &    mul:Mult => eval(env,mul) = eval(env, exp1(mul)) * eval(env, exp2(mul))
              )
   &   !(env,var).(       env:VARI --> NAT
                 &    var:VARI => eval(env,var) = env(var)
              )
   &   !(env,cte).(       env:VARI --> NAT
                 &    cte:ConstE => eval(env,cte) = ctNat(cte)
              )
          &   !(env,aff).(       env:VARI --> NAT
                 &   aff:Affect => eval(env,pgAff(aff)) = eval(env, pdAff(aff))
              )
             )
INVARIANT
   inv


INITIALISATION
   Inst :={} ||    Affect := {} ||    pdAff :={} ||    pgAff := {} ||

   Exp :={} ||     OB := {} ||    Add :={} ||    Mult := {} ||    ConstE := {} ||
   exp1 :={} ||    exp2 := {} ||    ctNat := {} ||    VARI := {}

OPERATIONS

    opAff(xVar) =
          PRE xVar : Exp
             & xVar: VARI
      & !(env,eval).(env : VARI-->NAT & isEval(eval) => eval(env,xVar) >0)
```

```
  THEN
     Exp, OB, Add, ConstE,ctNat, exp1, exp2, pgAff, pdAff, Inst, Affect : (  inv
       &  !(env,eval). (env: VARI-->NAT & isEval(eval) => eval(env,xVar) > 8)

       )
    END
  END
```

Le raffinement de cette machine, sert principalement à décrire le contenue de l'opération opAff et en basant sur cette machine et son raffinement, nous pouvons ensuite procéder à la phase de preuves, avec l'aide de l'outil Atelier B

```
REFINEMENT
  MachineAffect_r
REFINES
  MachineAffect

ABSTRACT_VARIABLES
  Exp ,
  OB ,
  Add ,
  Mult ,
  ConstE ,
  exp1 ,
  exp2 ,
  ctNat ,
  VARI ,
  Inst ,
  Affect ,
  pgAff ,
  pdAff

INITIALISATION
  Inst := {} ||    Affect := {} ||    pdAff := {} ||    pgAff := {} ||
  Exp := {} ||    OB := {} ||    Add := {} ||    Mult := {} ||    ConstE := {} ||
  exp1 := {} ||    exp2 := {} ||    ctNat := {} ||    VARI := {}

OPERATIONS
  opAff ( xVar ) =
  ANY
     newAff, newCte , newAdd
  WHERE
     newAdd : Expression - Exp &       newAff : Instruction - Inst &       newCte : Expression - Exp &
     newCte /= newAdd
  THEN
     Affect := Affect \/ {newAff};
     Inst := Inst \/ {newAff};
     Add := Add \/ {newAdd};
     OB := OB \/ {newAdd};
     ConstE := ConstE \/ {newCte};
     Exp := Exp \/ {newCte, newAdd};
     ctNat, exp1, exp2, pgAff, pdAff := ctNat <+ {newCte |-> 10 }, exp1<+{newAdd |-> xVar},
exp2<+{newAdd |-> newCte}, pgAff <+ { newAff |-> xVar} , pdAff <+ { newAff |-> newAdd}

  END

END
```

Cet interfaçage entre l'environnement de modélisation UML et B est général et systématique. Il existe des travaux de recherche sur ce passage de UML à B [Snoo92][MyerE01][Idani][Lale00]. Cependant, c'est une approche relativement lourde puisque l'on établit une correspondance entre chaque élément de modélisation UML et son équivalent en B.

Au premier chapitre, il a été rappelé que pour démontrer qu'un triplet de Hoare est valide, il faut démontrer que l'assertion qui lui est associée est valide. Il suffit donc, au niveau UML, de déduire cette assertion du triplet de Hoare que l'on cherche à démontrer et transmettre à l'Atelier B uniquement cette assertion, ainsi que les définitions des éléments de modélisation qui peuvent s'y rattacher.

C'est donc cette solution que nous proposons au paragraphe suivant. Cependant, pour déterminer l'assertion associée au triplet de Hoare, nous appliquons un algorithme appelé Plus Faible Pré-condition (pfp) qui, combiné avec la règle de conséquence, peut faciliter la démonstration de la validité du triplet de Hoare.

Cette solution fait appel à des propriétés basées sur la logique. Elle évite donc un changement de notation en transférant à un Atelier B tout un environnement de modélisation UML. Nous verrons, enfin, comment simplifier encore plus les appels à l'Atelier B, lorsque les types des variables et des constantes entrant dans la composition de l'assertion peuvent correspondre directement aux types prédéfinis du langage B.

II.2 Algorithme de la Plus Faible Pré-condition d'un triplet de Hoare

Après avoir rappelé brièvement le contexte d'un triplet de Hoare, nous introduisons sur un exemple le concept de plus faible pré-condition d'un triplet de Hoare et d'obligations de preuve qui sont les éléments essentiels de l'algorithme prenant en compte la sémantique opérationnelle des langages de programmation. Nous décrirons ensuite l'algorithme calculant la Plus Faible Pré-condition d'un triplet de Hoare et des obligations de preuve pour chaque type d'instruction du langage « L », l'affectation, la séquence, le test et la boucle, et les obligations de preuve correspondante.

Il s'agit donc d'effectuer, au niveau UML, un pré-traitement sur le triplet de Hoare à étudier, avant de faire appel à l'assistant de preuve. Nous nous inspirons de travaux décrits dans [Moh][MyerB92], mais que nous adaptons à un environnement de modélisation UML/OCL pour garder une certaine continuité de notation au sein de processus IDM.

II.2.a Algorithme de la Plus Faible Pré-condition (pfp) et les obligations de preuve

L'algorithme de la Plus Faible Pré-condition (pfp) vient de deux remarques, rappelées au chapitre I, portant sur les règles d'inférence, sur la séquence d'instructions et sur l'instruction d'affectation.

La règle d'inférence spécifiant la validité d'un triplet de Hoare pour une séquence d'instructions est la suivante:

$$\frac{\{\,\mathrm{P}\,\}\, Inst_i \,\{\,\mathrm{P1}\,\},\quad \{\,\mathrm{P1}\,\}\, Inst_j \,\{\,\mathrm{Q}\,\}}{\{\,\mathrm{P}\,\}\,\mathrm{Seq}(\, Inst_i,\ Inst_j\,)\,\{\,\mathrm{Q}\,\}}$$

Pour chercher à valider le triplet de Hoare { P } Seq($Inst_i$, $Inst_j$) { Q }, il s'agit de calculer l'assertion P1 validant les triplets de Hoare { P } $Inst_i$ { P1 } et { P1} $Inst_j$ { Q }. En fait, il peut y avoir un grand nombre d'assertions P1. Le problème consiste donc à voir comment déterminer une assertion P1 validant les deux triplets de Hoare.

Si on reprend le triplet de Hoare donné en exemple au paragraphe précédent :

{ x > 0 } x := x + 10 { x > 8 }

On peut remarquer que pour toute valeur de x > -2, avant exécution de l'instruction d'affectation, on aura près exécution de l'instruction x > -2 + 10 = 8. On peut donc en déduire que pour toute pré-condition P tel que x > -2, par exemple, x > -1, x > 0, x > 1, … , alors, le triplet de Hoare P suivant :

{ x > -2 } x := x + 10 { x > 8 } est valide.

Cette valeur -2, peut être calculée à partir de la post-condition x > 8 et de l'instruction d'affectation x := x + 10, en substituant, dans la post-condition, x par l'expression de l'instruction d'affectation. Soit : (x + 10) > 8 donne l'assertion x > -10 +8 soit : x > -2. Cette opération, décrite au premier chapitre, est appelée opération de substitution de la variable de la post-condition par l'expression de l'instruction d'affectation,

Une telle assertion x > -2 est appelée la Plus Faible Pré-condition (pfp) de

x := x + 10 { x > 8 }

et que l'on écrit de la manière suivante :

pfp(x := x + 10 ; x < 8) ≡ x > -2

Cette pfp rend le triplet de Hoare { x > -2 } x := x + 10 { x > 8 } valide.

Donc, une solution systématique pour voir si le triplet de Hoare { x > 0 } x := x + 10 { x > 8 } est valide, est de calculer pfp(x := x + 10 ; x > 8) qui donne x > -2, et de voir si la pré-condition implique la pfp, en l'occurrence ici : a-t-on, quelque soit x

$$x > 0 \Longrightarrow x > -2 \text{ valide}$$

En appliquant la règle d'inférence de conséquence :

$$\frac{x > 0 \Longrightarrow x > -2 \;;\; \{\, x > -2 \,\}\ x := x + 10\ \{\, x > 8 \,\}}{\{\, x > 0 \,\} \qquad\qquad x := x + 10\ \{\, x > 8 \,\}}$$

On en déduit que le triplet suivant :

{ x > 0 } x := x + 10 { x > 8 } est valide.

On peut remarquer, par exemple, que les triplets de Hoare suivants sont valides :

{ x > -2 } x := x + 10 { x > 8 }
{ x > 1 } x := x + 10 { x > 8 }
{ x > 0 } x := x + 10 { x > 8 }

…

Alors que $\{ x > -3 \}\ x := x + 10\ \{ x > 8 \}$ n'est pas valide.

D'une manière générale :

{ P } Inst { Q } est valide si $P \Rightarrow$ pfp(Inst, Q) est valide.

$P \Rightarrow$ pfp(Inst, Q) est appelée l'obligation de preuve du triplet { P } Inst { Q }.

On en déduit que pour démontrer que { P } Seq(Inst1, Inst2) { Q } est valide, il faut démontrer l'obligation de preuve suivante basée sur le calcul de la pfp de Inst2 et de Q, qui devient la post-condition du triplet de Hoare { P } Inst1 { pfp(Inst2, Q) }, soit :

$P \Rightarrow$ pfp(Inst1, pfp(Inst2, Q))

II.2.b Règles de calcul de la plus faible pré-condition pour les instructions du langage « L »

Les règles de calcul de la Plus Faible Pré-condition pour les instructions du langage « L » sont donc les suivantes :

-- pfp(Inst , Assertion) : Assertion

-- pfp(Skip , Q) ≡ Q

-- pfp(x := Exp , Q) ≡ subst(Q, x := Exp)

-- pfp(*Inst1* ; *Inst2*, Q) ≡ pfp(*Inst1*, Pfp(*Inst2*, Q)

-- pfp(si *Exp* alors *Inst*, Q) ≡ (*Exp* ⟹ pfp(*Inst*, Q)) ∧ (¬*Exp* ⟹ Q)

-- pfp(si *Exp* alors *Inst1* sinon *Inst2*) ≡ (*Exp* ⟹ Pfp(*Inst1*, Q) ∧ (¬*Exp* ⟹ pfp(*Inst2*, Q))

-- pfp(tant que *Exp* faire *Inst* { Invariant, Variant }) ≡ Invariant

Figure 16 : ***Règles de calcul de la plus faible pré-condition pour les instructions du langage « L »***

II.2.c Obligations de preuve pour les instructions du langage « L »

En ce qui concerne l'instruction de boucle, il n'y a pas de calcul spécifique pour en déterminer la pfp. On prend généralement comme pfp, l'invariant de boucle. Par contre, il ne faut pas oublier les obligations de preuve qui assurent le lien avec les autres assertions à démontrer dans un triplet de Hoare constitué d'une séquence d'instructions. Ces obligations de preuve sont les suivantes :

$$(Exp \wedge Inv \wedge Var = z) \Rightarrow Pfp(Inst, Inv \wedge Var < z)$$

$$Inv \Rightarrow Var \geq 0$$

$$(\neg Exp \wedge Inv) \Rightarrow Q$$

Le tableau suivant donne un récapitulatif des obligations de preuve pour les instructions du langage « L » :

Triplets de Hoare	Obligations de preuve
{ P } *Inst* { Q }	P ⟹ pfp(Inst , Q)
{ P } Skip { P }	P ⟹ P
{ P } *Inst1* ; *Inst2* { Q }	P ⟹ pfp(*Inst1*, pfp(*Inst2*, Q))
{ P } si *Exp* alors *Inst* { Q })	(*Exp* ⟹ pfp(*Inst* , Q)) ˄ (¬*Exp* ⟹ Q)
{ P } si *Exp* alors *Inst1* sinon *Inst2* { Q }	(*Exp* ⟹ pfp(*Inst1*, Q)) ˄ (¬*Exp* ⟹ pfp(*Inst2*, Q))
{ P } tant que *Exp* faire *Inst* , Invariant, Variant { Q }	P ⟹ Invariant (*Exp* ˄ Inv ˄ Var = z) ⟹ Pfp(*Inst*, Inv ˄ Var < z) Inv ⟹ Var ≥ 0 (¬*Exp* ˄ Inv) ⟹ Q

Figure 17 : ***Tableau récapitulatif des obligations de preuve pour les instructions du langage « L »***

II.3 Modélisation en UML/OCL des propriétés axiomatiques

La mise en œuvre en UML/OCL des propriétés axiomatiques du langage « L » se déduisent des propriétés décrites au paragraphe précédent.

Il s'agit tout d'abord de vérifier que l'invariant et le variant de boucle ont été définis :

```
Boucle ::ok() : Boolean =
        ( self.exp.type().oclIsTypeOf( Bool )       and self.inst.ok() )                        and
        ( self.invariant.type().oclIsTypeOf( Bool ) and self.variant.type().oclIsTypeOf( Rel ) )
```

II.3.a Fonctions OCL de substitution des instructions d'affectation du langage « L »

Les opérations de substitution relatives aux expressions sont les suivantes :

```
Assert ::substitution( aff : Affect ) : Assert = copy()
Not ::substitution( aff : Affect ) : Assert = Not(        self.ass.substitution( aff ) )
AssertBin::substitution( aff : Affect ) : Assert =        self
Eq ::substitution( aff : Affect ) : Assert = Eq(          self.ass1.substitution( aff ), self.ass2.substitution( aff ) )
Imp ::substitution( aff : Affect ) : Assert = Imp(        self.ass1.substitution( aff ), self.ass2.substitution( aff ) )
And ::substitution( aff : Affect ) : Asser = And(         self.ass1.substitution( aff ), self.ass2.substitution( aff ) )
Or::substitution( aff : Affect ) : Assert = Or(           self.ass1.substitution( aff ), self.ass2.substitution( aff ) )

Exp::substitution( aff : Affect ) : Assert = self
Variable::substitution( aff : Affect ) : Assertion =     let r1 : Exp = aff.exp.copy().oclAsType( Exp )
                                                   in let r2 : Exp = self.Variable( self.s )
                                                   in if self.s = aff.variable.s
                                                      then r1
                                                      else r2
                                                      endif
Const::substitution( aff : Affect ) : Assert =  self
C_Bool::substitution( aff : Affect ) : Assert =          self
True::substitution( aff : Affect ) : Assert =   True()
False::substitution( aff : Affect ) : Assert =  False()
C_Rel:: substitution( aff : Affect ) : Assert =          C_Rel( self.n )
Null::substitution( aff : Affect ) : Assert =   Null(    self.exp.substitution( aff ).oclAsType( Exp ) )
ExpBin::substitution( aff : Affect ) : Assert =          self
Add::substitution( aff : Affect ) : Assert =    Add(     self.exp1.substitution( aff ).oclAsType( Exp ),
```

```
                                              self.exp2.substitution( aff ).oclAsType( Exp ) )
…
```

Figure 18 : ***Fonction de substitution OCL/LA, dans une expression, d'une variable par une expression***

II.3.b Fonctions OCL calculant les pfp des instructions du langages « L »

Les opérations calculant les Plus Faible Pré-conditions (pfp) des instructions du langage « L » sont les suivantes :

```
Inst ::pfp( q : Assert ) : Assert = q
Skip ::pfp( q : Assert ) : Assert = q.copy()
Seq ::pfp( q : Assert ) : Assert = self.inst.pfp( self.suite.pfp( q ) )
Affect ::pfp( q : Assert ) : Assert = q.copy().substitution( self )
Test ::pfp( q : Assert ) : Assert = let al : Assert = And( Imp( self.exp.copy(),
                                                              self.alors.pfp( q )
                                                              ),
                                                         Imp( self.Not( self.exp.copy() ),
                                                             q
                                                             )
                                                        )
                                   in let al_si : Assert = And( al,
                                                                 Imp( Not( self.exp.copy() ),
                                                                        self.sinon.pfp( q )
                                                                     )
                                                               )
                                   in if self.sinon.oclIsKindOf( Inst ) and self.sinon <> oclUndefined( Inst )
                                      then al_si
                                      else al
                                      endif
Boucle::pfp( q : Assert ) : Assert = self.invariant.copy()
```

Figure 19 : ***Fonction OCL/LA retournant le pfp d'une instruction et d'une post-condition***

II.3.c Fonction OCL retournant les obligations de preuve des instructions du langage « L »

Les opérations retournant les obligations de preuve se définissent directement à partir de calcul de la pfp.

Remarque : Si les variables et les constantes apparaissant dans l'assertion sont de type prédéfini du langage « L » et si l'on peut établir une correspondance entre les opérateurs des types prédéfinis du langage « L » et ceux de B, alors l'assertion correspondante au triplet de Hoare pourra être traduite directement en B. En effet, si on prend en exemple le triplet de Hoare suivant :

$$\{ x > -2 \} \; x := x + 10 \; \{ x > 8 \}$$

où l'on suppose que la variable x et les constantes -2, 8 et 10 sont du type Rel, type prédéfini du langage « L ».

On peut alors faire correspondre le type prédéfini Rel du langage « L » au type prédéfini NAT du langage B. Les opérateurs se correspondant en « L » et en B ont donc les mêmes sémantiques comportementales.

Dans ces conditions pfp(x := x + 10 ; x > 8) retourne x > -2.

L'obligation de preuve est donc la suivante : $x > 0 \Rightarrow x > -2$

En B, cette assertion s'écrit : !xx.(xx : NAT => (xx > 0 => xx > -2))

Ce que B pourra démontrer, bien sûr.

II.3.d Synthèse des techniques présentées dans ce paragraphe

Avant de montrer sur quelques exemples tout l'intérêt que l'on peut porter sur la modélisation des triplets de Hoare à partir de l'opération de substitution et de son utilisation pour faciliter le calcul de la preuve, nous en rappelons les idées générales.

- Un triplet de Hoare est de la forme { P } *Inst* { Q }, où *Inst* est une séquence d'instruction d'un programme et où P et Q sont des assertions de type pré-conditions (P) et post-condition (Q).

- Définition et sémantique des triplets de Hoare : L'idée est de démontrer que la séquence d'instructions *Inst* d'un programme est correcte par rapport aux assertions P et Q. Un triplet de Hoare est valide si, à chaque fois que l'environnement d'exécution se trouve dans un état où P est vrai, alors l'exécution de *Inst* amène l'environnement d'exécution dans un état où Q est vrai.

- Sémantique du triplet Hoare :

$$\{ P \}\ \mathit{Inst}\ \{ Q \} \equiv \forall\ EnvExec_i \in EnvExec\ /\ eval(EnvExec_i, P) = true \Rightarrow eval(\ exec(EnvExec_i, \mathit{Inst}), Q) = true$$

ou, d'une manière plus implicite :

$$\{ P \}\ \mathit{Inst}\ \{ Q \} \equiv eval(EnvExec, P) = true \Rightarrow eval(\ exec(EnvExec, \mathit{Inst}), Q) = true$$

<u>Modélisation des triplets de Hoare en UML/OCL (Application au langage « L »)</u> :

- Notation :
  P, Q : Modèle d'expressions en OCL/LA du MM du langage « L » complété éventuellement par des opérateurs logiques
  Inst : Modèles d'instructions du MM du langage « L »

- Assertion déduite du triplet de Hoare :

Inst ::evalAssert(env) : Assert =

```
self.p.eval( env ).oclAsType( V_Booleen ).eq( true ).imp(

        self.q.eval( self.inst.exec( env ) ).oclAsType( V_Booleen ).eq( true )
```

)

- Démontrer que le triplet de Hoare est valide se fait donc en transformant l'assertion en B, puis en transférant le contrôle à un Atelier B qui se charge de prouver que le triplet de Hoare est valide, avec l'aide des Concepteurs, si nécessaire.

  On peut cependant faire remarquer que l'on peut chercher à évaluer l'expression au coup par coup, en restant au niveau modélisation UML/OCL, après avoir initialisé l'environnement d'exécution en conséquence. La preuve n'est pas apportée au sens de la validité des triplets de Hoare, mais l'assertion peut être vérifiée pour certaines valeurs. C'est une manière de définir des modèles de programmes tests.
  Cette interprétation de la propriété comportementale du triplet de Hoare est faite indépendamment du calcul de la Plus Faible Pré-condition. En fait, il n'était pas nécessaire au chapitre I de rappeler les règles d'inférences des propriétés axiomatiques du langage « L » dès le début du paragraphe sur les propriétés axiomatiques. Il suffisait de donner la sémantique du triplet de Hoare, et de reporter la présentation de ces propriétés lors du calcul des pfp en fonction des différents types des instructions du langage « L ».

<u>Calcul de la Plus Faible Pré-condition, triplet de Hoare</u> :

- Principe : Le calcul de la Plus Faible Pré-condition est une manière de déterminer une assertion P', appelée Plus Faible Pré-condition (pfp), à partir de l'instruction *Inst* et de la post-condition Q rendant le triplet de Hoare { P' } *Inst* { Q } valide. Prouver que le triplet de Hoare { P } *Inst* { Q }, revient donc à démontrer que P ⟹ pfp(*Inst*, Q) est valide, de part l'application de la règle de conséquence des propriétés axiomatiques :

  $$\{ P \} \mathit{Inst} \{ Q \} \equiv P \Longrightarrow \text{pfp}(\mathit{Inst}, Q)$$

  Le calcul de pfp(*Inst*, Q) se décline différent selon le type de l'instruction (opération de substitution pour l'instruction d'affection, l'invariant pour la boucle, …).

- Assertion déduite du triplet de Hoare, intégrant le calcul de la pfp :

  Inst ::evalAssert(env) : Assert =

  self.p.eval(env).oclAsType(V_Booleen).eq(true).imp(

  self.inst.pfp(self.q).eval(env).oclAsType(V_Booleen).eq(true)

)

- Démontrer la validité du triplet de Hoare est basé sur le même principe que précédemment. Cependant, l'intérêt, ici, réside dans le fait que l'on ramène le calcul de la preuve au niveau du calcul de la pré-condition, sans qu'il soit nécessaire d'exécuter l'instruction.

<u>Cas particulier</u> :

- Si l'assertion ne contient que des variables dont les types prédéfinis sont comparables aux types prédéfinis de B, alors transformer l'assertion en B est relativement facile. La sémantique comportementale des opérateurs étant la même : la transformation en B se limite à une transformation de modèle d'arbre, et la vérification de la validité d'un triplet de Hoare se limite à une évaluation classique d'une expression.

On rejoint, en fait, les travaux de recherche qui ont été réalisés autour du langage Why[1] développé dans le cadre de la preuve de programmes impératifs. Ce langage qui prend en compte l'appel de sous-programmes et les exceptions, intègre des structures de données complexes (tableaux, …). Par rapport à ce que nous avons présenté, on retrouve les expressions avec effets de bord. Cependant, l'approche présentée, dans cette étude, en UML/OCL peut être appliquée tout au long d'un processus IDM permettant aux Concepteurs de s'assurer que les modèles ont bien les propriétés requises, et que les transformations de modèles sont correctes.

Le langage Alloy a été présenté dans le chapitre II. Il peut aider les Concepteurs qui manipulent des propriétés définissant de la sémantique dans des Méta-Modèles. En effet, ce langage permet, en particulier, de chercher dans un méta-modèle une instance répondant à un prédicat, ou un contre-exemple.

III Quelques exemples d'application

On présente dans ce paragraphe la modélisation en UML/OCL des exemples classiques d'application des propriétés comportementales et axiomatiques visant à démontrer que des fragments de code sont corrects. Il s'agit principalement de montrer les structures de données que l'on doit mettre en place pour pouvoir effectuer les calculs des Plus Faibles Pré-conditions et des obligations de preuve sur des modèles.

Un premier exemple, très simple, montre les structures de données que l'on peut mettre en place pour calculer les assertions déduites d'un triplet de Hoare.

Le deuxième exemple reprend le calcul de la racine carrée approchée d'un nombre entier. Il montre bien l'aspect statique de la vérification du code, et la modélisation en UML/OCL. Le langage « L » est utilisé comme langage de programmation décrivant les éléments de programmation du triplet de Hoare correspondant. On peut voir que les techniques vérifiant qu'un fragment de code « L » est correct se transposent facilement sur des modèles de codes.

Le troisième exemple introduit l'application des techniques de preuve à la transformation de modèles. On montre d'abord comment cette factorisation écrite à l'aide d'un programme « L » peut être modélisée en UML/OCL. On montre, ensuite, comment on peut appliquer l'ensemble de ces techniques à la vérification des transformations de modèles. Cet exemple introduit, en fait, le chapitre V portant sur la transformation d'un diagramme d'activité en un modèle de code. Ce deuxième exemple montre que les modèles source et cible d'une transformation peuvent être des instances de Méta-Modèles différents. Cependant, il est nécessaire que les propriétés syntaxiques et sémantiques des langages des modèles sources et cibles puissent être spécifiées dans le langage du programme de transformation de manière à

[1] http://research.microsoft.com/en-us/um/redmond/events/smt08/filliatre.pdf

pouvoir exprimer les propriétés comportementales des deux modèles sur le même environnement d'exécution.

III.1 Exemple d'un Modèle de code du langage « L »

III.1.a Modèle de code

Nous supposons qu'au programme « L » suivant :

```
programme p1
        x : Rel
        x := 1
                -- p : { x > 0 }
        x := x + 10
                -- p : { x > 8 }
        -- resultat x
```

correspond le modèle suivant, instance du Méta-Modèle du langage « L », enrichi des éléments de modélisation permettant de prendre en compte les triplets de Hoare :

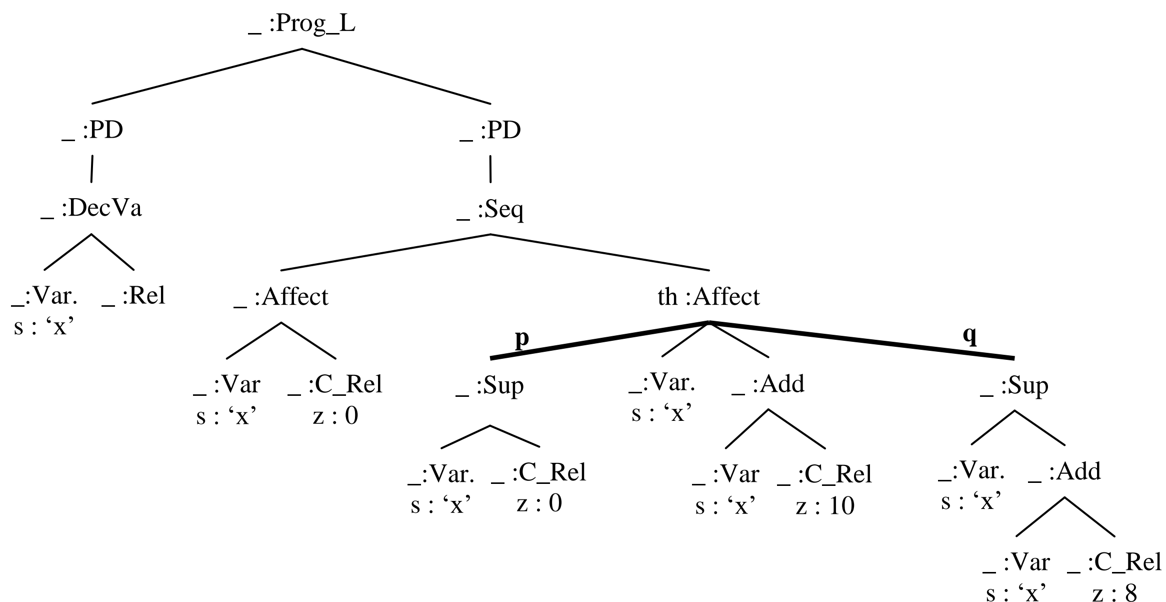


Figure 20 : ***Exemple simple d'un Modèle de code du langage « L »***

Ce modèle fait suggérer qu'un Concepteur souhaite prouver qu'un fragment de code, en l'occurrence l'instruction d'affectation mettant à jour la variable x, est correct par rapport aux assertions p et q.

La sémantique comportementale du triplet de Hoare rappelée en fin de paragraphe précédente montre comment l'assertion associée au triplet de Hoare modélisé au niveau de l'objet de nom th, peut être traduit soit en B, soit exécuté sur des données ponctuelles.

III.1.b Arbres déduits du triplet de Hoare et du calcul de la pfp

En calculant, toujours au niveau UML, l'obligation de preuve à partir du calcul de la Plus Faible Pré-condition, on obtient une obligation de preuve encore plus simple, facilitant ainsi le passage à B pour vérifier statiquement la preuve. En supposant que l'on crée un arbre d'expression Assert, tel que le montre la figure suivante, en reprenant l'exemple du programme précédent :

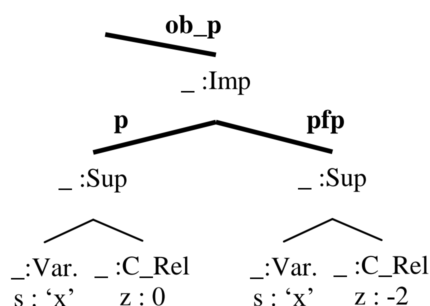


Figure 21 : ***Arbre de l'Obligation de Preuve après le calcul de la pfp***

On peut voir facilement comment cette obligation de preuve peut être traduit en B, en vue de voir si elle est valide, ou éventuellement en vue d'être exécutée au niveau UML, sur quelques données pertinentes.

Dans le cas, où le typage des types prédéfinis du langage « L » et de B, ainsi que les opérateurs, se correspondent, on peut simplifier le processus d'interprétation précédent en transformant l'arbre de l'obligation de preuve en « L » (1), en un arbre B (2), tel que le montre la figure suivante :

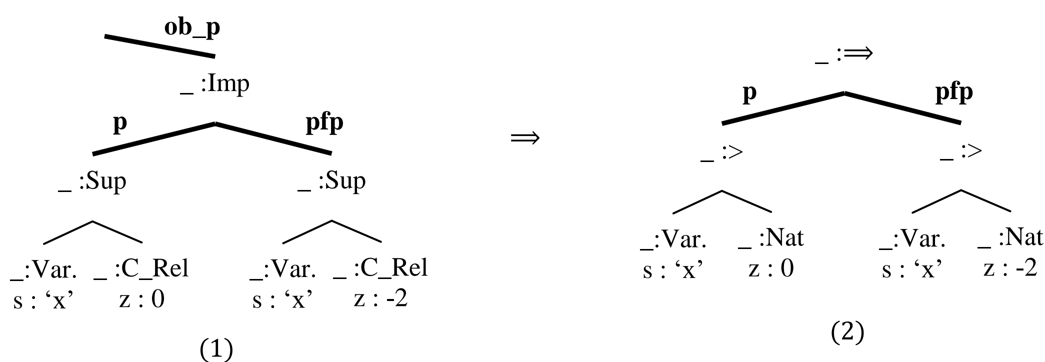


Figure 22 : ***Correspondance entre les types et les opérateurs du Langage « L » et de B***

Dans ce cas, l'expression B est très facile à produire, et les preuves statiques, en B, ou dynamiques, au niveau UML sont aussi accessibles aux Concepteurs.

III.2 Retour au calcul de la racine carrée approchée d'un nombre entier

III.2.a Programme « L »

```
programme p1
        x, y : Rel
        yc, y1c : Rel
        x := …
        y := 0
        yc := 0
        y1c := 1
                        -- p : { x > 0 and y1c = 1 and yc = 1 }

        tant que  not( x < y1c )
                y := y + 1
                yc := y1c
                y1c := yc + 2 * y + 1
        -- resultat y

                        -- q : {  ( y * y <= x )  and ( x < ( y + 1 ) * ( y + 1 ) ) }
```

III.2.b Invariant, Variant, Triplet de Hoare et Obligations de preuve

-- invariant : (y * y <= x) and ((yc = y * y) and (y1c <= (y1 + 1) * (y1 + 1))

-- variant : x – y1c

-- triplet de Hoare :

{ x > 0 and y1c = 1 and yc = 1 } tant que not (x < y1c) … { (y * y) <= x and (x < ((y + 1) * (y + 1)) }

-- assertion associée au triplet de Hoare :

(x > 0 and y1c = 1 and yc = 1) ⟹ (y * y <= x) and (yc = y * y) and (y1c < (y1 + 1) * (y1 + 1))

-- obligations de preuve :

assertion associée au triplet de Hoare
invariant ^ (*Exp* = true) ^ (variant = z) : A l'entrée de la boucle l'invariant doit être vérifié,
invariant ^ (*Exp* = false) ⟹ Q : En sortie de boucle la post-condition Q doit être vérifiée,
invariant ^ (Not(x < y1c) ⟹ (variant ≥ 0) : Le variant doit rester positif
invariant ^ (Not(x < y1c) ⟹ pfp(v := variant, Inst ; (variant < v))

III.2.c Modèle du programme « L »

La figure suivante donne le fragment de modèle de la boucle du programme :

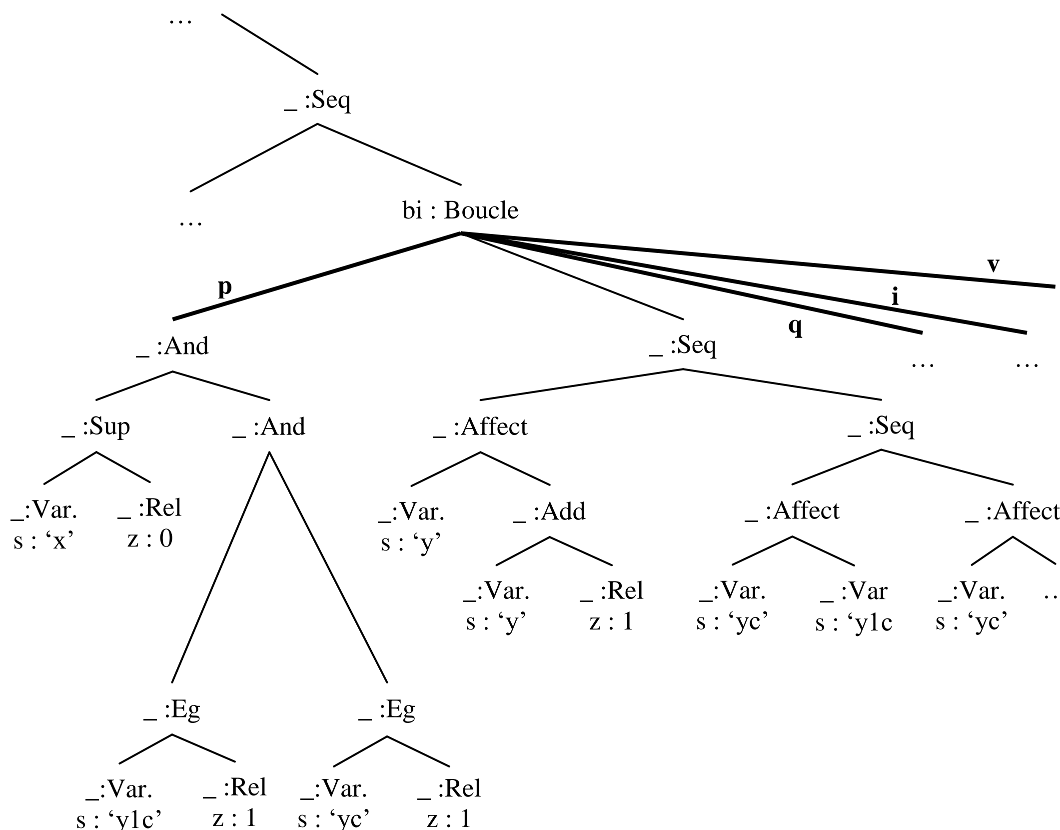


Figure 23 : ***Fragment du modèle du programme faisant apparaître l'instruction de boucle***

Sur cette figure, seul l'arbre de la pré-condition a été développé. Les arbres de la post-condition, de l'invariant et du variant n'apparaissent pas dans la figure, mais se déduisent des expressions qui sont rappelées plus haut.

On suppose bien sûr que les opérations définissant en UML/OCL les propriétés comportementales et axiomatiques du langage « L » sont implantées. En particulier :

Inst ::pfp(envExec : EnvExec) : Assert
Assert ::op(envExec : EnvExec) : Assert
Assert ::op2B(envExec : EnvExec) : String

III.2.d Calcul de l'assertion associée au triplet de Hoare

L'exécution de l'opération :

bi.pfp(env) : Assert

retourne la structure de données suivante, correspondante au triplet de Hoare pour l'instruction de boucle référencée à la racine du sous-arbre bi :

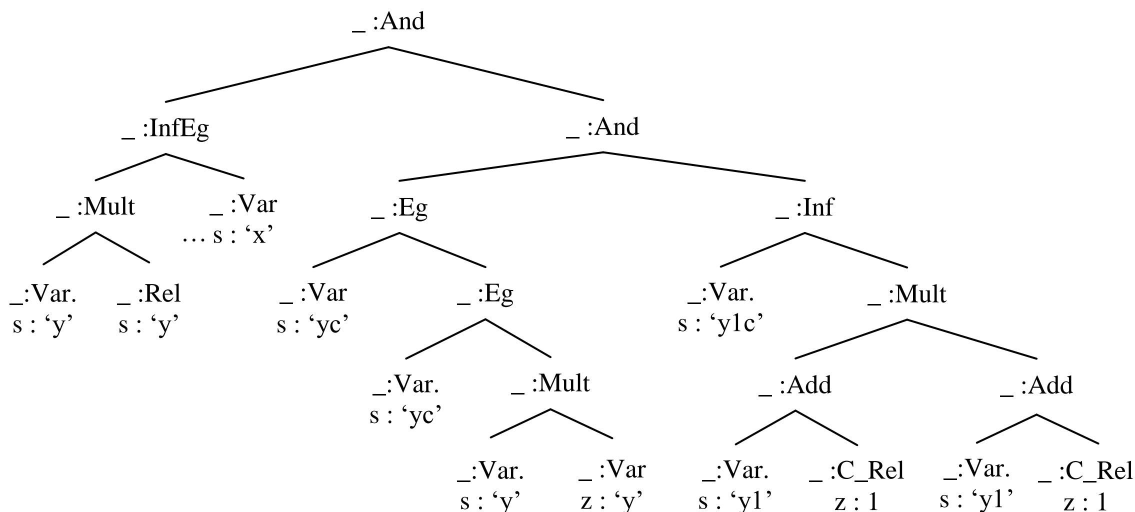


Figure 24 : ***Calcul de l'assertion associée au triplet de Hoare pour une instruction de boucle***

L'assertion associée au triplet de Hoare pour une instruction de boucle, n'est autre que l'invariant, par définition.

L'exécution de l'opération

pi.pfp(env).op(env)

retourne la structure de données suivante représentant l'obligation de preuve associée à la boucle :

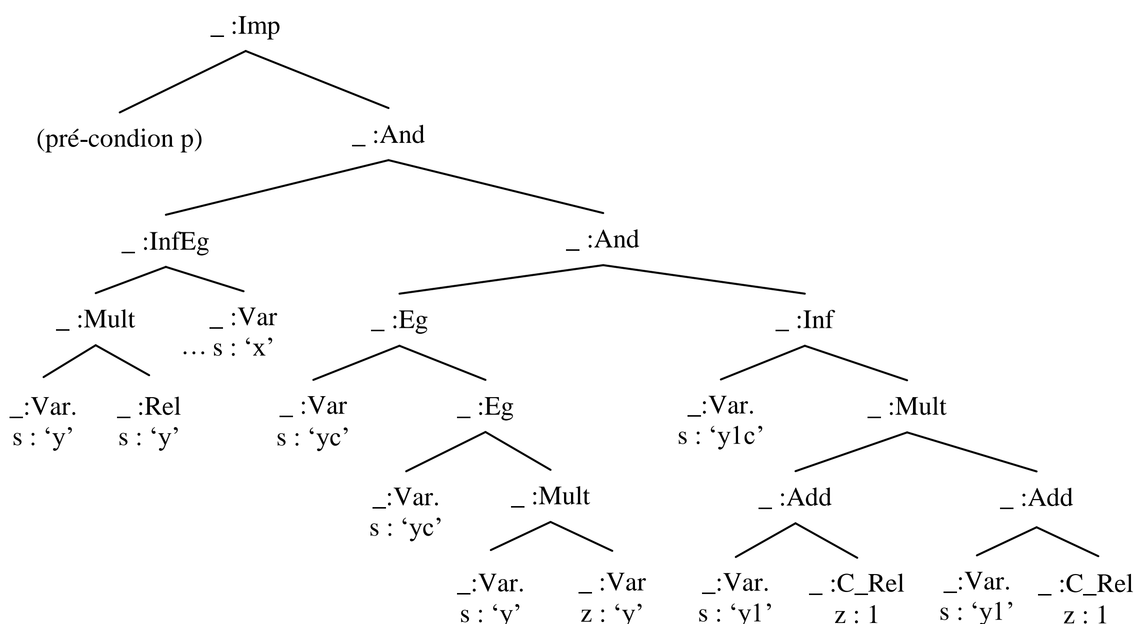


Figure 25 : ***Structure de données relative à l'obligation de preuve pour l'instruction de boucle***

Cette structure de données permettra de déterminer tout l'environnement de modélisation B qui sera ensuite transmis à l'Atelier B pour en vérifier son éventuelle validité.

Si les types des variables d'une telle assertion peut être mis en correspondant avec les types B, alors il suffira de retourner cette assertion B sous la forme textuelle suivante :

```
!( xx, yy, yc, y1c ).( xx : NAT & yx : NAT & yc : NAT & y1c : NAT ⟹ (
        ( xx > 0 & ( y1c = 1 & yc = 1 ) ) ⟹ ( ( yy * yy <= xx ) & ( yc = ( yy  * yy ) ) and
                                                    ( x1c < ( y1 + 1 ) * ( y1 + 1 ) ) )
```

III.3 Factorisation d'une expression

Supposons l'expression entière suivante : x * y + x * z. En appliquant les identités remarquables, on sait que l'on peut simplifier le nombre d'opérations de cette expression en mettant x en facteur. Une telle expression entière x * y + x * z apparaissant dans un programme pourrait donc être remplacée par l'expression x * (y + z). Cette simplification, et bien d'autres se déduisant d'identités remarquables, sont directement effectuées par le traducteur, en interne.

III.3.a Fragment de programme

Le fragment de programme « L » suivant effectue cette transformation, en supposant que toutes les variables sont de type REL, et que les variables xij sont initialisées :

```
…
x := x11 * x12 + x21 * x22
y11 := x11
y12 := x12
y22 := x22
y := y11 * ( y12 + y22 )
…
```

Le changement des variables en x et en y permet de construire la nouvelle expression y.

Si l'on veut vérifier que la séquence d'instructions construisant l'expression y est correcte, alors on peut rajouter la pré-condition et la post-condition suivantes :

```
…
x := x11 * x12 + x21 * x22
                         -- p = { x11 = x21 }
y11 := x11
y12 := x12
y22 := x22
                         -- q = { x11 * x12 + x21 * x22  = y11 * ( y12 + y22 )  }
y := y11 * ( y12 + y22 )
…
```

La pré-condition indique que pour pouvoir appliquer la factorisation, il faut que x11 et x21 aient les mêmes valeurs, ce qu'exprime la pré-condition p. La post-condition q exprime le fait que quelques soient les valeurs des différentes variables, les expressions x et y ont le même comportement.

III.3.b Triplet de Hoare

Pour essayer donc de prouver que la séquence des instructions est correcte, on peut chercher à voir si le triplet de Hoare suivant est valide :

{ x11 = x21 } y11 := x11 ; y12 := x12 ; y22 := x22 { x11 * x12 + x21 * x22 = y11 * (y12 + y22) }

III.3.c Calcul de l'obligation de preuve associée au triplet de Hoare

A partir du triplet de Hoare (1), on applique la règle de calcul de la pfp, étape par étape, de la troisième instruction d'affectation en remontant jusqu'à la première instruction d'affectation. Les triplets de Hoare de (2), (3) et (4) montrent l'évolution du triplet de Hoare. On en déduit, alors, que pour valider le triplet de Hoare, il faut démontrer que l'assertion (5) est valide :

p *Inst* q

(1) : { x11 = x21 } y11 := x11 ; y12 := x12 ; **y22** := x22 { x11 * x12 + x21 * x22 = y11 * (y12 + **y22**) }

{ x11 * x12 + x21 * x22 = y11 * (y12 + **x22**) }

(2) : { x11 = x21 } y11 := x11 ; **y12** := x12 ; { x11 * x12 + x21 * x22 = y11 * (**y12** + x22) }

{ x11 * x12 + x21 * x22 = y11 * (**x12** + x22) }

(3) : { x11 = x21 } **y11** := x11 { x11 * x12 + x21 * x22 = **y11** * (y12 + x22) }

{ x11 * x12 + x21 * x22 = **x11** * (x12 + x22) }

(4) : { x11 = x21 } { x11 * x12 + x21 * x22 = x11 * (y12 + x22) }

(5) : x11 = x21 ⇒ (x11 * x12 + x21 * x22 = x11 * (y12 + x22))

On en déduit qu'il faut démontrer que l'obligation de preuve suivante :

∀ x11, x12, x21, x22 ∈ Rel | (x11 = x21) ⟹ (x11 * x12 + x21 * x22 = x11 * (x12 + x22))

Cette assertion a une écriture simplifiée, puisque l'on suppose que le type Rel fait référence au type des nombres entiers, positifs ou négatifs. A ce type REL du langage « L », correspond le type NAT en B. les opérateurs de multiplication et d'addition du langage « L » et de B se correspondent.

L'obligation de preuve s'écrit en B de la manière suivante :

III.3.c.1 Modélisation en B de l'assertion définissant le triplet de Hoare

```
MACHINE
factorisation

SETS
Expression

ABSTRACT_VARIABLES
Exp, OB , Add , Mult, ConstE,
exp1 , exp2 , ctNat, VARI

DEFINITIONS
inv == (
Exp <: Expression & Add <: Expression & Mult <: Expression & OB <: Expression &
ConstE <: Expression & VARI <: Expression &
OB <: Exp & Add <: OB & Mult <: OB &
ConstE <: Exp & VARI <: Exp &
Add /\ Mult = {} & Add \/ Mult = OB &
OB /\ ConstE = {} & OB /\ VARI = {} &
ConstE /\ VARI = {} &
OB \/ ConstE \/ VARI = Exp &
exp1 : Expression +-> Exp & exp2 : Expression +-> Exp &
exp1 : OB --> Exp & exp2 : OB --> Exp &
ctNat : ConstE --> NAT );

isEval(eval) == (
eval: (VARI --> NAT) * Exp --> NAT
& !(env, add).( env: VARI --> NAT
& add:Add => eval(env,add) = eval(env, exp1(add)) + eval(env, exp2(add))
)
& !(env, mul).( env: VARI --> NAT
& mul:Mult => eval(env,mul) = eval(env, exp1(mul)) * eval(env, exp2(mul))
)
& !(env,var).( env:VARI --> NAT
& var:VARI => eval(env,var) = env(var)
)
& !(env,cte).( env:VARI --> NAT
& cte:ConstE => eval(env,cte) = ctNat(cte)
)
)
INVARIANT
inv

INITIALISATION
Exp :={} || OB := {} || Add :={} || Mult := {} || ConstE := {} || VARI := {} ||
exp1 :={} || exp2 := {} || ctNat := {}

OPERATIONS
res <-- transfo(exp) =
PRE exp: Exp
& exp : Add
& exp1(exp) : Mult
& exp2(exp) : Mult
& exp1(exp1(exp)) = exp1(exp2(exp))
THEN
res, Exp, OB, Add, Mult, exp1, exp2 : ( inv & res: Exp
& !(env,eval). (env: VARI-->NAT & isEval(eval) => eval(env,res) = eval(env,exp))
)
END
END
```

```
REFINEMENT
factorization_r

REFINES
factorisation

ABSTRACT_VARIABLES
Exp , OB , Add , Mult , ConstE , exp1 , exp2 , ctNat , VARI

INITIALISATION
Exp := {} || OB := {} || Add := {} || Mult := {} || ConstE := {} || exp1 := {} || exp2 := {} || ctNat := {} || VARI :=
{}

OPERATIONS
res <-- transfo(exp) =
ANY       newAdd, newMult
WHERE
          newAdd : Expression - Exp & newMult : Expression - Exp & newAdd /= newMult
THEN
Mult := Mult \/ {newMult};
Add := Add \/ {newAdd};
OB := OB \/ {newMult, newAdd};
Exp := Exp \/ {newMult, newAdd};
exp1, exp2 := exp1 <+ { newAdd |-> exp2(exp1(exp)) , newMult |-> exp1(exp1(exp)) } , exp2 <+ { newAdd |->
exp2(exp2(exp)) , newMult |-> newAdd};
res := newMult
END
END
```

III.3.c.2 Modélisation en B de l'assertion du triplet de Hoare après le calcul de la pfp

```
MACHINE
factorisation

SETS
Expression

ASSERTIONS
!(env, eval, x11, x12, x21, x22, add1, mm1, mm2,mm3, add2 ).( x11: Exp & x12 : Exp & x21: Exp & x22 : Exp
& add1: Add & add2 : Add & mm1 : Mult & mm2 : Mult & mm3 : Mult & mm1 : Mult
& {(add1 |-> mm1),(mm1 |->x11), (mm2 |->x21), (mm3 |->x11), (add2 |-> x12)} <: exp1
& {(add1 |-> mm2),(mm1 |->x12), (mm2 |->x22), (mm3 |->add2), (add2 |-> x22)} <: exp2
& env: VARI-->NAT & isEval(eval)
=> (( eval(env, x11) = eval(env,x21) )=>( eval(env,mm3) = eval(env, add1)) ))

ABSTRACT_VARIABLES
Exp, OB , Add , Mult, ConstE,
exp1 , exp2 , ctNat, VARI

DEFINITIONS
inv == (
Exp <: Expression & Add <: Expression & Mult <: Expression & OB <: Expression &
ConstE <: Expression & VARI <: Expression &
OB <: Exp & Add <: OB & Mult <: OB &
ConstE <: Exp & VARI <: Exp &
Add /\ Mult = {} & Add \/ Mult = OB &
OB /\ ConstE = {} & OB /\ VARI = {} &
```

```
ConstE /\ VARI = {} &
OB \/ ConstE \/ VARI = Exp &
exp1 : Expression +-> Exp & exp2 : Expression +-> Exp &
exp1 : OB --> Exp & exp2 : OB --> Exp &
ctNat : ConstE --> NAT );

isEval(eval) == (
eval: (VARI --> NAT) * Exp --> NAT
& !(env, add).( env: VARI --> NAT
& add:Add => eval(env,add) = eval(env, exp1(add)) + eval(env, exp2(add))
)
& !(env, mul).( env: VARI --> NAT
& mul:Mult => eval(env,mul) = eval(env, exp1(mul)) * eval(env, exp2(mul))
)
& !(env,var).( env:VARI --> NAT
& var:VARI => eval(env,var) = env(var)
)
& !(env,cte).( env:VARI --> NAT
& cte:ConstE => eval(env,cte) = ctNat(cte)
)
)
INVARIANT
inv

INITIALISATION
Exp :={} || OB := {} || Add :={} || Mult := {} || ConstE := {} || VARI := {} ||
exp1 :={} || exp2 := {} || ctNat := {}
END
```

III.3.c.3 Cas particulier : Les types prédéfinis sont comparables

```
!( x11, x12, x21, x22 ). ( x11 : NAT & x12 : NAT & x21 : NAT & x22 : NAT
                    =>( x11 = x21 => x11 * x12 + x21 * x22 = x11 * ( x12 + x22 ) ) ).
```

L'atelier B pourra donc valider cette assertion.

III.3.d Modélisation en UML/OCL

En reprenant le fragment de programme « L » décrit dans le paragraphe *III.3.a* page 31, il s'agit de modéliser en UML/OCL cet exemple de fragment, et de voir comment le détail des calculs peut se retrouver.

La figure suivante montre comment est structuré le triplet de Hoare au sein du modèle du programme. Le triplet de Hoare englobe les trois instructions d'affectations. La pré-condition précède la première instruction d'affectation. Dans le modèle du programme, elle est donc rattachée à l'objet afi de type Affect référençant cette instruction. La post-condition, qui n'apparat pas détaillée sur la figure, se situe après la dernière instruction d'affectation. Elle est donc rattachée à l'objet afj de type Affect référençant cette instruction. La pré-condition et la post-condition encadre donc bien les trois instructions d'affectation.

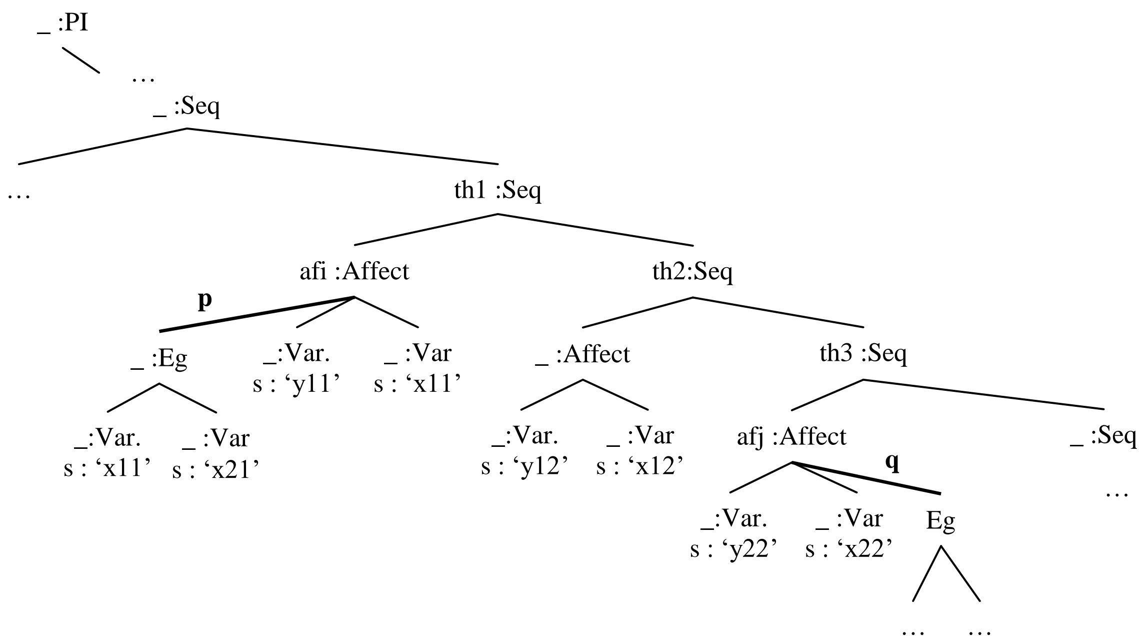


Figure 26 : ***Fragment du Modèle de programme concernant le triplet de Hoare***

La structure de données représentant le modèle de la post-condition du triplet de Hoare est donnée par la figure suivante :

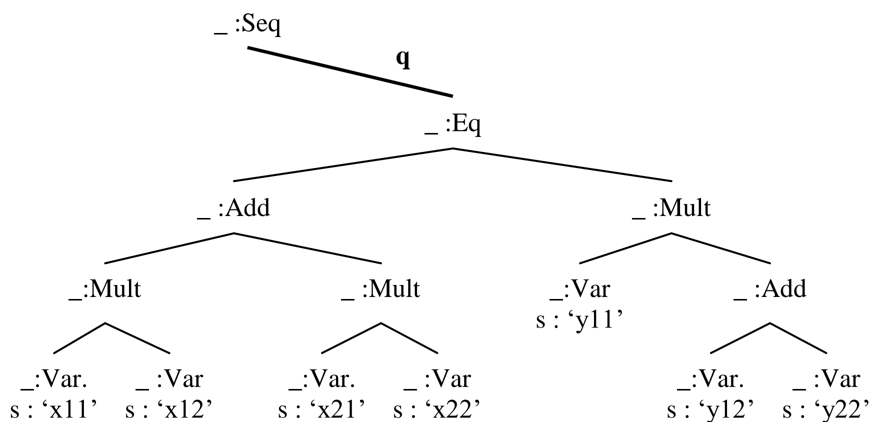


Figure 27 : ***Modèle de la post-condition du triplet de Hoare***

III.3.e Calcul de l'obligation de preuve

L'opération *Inst*.pfp(envExec : EnvExec) retourne la structure de données de type Assert, représentant l'assertion associée au triplet de Hoare se trouvant référencée par une instance de Inst. L'exécution de l'opération th1.pfp(env) retourne donc l'arbre suivant, de type Assert :

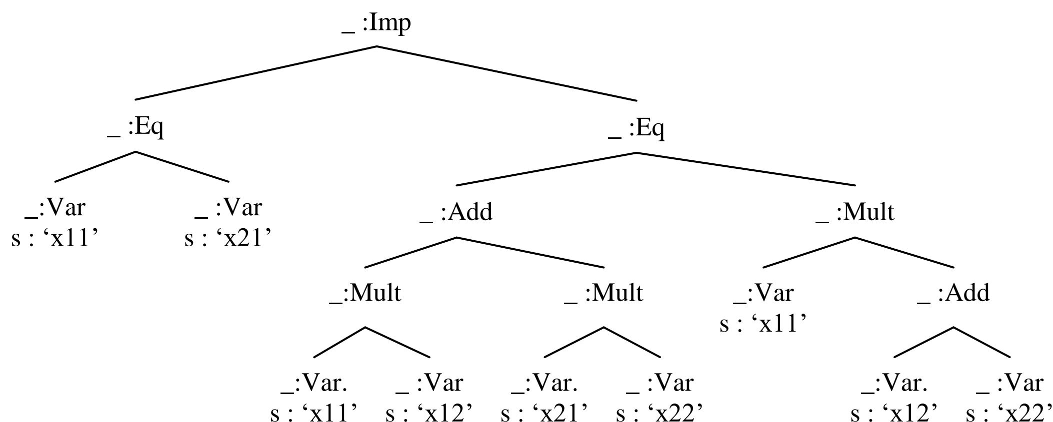


Figure 28 : ***Modèle de l'assertion associée au triplet de Hoare***

L'opération Assert.op(envExec : EnvExec) retourne la structure de données représentant passé en paramètre.

Enfin : th1.pfp(envExec).op2B(envExec) retourne l'obligation de preuve traduite en B, en supposant que l'on reste dans notre contexte où les variables sont de type Rel, et du fait de la correspondance entre les opérateurs du langage « L » et B.

III.4 Vers la vérification de transformations de modèles

Dans un environnement de transformation de modèles, les modèles source et cible sont, tel que le montre la figue suivante, dissociés :

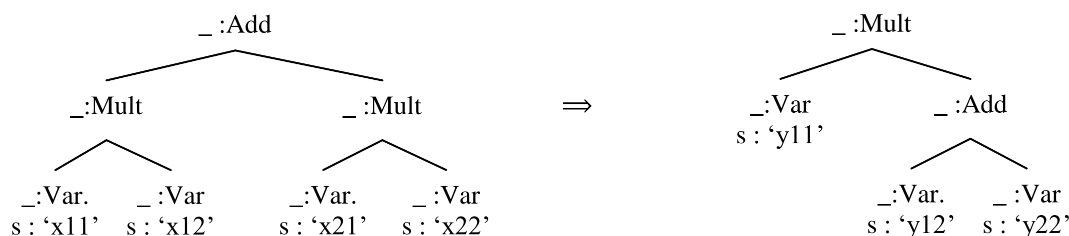


Figure 29 : ***Modèle Source, Modèle cible ; programme de transformation***

Le modèle de gauche, représentant une expression du langage « L », peut en fait être un modèle, instance d'un Méta-Modèle quelconque. Il peut en être de même pour le modèle de droite. Le modèle source et le modèle cible sont, ici, par rapport à l'exemple précédent, dissociés c'est-à-dire n'appartement pas au modèle d'un même programme. Dans une transformation de modèles, on aura donc à écrire un programme dans un langage qui assurera la transformation du modèle de gauche, le modèle source, en un modèle de droite, e modèle cible. Dans un environnement de modélisation UML, les langages source et cible et leurs propriétés sont modélisés en UML ou en MOF, mais au départ les langages peuvent être différents. Le langage d'action UML permettra de décrire la transformation, alors que dans l'exemple précédent la transformation était écrit en langage « L ». L'opération qui assurera la transformation devra donc passer en paramètre, les Méta-Modèles source et cible, les Modèles

source et cible ainsi l'environnement d'exécution qui servira à définir les comportements de référence des langages des modèles source et cible sur le même environnement d'exécution.

Dans le chapitre suivant, nous prendrons comme exemple de langage du modèle source, le diagramme d'activité qui permet, en particulier, de définir des algorithmes de haut niveau qui pourront être utilisés par les Analystes/Concepteurs, au cours du processus de développement, en amont des modèles de programmes. Ce qui signifie que l'on rajoute un niveau de modélisation entre les modèles issus des exigences des applications à développer et les modèles de programmes.

A ce niveau de modélisation, il s'agit donc de modéliser des algorithmes de niveau PIM, indépendants donc des spécificités des plates-formes et des langages cibles. Rajouter, au cours du processus de développement, un niveau de modélisation des algorithmes à l'aide d'un diagramme d'activité, en amont des modèles de codes, et par voie de conséquence des codes nous parait méthodologiquement important.

En effet, le Méta-Modèle du langage cible se déduit des propriétés de la grammaire et des propriétés du langage. Le Méta-Modèle est donc figé, éventuellement dans la mesure où les modifications ne remettent pas en cause les propriétés du langage. En effet, en phase finale du processus, toute instance du Méta-Modèle du langage devra être transformée en du code qui devra être pris en compte par le traducteur du langage.

Le Méta-Modèle UML décrit la structure syntaxique des différents diagrammes UML, en particulier, le diagramme d'activité qui peut être utilisé pour modéliser des algorithmes de haut niveau. Les experts d'un domaine d'application peuvent donc, comme on l'a fait pour le Méta-Modèle d'un langage, définir des propriétés comportementales et axiomatiques mais indépendantes des spécificités des langages cibles, et pouvant tenir compte des spécificités métier du domaine d'applications. Ce qui donnera la possibilité aux Analystes/Concepteurs des applications du domaine à réaliser l'activité de génération des codes à partir des modèles de conception en trois grandes étapes successives de raffinage :

- La transformation des modèles de conception en diagramme d'activité faisant apparaitre les activités du système à développer et leurs enchainements.
- La transformation des diagrammes d'activités en modèles de programme prenant en compte les spécificités des langages cibles issues de leur grammaire abstraite.
- La génération proprement-dite des codes.

Le rajout, au niveau de Méta-Modèle du diagramme d'activité de propriétés comportementales et axiomatiques, permettra de vérifier à chacune des étapes de raffinage que les transformations respectent les propriétés des modèles sources et cibles.

V – Diagrammes d'Activité et exemple de Propriétés Syntaxiques et Sémantiques

Les diagrammes d'activités UML 2.0 servent à représenter graphiquement la logique procédurale des différentes actions d'un processus métier et des flots de contrôle qui enchaînent ces actions. Ce sont les diagrammes d'activités qui ont connu les changements les plus notables au fil des différentes versions d'UML.

Les diagrammes d'activité permettent de spécifier des traitements en donnant une vision proche de celle des langages de programmation impératifs comme C++ ou Java, si l'on s'en tient aux aspects séquentiels de ses éléments de modélisation. Ils peuvent être ainsi utiles aux Analystes/Concepteurs en leur donnant la possibilité de décrire des algorithmes de haut niveau, en amont des activités de génération des codes d'un processus IDM. La modélisation d'une opération reste cependant assimilée à une utilisation d'UML comme langage de programmation visuelle.

Dans le cadre de cette étude, les digrammes d'activité pourront être utilisés pour spécifier, construire, visualiser ou documenter le comportement d'une méthode ou le déroulement d'un cas d'utilisation.

Au cours de ce chapitre, nous rappellerons les différents éléments de modélisation offerts par un diagramme d'activité pour modéliser des algorithmes séquentiels.

Nous nous inspirerons, ensuite, des concepts et techniques que nous avons décrits aux chapitres III et IV pour appliquer sur le langage UML, (le Méta-Modèle UML) des propriétés syntaxiques et sémantiques applicables aux diagrammes d'activité. Les Analystes/Concepteurs pourront donc modéliser avec précision le comportement des algorithmes de haut niveau décrits à l'aide des diagrammes d'activité.

Rajouter des propriétés syntaxiques et sémantiques applicables à des diagrammes d'activité, et plus généralement à d'autres diagrammes UML, est relativement délicat dans la mesure où il s'agit de décrire des algorithmes de haut niveau, indépendants des langages des plates-formes cibles mais pouvant, éventuellement, prendre en compte et intégrer dans les propriétés des spécificités du domaine métier. C'est la raison pour laquelle le rajout de propriétés applicables à des diagrammes d'activité ne peut se faire que par des experts de manière à ce que les Analystes/Concepteurs puissent créer des diagrammes d'activité se situant à la charnière entre les activités de conception et de génération de codes d'un processus IDM.

Au cours du chapitre suivant, nous serons alors en mesure de montrer comment il est possible d'aider les Analystes/Concepteurs à vérifier que les codes des applications reflètent bien les intentions des Concepteurs.

V – Diagrammes d'Activité, et exemple de Propriétés Syntaxiques et Sémantiques

Table des matières

I Processus IDM pour l'Activité de Génération de Code ... 146
I.1 Les diagrammes d'activité ... 146
I.2 Les actions ... 147
II Eléments de Modélisation du Diagramme d'Activité ... 147
II.1 Les Nœuds d'Activité et les Flots de Contrôle ... 147
II.1.a Représentation graphique des Nœuds d'Activité et des Flots de Contrôle ... 147
II.1.b Poids et Garde des Flots de Contrôle ... 148
II.1.c Nœuds d'Activité : Nœuds Exécutables et Nœuds de Contrôle ... 148
II.1.d Diagramme d'Activité et Algorithme séquentiel ... 151
II.2 Les différents types d'Actions Exécutables ... 152
II.2.a Appel à un Diagramme d'Activité ... 152
II.2.b Diagramme d'Activité et Diagramme de Classes ... 152
II.2.c Nœuds d'Activité Structurée ... 153
II.2.d Action de mise à jour la valeur d'une variable (AddVariableValueAction) ... 154
III Fragment du méta-modèle UML 2.0 du diagramme d'activité ... 155
III.1 Nœuds Exécutables, Nœuds de Contrôle et Flots de Contrôle ... 155
III.1.a Fragment de Méta-Modèle du Diagramme d'activité ... 155
III.1.b Représentation des flots de contrôle ... 157
III.1.c Représentation des Nœud de Contrôle de type Nœud de Décision (DecisionNode) ... 157
III.2 Représentation des différents types d'Actions ... 158
III.2.a Fragment du Méta-Modèle UML détaillant quelques types d'actions ... 158
III.2.b Appel à un Diagramme d'Activité ... 159
III.2.c Action de mise à jour la valeur d'un variable ... 161
III.2.d Action et Expressions Opaques ... 163
III.2.e Noeuds d'Activité Structurée ... 163
III.2.f Diagramme de classes et diagramme d'activité ... 166
IV Modélisation de Propriétés Syntaxiques et Sémantiques en UML/OCL applicables au Diagramme d'Activité ... 169
IV.1 Modélisation en UML/OCL de Propriétés Syntaxiques et Comportementales ... 169
IV.1.a Modélisation en UML/OCL des Propriétés Syntaxiques pour un Diagramme d'Activité ... 169
IV.1.b Environnement d'exécution pour le fragment du Diagramme d'Activité du MM UML ... 170
IV.1.c Modélisation des propriétés comportementales d'un Diagramme d'Activité ... 171
IV.1.d Triplet de Hoare et propriétés axiomatiques d'un Diagramme d'Activité ... 173
IV.2 Le rôle du diagramme d'activité ... 174
IV.2.a Graphe de Nœuds d'Activité ou Arbre de Nœuds d'Activité Structurée ? ... 174
IV.2.b Vers un Processus IDM pour la Génération de Codes ... 174

V – Diagrammes d'Activité, et exemple de Propriétés Syntaxiques et Sémantiques

Table des figures

Figure 1 : Composants logiciels impliqués dans l'activité de génération de codes d'un processus IDM 146
Figure 2 : Nœuds d'Activité et Flot de Contrôle d'un Diagramme d'Activité 148
Figure 3 : Nœud d'activité de type Nœud Exécutable et Nœud de Contrôle 149
Figure 4 : Différents types de Nœuds de Contrôle 150
Figure 5 : Exemple d'un Diagramme d'Activité 150
Figure 6 : Le poids w et la garde g des Flots de Contrôle 151
Figure 7 : Appel d'un diagramme d'activité (2) à partir d'un nœud exécutable de type CallBehaviorAction (1) défini dans un diagramme d'activité 152
Figure 8 : diagramme d'activité modélisant la méthode d'une classe 153
Figure 9 : Diagramme d'activité élaboré à l'aide de nœuds d'activité structurée 154
Figure 10 : Segment d'un diagramme d'activité montrant l'affectation d'une valeur à une variable 154
Figure 11 : Les nœuds exécutables, les nœuds de contrôle et les flots de contrôle dans le Méta-Modèle UML 2.0 156
Figure 12 : Exemple d'un fragment d'un Diagramme d'Activité enchaînant deux Actions 157
Figure 13 : Exemple d'un fragment d'un diagramme d'activité contenant un nœud de décision 158
Figure 14 : Les différents types d'Actions du Méta-Modèle UML 2.0 159
Figure 15 : Appel d'un diagramme d'activité à partir d'un nœud exécutable de type CallBehaviorAction défini dans un diagramme d'activité 160
Figure 16 : Représentation interne (Instance du Méta-Modèle UML 2.0) d'une action référençant un diagramme d'activité 161
Figure 17 : Actions sur les Variables dans le fragment du Méta_Modèle UML 2.0 du Diagramme d'Activité 162
Figure 18 : Fragment d'un diagramme d'activité montrant une action de type AddVariableValueAction 162
Figure 19 : Représentations internes d'un fragment de Diagramme d'Activité référençant une action de type AddVariableValueAction 163
Figure 20 : Les nœuds d'activité structurée dans le fragment du méta-modèle UML 2.0 du diagramme d'activité 164
Figure 21 : Fragment d'un diagramme d'activité montrant un nœud d'activité structurée de type LoopNode 165
Figure 22 : Représentations internes d'un fragment de Diagramme d'Activité référençant un nœud de type LoopNode 166
Figure 23 : Fragment du Méta-Modèle UML 20 montrant les liens existant entre un Diagramme de Classes et le Diagramme d'Activité 167
Figure 24 : Diagramme d'Activité modélisant la méthode d'une classe 167
Figure 25 : Représentation interne (Instance du Méta-Modèle UML 2.0) du diagramme de classes (1) et du diagramme d'activité (2) 168
Figure 26 : Propriétés syntaxiques portant sur les Diagrammes d'Activité modélisant des algorithmes 170
Figure 27 : Environnement d'exécution définissant le comportement de référence attendu du Diagramme d'Activité 171
Figure 28 : Spécification comportementales des nœuds d'activité et des spécifications de valeurs 171
Figure 29 : Sémantique opérationnelle des Nœuds de Contrôle du Diagramme d'Activité 172
Figure 30 : Sémantique opérationnelle des Nœuds de Contrôle du Diagramme d'Activité 172
Figure 31 : Sémantique opérationnelle des actions du Diagramme d'Activité 173

I Processus IDM pour l’Activité de Génération de Code

La figure suivante montre l’activité de génération de codes intégrant un niveau de modélisation se situant à la charnière entre les modèles de conception issus des exigences d’une application à développer et l’activité de génération de codes passant par une modélisation des codes des langages cibles, en l’occurrence le langage L :

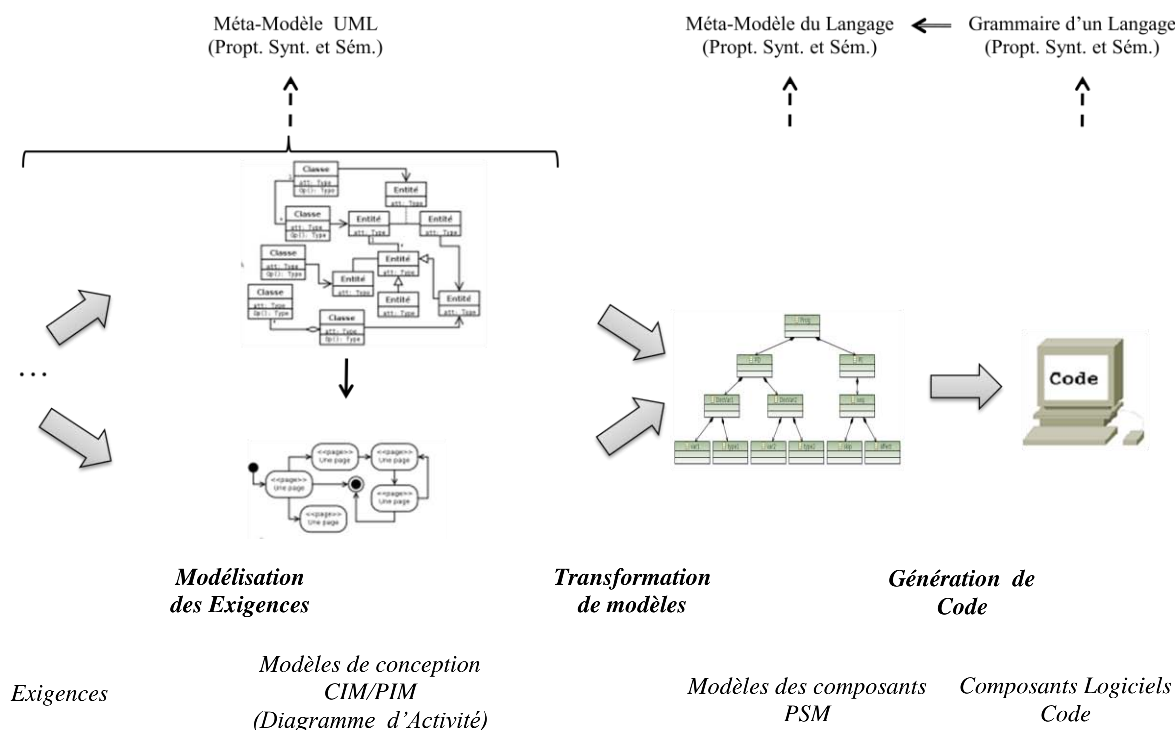


Figure 1 : ***Composants logiciels impliqués dans l’activité de génération de codes d’un processus IDM***

C’est donc au niveau du MM UML que nous intégrons des propriétés syntaxiques et sémantiques portant sur les éléments de modélisation du diagramme d’activité se situant donc en amont de l’activité de génération de codes du processus IDM. Cette activité de génération de codes se fait d’une manière analogue aux techniques de raffinage de code, en raffinant les modèles depuis les diagrammes d’activité jusqu’au code, après avoir pris en compte au niveau du Méta-modèle du langage les caractéristiques techniques du langage cible. Les propriétés syntaxiques et sémantiques, au niveau du Méta-Modèle du langage de modélisation UML et au niveau du Méta-Modèle du langage de programmation L, permettront de vérifier la cohérence entre le diagramme d’activité, assimilé à un modèle de conception, et le code.

I.1 Les diagrammes d’activité

Les diagrammes d’activités permettre de mettre l’accent sur les traitements. Ils sont donc particulièrement adaptés à la modélisation du cheminement de flots de contrôle et de flots de données. Ils permettent ainsi de représenter graphiquement le comportement d’une méthode ou le déroulement d’un cas d’utilisation. Les diagrammes d’activité peuvent donc tout naturellement intervenir dans un processus IDM en amont des activités de génération des codes. Un exemple de propriétés syntaxiques et sémantiques applicables à des diagrammes d’activité sera donné après avoir détaillé dans le paragraphe suivant les éléments de

modélisation pouvant apparaître dans un diagramme d'activité utilisé pour modéliser des algorithmes séquentiels de plus ou moins haut niveau.

I.2 Les actions

Une action est le plus petit traitement qui puisse être exprimé dans un diagramme d'activité. Une action a une incidence sur l'état du système ou en extrait une information. Les actions sont des étapes discrètes à partir desquelles se construisent les comportements. La notion d'action est à rapprocher de la notion d'instruction élémentaire d'un langage de programmation (comme C++ ou Java). Une action peut être, par exemple :

- Une affectation de valeur à des attributs;
- Un accès à la valeur d'une propriété structurelle (attribut ou terminaison d'association);
- Une création d'un nouvel objet ou lien;
- Un calcul arithmétique simple;
- Une émission d'un signal;
- réception d'un signal;
- …

Parmi les actions les plus courantes, on peut citer l'action *call operation* qui correspond à l'invocation d'une opération sur un objet de manière synchrone ou asynchrone. Lorsque l'action est exécutée, les paramètres sont transmis à l'objet cible, ou l'action *call behavior* qui invoque directement une activité plutôt qu'une opération.

Dans les premiers exemples de diagrammes d'activité que l'on donnera dans ce chapitre, on citera l'action opaque qui pourra au cours du processus contenir du code dans un langage qui devra être précisé.

II Eléments de Modélisation du Diagramme d'Activité

Nous présentons tout d'abord les artéfacts de modélisation de plus haut niveau d'abstraction, les nœuds d'activités et les flots de contrôle. Tous les éléments de modélisation que l'on peut trouver dans un diagramme d'activité se définissent à partir de ces nœuds et de ces flots.

II.1 Les Nœuds d'Activité et les Flots de Contrôle

Un diagramme d'activité est composé d'un ensemble de nœuds d'activité qui représentent d'une manière générale les différentes actions du système à modéliser et d'un ensemble de flots de contrôle définissant les modalités de passage entre les nœuds d'activité.

II.1.a Représentation graphique des Nœuds d'Activité et des Flots de Contrôle

A tout nœud d'activité et à tout flow de contrôle est associé un nom. La figure suivante montre les deux nœuds d'activité NA1 et NA2 ont le flot de contrôle FC12 en définit les modalités de passage :

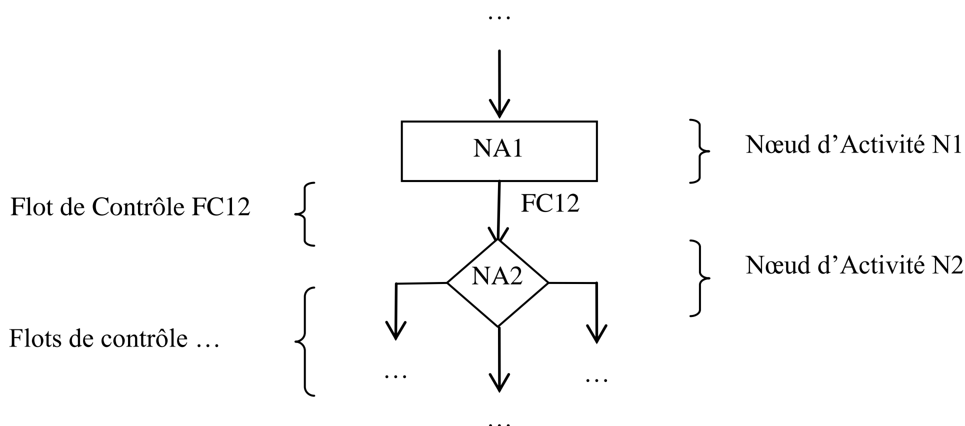


Figure 2 : ***Nœuds d'Activité et Flot de Contrôle d'un Diagramme d'Activité***

Les nœuds d'activité sont représentés par des figures (Rectange, Losange et Rond), et les flots de contrôle par des arcs.

II.1.b Poids et Garde des Flots de Contrôle

Tout flot de contrôle définit les modalités de passage entre deux nœuds d'activité, dont les règles sont les suivantes :

- Tout flot de contrôle est orienté, indiquant l'ordonnancement de l'exécution des deux nœuds d'activité : L'exécution du nœud d'activité N1 sera suivie de l'exécution du nœud d'activité N2, si le flot de contrôle FC12 qui les réunit a pour extrémité initiale N1, et extrémité terminale N2. On dit que N1 a pour successeur N1, que FC12 est un flot de contrôle sortant pour N1, et entrant pour N2. Tout nœud d'activité peut recevoir plusieurs flots de contrôle entrants, et inversement peut être à l'origine du départ de plusieurs flots de contrôle sortants.
- Tout flot de contrôle est défini par deux paramètres qui sont le poids (weight), et la garde (guard) qui donnent les règles de passage entre les deux nœuds d'activité qu'elle relie. Le poids est un nombre entier positif, la garde est un booléen : Dès qu'un nœud d'activité a terminé son exécution, le contrôle est alors donné au nœud d'activité sortant en fonction des valeurs courante des paramètres du flot de contrôle.

II.1.c Nœuds d'Activité : Nœuds Exécutables et Nœuds de Contrôle

Les nœuds d'activité peuvent être des nœuds exécutables ou des nœuds de contrôle reliés à un ou plusieurs flots entrants et sortants, tel que le montre la figure suivante :

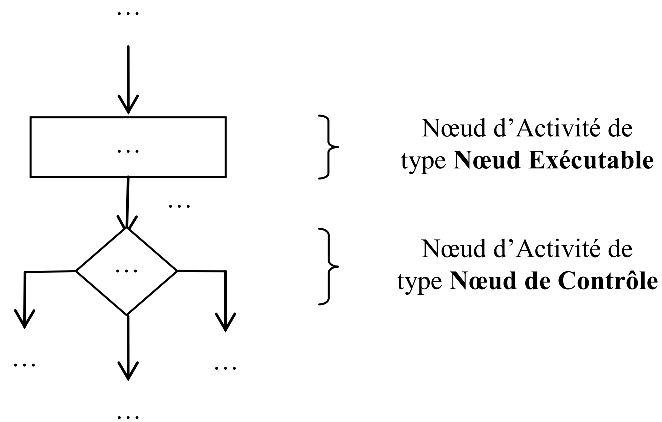
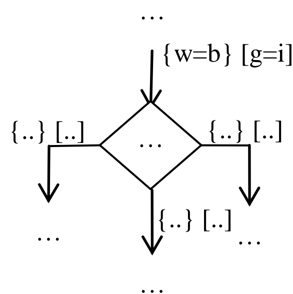


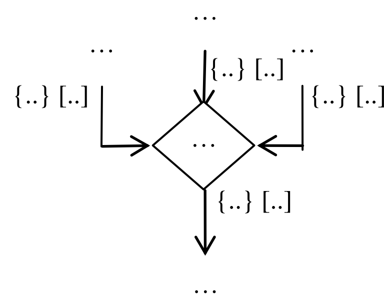
*Figure 3 : **Nœud d'activité de type Nœud Exécutable et Nœud de Contrôle***

Le poids et la garde de chaque flot de contrôle sont indiqués sur la flèche correspondante. Les noms des flots de contrôle et des nœuds de contrôle n'ont pas été indiqués dans la figure.

Compte tenu du fait que nous considérons les diagrammes d'activité élaborés en vue d'une réalisation en programmes de type procédural, un nœud exécutable correspond à une action du système à modéliser et est relié à un seul flot entrant et un seul flot sortant.

Un nœud de contrôle peut être un nœud de décision (*DecisionNode*) relié à un flot entrant et plusieurs flots sortants ou un nœud de regroupement (*MergeNode*) recevant plusieurs flots de contrôle entrants et relié à un flot sortant. En fait, on peut utiliser en même temps un nœud de décision et un nœud de regroupement ayant plusieurs flots de contrôle entrants et plusieurs flots de contrôle sortants. Tout diagramme d'activité doit contenir un nœud initial (*InitialNode*) recevant le contrôle lorsque le diagramme d'activité est activé, et un nœud d'activité terminale (*ActivityFinalNode*) terminant l'exécution du diagramme d'activité. Ces deux derniers nœuds sont des nœuds de contrôle. Les deux figures suivantes montrent les différents types de Nœuds de Contrôle :





Nœud de Contrôle de
type **Nœud de Décision**

Nœud de Contrôle de
type **Nœud de Regroupement**

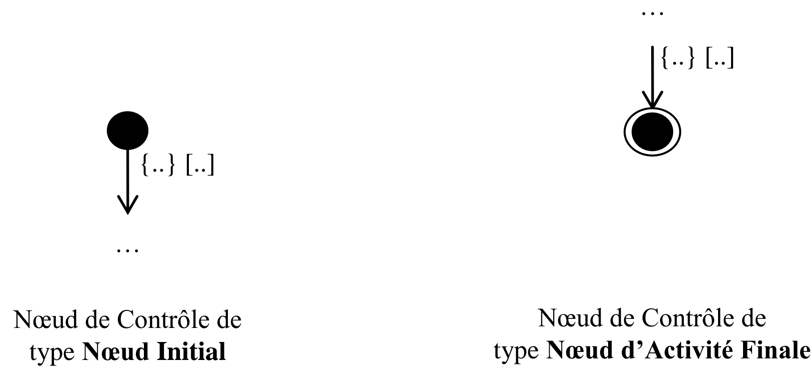


*Figure 4 : **Différents types de Nœuds de Contrôle***

Le nœud initial, correspondant au début des activités d'un diagramme d'activité, est représenté par un rond noir, et le nœud terminal par un rond et un cercle noirs. Tout diagramme d'activité débute par un nœud initial et se termine par un nœud d'activité finale.

La figure suivante montre un exemple de diagramme d'activité où les nœuds exécutables sont représentés par des rectangles, et où les nœuds de décision et de regroupement sont représentés par des losanges :

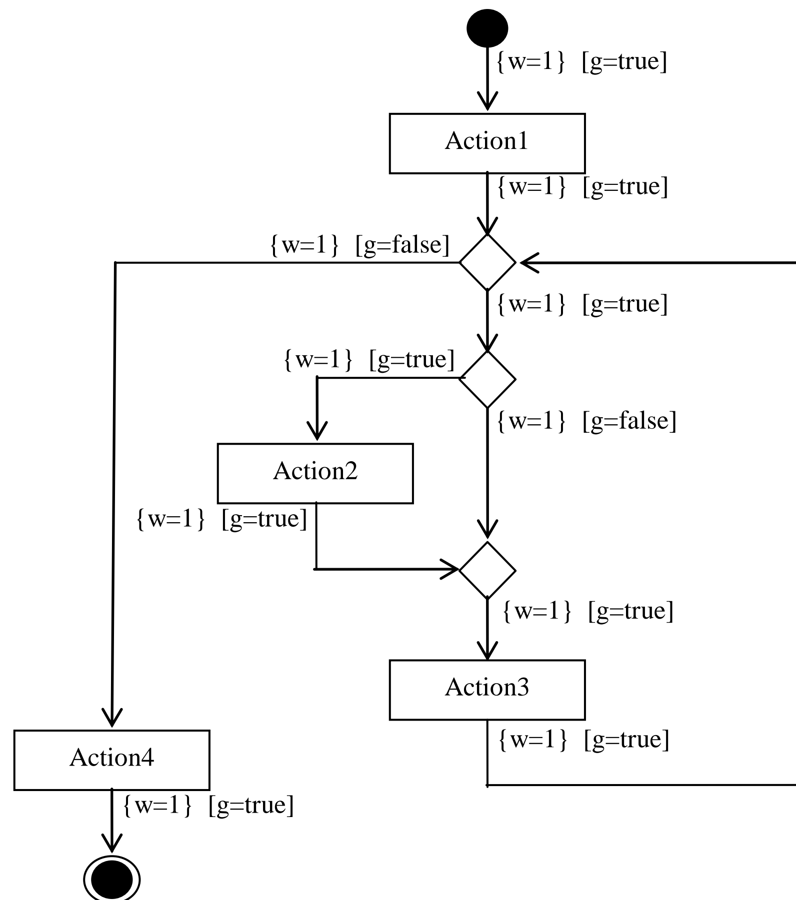


*Figure 5 : **Exemple d'un Diagramme d'Activité***

Cette figure montre, en particulier, que l'action Action1 sera la première action à être exécutée quand le diagramme d'activité recevra le contrôle. Selon la valeur donnée en sortie

de cette action, le contrôle de l'exécution sera ensuite passé à l'action Action4, ou au nœud de décision qui succèdent à cette première action.

II.1.d Diagramme d'Activité et Algorithme séquentiel

Le diagramme d'activité offre un très grand nombre d'éléments de modélisation qui en fait un outil extrêmement riche nécessitant une très grande expérience pour l'exploiter pleinement et de façon très efficace. Cependant, dans le cadre de cette étude, nous nous limitons à la modélisation d'algorithmes séquentiels nous permettant ainsi de réduire de façon très significative son utilisation. Si cette restriction contraint les Concepteurs à rester dans un cadre bien défini, cela permet, en contre partie, de mettre en place des règles, à l'aide de propriétés, vérifiant que les diagrammes ainsi élaborés restent bien dans ce cadre, préparant et anticipant du même coup le travail des Analystes/Programmeurs.

La figure suivante montre le cadre dans lequel les diagrammes d'activité devront rester :

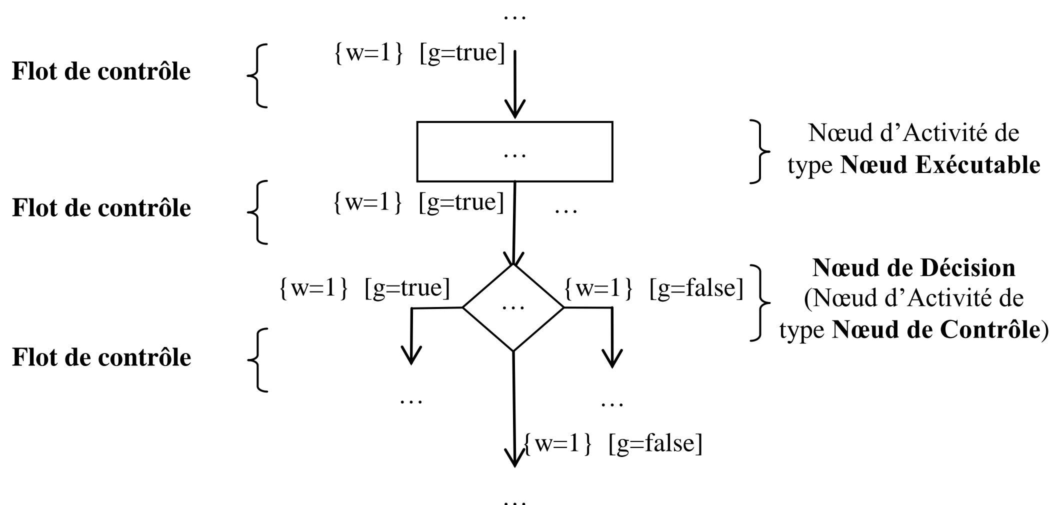


Figure 6 : ***Le poids w et la garde g des Flots de Contrôle***

Les paramètres des flots de contrôle permettent d'exprimer du parallélisme : le poids sera toujours égal à 1, indiquant qu'à la fin de l'exécution d'un nœud exécutable, le contrôle doit être transmis au nœud d'activité suivant, en fonction des gardes des flots sortants.

Tout nœud exécutable ne peut recevoir qu'un seul flot de contrôle, et peut être le point de départ d'un seul flot de contrôle. Leur garde est toujours égale à vrai (true).

La garde d'un nœud de contrôle traduit les branchements des langages de programmation : en sortie d'un nœud de décision il peut, bien sûr, y avoir plusieurs flots de contrôle en sortie, mais un seul ayant la valeur de la garde égale à 1, traduisant le fait que l'exécution du nœud de décision a pour effet de passer le contrôle directement (w=1) au nœud d'activité se trouvant en sortie du flot de contrôle dont la garde a pour valeur vrai (true).

II.2 Les différents types d'Actions Exécutables

Dans un diagramme d'activité, on peut trouver une très grande variété de types d'actions qui sont les éléments exécutables de modélisation. Nous nous limiterons aux actions les plus significatives, telles que des actions opaques qui sous-tendent du code dans un langage qui doit être précisé, des appels à des diagrammes d'activités. Les diagrammes UML peuvent dépendre les uns des autres, d'où l'existence d'associations entre certains éléments de modélisation appartenant à des types de diagrammes différents. Nous terminerons ce paragraphe en décrivant les nœuds d'activité structurée reprenant le principe des instructions de contrôle des langages impératifs.

II.2.a Appel à un Diagramme d'Activité

La figure suivante montre un fragment de diagramme d'activité référençant un diagramme d'activité :

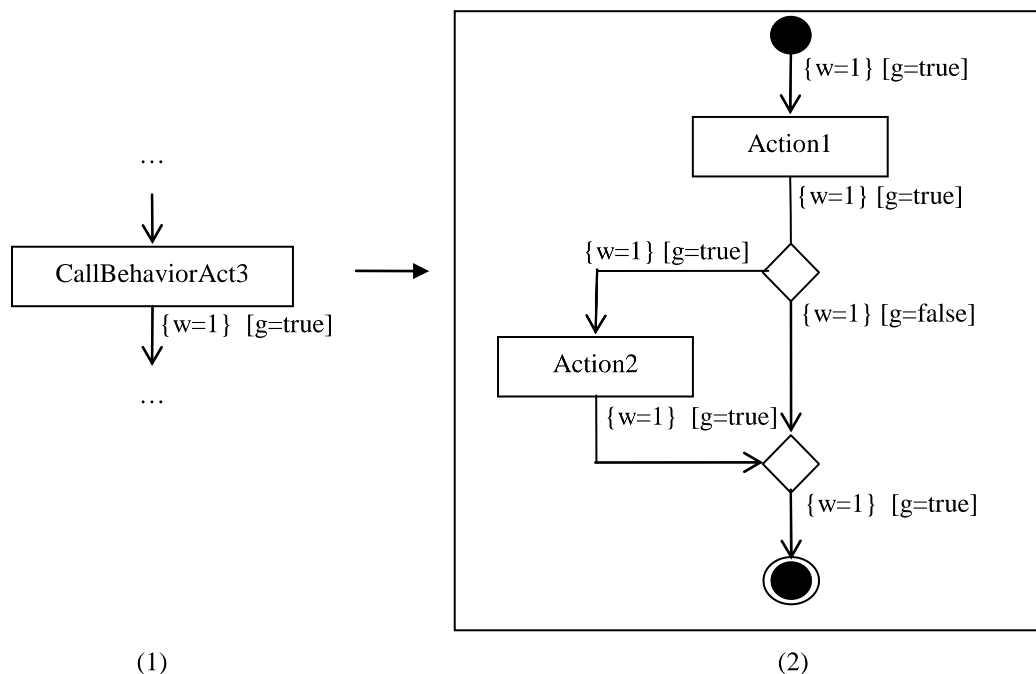


Figure 7 ***: Appel d'un diagramme d'activité (2) à partir d'un nœud exécutable de type CallBehaviorAction (1) défini dans un diagramme d'activité***

Le lien entre le nœud de nom CallBehaviorAct3 et le diagramme de droite n'apparait pas explicitement sur l'écran. En principe, un tel lien *est* géré par l'interface graphique qui, par exemple, fait apparaitre dans une fenêtre le diagramme d'activité si on clique sur le nœud CallBeaviorAct3.

II.2.b Diagramme d'Activité et Diagramme de Classes

Les différents diagrammes UML ne sont pas indépendants et peuvent, selon les cas, être interconnectés entre eux. En particulier, la méthode d'une opération peut être modélisée à l'aide d'un diagramme d'activité. La figure suivante montre un exemple de diagramme de classes et le diagramme d'activité modélisant la méthode f1 de l'opération de la classe C1 :

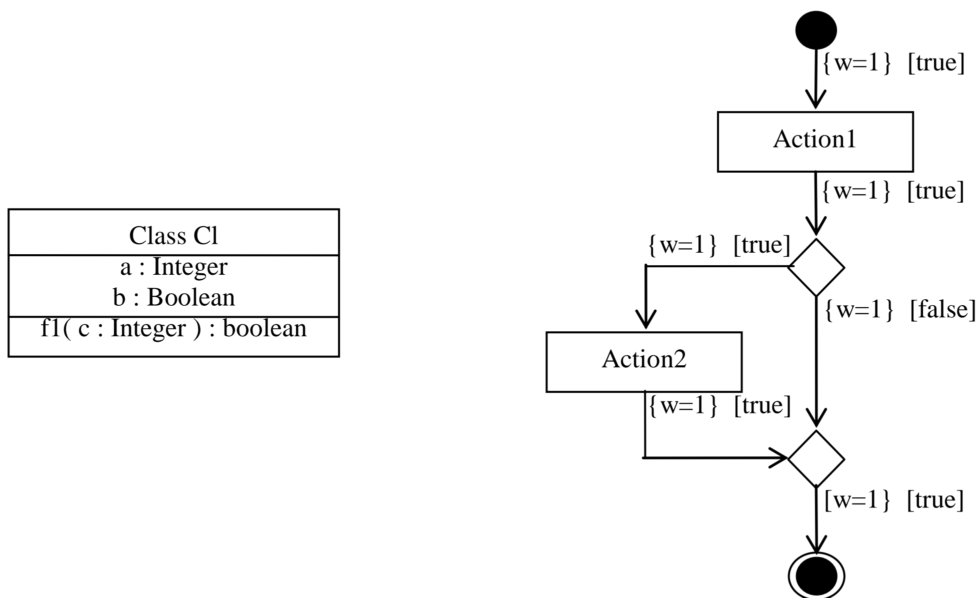


Figure 8 ***: diagramme d'activité modélisant la méthode d'une classe***

II.2.c Nœuds d'Activité Structurée

Les nœuds d'activité structurée (*StructuredActivityNode*) regroupent des actions similaires aux structures de contrôle classiques des langages de programmation telles que la séquence (SequenceNode), le test (ConditionalNode) et la boucle (LoopNode), chacune de ces actions étant reliées par des liens de composition aux nœuds structurés qui les définissent. Le diagramme d'activité suivant reprend l'exemple de la figure précédente, vu au travers de nœuds d'activité structurée dans la mesure où les poids des flots de contrôle ont, chacun, pour valeur 1, et que les différentes formes des nœuds de contrôle ont permis de transformer un tel diagramme en des structures de contrôle de séquence, de boucle et de test.

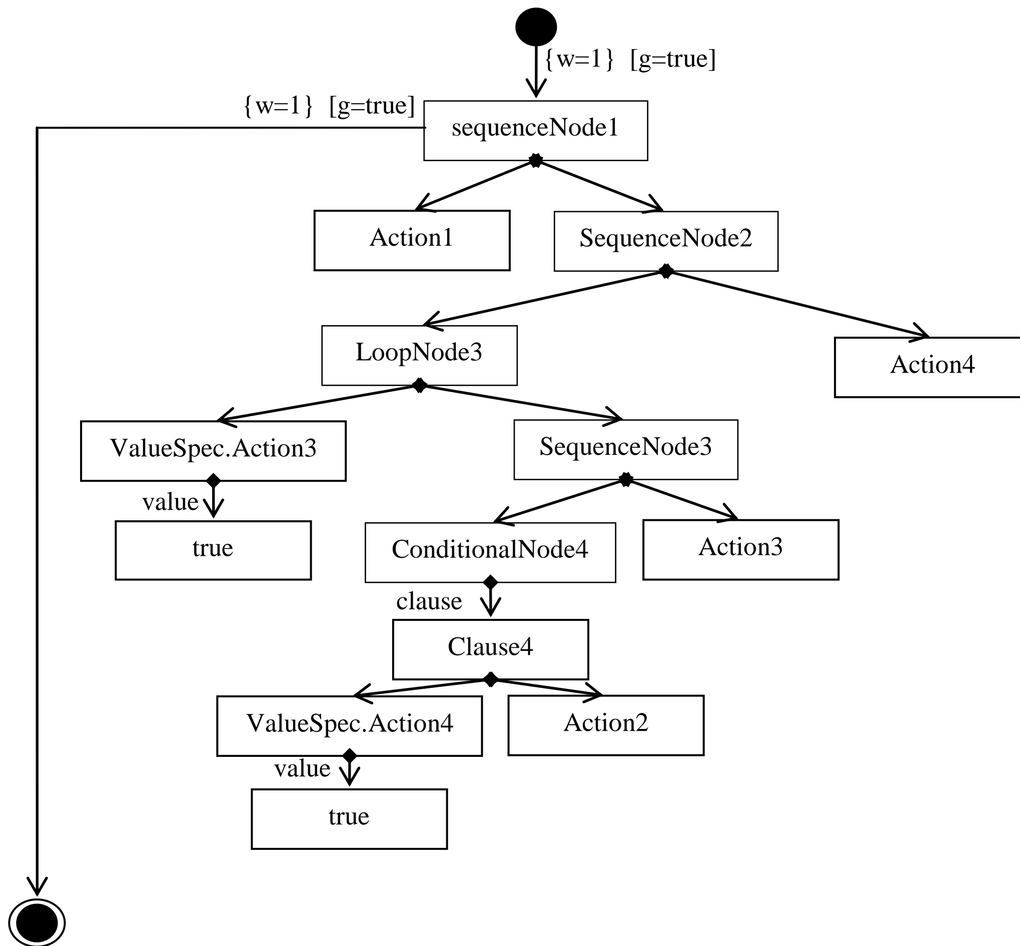


Figure 9 : ***Diagramme d'activité élaboré à l'aide de nœuds d'activité structurée***

II.2.d Action de mise à jour la valeur d'une variable (AddVariableValueAction)

Ce type d'action sert principalement à ajouter une valeur dans une variable (qui est potentiellement multiple et ordonnée). Cette action utilise les pins (InputPin) pour décrire la valeur qu'on souhaite affecter à une variable.

Les Pins sont des nœuds d'activité objet (*ObjectNode*) qui représentent les objets disponibles en entrée/sortie des actions d'une activité. Ces pins sont reliés à l'action par deux liens différents pour différencier leurs rôles (*input* ou *output)* et en plus, ces liens seront raffinés selon chaque type d'action exact. Dans ce cas de AddVariableValueAction, le pin d'entrée représentant la valeur à ajouter dans la variable est relié à l'action en tant que *value* ; et il doit avoir le même type avec la variable.

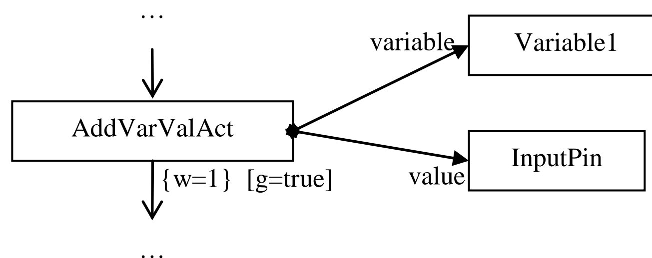


Figure 10 : ***Segment d'un diagramme d'activité montrant l'affectation d'une valeur à une variable***

III Fragment du méta-modèle UML 2.0 du diagramme d'activité

Dans ce paragraphe, nous rappelons la description des principaux éléments de modélisation du diagramme d'activité issue du Méta-Modèle UML 2.0, en nous limitant à des diagrammes décrivant des actions devant s'exécuter séquentiellement. En principe le Méta-Modèle UML est défini par l'OMG. C'est donc un standard, et il devait être inutile de les rappeler. De plus, si décrire une propriété à partir de sa représentation graphique est relativement facile, spécifier son expression OCL ou dans un langage d'actions UML au niveau de sa représentation interne peut être relativement complexe. En effet, dans le but justement d'être standard, le Méta-Modèle UML prend en compte des spécificités d'experts en Modélisation dans des domaines métiers qui rend difficile la compréhension de certains fragments du Modèle, surtout si l'on n'en connait pas les raisons.

Les exemples de diagramme d'activité décrits au paragraphe précédent font apparaître trois types de symboles qui sont : les rectangles représentant les actions exécutables du système à développer, les losanges permettant de définir des conditions préalables à l'exécution des actions et les arcs définissant d'une part les interactions entre les actions et les modalités de passage d'une action à une autre lorsque les actions peuvent s'exécuter parallèlement. Il existe en plus le disque noir, pour le nœud initial, qui démarrera l'exécution du diagramme d'activité lorsqu'il recevra le contrôle et qui se terminera au niveau du nœud d'activité finale représenté par un disque noir inscrit dans un cercle.

Dans ce paragraphe nous décrivons le fragment du méta-modèle UML 2.0 concernant le diagramme d'activité en partant des actions et des nœuds de contrôle. Nous déclinerons ensuite les différents types d'actions les plus importantes pour modéliser des actions séquentielles et nous replacerons les différents éléments de modélisation du diagramme d'activité en lien avec le fragment du méta-modèle du diagramme de classes où nous pourrons trouver les propriétés des attributs qui pourront être référencés dans le diagramme d'activité lors du raffinage des actions.

III.1 Nœuds Exécutables, Nœuds de Contrôle et Flots de Contrôle

III.1.a Fragment de Méta-Modèle du Diagramme d'activité

La figure suivante montre comment les nœuds exécutables (*ExecutableNode*), les nœuds de contrôle (*ControlNode*) et les flots de contrôle (ControlFlow) sont définis dans le Méta-Modèle UML 2.0 à partir de la méta-classe Activity qui permet de référencer un diagramme d'activité :

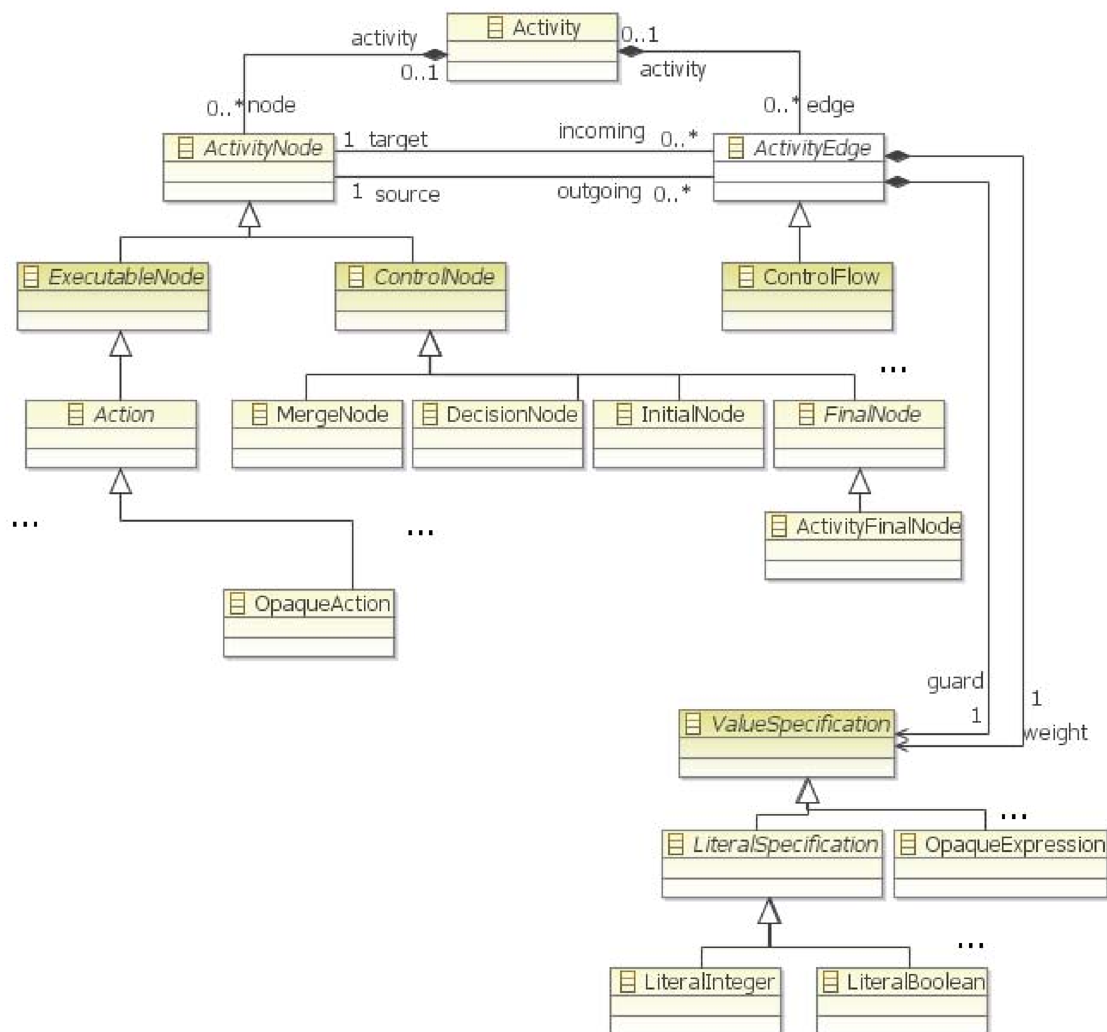


Figure 11 : ***Les nœuds exécutables, les nœuds de contrôle et les flots de contrôle dans le Méta-Modèle UML 2.0***

Au plus haut niveau d'abstraction, le fragment du Méta-Modèle UML décrit la structure d'un diagramme d'activité comme un ensemble de nœuds d'activité (*ActivityNode*) et de flots de contrôle (*ControlFlow*) qui définissent les modalités d'exécution entre les nœuds d'activité.

Les nœuds d'activité regroupent les nœuds exécutables (*ExecutableNode*), c'est-à-dire les actions (*Action*) du système à modéliser dont l'enchaînement est défini par les flots de contrôle et les nœuds de contrôle (*ControlNode*) qui peuvent être des nœuds de décision (*DecisionNode*) à l'origine de plusieurs branches de nœuds d'activité, et des nœuds de regroupements de plusieurs branches (*MergeNode*). Un nœud de décision est donc à l'origine de plusieurs branches de nœuds d'activité dont les conditions d'exécution (*ValueSpecification*) doivent apparaître au niveau des flots de contrôle sortants correspondants.

Ce Méta-Modèle montre, en particulier, qu'un nœud exécutable peut être, par exemple, une action opaque (OpaqueAction) ou un nœud de contrôle (*ControlNode*) de type nœud de décision (DecisionNode), entre autre. A un certain niveau de raffinage, le nœud exécutable se transformera en une instruction du langage de la plate-forme cible.

III.1.b Représentation des flots de contrôle

La figure suivante montre un exemple de fragment de diagramme d'activité représentant deux actions consécutives reliées par un flot de contrôle :

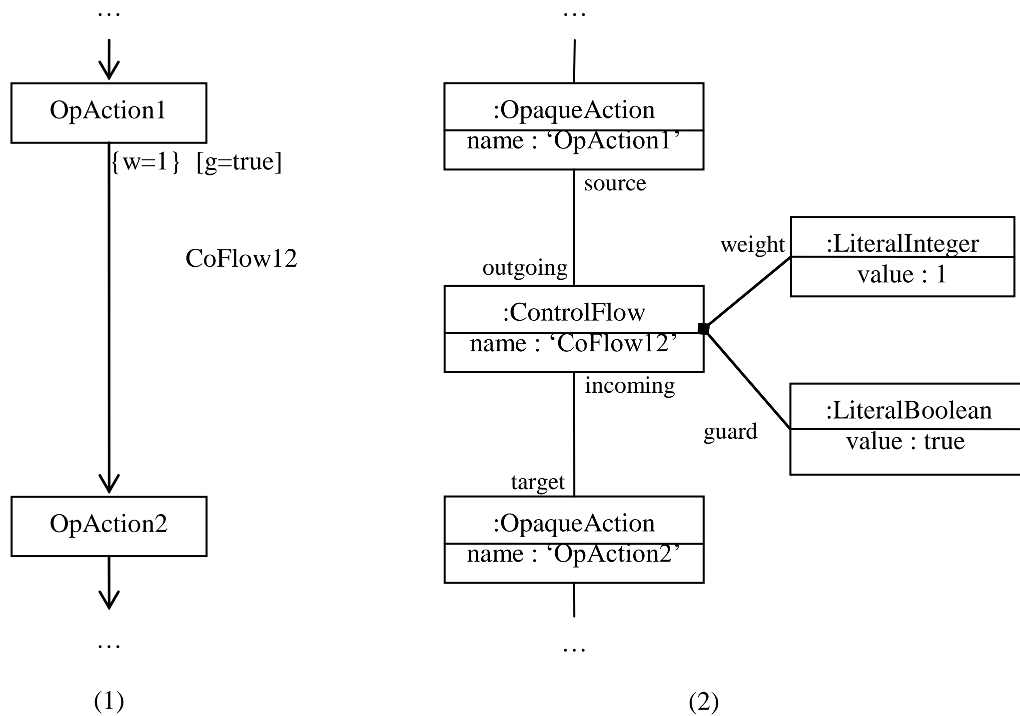


Figure 12 : ***Exemple d'un fragment d'un Diagramme d'Activité enchaînant deux Actions***

Cette figure montre la représentation graphique du flot de contrôle (1), et sa représentation interne (2).

III.1.c Représentation des Nœud de Contrôle de type Nœud de Décision (DecisionNode)

Un exemple de fragment de diagramme d'activité contenant un nœud de décision initiant deux branches de nœuds d'activité (1), et de sa représentation interne (2), instance du Méta-Modèle UML 2.0 pourrait être le suivant :

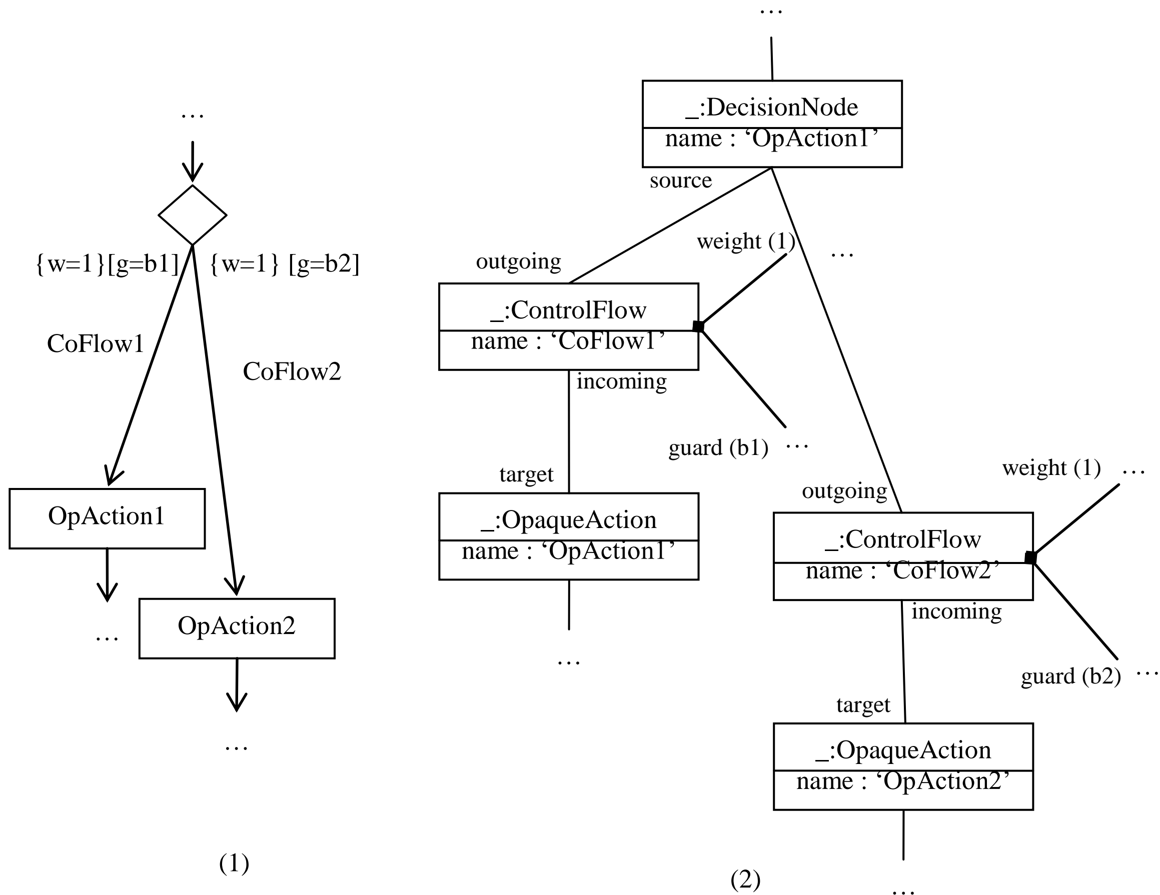


Figure 13 : ***Exemple d'un fragment d'un diagramme d'activité contenant un nœud de décision***

Cette figure montre la représentation graphique (1) et la représentation interne (2) d'un fragment de diagramme d'activité contenant un nœud de contrôle de type décision, où il peut exister plusieurs flots de contrôle sortants. La représentation graphique et interne d'un nœud de contrôle de type regroupement s'en déduit selon le même principe, sachant que plusieurs flots de contrôle peuvent y arriver.

III.2 Représentation des différents types d'Actions

On décrit, dans ce paragraphe, quelques types d'actions que l'on pourrait rencontrer couramment dans un diagramme d'activité.

III.2.a Fragment du Méta-Modèle UML détaillant quelques types d'actions

La figure suivante montre le fragment du Méta-Modèle UML 2.0 définissant quelques différents types d'actions :

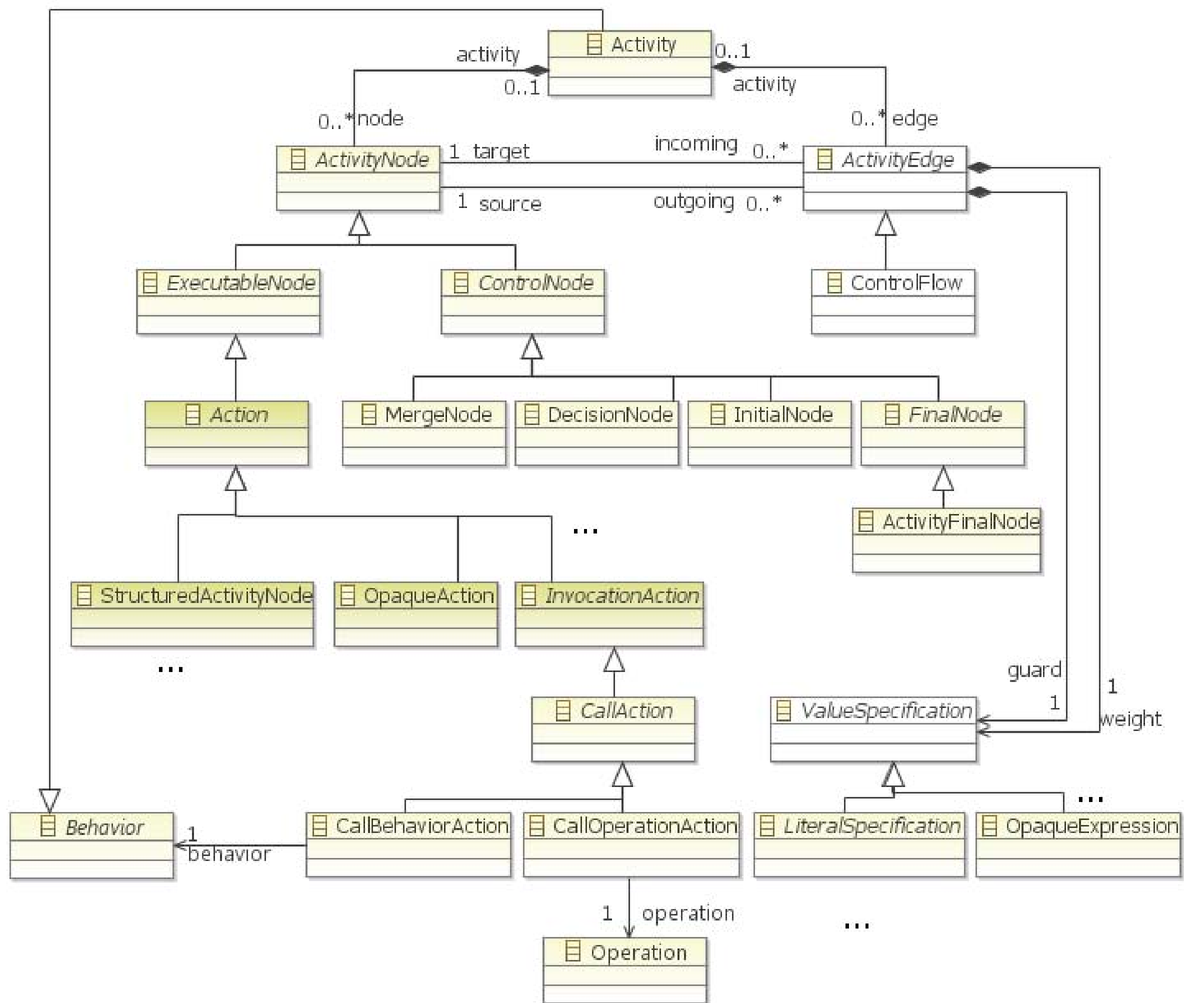


Figure 14 : ***Les différents types d'Actions du Méta-Modèle UML 2.0***

III.2.b Appel à un Diagramme d'Activité

La figure suivante montre un fragment de diagramme d'activité référençant un algorithme modélisé à l'aide d'un diagramme d'activité :

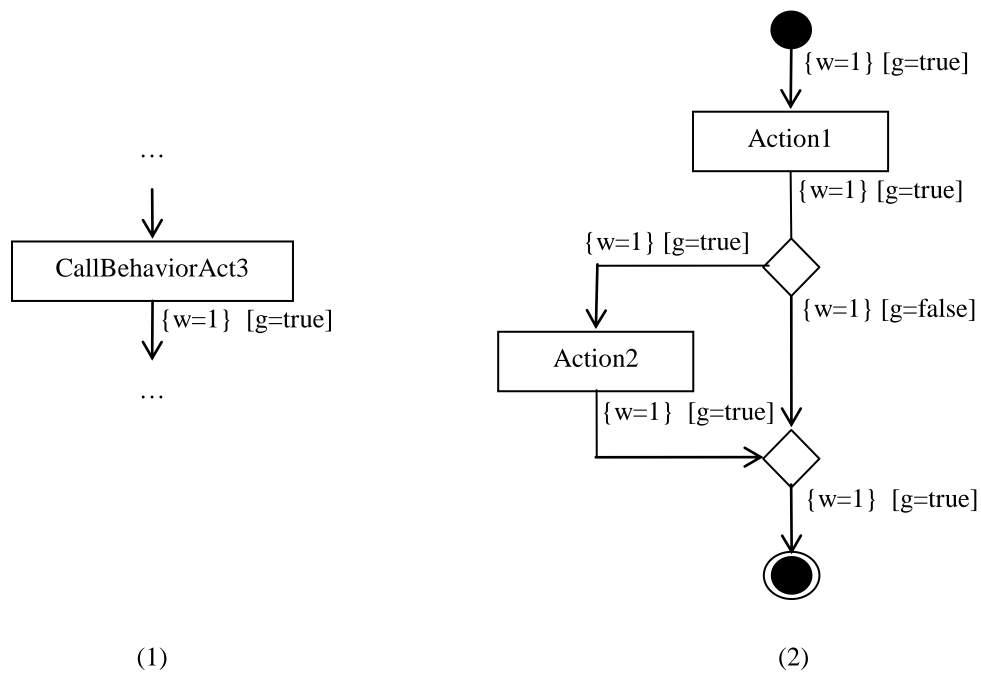


Figure 15 ***: Appel d'un diagramme d'activité à partir d'un nœud exécutable de type CallBehaviorAction défini dans un diagramme d'activité***

La figure suivante détaille la représentation interne des liens permettant à une action de référencer un diagramme d'activité :

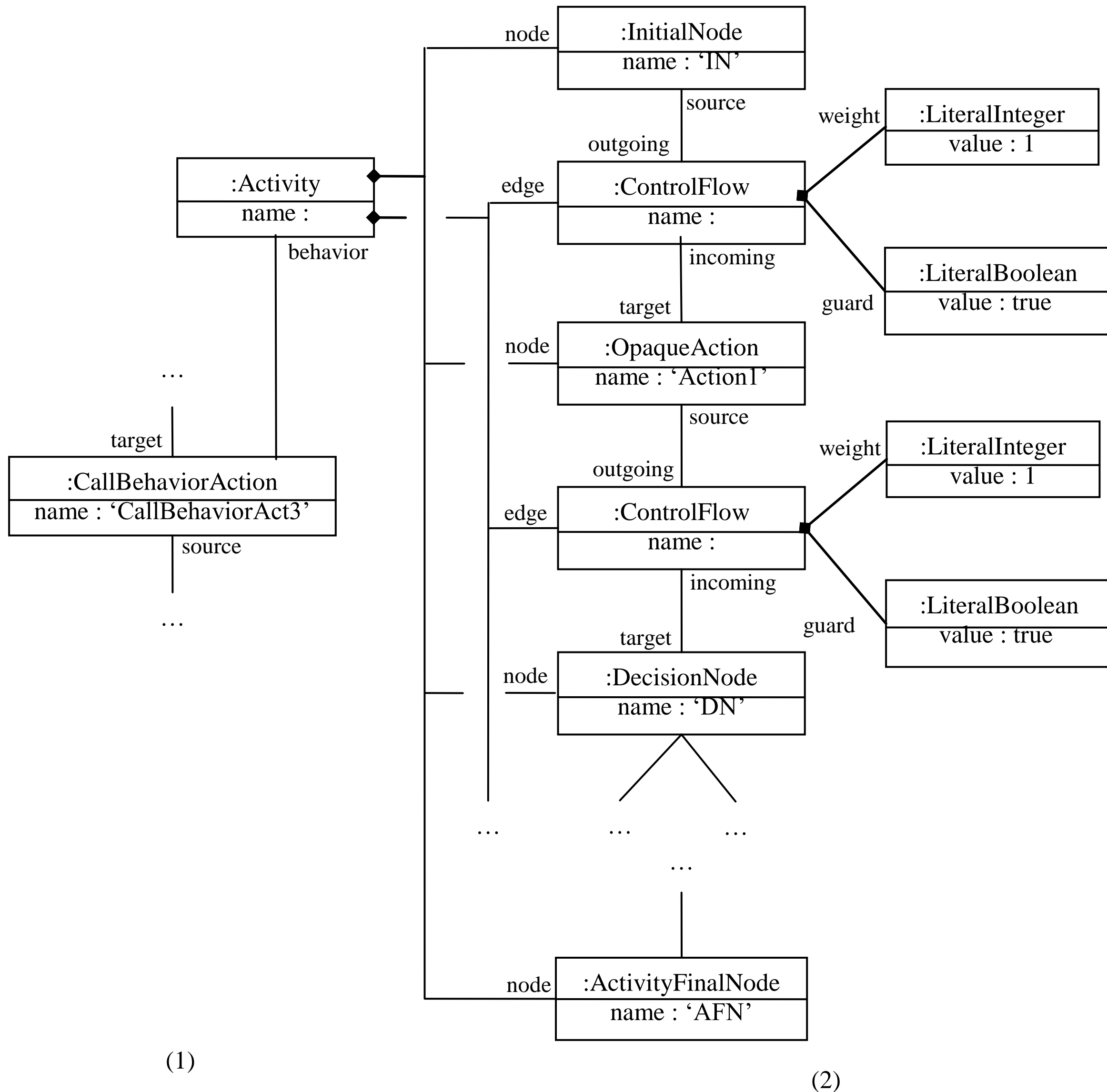


Figure 16 : ***Représentation interne (Instance du Méta-Modèle UML 2.0) d'une action référençant un diagramme d'activité***

III.2.c Action de mise à jour la valeur d'un variable

La figure suivante montre le fragment du Méta-Modèle UML décrivant l'affectation d'une valeur à une variable :

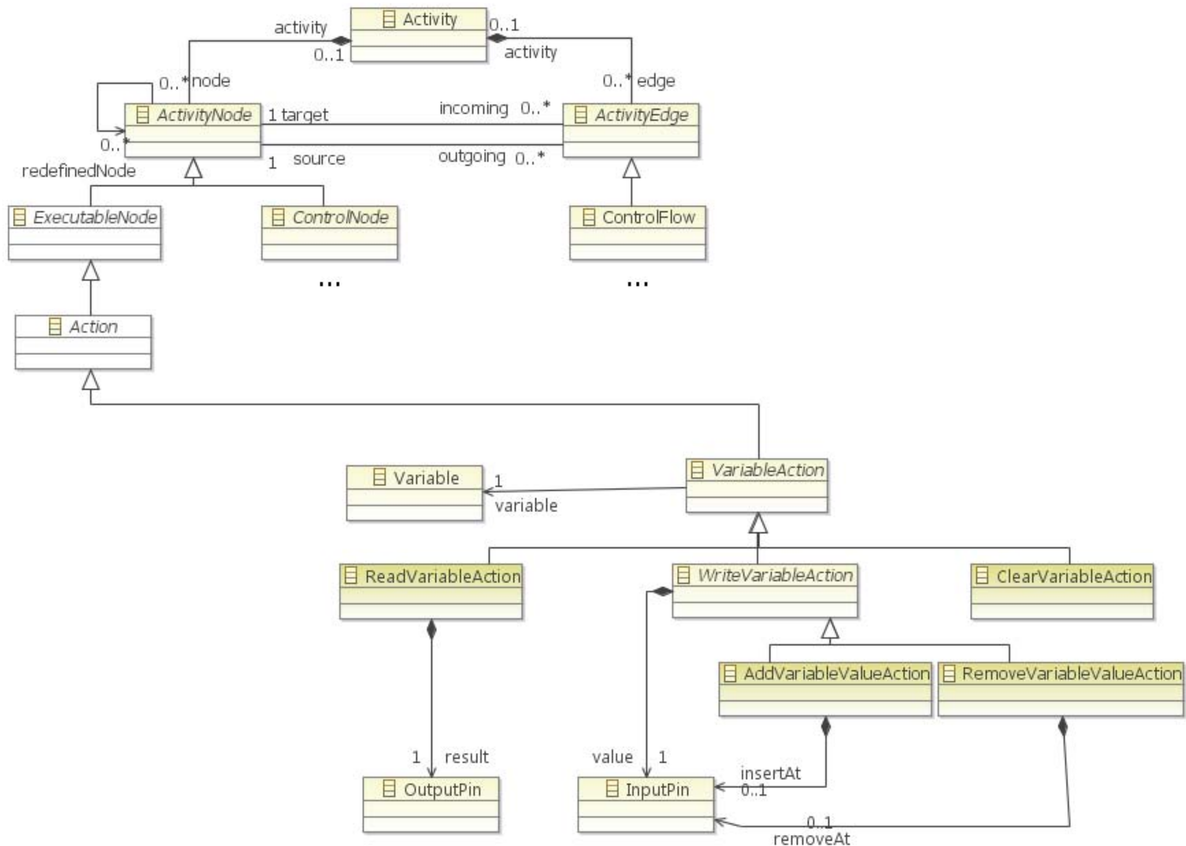


Figure 17 : ***Actions sur les Variables dans le fragment du Méta_Modèle UML 2.0 du Diagramme d'Activité***

On reprend l'exemple décrivant l'affectation d'une variable en montrant dans les deux figures suivantes, leurs représentations graphique et interne :

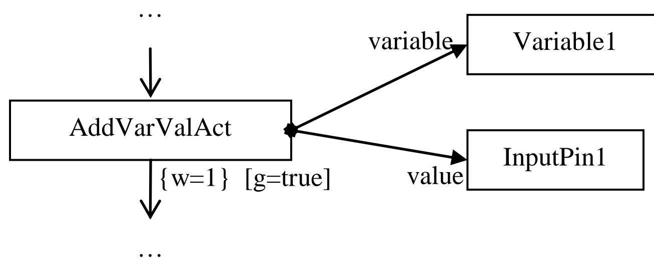


Figure 18 : ***Fragment d'un diagramme d'activité montrant une action de type AddVariableValueAction***

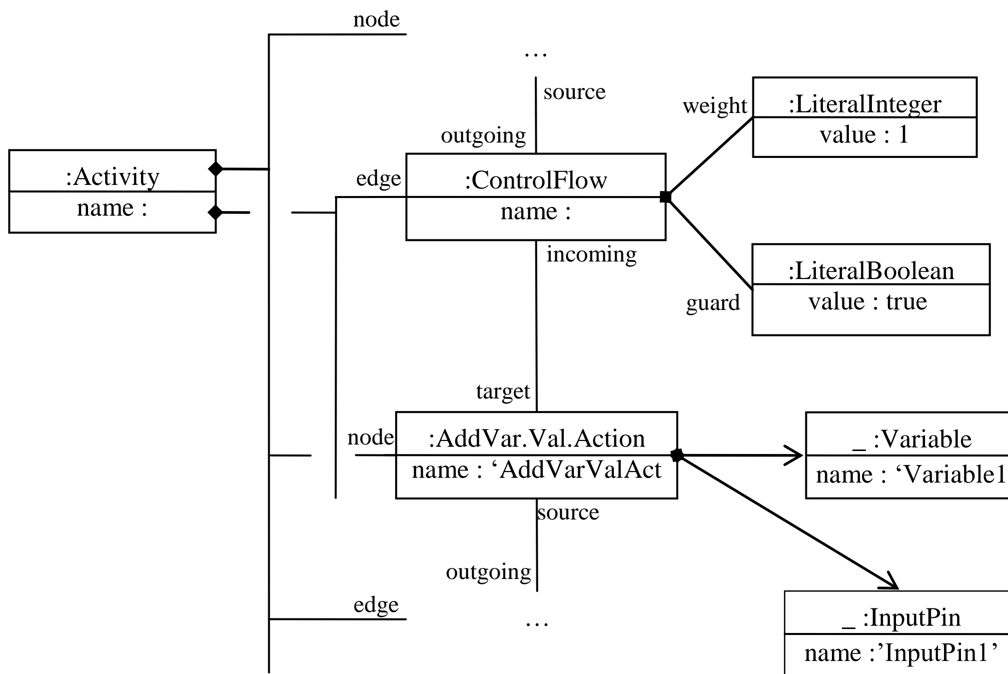


Figure 19 ***: Représentations internes d'un fragment de Diagramme d'Activité référençant une action de type AddVariableValueAction***

III.2.d Action et Expressions Opaques

Une action opaque (OpaqueAction) représente, à un certain niveau d'abstraction, une action qui, au niveau d'abstraction où se trouve le diagramme d'activité, n'a pas besoin d'être précisée. Il en est de même pour les expressions opaques (OpaqueExpression).

III.2.e Noeuds d'Activité Structurée

Un dernier type d'actions, très important dans le cadre de notre étude, concerne les nœuds d'activité structurée (*StructuredActivityNode*) qui seront utilisés dans une étape préliminaire à l'activité de génération de code d'un diagramme d'activité. Ils permettent, en particulier, de modeler le diagramme d'activité selon une structure arborescente, tout en restant conforme au Méta-Modèle UML 2.0. Les Analystes/Concepteurs pourront donc voir, au niveau de cette étape préliminaire, si les flots de contrôle reliant les différentes actions de leurs modèles peuvent représenter des structures arborescentes, ce qui est une des premières exigences (minimale !) des langages de programmation.

La figure suivante montre les différents types de nœuds d'activité structurée et leurs liens avec les autres éléments de modélisation du diagramme d'activité dans le Méta-Modèle UML 2.0 :

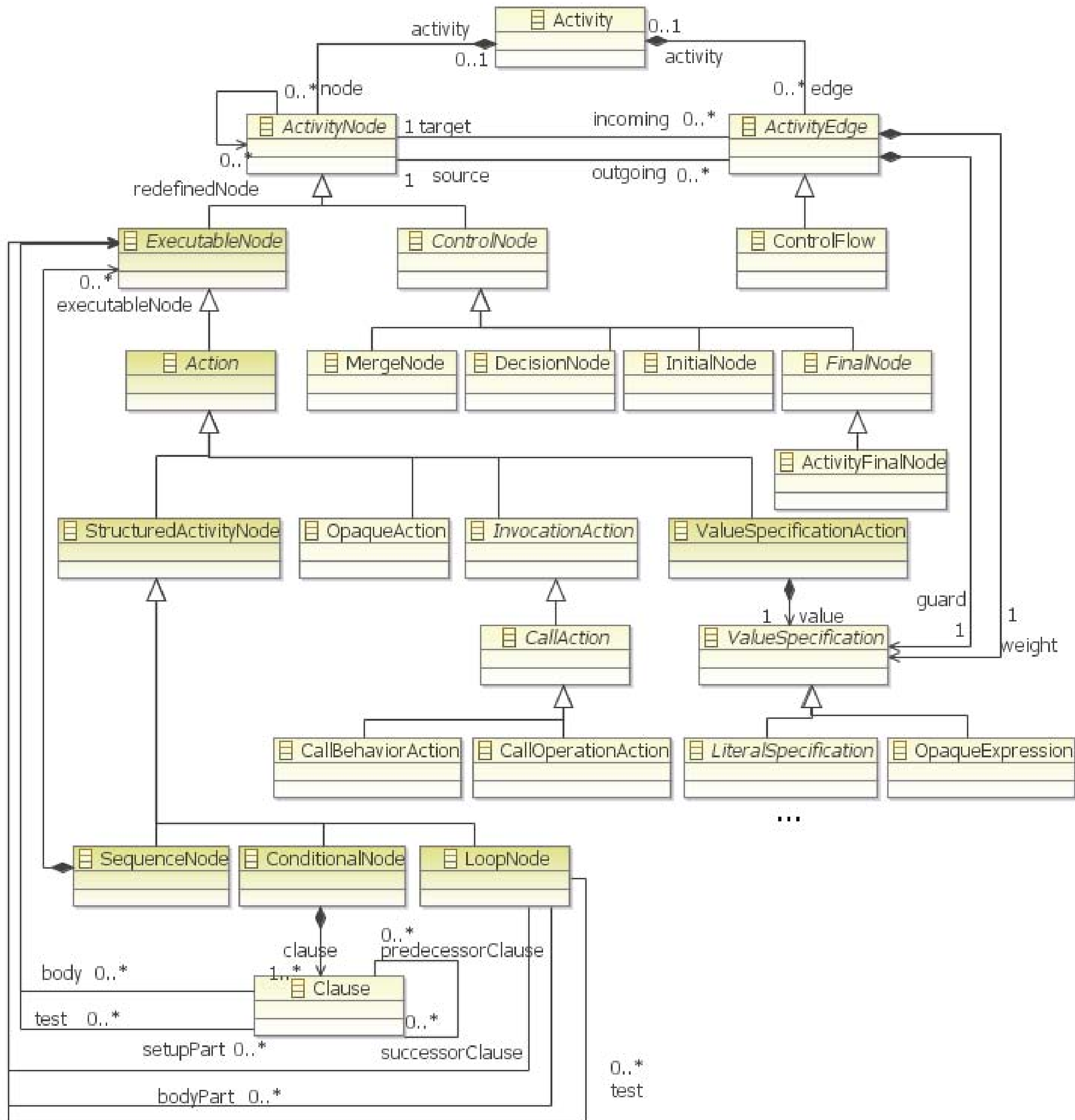


Figure 20 : ***Les nœuds d'activité structurée dans le fragment du méta-modèle UML 2.0 du diagramme d'activité***

En fait, ce fragment de Méta-Modèle UML 2.0 du diagramme d'activité fait apparaître les éléments de modélisation du diagramme d'activité nécessaire à l'étude que nous nous sommes fixés. On peut remarquer, en particulier, qu'un nœud d'activité structurée (*StructuredActivitydNode*) de type SequenceNode peut être en lien direct de composition avec plusieurs nœuds exécutables (*ExecutableNode*), comme une structure de contrôle séquentielle d'un langage de programmation. Les nœuds d'activité entrant dans la composition d'un nœud de séquence ne sont pas liés par des flots de contrôle. Un nœud conditionnel (ConditionalNode), synonyme d'une instruction de tests, est composé de plusieurs clauses (*Clause*). Chaque clause associée à un ou plusieurs tests de type ExécutableNode devant logiquement référencer une expression opaque (OpaqueExpression) dépendant, comme pour une action opaque, du langage de la plate-forme cible, et à un corps qui est logiquement un ensemble d'actions devant être exécuté si le test correspondant est vrai. On retrouve

l'instruction de test avec plusieurs tests, chacun définissant un ensemble de nœuds exécutables si le test est vrai. Enfin, le nœud d'activité structurée de type LoopNode est, bien sûr, l'équivalent d'une instruction de boucle.

La figure suivante montre un exemple de nœud d'Activité Structurée de type LoopNode :

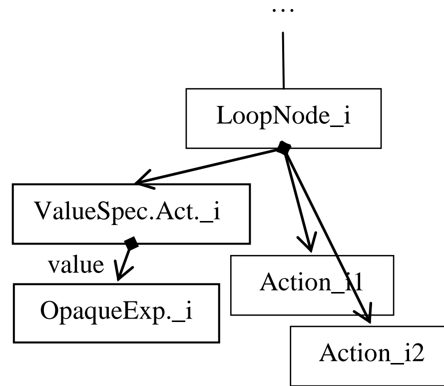
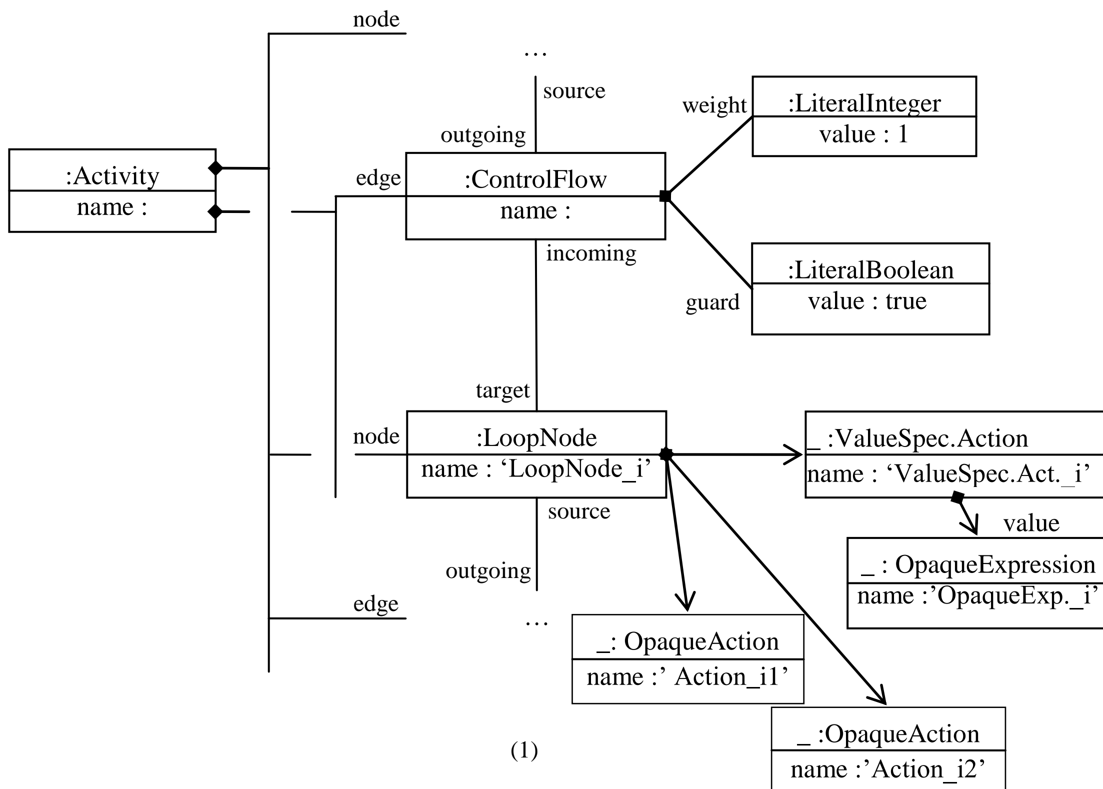


Figure 21 : ***Fragment d'un diagramme d'activité montrant un nœud d'activité structurée de type LoopNode***

Le nœud d'activité structurée LoopNode_i peut se trouver en sortie d'un flot de contrôle ou peut se trouver déjà en lien de composition d'un nœud d'activité structurée. Selon le cas, la représentation interne d'un tel fragment sera différente. Les deux figures suivantes montrent les deux représentations internes possibles. La figure suivante montre que la boucle reçoit un flot de contrôle. De cette boucle, part un flot de contrôle :



La figure suivant montre que la boucle est rattachée directement à un noeud d'activité structurée.

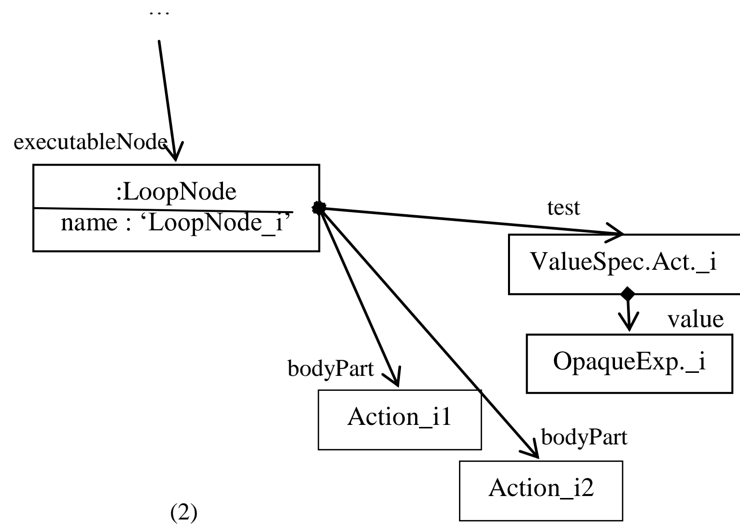


Figure 22 ***: Représentations internes d'un fragment de Diagramme d'Activité référençant un nœud de type LoopNode***

III.2.f Diagramme de classes et diagramme d'activité

La figure suivante montre donc le fragment du Méta-Modèle UML 2.0 montrant les liens entre un diagramme d'activité (Activity) modélisant une opération (Operation) d'une classe (Class) définie par un certain nombre de propriétés caractérisées par des attributs (*StructuralFeature*) typés (*Type*) :

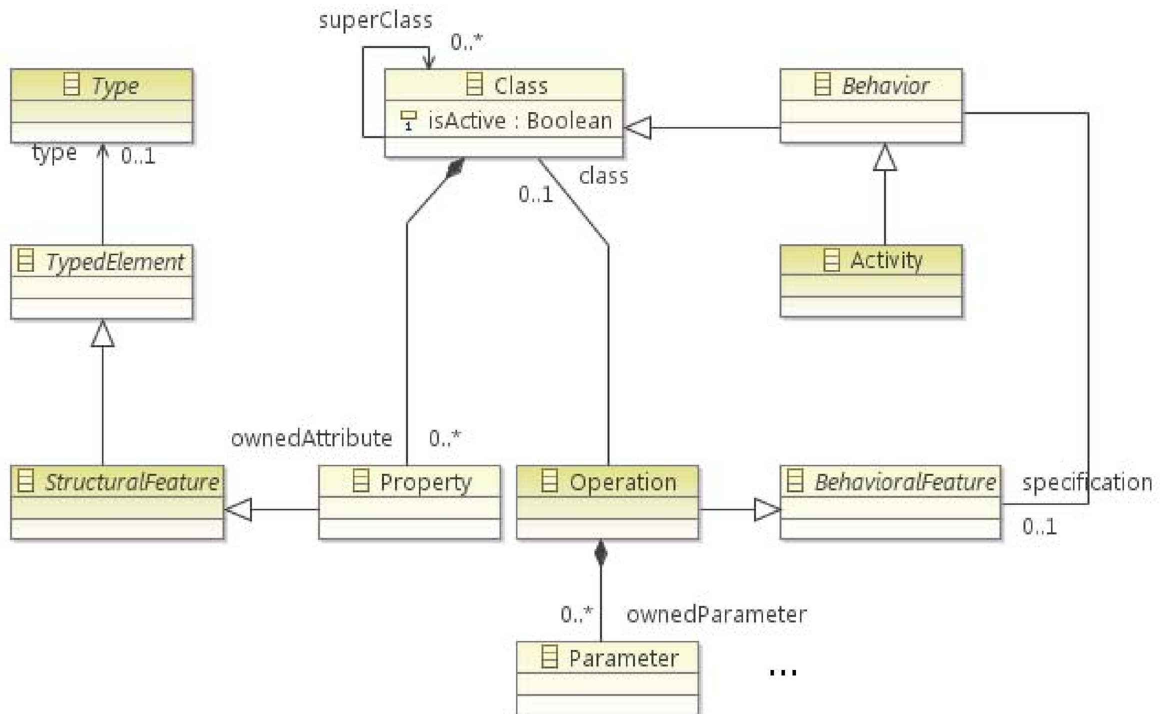


Figure 23 : ***Fragment du Méta-Modèle UML 20 montrant les liens existant entre un Diagramme de Classes et le Diagramme d'Activité***

La figure suivante montre un exemple de diagramme de classes et le diagramme d'activité modélisant la méthode f1 de l'opération de la classe C1 :

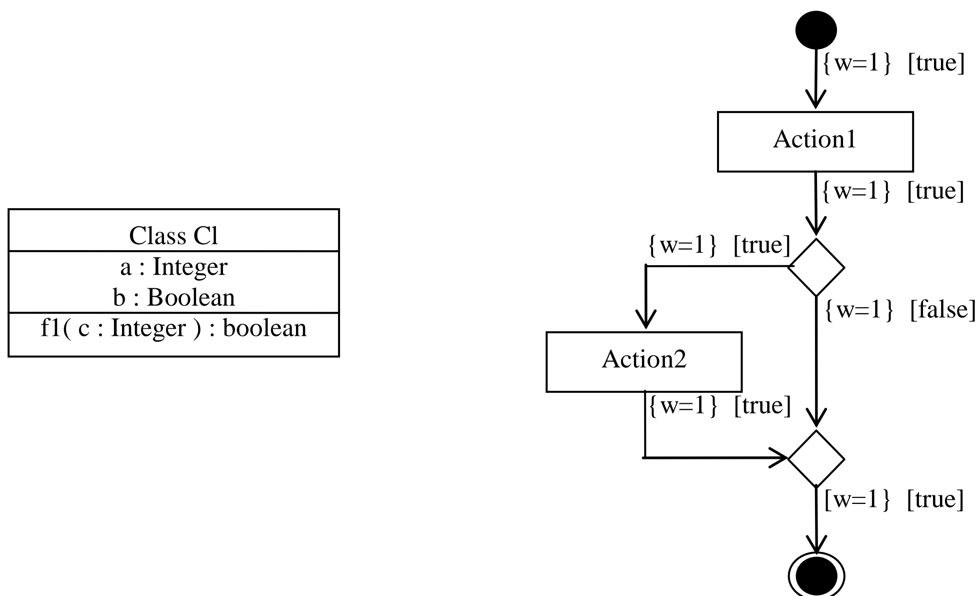


Figure 24 : ***Diagramme d'Activité modélisant la méthode d'une classe***

La figure suivante donne une représentation interne (Instance du Méta-Modèle UML 2.0) du modèle au niveau de la figure précédente :

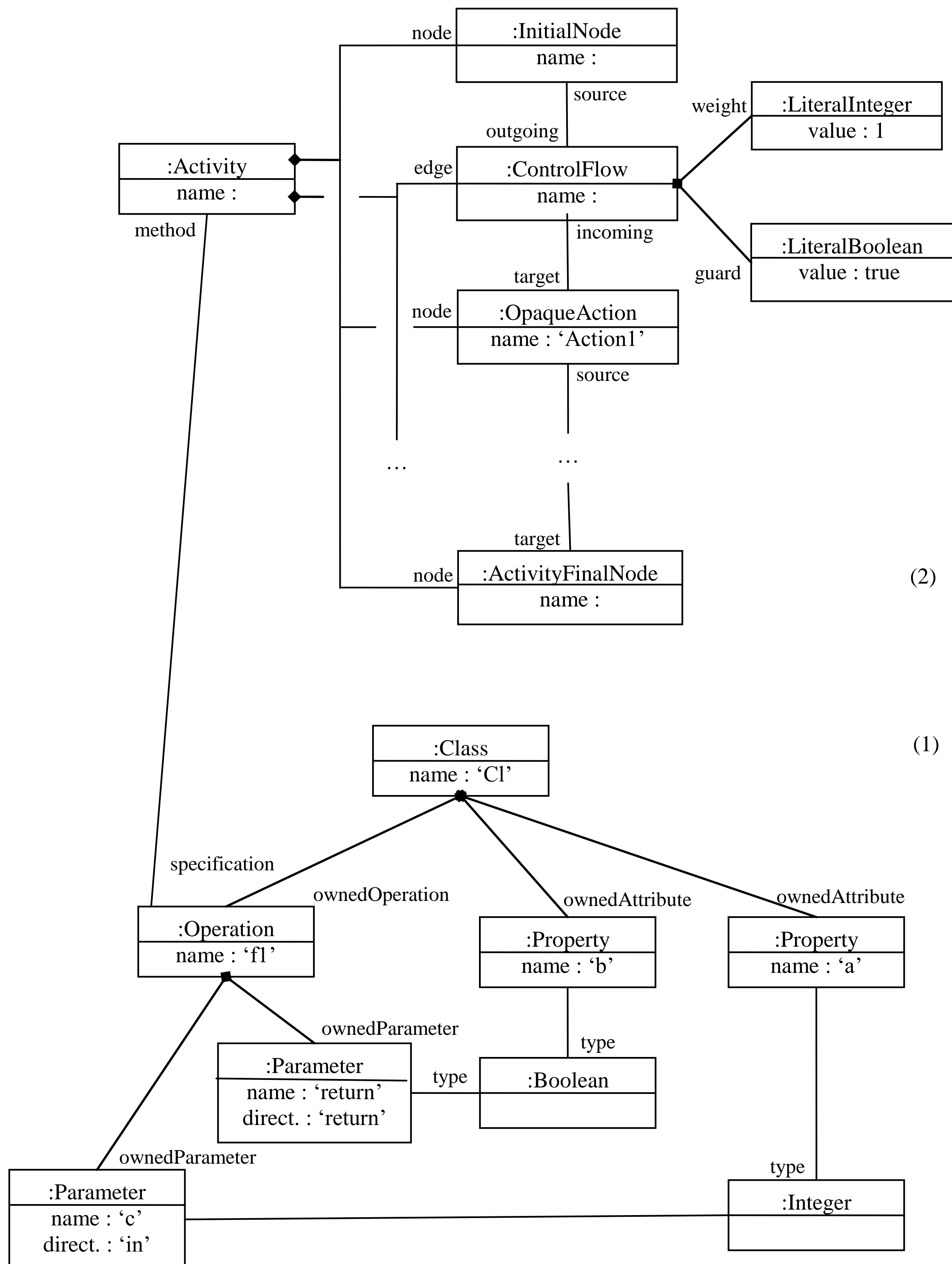


Figure 25 : ***Représentation interne (Instance du Méta-Modèle UML 2.0) du diagramme de classes (1) et du diagramme d'activité (2)***

La présentation des principaux éléments de modélisation du fragment du Méta-Modèle UML permettant de modéliser des algorithmes séquentiels de haut niveau à l'aide d'un

diagramme d'activité étant faite, nous décrivons dans le paragraphe suivant les expressions OCL spécifiant des propriétés comportementales et axiomatiques.

IV Modélisation de Propriétés Syntaxiques et Sémantiques en UML/OCL applicables au Diagramme d'Activité

Nous avons défini des propriétés comportementales et axiomatiques au niveau du Méta-Modèle du langage L, nous pouvons, de la même manière, définir des propriétés comportementales et axiomatiques applicables aux diagrammes d'activités. Ces propriétés assureraient que ces diagrammes ont bien les qualités requises à ce niveau de modélisation, et permettraient de vérifier la cohérence entre les modèles de conception et les modèles de codes. Cela signifie, tout d'abord, que l'on doit modifier le MM UML, ce qui, a priori, n'est pas recommandé pour les raisons qui ont été évoquées au chapitre I.

Cependant, au cours de ce paragraphe, nous proposons une sémantique du diagramme d'activité en injectant des propriétés comportementales et axiomatiques au niveau du MM UML, et nous verrons plus tard les solutions évitant d'altérer le MM UML.

Ces propriétés sont inspirées des propriétés des langages de programmation, mais elles devraient rester plus générales et devraient pouvoir intégrer des spécificités métiers.

IV.1 Modélisation en UML/OCL de Propriétés Syntaxiques et Comportementales

Dans ce paragraphe, nous cherchons à définir des propriétés syntaxiques et comportementales. Ces propriétés insérées au niveau du fragment du Méta-Modèle UML concernant le diagramme d'activité, devraient, en principe, être définies par les Experts des différents domaines d'application obligeant ainsi les diagrammes d'activité, élaborés par les Analystes/Concepteurs, à respecter les propriétés qui leur sont imposées.

IV.1.a Modélisation en UML/OCL des Propriétés Syntaxiques pour un Diagramme d'Activité

On suppose que tout éditeur UML effectue un certain nombre de vérifications par rapport aux aspects graphiques des différents éléments de modélisation des diagrammes. Dans ce paragraphe, on définit les propriétés qui vérifient que tout diagramme d'activité reste bien dans le cadre que l'on s'est fixé. On doit donc définir les propriétés syntaxiques assimilant ainsi un diagramme d'activité à des formules bien formées, respectant à la fois les propriétés structurelles définies au niveau UML et les propriétés tenant compte de notre contexte d'études.

Ces propriétés ont été rappelées plus haut :

Dans notre contexte :

- p1 : Le poids de tout flot de contrôle doit être égal à 1.
- p2 : Tout nœud exécutable ne peut recevoir qu'un seul flot de contrôle, et peut être le point de départ d'un seul flot de contrôle.
- p3 : La garde de tout flot de contrôle en sortie d'un nœud exécutable est égale à la valeur vrai.

- p4 : Un seul flot de contrôle en sortie d'un nœud de décision doit être égal à vrai (true).
- p5 : Le nœud initial ne reçoit pas de flot de contrôle entrant, et n'a qu'un seul flot de contrôle sortant.
- …

Ces propriétés syntaxiques qui complètent le fragment du MM UML concernant les éléments du diagramme d'activité sont donc en OCL les suivantes, sachant que tout diagramme d'activité est référencé par un objet de type Activity :

```
context Activity

inv p1 : self.edge->forAll( ae |
          ae.oclIsTypeOf( ControlFlow ) implies
                          ae.oclAsType( ControlFlow ).weight.oclAsType( LiteralInteger ).value = 1
                         )

inv p2 : self.node->forAll( an |
          an.oclIsKindOf( ExecutableNode ) implies        an.incomming->size() = 1        and
                                                          an.outgoing->size() <= 1
                          )

inv p3 : self.node->forAll( an |
          an.oclIsKindOf( ExecutableNode ) implies
                an.outgoing->select( fc |
                          fc.oclIsTypeOf( ControlFlow
                                    )->asSequence()->at( 1 ). guard.oclAsType( LiteralBoolean
                                                                            ).value = true
                          ).

inv p4 : self.node->forAll( an |
          an.oclIsTypeOf( ControlDecisionNode ) implies
                          an.outgoing->select( fc |
                                fc.oclIsTypeOf( ControlFlow )              and
                                fc.oclAsType( ControlFlow ).guard.oclAsType(LiteralBoolean ).value = True
                                      )->size() = 1
                            )

inv p5 : self.node.oclIsTypeOf( InitialNode ) and      ( self.node.incoming->size() = 0 and
                                                        self.node.outgoing->size() = 1        )

…
```

Figure 26 : ***Propriétés syntaxiques portant sur les Diagrammes d'Activité modélisant des algorithmes***

IV.1.b Environnement d'exécution pour le fragment du Diagramme d'Activité du MM UML

On rappelle que la spécification des propriétés comportementales se définit par rapport à un comportement attendu des différentes constructions syntaxiques du langage sur un environnement d'exécution de référence. Cet environnement de référence associe à chaque variable du programme la valeur (de référence) qui est calculée au cours de l'exécution du programme.

Il en est de même pour définir les propriétés comportementales des diagrammes d'activité, ce qui est relativement facile dans notre contexte puisqu'un algorithme est décrit à

l'aide de variables, et d'opérations sur ces variables. L'environnement d'exécution définissant les propriétés comportementales et axiomatiques du diagramme d'activité est donc défini de la même manière que pour les propriétés du langage « L », tel que le montre la figure suivante en reprenant les notations des grammaires utilisées au chapitre IV :

EnvExecDA →	ensMem : Mémoire*		
Mémoire →	Variable,	ValueSpecification	
[*ValueSpecification* →	OpaqueExpression	| *LiteralSpecification*	
LiteralSpecification →	LiteralNul	| LiteralString	| LiteralInteger | LiteralBooelan
LiteralNul →	value : Integer		
LiteralString →	value : String		
LiteralInteger →	value : Integer		
LiteralBooelan →	value : Boolean		
]			

Figure 27 : ***Environnement d'exécution définissant le comportement de référence attendu du Diagramme d'Activité***

Cet environnement est défini à l'aide des deux premières règles de construction syntaxique de la grammaire, les autres règles exprimant les éléments de modélisation du fragment du MM UML définissant les types prédéfinis du MM UML.

IV.1.c Modélisation des propriétés comportementales d'un Diagramme d'Activité

On peut trouver dans le document de l'OMG[1] une description très précise de la sémantique de chacun des éléments de modélisation du diagramme d'activité qui ont été présentés en début du chapitre. A partir de cette description on peut en déduire leur spécification OCL. Cependant, nous décrivons ces propriétés en tenant compte du fait que les propriétés syntaxiques décrites dans le paragraphe précédent sont vérifiées.

Les spécifications générales définissant les propriétés comportementales d'un diagramme d'activité sont les suivantes :

ActivityNode ::exec(env : EnvExecDA) : EnvExecDA
ValueSpecification::eval(env : EnvExecDA) : ValueSpecification

Figure 28 : ***Spécification comportementales des nœuds d'activité et des spécifications de valeurs***

Dans le cadre de cette étude, le diagramme d'activité est utilisé pour modéliser des algorithmes séquentiels. On ne définit donc pas de propriétés comportementales pour les flots de contrôle, dans la mesure où lorsqu'un nœud d'activité termine son exécution, le contrôle est transmis aussitôt au nœud de contrôle désigné par le flot sortant dont la garde a la valeur vrai. Dans la mesure où un nœud d'activité peut être une action (de type Action) ou un nœud de contrôle (de type ControlNode), il s'en suit que

- Si le nœud exécutable courant (self) est une action, il s'agit donc d'exécuter l'action correspondante et de passer ensuite le contrôle à l'action exécutable qui suit, si elle existe :

[1] http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

```
Action ::exec( env : EnvExecDA ) : EnvExecDA =

    let     ensSuiv : ExecutableNode = self.outcoming.target
    in      if   ensSuiv->size() = 1
            then ensSuiv->asSequence()->at( 1 ).exec( self.exec( env )     )
            else                      self.execAction( env )
            end

Action::execAction( env : EnvExecDA ) : EnvExecDA
```

- Si le nœud exécutable courant (self) est un nœud de contrôle, il peut y avoir plusieurs nœuds exécutables en sortie, selon le type du nœud de contrôle courant : le choix se fera en fonction de la valeur courante des gardes des flots de contrôle reliant le nœud courant aux nœuds exécutables en sortie :

```
MergeNode::exec( env : envExecDA ) : EnvExecDA =
            self.outgoing.target->asSequence()->at( 1 ).exec( env )

DecisionNode::exec( env : envExecDA ) : EnvExecDA =
            self.outgoing.select( cf |  cf.guard.oclAsType( LiteralBoolean ).value = true
                        )->asSequence()->at( 1 ).target.exec( env )

InitialNode::exec( env : envExecDA ) : EnvExecDA =
            self.outgoing.target->asSequence()->at( 1 ).exec( env )

ActivityFinalNode::exec( env : envExecDA ) : EnvExecDA =
            env
```

Figure 29 ***: Sémantique opérationnelle des Nœuds de Contrôle du Diagramme d'Activité***

Pour un nœud de regroupement, le contrôle est passé directement au nœud exécutable qui suit, pour un nœud de décision, le contrôle est donné en fonction de la garde des flots de contrôle sortant. Le nœud initial passe le contrôle au nœud exécutable suivant qui sera donc le premier à être exécuté. Le nœud d'activité final terme l'exécution du modèle du diagramme d'activité.

Les propriétés comportementales des nœuds exécutables élémentaires sont les suivantes :

```
OpaqueExpression ::eval( env : EnvExecDA ) = ValueSpecification

OpaqueAction::execAction( env : EnvExecDA ) : EnvExecDA =  env

CallBehaviorAction ::execAction( env : EnvExecDA ) : EnvExecDA =
                        self.behavior.oclAsType( Activity ).exec( env )

…
```

Figure 30 ***: Sémantique opérationnelle des Nœuds de Contrôle du Diagramme d'Activité***

Les propriétés comportementales des nœuds d'activité structurée sont les suivantes :

```
SequenceNode ::execAction( env : EnvExecDA ) : EnvExecDA =
```

```
            self.executableNode->at( 2 ).exec( self.executableNode->at( 1 ).exec( env )
                                                                   )

ConditionnalNode::execAction( env : EnvExecDA ) : EnvExecDA =
        let cla : Clause =
          self.clause->select( cl | cl.test->asSequence()->first().oclAsType( ValueSpecificationAction
                                                                          ).value.eval( env )
                        ).asSequence()->first()
        in  cla.body->asSequence()->first().exec( env )

LoopNode ::execAction( env : EnvExecDA ) : EnvExecDA =
        let t : ValueSpecification =
              self.test->asSequence()->first().oclAsType( ValueSpecificationAction ).value
        in if t.eval( env )
          then self.bodyPart->asSequence()->first().exec( env )
         else                                             env
          endif
```

*Figure 31 : **Sémantique opérationnelle des actions du Diagramme d'Activité***

La simulation d'un diagramme d'activité pourra être lancée à l'aide de l'opération suivante :

```
Activity ::exec( env : EnvExecDA ) : EnvExecDA =
        self.node->select( no | no.oclIsTypeof( InitialNode )
                        ).oclAsType( InitialNode )->asSequence()->first().exec( env )
```

Injecter techniquement des propriétés au niveau Méta-Modèle UML pour définir une sémantique comportementale du diagramme d'activité ne semble donc pas, a priori, difficile.

Cependant, définir une sémantique opérationnelle au niveau des nœuds exécutables et des flots de contrôle donne une vision du comportement d'un nœud exécutable et de son suivant, et pas au-delà : on ne contrôle donc pas de ce qui peut se passer après !

Ce n'est pas comme en programmation classique où l'enchaînement de l'exécution de structures de contrôle arborescentes à l'aide de fonctions récursives donne une vision globale du comportement portant sur toute partie d'un programme, c'est-à-dire du sous-arbre du modèle du programme correspondant.

Structurellement parlant, la terminaison de l'exécution d'un diagramme telle qu'elle est définie à l'aide de la spécification précédente n'est nullement garantie.

D'autre part, si un expert d'un domaine d'applications était chargé de mettre en place de telle propriétés, il pourrait tenir compte de spécificités métier du domaine et définir ainsi une sémantique comportementale métier. Cela revient à modéliser le processus métier d'un domaine d'application à l'aide d'un langage de modélisation, ce que l'on appelle généralement un DSML[2]. Associé à ce langage de modélisation, il sera aussi nécessaire de prévoir les règles assurant la transformation des diagrammes d'activité en modèles de programmes des langages de la plate-forme cible, surtout si l'on souhaite qu'elle soit automatique.

IV.1.d Triplet de Hoare et propriétés axiomatiques d'un Diagramme d'Activité

[2] Domain Specific Model language

Dans le chapitre IV, on a présenté le triplet de Hoare que l'on définit par rapport à la sémantique comportementale d'un segment de programme et par rapport à des propriétés de type pré-condition et post-condition. Les propriétés comportementales des différents nœuds exécutables d'un diagramme d'activité que l'on a définies sont valables au niveau de chaque nœud. Cependant, de part sa représentation sous la forme d'un graphe, il n'est pas sûr qu'un diagramme d'activité puisse s'exécuter comme un programme ou un modèle de programme, l'exécution des nœuds exécutables ne pouvant s'enchainement récursivement comme pour un programme. Il n'est donc pas possible de définir, a priori, de sémantique d'un triplet de Hoare dans un diagramme d'activité de la même manière que pour le langage de programmation dont les programmes ont une structure arborescente. Dans le chapitre suivant, nous verrons les conditions particulières que tout diagramme d'activité devra respecter pour que l'on puisse définir une sémantique comportementale aux triplets de Hoare et pour en déduire des propriétés axiomatiques.

IV.2 Le rôle du diagramme d'activité

Un concepteur élaborant un modèle à l'aide du diagramme d'activité dispose d'une très grande variété d'artéfacts de modélisation pouvant se situer à différents niveaux d'abstraction. Quel en est donc son rôle et sa place dans une démarche UML, en l'occurrence la démarche donnée en exemple en début de ce document ?

IV.2.a Graphe de Nœuds d'Activité ou Arbre de Nœuds d'Activité Structurée ?

Le rappel fait en début de chapitre sur les différents éléments de modélisation du diagramme d'activité, bien que limité à la modélisation d'algorithmes séquentiels, montre qu'un diagramme d'activité peut être un outil de modélisation très puissant : il peut être utilisé à un haut niveau d'abstraction si l'on s'en tient à une modélisation sous la forme d'un ensemble de nœuds d'activité et de flots de contrôle, ou si l'on s'en tient à un bas niveau d'abstraction en structurant hiérarchiquement les actions élémentaires du système à modéliser à l'aide de nœuds d'activité structurée. Dans le premier cas, le système à modéliser apparait sous la forme d'un graphe, dans le deuxième sous la forme classique d'une structure d'arbre proche de la programmation.

IV.2.b Vers un Processus IDM pour la Génération de Codes

Dans le cas où l'on se place à un niveau d'abstraction relativement élevé, il est tout naturellement plus facile de modéliser les différentes actions du système et leurs enchainements sous la forme d'un graphe que sous la forme d'une structure arborescente. Même si le diagramme d'activité vérifie les propriétés rajoutées au niveau du fragment du Méta-Modèle UML portant sur la définition de ses éléments de modélisation, rien ne garantit qu'une structure de graphe puisse s'exécuter comme un langage de programmation dont la structure hiérarchique des instructions des programmes assure un retour à la racine du programme, si les calculs qui y sont décrits sont corrects. D'où l'étude, en programmation, de propriétés axiomatiques sur des structures de contrôle bien clairement définies, ayant pour objectif de faciliter la vérification du fonctionnement correct des programmes, en particulier les invariants et les variants de boucle étant définis pour en vérifier ce que l'on appelle la

correction partielle et totale. De plus, aura-t-on à un haut niveau d'abstraction les éléments de modélisation (variables, type des variable, opérations, …) pour décorer le fragment du Méta-Modèle du diagramme d'activité de propriétés de type pré-condition et post-condition, comme en programmation, vérifiant à l'aide d'un triplet de Hoare que le fragment de diagramme d'activité est correct ?

Dans le cas où l'on se place à un niveau d'abstraction plus proche de la programmation, on aura certainement défini avec précision, comme en programmation, tous les éléments de modélisation nécessaires à la spécification des triplets de Hoare. Dans ce cas, le diagramme d'activité n'est-il pas trop détaillé et trop proche de la programmation ? Il est en effet important qu'un modèle reste une étape intermédiaire entre les niveaux d'abstraction se situant en amont et en aval, mais ne soit pas une réplique, à la syntaxe prés, d'un autre modèle du processus IDM.

Le rôle du diagramme d'activité dépend en fait du niveau d'abstraction où l'on se situe. Il peut être donc utilisé à un niveau de modélisation relativement abstrait pour élaborer des modèles de conception. Il peut aussi être utilisé à un niveau de raffinement très proche des artefacts de programmation. Le diagramme d'activité est donc un outil indispensable si l'on veut dans un processus IDM décrire de façon détaillée le passage entre les activités de conception et de génération de codes.

Nous avons montré dans les chapitres précédents que le langage de contraintes OCL, partie intégrante de UML, est le complément indispensable pour spécifier, au niveau des modèles de codes, toutes les propriétés garantissant que les codes ont bien les qualités requises. Avec les triplets de Hoare, les Analystes/Concepteurs ont les moyens de vérifier, toujours au niveau des modèles, que les codes sont corrects. Sans même attendre qu'un prouveur, comme un atelier B par exemple, puisse le démontrer formellement, les Analystes/Concepteurs peuvent, s'ils disposent d'un environnement d'exécution UML/OCL, le vérifier sur des jeux de données.

Au cours des chapitres précédents, nous avons montré les avantages de s'appuyer sur les méthodes et les techniques des traducteurs pour les appliquer au niveau de modèles de langages. Nous montrons au chapitre suivant, le chapitre VI, comment nous pouvons aussi les appliquer au niveau du diagramme d'activité, permettant ainsi aux Analystes/Concepteurs de vérifier la qualité d'un tel modèle de conception, et s'assurer que leurs transformations vers les modèles de codes sont correctes, montrant ainsi que les codes reflètent bien les intentions des Concepteurs. Ces transformations restent internes au même Méta-Modèle, le MM UML, ce qui ne pose donc pas de problèmes pour exprimer les pré-conditions et les post-conditions des triplets de Hoare impliquées dans les propriétés de transformations. Ce qui n'est pas le cas lorsque l'on cherchera à transformer un diagramme d'activité en un modèle de code.

VI – Vers un Processus IDM incrémental pour l'activité de Génération de Codes à partir d'un Diagramme d'Activité

Au cours des chapitres III et IV, nous avons modélisé les propriétés des langages de manière à pouvoir appliquer au niveau des modèles de codes des techniques généralement réservées aux traducteurs. L'intérêt est double puisque les Analystes/Concepteurs disposent d'un environnement de Méta-Modélisation UML devenu un standard de fait, et des plates-formes comme Eclipse/EMF ou comme Topcased et Kermeta. La modélisation en UML/OCL des langages de programmation prend donc toute son importance puisqu'il devient possible aux Concepteurs de vérifier que les modèles de codes sont corrects, en « remontant » à ce niveau de modélisation les méthodes et les techniques de vérification des codes. Il est aussi possible de paramétrer les modèles de codes en fonction des spécificités des différents domaines d'applications.

Au cours du chapitre précédent, nous avons étudié le diagramme d'activité en remarquant que l'on peut disposer d'un très grand nombre d'éléments de modélisation qui en fait donc un outil indispensable dans un processus de développement, aussi bien en tout début du processus pour en décrire un premier niveau de modélisation du système à développer que dans les phases finales du processus pour modéliser dans le détail les aspects calculatoires d'une opération particulière.

Peut-on donc « remonter » et appliquer au niveau du diagramme d'activité toutes les techniques et les méthodes des langages et des traducteurs que nous avons déjà remontées pour modéliser leurs propriétés ? Le problème est en fait plus difficile puisque le diagramme d'activité est déjà défini par le Méta-Modèle UML alors que le Méta-Modèle du langage avait été développé avec une certaine liberté.

Cependant, le diagramme d'activité est déjà un outil difficile à maitriser pleinement compte tenu de ses très nombreux éléments de modélisation qu'il peut offrir, même si l'on s'en tient à des modèles ciblant dans un processus des langages de programmation séquentiels. De plus, il vient en complément d'autres diagrammes UML, comme le diagramme de classes ou d'états/transitions en particulier.

Le fait d'enrichir le diagramme d'activité de propriétés syntaxiques et sémantiques basées sur la logique permettra de mieux définir leurs rôles dans le processus et donc de bien situer leurs positions en fonction de leurs niveaux d'abstraction et par rapport aux modèles qui leurs sont en amont et en aval.

L'objectif de ce chapitre est donc de définir un processus IDM spécifique à l'activité de génération de codes, à partir d'un diagramme d'activité. Il s'agit en particulier de voir les conditions dans lesquelles il est possible de définir une sémantique opérationnelle portant sur l'ensemble des actions d'un diagramme d'activité et les flots de contrôle assurant ainsi aux Concepteurs que la génération des codes est possible.

VI – Vers un Processus IDM incrémental pour l'activité de Génération de Codes à partir d'un Diagramme d'Activité

Table des matières

I Diagramme d'activité : Démarche de Conception ou Outil de Programmation de haut niveau ?......... 180
I.1 Flots de contrôle et nœuds d'activité structurée et équivalence comportementale 180
I.1.a Exemple de deux fragments de modèles structurellement différents, mais définissant le même comportement.. 180
I.1.b Equivalence comportementale entre les deux fragments.. 181
I.2 Quelques exemples de fragments de modèles ayant les mêmes propriétés comportementales. 183
I.2.a Quelques exemples de correspondances structurelles 183
I.2.b Equivalence comportementale des fragments de modèles 184
I.2.c Règles formelles de correspondance comportementale.. 185
I.3 Règles de transformation et Enchainement des Propriétés Comportementales 186
I.3.a Substitution d'un fragment de modèle par un fragment de modèle sémantiquement équivalent .. 186
I.3.b Règles de réécriture de transformation.. 188
I.3.c Exemple de transformation de modèle et d'enchainement des Propriétés 189
I.3.d L'approche règles de réécriture appliquée à la transformation de modèles................................. 192
I.3.e Conditions d'Exécutabilité d'un Diagramme d'Activité 193
II Processus IDM incrémental pour l'activité de génération de codes.. 194
II.1 Activité de génération de codes réalisée par étapes successives de raffinage 194
II.1.a Diagramme d'activité et génération de codes.. 194
II.1.b Raffinage de modèles au niveau du diagramme d'activité 195
II.1.c Propriétés sur les modèles 196
II.2 Quelques éléments bibliographiques sur les transformations de modèles................................. 198
II.2.a Modèle source, Modèle cible et transformation de modèles 198
II.2.b Architecture d'un Processus de développement de logiciels.. 199
II.2.c Différents types de transformation de modèles 200
III Mise en œuvre des Transformations de Modèles de l'activité de génération de codes 201
III.1 Transformation d'un graphe de flots de contrôle en une arborescence de nœuds d'activité structurée... 202
III.1.a Transformations élémentaires et enchainement des transformations...................................... 202
III.1.b Mise en œuvre de la règle 1 202
III.1.c Mise en œuvre de la règle 2, 3 et 4 207
III.1.d Pré-condition à la hiérarchisation : Algorithme basé sur la trace du graphe 209
III.1.e Hiérarchisation d'un diagramme d'activité.. 212
III.1.f Validation de la hiérarchisation d'un diagramme d'activité contenant un modèle de graphei.. 214
III.2 Transformation d'un diagramme d'activité hiérarchisé en un modèle de code........................ 215
III.2.a Environnement d'exécution de la transformation 215
III.2.b Déclaration des variables 216
III.2.c Transformation des nœuds de l'arbre : les structures de contrôle... 216
III.2.d Transformation des feuilles de l'arbre : Les actions élémentaires 216

VI – Vers un Processus IDM incrémental pour l'activité de Génération de Codes à partir d'un Diagramme d'Activité

Table des figures

Figure 1 : Flot de contrôle enchainant deux actions, et nœud d'activité structurée de type SequenceNode __ 181
Figure 2 : Environnements d'exécution avant et après exécution des deux fragments de modèle ___________ 182
Figure 3 : Equivalence comportementale entre deux fragments de modèles d'un diagramme d'activité _____ 182
Figure 4: Fragments de modèles regroupés en nœuds d'activité structurée ___________________________ 184
Figure 5: Exemple d'un fragment de modèle non exprimable en nœuds d'activité structurée ______________ 185
Figure 6 : Quelques règles définissant les équivalences élémentaires entre des fragments de modèles d'un diagramme d'activité ______ 186
Figure 7 : Exemples de règles de substitution de fragments de modèles sémantiquement équivalents _______ 187
Figure 8 : Substitution d'un fragment de modèle par un fragment de modèle ayant le même comportement __ 188
Figure 9 : Règles de transformation d'un fragment de modèle de nœuds exécutables et de flots de contrôle en un fragment de modèle de nœuds d'activité structurée concernant les propriétés comportementales __________ 189
Figure 10 : Diagramme d'activité comprenant un modèle de nœuds d'activité et de flots de contrôle _______ 189
Figure 11 : Equivalence comportementale de deux fragments de modèles ______________________________ 190
Figure 12 : Diagrammes se déduisant les uns des autres ((3) se déduit de (2), et (2) se déduite de (1) ______ 191
Figure 13 : Exemples de diagrammes d'activité ayant les mêmes propriétés comportementales ___________ 192
Figure 14 : Génération de codes réalisée à partir d'un diagramme d'activité et d'un diagramme de classes _ 195
*Figure 15 : Génération de codes faisant apparaitre deux niveaux de modèle à l'aide du diagramme d'activité*195
Figure 16 : Transformation des modèles ______ 198
Figure 17 : Concepts de base de la transformation de modèles ______ 198
Figure 18 : Architecture d'un processus de développement proposé par l'OMG ________________________ 200
Figure 19 : Différents types de transformations pouvant apparaitre au cours d'un processus de développement ______ 201
Figure 20 : Mise en œuvre de la première règle ______ 203
Figure 21 : Regroupement de deux actions consécutives en un nœud d'activité structurée de type SequenceNode ______ 203
Figure 22 : Représentation interne (Instance du Méta-Modèle UML 2 .0) des deux fragments de modèle ____ 204
Figure 23 : Mise en œuvre d'une transformation élémentaire hiérarchisant un diagramme d'activité _______ 206
Figure 24 : Fragment de modèle transformable en un nœud d'activité de type LoopNode ________________ 208
Figure 25 : Fragment de modèle transformable en un nœud d'activité structurée de type ConditionalNode _ 209
Figure 26 : Segment de modèle permettant d'identifier un sous-arbre, et les nœuds qui le composent _______ 210
Figure 27 : Règles définissant pour chaque nœud d'activité à quel sous-arbre il appartient ______________ 211
Figure 28 : Exemple de recherche d'une composante fortement connexe dans un graphe de flots de contrôle 212
Figure 29 : Environnement d'exécution pour le calcul de la pré-condition de hiérarchisation d'un modèle __ 212
Figure 30 : Processus de hiérarchisation d'un modèle d'un diagramme d'activité ______________________ 213
Figure 31 : Les environnements d'exécution pour une transformation dont les modèles source et cible ne sont pas définis par le même Méta-Modèle ______ 216

I Diagramme d'activité : Démarche de Conception ou Outil de Programmation de haut niveau ?

L'étude au chapitre précédent du fragment du Méta-Modèle UML concernant le diagramme d'activité fait état d'un très grand nombre d'éléments de modélisation applicables aussi bien à un niveau d'abstraction relativement élevé avec les nœuds d'activité et les flots de contrôle, qu'au niveau proche de la programmation avec les nœuds d'activité structurée.

C'est peut-être la raison pour laquelle le diagramme d'activité n'a pas de position bien définie lors d'un processus IDM, d'autant plus qu'il est difficile lors de l'élaboration d'un modèle d'avoir une certaine cohérence entre le niveau d'abstraction de ses artéfacts de modélisation et la position du modèle dans le processus.

En fait, le diagramme d'activité peut apparaitre dans les phases en amont du processus IDM dans la phase de conception, ou dans les phases plus en aval après avoir pris en compte des artéfacts de programmation anticipant les activités de génération de codes. Il s'ensuit qu'un diagramme d'activité, se trouvant plus ou moins bien positionné dans un processus ou dans une démarche, peut être s'avérer être difficile à interpréter par les Analystes/Concepteurs et donc mal compris.

Si le diagramme d'activité offre de très nombreuses possibilités pour modéliser un système à développer, on s'aperçoit en fait que l'on peut avoir souvent plusieurs solutions pour résoudre un problème particulier. Dans ce paragraphe, nous donnons quelques exemples de fragments de modèles équivalents, certains pouvant apporter une vision relativement abstraite du problème que l'on cherche à modéliser, d'autres plus concrète. Ce qui nous permettra alors d'en déduire quelques règles de bonne utilisation d'un diagramme d'activité par rapport au rôle qu'on cherche à lui attribuer dans le processus. On rappelle cependant que dans le cadre de cette étude, le diagramme d'activité doit assurer la transition entre les modèles de conception et les codes, limités dans le cadre de cette étude à des langages séquentiels.

I.1 Flots de contrôle et nœuds d'activité structurée et équivalence comportementale

Il s'agit principalement de voir les différentes correspondances que l'on peut faire entre les flots de contrôle et les nœuds d'activité structurée qui offrent deux types d'enchainement des actions. Nous nous appuierons sur les propriétés comportementales pour bien montrer les correspondances que l'on peut faire entre différentes solutions équivalentes.

I.1.a Exemple de deux fragments de modèles structurellement différents, mais définissant le même comportement

La figure suivante montre deux fragments de modèles d'un diagramme d'activité où l'enchainement des deux actions j et k est décrit de manière différente :

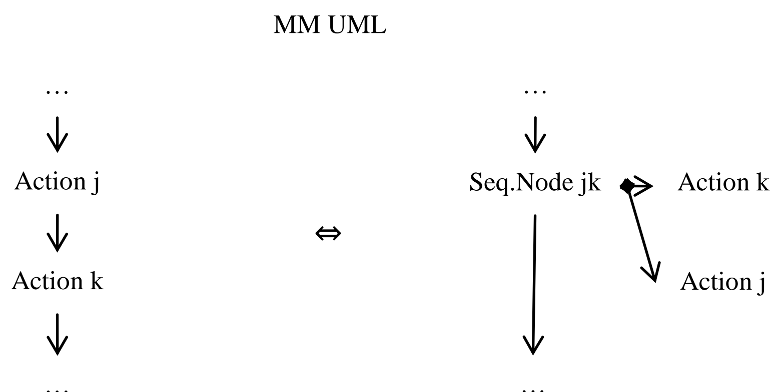


*Figure 1 : **Flot de contrôle enchainant deux actions, et nœud d'activité structurée de type SequenceNode***

Ces deux fragments de modèles, instances du Méta-Modèle UML, sont intuitivement équivalents, au sens où l'enchainement des deux actions reste le même. Dans celui de gauche, l'enchainement est décrit à l'aide d'un flot de contrôle, dans celui de droite c'est le nœud d'activité structurée de type SequenceNode qui définit l'ordonnancement des deux actions j et k, comme pour une structure de contrôle de type Sequence d'un langage de programmation. Dans les deux cas, l'action j doit s'exécuter avant l'action k. Mais, peut-on le démontrer en précisant le sens que l'on donne à la notion d'équivalence de modèles ?

I.1.b Equivalence comportementale entre les deux fragments

La propriété comportementale concernant le fragment de gauche dont l'enchainement des actions est décrit à l'aide d'un flot de contrôle, peut être définie de la manière suivante :

```
Action ::exec( env : EnvExecDA ) : EnvExecDA =
   self.outcoming.target ->asSequence()->at( 1 ).oclAsType( Action ).execAction( self.execAction( env )  )
```

La propriété comportementale concernant le fragment de droite dont l'enchainement des actions est décrit à l'aide d'un nœud d'activité structurée, peut être définie de la manière suivante :

```
SequenceNode ::exec( env : EnvExecDA ) : EnvExecDA =
                    self.executableNode->at( 2 ).oclAsType( Action ).execAction(
                              self.executableNode->at( 1 ).oclAsType( Action ).execAction( env )
                                                                                   )
```

On a donc deux fragments de modèles structurellement différents. A priori, ces deux fragments de modèles sont équivalents au sens où l'on doit pouvoir vérifier qu'ils ont les mêmes comportements sur le même environnement d'exécution. On dira que les deux fragments de modèles ont les mêmes propriétés comportementales si les fragments ont le même effet sur le même environnement d'exécution.

Cette propriété peut se schématiser, telle que le montre la figure suivante :

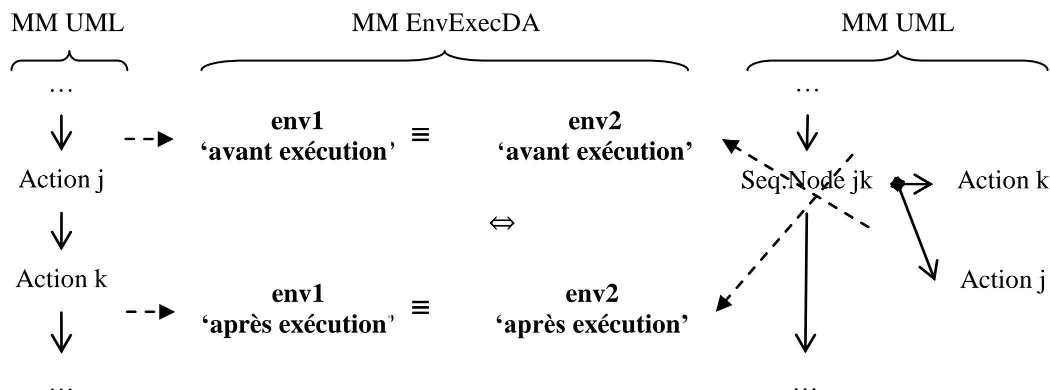


Figure 2 : Environnements d'exécution avant et après exécution des deux fragments de modèle

Cette figure montre que le fragment de modèle de gauche (modèle « de graphe ») prend un environnement d'exécution env1 dans un certain état appelé **'avant exécution'**. A la fin de l'exécution du fragment de ce modèle de graphe, l'environnement d'exécution env1 se trouve dans un état **'après exécution'**. Il en est de même pour le fragment de modèle de droite (modèle « d'arbre »), en ce qui concerne l'environnement d'exécution env2. Les deux fragments de modèle ont des comportements équivalents si les deux environnements **env1 et env2 sont identiques 'avant l'exécution'** de chacun des deux fragments de modèles, alors l'exécution des deux fragments de modèles, chacun sur son environnement d'exécution, font que les environnements **env1 et env2 sont identiques.** Le Méta-Modèle EnvExecDA définit les deux environnements d'exécution env1 et env2, de la même manière que l'on avait défini les propriétés comportementales du langage « L » à l'aide d'un environnement d'exécution appelé EnvExec.

Cette propriété se spécifie, en UML/OCL, de la manière suivante :

```
∀ env1 ∈ EnvExecDA, ∀ env2 ∈ EnvExecDA
                        env1.eqDA( env2 )
                                    ).imp(
        aj.exec( env1 ) ).eqDA( snjk.exec( env2 ) )
                                    )
```

Figure 3 : ***Equivalence comportementale entre deux fragments de modèles d'un diagramme d'activité***

Sachant que :

- Action j, Action k représentent des actions d'un diagramme d'activité. Ce sont des valeurs de l'attribut name défini au niveau de la Méta-Classe du Méta-Modèle UML ModelElement, dont hérite une grande partie des Méta-Classes. On a supposé que ai et aj sont les noms des objets correspondants. Il en est de même pour le nom de l'objet dont SeqNode jk est la valeur de l'attribut name.
- aj est un objet référençant la première action du fragment du modèle de gauche, snjk le nœud de type SequenceNode du fragment de modèle de droite.
- L'opérateur eqDA retourne vrai si deux environnements d'exécution sont identiques. L'opérateur imp est l'opérateur booléen classique de l'implication. EnvExecDA

définit l'environnement d'exécution associant les variables manipulées dans le diagramme d'activité et leur valeur. Leur spécification est la suivante :

```
EnvExecDA ::eqDA( env : EnvExecDA ) : Booleen
Booleen ::imp( b : Booleen ) : Booleen
```

Comme on l'a vu au chapitre précédent, cette assertion pourrait être traduite en B avec tous les éléments la concernant. Un Atelier B pourrait ainsi prouver l'équivalence sémantique des deux fragments de modèles.

Par contre, au niveau UML/OCL, cette assertion peut servir de programmes de tests ponctuels, en définissant préalablement les environnements d'exécution contenant des données judicieusement choisies.

Il est à remarquer que cette propriété à valider est des plus importantes dans le contexte de ce travail puisque les propriétés comportementales sont mises en œuvre au niveau du Méta-Modèle UML, en particulier au niveau des nœuds d'activité pour le fragment de modèle source et au niveau des actions pour le fragment de modèle cible. Les propriétés comportementales ne se correspondent donc pas deux à deux, mais globalement au niveau des fragments : les expressions définissant donc chacun des fragments sont donc différentes structurellement.

I.2 Quelques exemples de fragments de modèles ayant les mêmes propriétés comportementales

Etant donné que l'on se situe dans un contexte de modélisation des algorithmes séquentiels, cela revient à voir comment on pourrait écrire des séquences d'instructions contrôlées avec des branchements conditionnels (structure de contrôle de tests) et inconditionnels (instruction go to !) d'une manière « plus structurée » à l'aide de structures de boucle ou de test.

I.2.a Quelques exemples de correspondances structurelles

Il s'agit de voir, donc, s'il existe dans un modèle d'autres formes de fragments «équivalents » qui peuvent s'exprimer en termes de nœuds d'activité structurée, telles que la séquence (SequenceNode), le test (ConditionnalNode) et la boucle (LoopNode). La figure suivante montre quatre cas élémentaires de fragments de modèles différents mais sémantiquement

équivalents :

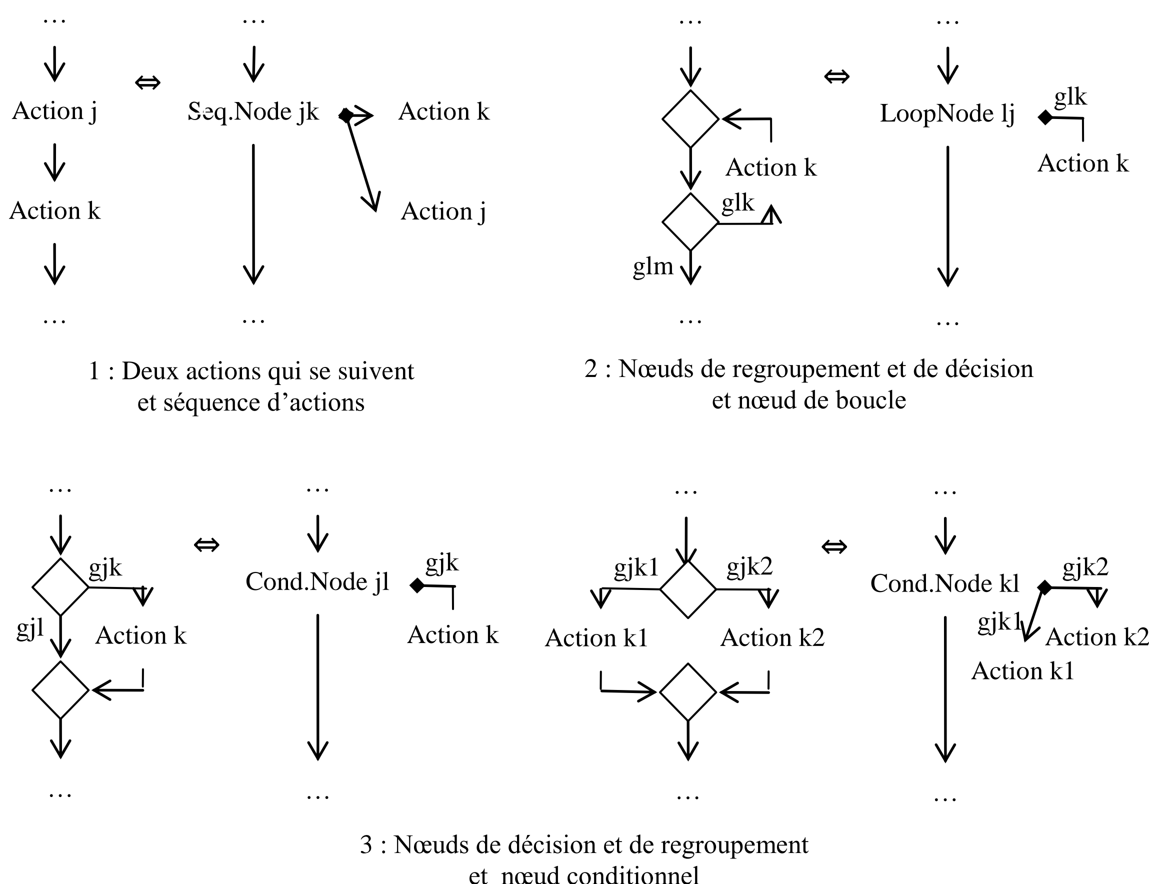


Figure 4: ***Fragments de modèles regroupés en nœuds d'activité structurée***

Le premier cas reprend l'exemple des deux actions séquentielles, le deuxième correspond à un fragment de modèle où l'action k peut être contrôlée par un nœud d'activité structurée de type LoopNode. Enfin, les deux autres figures du bas, sont les exemples classiques de tests.

I.2.b Equivalence comportementale des fragments de modèles

De la même manière que deux actions qui se suivent et deux actions entrant dans la composition d'un nœud de séquence ont la même sémantique comportementale, on peut écrire les propriétés comportementales des deux fragments de modèles des trois autres exemples, et en déduire leur équivalence comportementale.

Tout fragment de modèle élaboré à l'aide de nœuds d'activité et de flots de données peut ne pas s'exprimer en termes d'actions et de nœuds d'activité structurée. La figure suivante en montre un exemple classique :

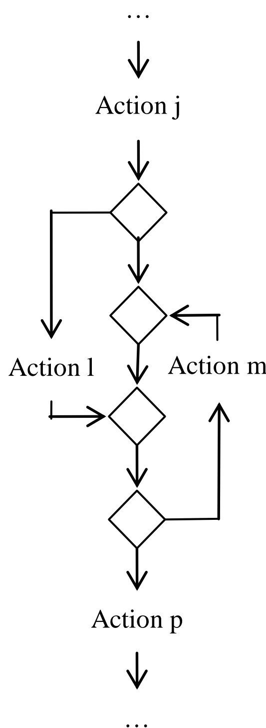


Figure 5: ***Exemple d'un fragment de modèle non exprimable en nœuds d'activité structurée***

Dans le Méta-Modèle UML, la définition récursive des nœuds d'activité structurée permet d'étendre les propriétés comportementales à tout fragment de modèle représenté sous la forme d'une structure arborescente de nœuds d'activité structurée. On retrouve, effectivement, les propriétés de base des langages de programmation classiques, comme le langage « L » par exemple, où les grammaires définissent des programmes qui se représentent à l'aide de structure de données arborescentes récursives.

Le diagramme d'activité permet donc de modéliser les enchainements des actions d'un système à développer, soit sous la forme d'un graphe, soit sous la forme d'une structure arborescente, proche de la programmation. La manière de modéliser un système à développer réalisé à l'aide d'un diagramme d'activité devrait donc dépendre de sa position dans le processus. Un modèle dont l'enchainement des actions est décrit à l'aide de nœuds d'activité structurée devrait être un modèle relativement proche de la programmation.

I.2.c Règles formelles de correspondance comportementale

Au paragraphe précédent, nous avons montré des exemples de fragments de modèles UML qui ont des propriétés comportementales équivalentes. Ces règles peuvent être écrites en OCL/LA de la manière suivante :

R1 : aj, ak : Action ; cfjk : ControlFlow ; snij : SequenceNode

cfij(aj, ak) ≡ snjk(aj, ak) (équivalence comportementale)

ou plus simplement : aj → ak ≡ snij(aj, ak), la flèche représentant un flot de contrôle
et plus généralement : Action, Action ≡ SequenceNode

R2 : mnj: MergeNode ; ak : Action ; dnl : DecisionNode ; glk, glm : ValueSpecification ; lnjl : LoopNode

mnj → dnl ; dnl(glk) → ak ; ak → mnj ; dnl(glm) → … ≡ lnjl(gkl, ak)

Merge, Action, DecisionNode ≡ LoopNode

R3 : dnj : DecisionNode ; ak : Action ; gjk, gjl: ValueSpecification ; ml : MergeNode ; cnjl : ConditionalNode

dnj → ml ; dnj(gjk) → ak ; dnj(gjl) → ml ; ak → mnl ≡ cnjl((gjk, ak))

DecisionNode, Action, MergeNode ≡ ConditionalNode

R4 : dnj : DecisionNode ; ak1, ak2 : Action ; gjk1, gjk2 : ValueSpecification ; mnl : MergeNode ; cnjl : ConditionalNode

dnj(gik1) → ak1 ; dnj(gik2) → ak2 ; ak1 → ml ; ak2 → mnl ≡ cnjl((gjk1, ak1), (gjk2, ak2))

DecisionNode, Action, Action, MergeNode ≡ ConditionalNode

*Figure 6 : **Quelques règles définissant les équivalences élémentaires entre des fragments de modèles d'un diagramme d'activité***

Ces exemples de règles définissant l'équivalence des comportements des fragments de modèles montrent bien le rapprochement entre les techniques des langages et des traducteurs qui permettent d'apporter les démonstrations des propriétés énoncées sur les modèles, et les techniques des modèles qui offrent aux Concepteurs des représentations graphiques des éléments de modélisation. Le langage de modélisation UML, et le langage de contraintes OCL étendu à un langage d'action renforcent pleinement le rapprochement entre ces deux technologies.

I.3 Règles de transformation et Enchainement des Propriétés Comportementales

Les règles définies au paragraphe précédent peuvent être reprises pour définir les transformations d'un modèle de graphe orienté de nœuds exécutables et de flots de contrôle, en un modèle d'arbre de nœuds d'activité structurée et d'action. Ces règles nous amènerons ensuite à vérifier que les transformations seront correctes.

I.3.a Substitution d'un fragment de modèle par un fragment de modèle sémantiquement équivalent

Ces règles sont dont les suivantes :

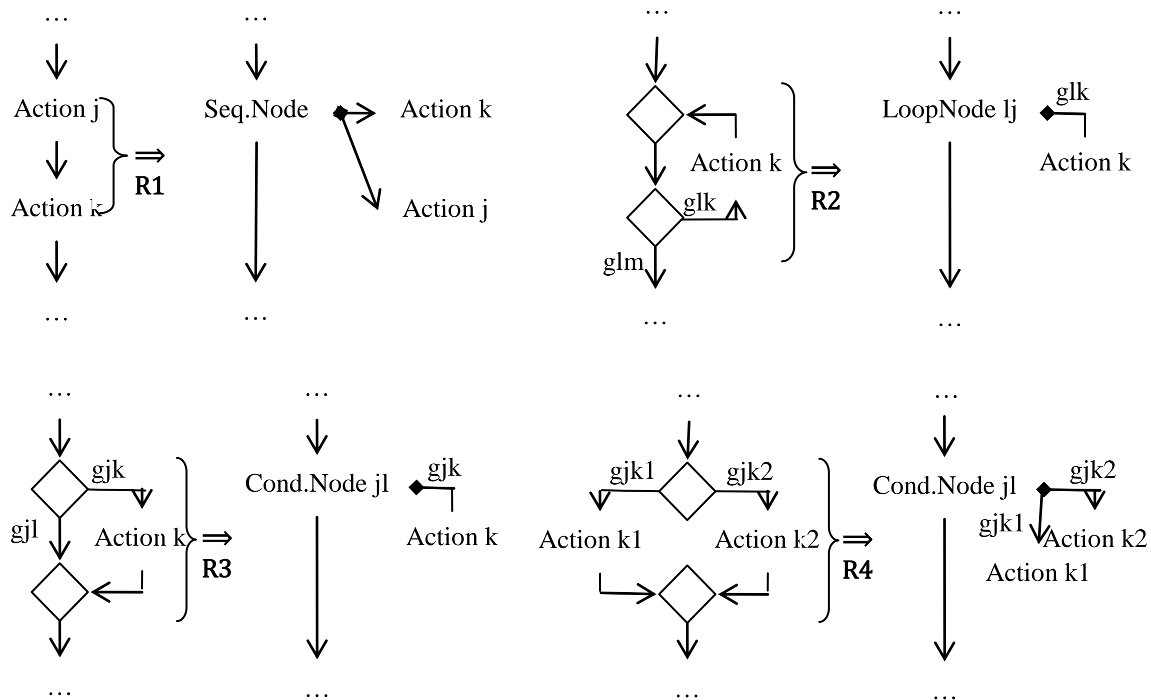


*Figure 7 : **Exemples de règles de substitution de fragments de modèles sémantiquement équivalents***

La figure suivante montre à gauche (1) un exemple de diagramme d'activité dont le modèle se représente sous la forme d'un graphe. On peut remarquer que le fragment de modèle comprenant les actions k1 et k2 encadrées par les nœuds de contrôle de type DecisionNode et de type MergeNode correspond au fragment de modèle de la partie gauche de la 4ième règle. On peut le substituer par le fragment de modèle de la règle qui lui est équivalent sémantiquement :

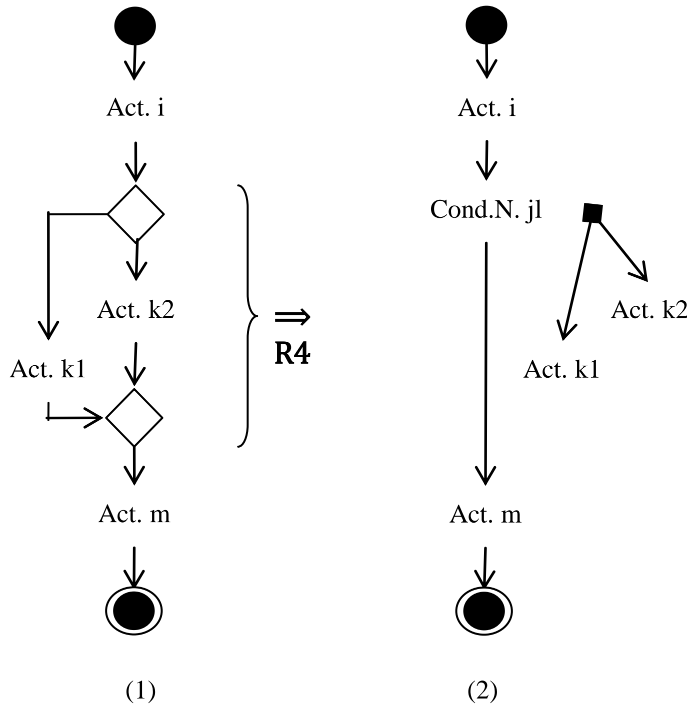


Figure 8 : ***Substitution d'un fragment de modèle par un fragment de modèle ayant le même comportement***

I.3.b Règles de réécriture de transformation

Comme pour les règles montrant des fragments de modèles différents mais ayant les mêmes propriétés comportementales, nous explicitons les règles de transformation élémentaires sur des fragments de modèles, sous la forme de règles de réécriture en supposant que la flèche avec le double trait exprime la transformation élémentaire :

R1 : Regroupement en structure de séquence deux actions qui se suivent :

aj, ak : Action ; cfjk : ControlFlow

cfjk(aj, ak) ⟹ snjk(aj, ak)

R2 : Fragment de modèle définissant une structure de boucle (en écrivant cfjk(aj,ak) sous la forme aj→ak) :

mnj : MergeNode ; ak : Action ; dnl : DecisionNode ; glk, glm : Guard

mnj → dnl ; dnl(glk) → ak ; ak → mnj ; dnl(glm) → … ⟹ LoopNode(glk, ak)

R3 : Fragment de modèle définissant une structure de test simple :

dnj : DecisionNode ; mnl : MergeNode ; ak : Action ; gjl, gjk : Guard

dnj → ml ; dnj(gjk) → ak ; ak → mnl ⟹ ConditionalNode((gjk, ak))

R4 : Fragment de modèle définissant une structure de test à deux branchements :

dnj : DecisionNode ; ml : MergeNode ; ak1, ak2 : Action ; gjk1, gjk2 : Guard

dnj(gjk1) → ak1 ; dnj(gjk2) → ak2 ; ak1 → ml ; ak2 → mnl ⟹ cnjl((gjk1, ak1), (gjk2, ak2))

Figure 9 : ***Règles de transformation d'un fragment de modèle de nœuds exécutables et de flots de contrôle en un fragment de modèle de nœuds d'activité structurée concernant les propriétés comportementales***

I.3.c Exemple de transformation de modèle et d'enchainement des Propriétés

L'application que nous prenons comme exemple doit réaliser 4 actions, la première étant Act. i et la dernière Act. m. Suite à l'action Act. i, l'application doit réaliser l'action k1 ou k2 selon le résultat d'une expression définie à l'aide d'un nœud de décision dont les gardes des flots de contrôle sont gjk1 et gjk2. Un nœud de regroupement indique qu'à la suite des actions k1 ou k2, le contrôle est donné à l'Act. m. Le diagramme d'activité de la figure suivante donne un modèle de cette application :

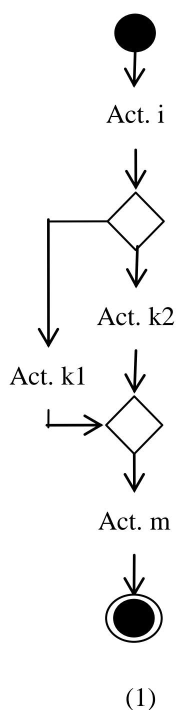


Figure 10 : ***Diagramme d'activité comprenant un modèle de nœuds d'activité et de flots de contrôle***

Ce diagramme d'activité comprend un fragment pour lequel, d'après les quatre règles définies précédemment, il en existe une (la règle 4) qui fait état d'un fragment de modèle qui lui est équivalent sémantiquement. Après avoir effectué la substitution correspondante, la figure suivante montre ce que ce devient le modèle :

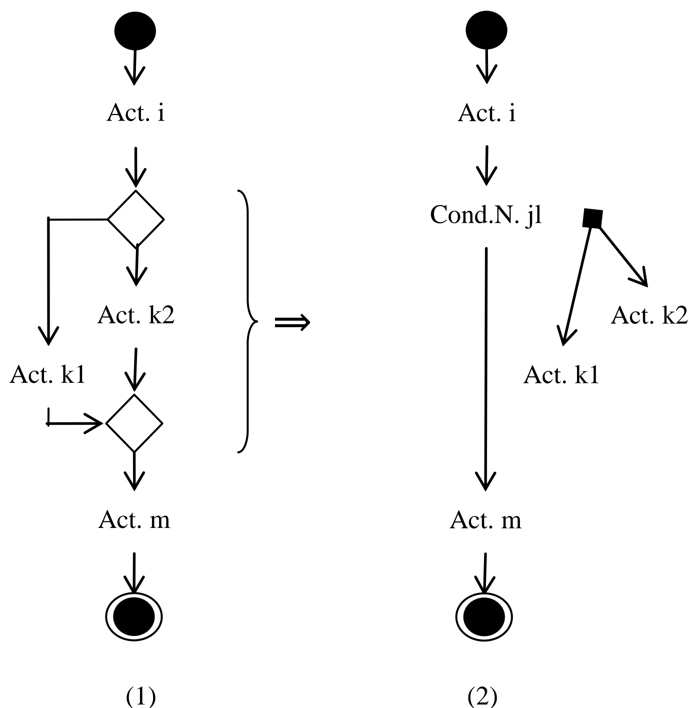


Figure 11 : ***Equivalence comportementale de deux fragments de modèles***

En appliquant les règles énoncées précédemment sur des fragments de modèles élémentaires équivalents, les propriétés comportementales du fragment de modèle du diagramme de gauche (1) se déduisent des propriétés du fragment de modèle du diagramme d'activité de droite (2).

Sur le diagramme (2), apparait le nœud conditionnel qui, par définition, est un nœud d'activité structurée qui est, plus généralement, une action.
On en déduit que dans ce diagramme (2), apparait deux actions qui se suivent et pour lesquelles on peut les regrouper en un nœud d'activité de type SequenceNode, tel que le montre le diagramme (3) de la figure suivante :

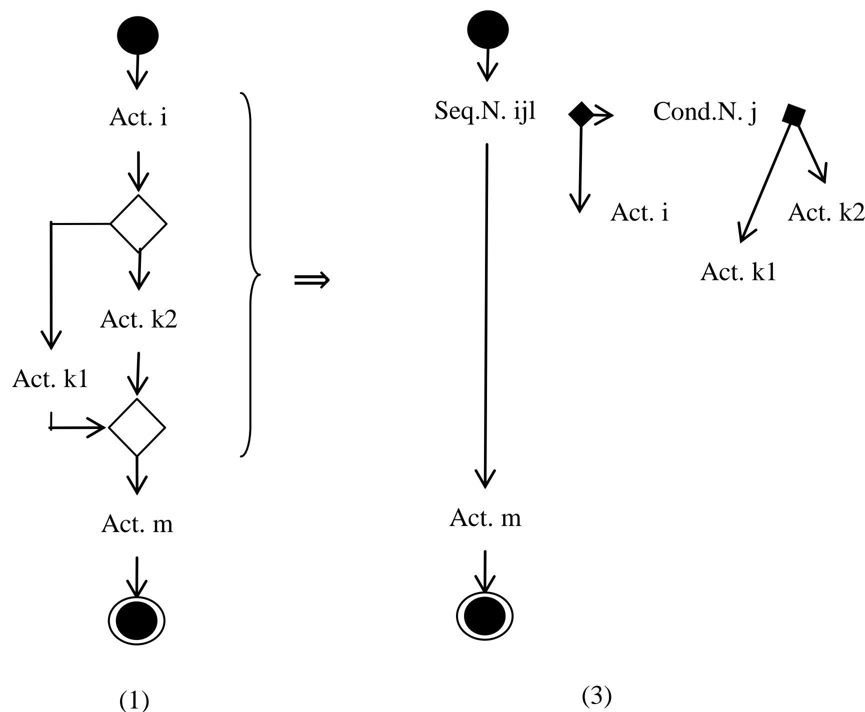


Figure 12 : ***Diagrammes se déduisant les uns des autres ((3) se déduit de (2), et (2) se déduite de (1)***

Ce qui signifie que le fragment de modèle se situant entre l'action i et le nœud MergeNode l, dans le diagramme (1), a la même propriété comportementale que le sous-arbre dont le nœud Seq.N. ijl est la racine, dans le diagramme (3).

Le diagramme (3) fait apparaitre aussi deux activités en séquence que l'on peut regrouper en un nœud d'activité structurée, tel que le montre le diagramme (4) de la figure suivante :

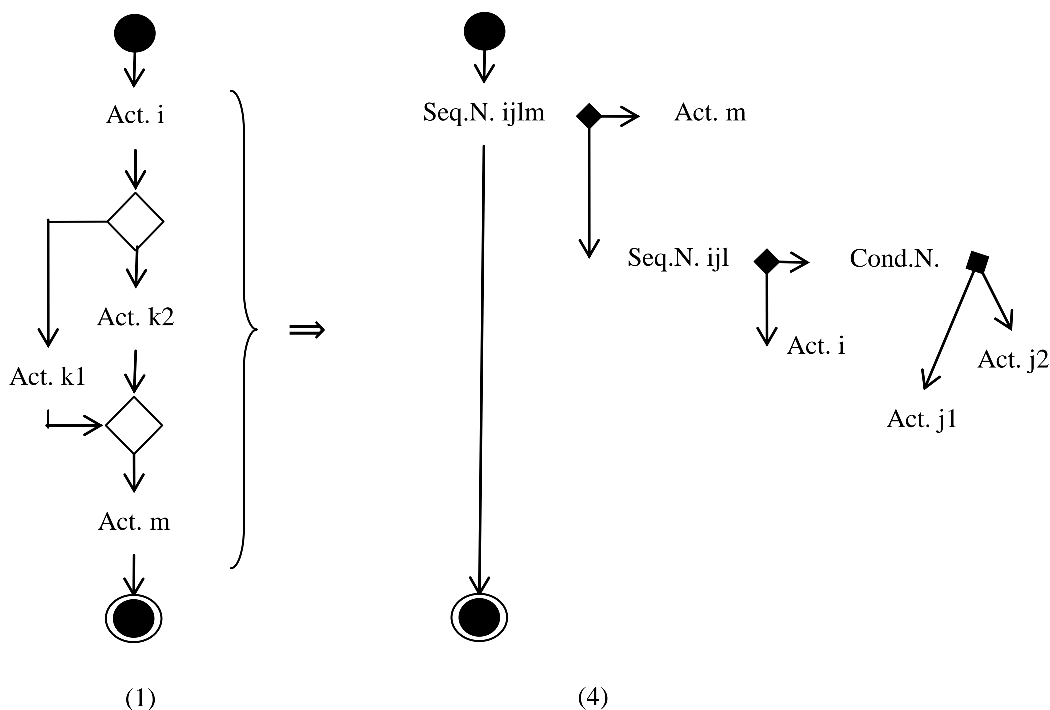


Figure 13 : ***Exemples de diagrammes d'activité ayant les mêmes propriétés comportementales***

On en déduit que les deux modèles ont la même propriété comportementale.

On peut remarquer que si l'on parcourt l'arbre du modèle du diagramme d'activité de la figure (4) en descendant de gauche à droite, la première action qui sera prise en compte est l'action qui se trouve à la suite du nœud de contrôle de type InitialNode.

On en déduit que si un diagramme d'activité représentable sous la forme d'un graphe de nœuds d'activité et de flots de contrôle peut se transformer, par étapes successives, en une structure arborescente de nœuds d'activité structurée et d'actions, alors il a la même sémantique comportementale de que cette structure arborescente. Ce qui signifie que le diagramme d'activité, tel qu'il apparait dans les figures précédentes (1) peut s'exécuter comme un programme, dans la mesure où les calculs qui y sont décrits sont élaborés en conséquence.

I.3.d L'approche règles de réécriture appliquée à la transformation de modèles

En reprenant les notations utilisées pour décrire les règles de transformation, l'exemple précédent peut se détailler de la manière suivante, en supposant que l'objet init réfère le nœud initial du diagramme d'activité qui correspond au point d'entrée du modèle, et fen le nœud terminal du diagramme d'activité :

- Au départ, le modèle se représente de la manière suivante :

  { init → ai ; ai → dnj ; dnj → ak1 ; dnj → ak2 ; ak1 → mnl ; ak2 → mnl ; mnl → am ; am → fen }

- En recherchant dans le système de règles, on peut voir que l'on a des éléments reliés par des flots de contrôle de même structure que le modèle de la partie de la règle 4 :

{ init → ai ; ai → **dnj ; dnj → ak1 ; dnj → ak2 ; ak1 → mnl ; ak2 → mnl ; mnl** → am ; am → fen }

- On peut donc substituer ce fragment par le fragment de modèle qui découle de la transformation de la règle :

{ init → ai ; ai → **cnjl ; cnjl** → am ; am → fen }

avec cnjl = ConditionalNode((gjk1, ak1), (gjk2, ak2))

- Il suffit de recommencer le processus avec le fragment de modèle :

{ init → ai ; ai → cnjl ; cnjl → am ; am → fen }

- On reconnaît deux actions qui se suivent :

{ init → **ai ; ai → cnjl ; cnjl** → am ; am → fen }

- On obtient donc la substitution suivante :

{ init → **snijl ; snijl** → am; am → fen }

avec snijl = sequenceNode(ai, cnjl).

- Avec une troisième itération :

{ init → snijl ; snijl → am; am → fen }

- On reconnaît deux actions qui se suivent :

{ init → **snijl ; snijl → am; am** → fen }

- Et qui, après avoir appliqué le principe de substitution sur la première règle donne :

{ init → **snijlm ; snijlm** → fen }

Avec snijlm = SequenceNode(snijl, am);

Ce qui permet d'obtenir le modèle arborescent de nœuds d'activité structurée et d'actions suivant :

{ init → snijlm ; snijlm → fen }

Avec : snijlm = SequenceNode(SequenceNode(ai, ConditionalNode((gjk1, ak1), (gjk2, ak2))))

I.3.e Conditions d'Exécutabilité d'un Diagramme d'Activité

Une condition nécessaire pour qu'un diagramme de nœuds d'activité et de flots de contrôle soit exécutable est qu'il puisse tout d'abord se transformer en un diagramme de nœuds d'activité structurée et d'actions. Compte tenu de l'étude faite dans les chapitres précédents sur les triplets de Hoare, cette condition n'est donc pas suffisante.

Ces équivalences entre modèles permettent de suggérer deux niveaux d'utilisation d'un diagramme d'activité dans un processus IDM en définissant avec précision les propriétés attendues à chacun des niveaux. Ce qui amène à définir un processus IDM incrémental pour l'activité de génération de codes à partir d'un diagramme d'activité considéré comme un modèle de conception.

II Processus IDM incrémental pour l'activité de génération de codes

Nous présentons dans ce paragraphe les principales caractéristiques d'un processus IDM incrémental pour l'activité de génération de codes en précisant à chaque niveau d'abstraction les modèles, leurs rôles et les propriétés attendues, puis les différentes transformations de modèles ainsi que leurs propriétés attendues justifiant qu'elles sont correctes.

II.1 Activité de génération de codes réalisée par étapes successives de raffinage

Pour s'assurer de la cohérence entre les modèles et les codes, nous avions proposé en introduction d'insérer un niveau de modélisation entre les modèles et les codes. Un tel niveau de modélisation basé sur la grammaire abstraite du langage cible et de ses propriétés comportementales permet de réaliser la génération de codes en deux étapes. La première étape transforme les modèles de conception en modèles de programmes, l'activité de génération de codes se faisant alors à partir des modèles de codes. L'intérêt réside dans le fait que c'est au niveau des modèles de codes que l'on peut vérifier si les codes sont cohérents par rapport aux modèles de conception, vérifiant ainsi que les intentions des Concepteurs sont bien comprises par les Analystes/Programmeurs.

II.1.a Diagramme d'activité et génération de codes

Si l'on prend un modèle de conception réalisé à l'aide du diagramme d'activité, alors, la figure suivante peut être un exemple de processus IDM pour réaliser l'activité de génération de codes :

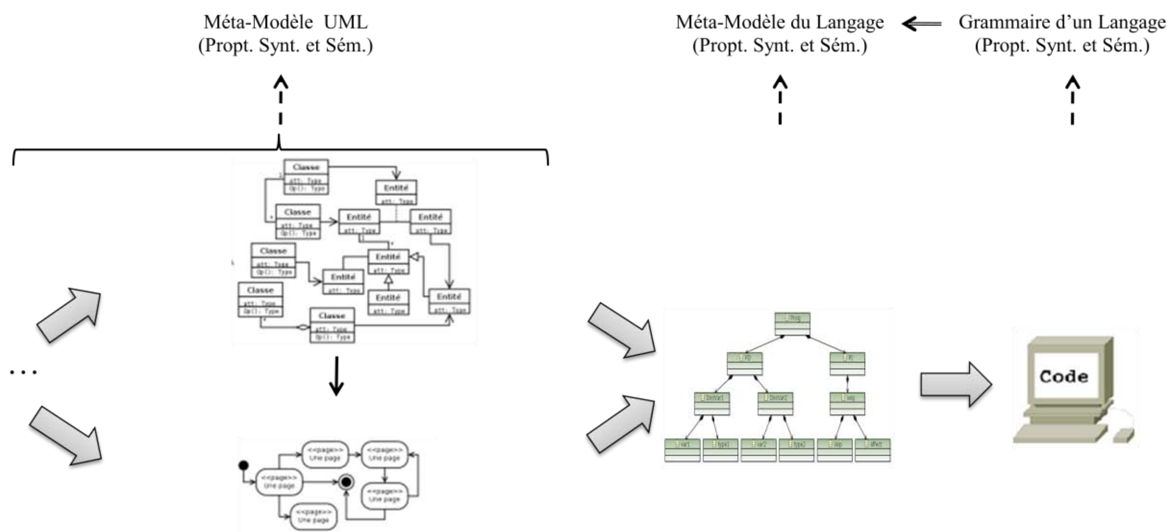


Figure 14 : ***Génération de codes réalisée à partir d'un diagramme d'activité et d'un diagramme de classes***

Cette figure suppose que l'activité de conception se termine par un diagramme d'activité contenant sans doute le modèle d'un organigramme ou d'un algorithme dont la spécification est définie dans un diagramme de classes. Au niveau du Méta-Modèle du langage, nous avons modélisé les propriétés syntaxiques et sémantiques du langage cible, et nous procédons de la même manière pour intégrer au niveau du Méta-Modèle UML des propriétés syntaxiques et sémantiques permettant aux Analystes/Concepteurs de modéliser des algorithmes ou, plus simplement, des organigrammes.

II.1.b Raffinage de modèles au niveau du diagramme d'activité

Puisque l'on trouve dans le fragment du Méta-Modèle UML différents éléments de modélisation qui peuvent être exploités à différents niveaux d'abstraction, nous proposons d'utiliser le diagramme d'activité à deux niveaux d'abstraction, tel que le montre la figure suivante :

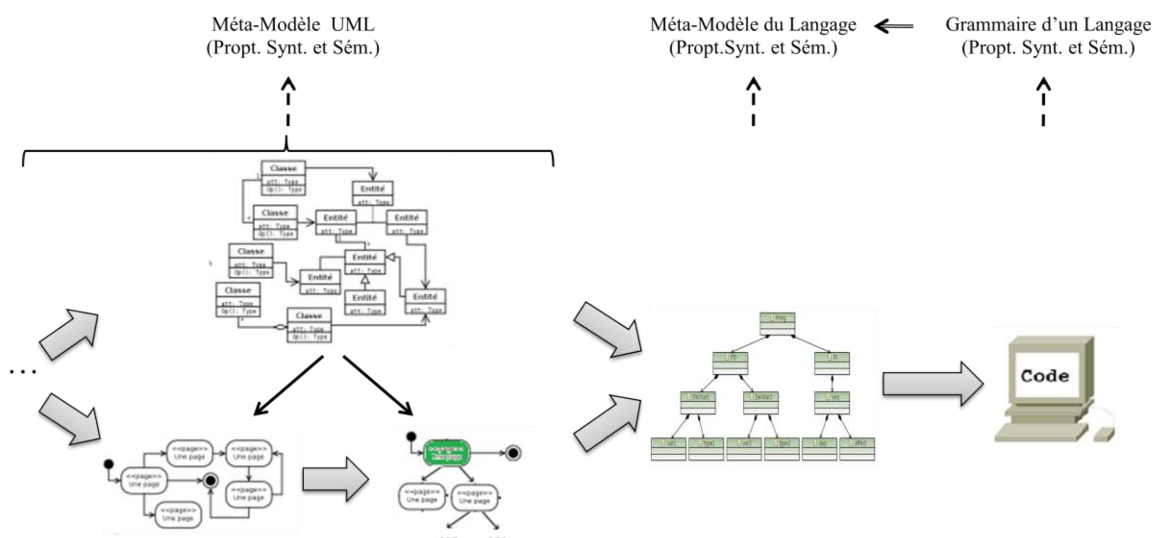


Figure 15 : ***Génération de codes faisant apparaitre deux niveaux de modèle à l'aide du diagramme d'activité***

A un premier niveau, plus abstrait, les Concepteurs peuvent modéliser le système à développer sous la forme d'un graphe orienté dont les nœuds d'activité du système à développer constituent les nœuds du graphe et dont les flots de contrôle représentant leurs enchainements constituent les transitions du graphe. Puis, à un niveau de raffinage plus détaillé anticipant ainsi la génération de code, le modèle de graphe peut être transformé en une structure arborescente dont les actions élémentaires que le système à développer doit réaliser, constituent les feuilles de l'arbre, et dont les nœuds d'activité structurée décrivant les enchainements constituent les nœuds de l'arbre.

La génération de codes, réalisée à partir d'un diagramme d'activité pouvant être en relation avec d'autres diagrammes UML, peut donc se faire en deux étapes préliminaires de raffinage de modèles, une première étape transformant le modèle de graphe et un modèle d'arbre, tous deux conformes au Méta-Modèle UML, puis une étape de transformation de ce modèle d'arbre en un modèle conforme au Méta-Modèle du langage cible, en l'occurrence le langage « L ».

Les deux niveaux de modélisation du diagramme d'activité permettent de préciser pour chacun d'entre eux les propriétés syntaxiques que devront donc respecter les Concepteurs s'ils veulent suivre le processus suggéré. Dans ces conditions, les propriétés comportementales pourront donc en être clarifiées. Le processus IDM proposé est donc un processus de type incrémental vérifiant à chaque niveau les propriétés des modèles et assurant que les transformations sont correctes.

II.1.c Propriétés sur les modèles

Au niveau du diagramme d'activité modélisant le système à développer sous la forme d'un graphe, il est composé d'un ensemble de nœuds d'activité dont les enchainements sont décrits par des flots de contrôle. On considèrera que le modèle de graphe ne contient pas de nœuds d'activité structurée pour bien le distinguer d'un modèle représentant une structure arborescente. Par rapport à notre cas d'étude limitée à la programmation séquentielle, les propriétés syntaxiques d'un modèle de graphe d'un diagramme d'activité sont les suivantes :

- gr1 : Il n'a pas de nœuds d'activité structurée.
- gr2 : Toute action reçoit un flot de contrôle entrant et est à l'origine d'un flot de contrôle sortant.

En supposant que l'objet da de type Activity référence un diagramme d'activité contenant un modèle de graphe, ces propriétés OCL sont les suivantes :

```
-   gr1 : da.node->forAll( en | not en.oclIsKindOf( StructuredActivityNode )
-   gr2 : da.node->forAll( ac | ac.oclIsTypeOf( Action ) implies   ac.incoming->size() = 1 and
                                                                   ac.outcoming->size() = 1      )
    …
```

Ces propriétés syntaxiques complètent les propriétés déjà évoquées au chapitre précédent, en particulier celles que nous avions définies pour les nœuds de contrôle.

Au niveau du diagramme d'activité modélisant le système à développer sous la forme d'une structure arborescente, il ne contient que des nœuds d'activité structurée, et des actions élémentaires, ces dernières n'étant reliées à aucun flot de contrôle qu'il soit entrant ou qu'il soit sortant, sauf en ce qui concerne l'action qui se trouve à la racine de l'arbre. Ce nœud

reçoit un flot de contrôle venant du nœud initial. De ce nœud est issu un flot de contrôle sortant vers le nœud d'activité finale, après avoir, bien sûr, réalisé le traitement en fonction de sa propriété comportementale. Par rapport à notre cas d'étude limitée à la programmation séquentielle, les propriétés syntaxiques d'un modèle d'arbre d'un diagramme d'activité sont les suivantes :

- ar1 : Le modèle n'est composé que d'actions sauf en ce qui concerne le nœud initial et le nœud d'activité final
- ar2 : Les actions ne sont reliées à aucun flot de contrôle sauf en ce qui concerne la racine de l'arbre.

En supposant que l'objet da référence un diagramme d'activité contenant un modèle arborescent de graphe, ses propriétés OCL sont les suivantes :

```
ar1 : da.node->forAll(   ac |   ac.oclIsKindOf( Action      ) or
                                ( ac.oclIsTypeOf( IntialNode )  or ac.oclIsType( FinalActivityNode ) )
                               )
ar2 : da.node->forAll( ac |
           ( not ac.incoming->asSequence()->first().oclIsTypeOf( InitialNode )  or
             not ac.outgoing->asSequence()->first().oclIsTypeOf( FinalActivityNode )  )
                                        implies
                    ac.outgoing->size() = 0 and ac.outgoing()->size() = 0
                              )
…
```

Ces propriétés syntaxiques complètent les propriétés déjà évoquées au chapitre précédent, en particulier celles que nous avions définies pour les nœuds de type Action.

Ces propriétés syntaxiques du modèle de graphe et du modèle d'arbre sont contradictoires, ce qui est gênant puisque ces modèles sont des instances du même Méta-Modèle. En effet, on impose en particulier que tout nœud exécutable dans le modèle de graphe doit être en lien avec des flots de contrôle qu'ils soient entrants ou sortants. Mais, dans le modèle d'arbre, une action n'a pas de lien avec les flots de contrôle sauf pour le nœud initial et le nœud d'activité finale. En fait, les propriétés énoncées dans ce paragraphe sont des pré-conditions et des post-conditions pour les transformations de modèles :

- Les propriétés gr1, gr2, … sont des pré-conditions de la transformation du modèle de graphe en modèle d'arbre.
- Les propriétés ar1, ar2, … sont des post-conditions de la transformation du modèle de graphe en modèle d'arbre, et des pré-conditions de la transformation du modèle d'arbre vers le modèle de code.

En ce qui concerne les propriétés comportementales, ce sont les mêmes que celles énoncées au chapitre précédent. En effet, la sémantique comportementale d'un nœud d'activité d'un modèle de graphe (operation exec) est définie de façon différente selon qu'il est de type ExecutableNode ou ControlNode, alors que la sémantique comportementale d'une action d'un modèle d'arbre (opération execAction) est définie de façon différente selon qu'elle représente un nœud d'activité structurée ou une action élémentaire.

Avant d'étudier les transformations proposées dans le cadre de cette étude, nous présentons préalablement, au paragraphe suivant, les différents travaux de recherche déjà existants dans ce domaine. Nous resituons ainsi nos travaux dans ce contexte. Cependant, ces transformations doivent se placer dans un processus IDM incrémental déjà bien défini et bien structuré.

II.2 Quelques éléments bibliographiques sur les transformations de modèles

Les transformations des modèles ont un rôle essentiel en Ingénierie Dirigée par les Modèles. Il s'agit d'étudier les transitions de modèles entre les différentes phases principales du processus en Y de l'IDM.

II.2.a Modèle source, Modèle cible et transformation de modèles

D'une manière générale, une transformation de modèles est définie comme une génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources, conformément à une définition de transformation [WP5], tel que le montre la figure suivante :

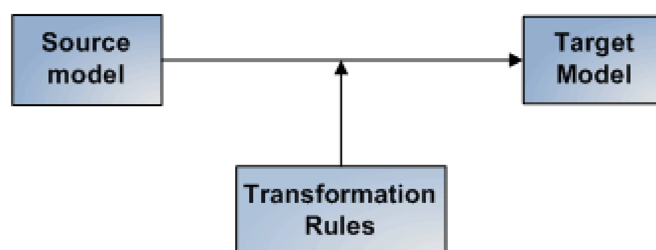


Figure 16 : ***Transformation des modèles***

La figure suivante remplace les fonctions de transformation par rapport à leur Méta-Modèle :

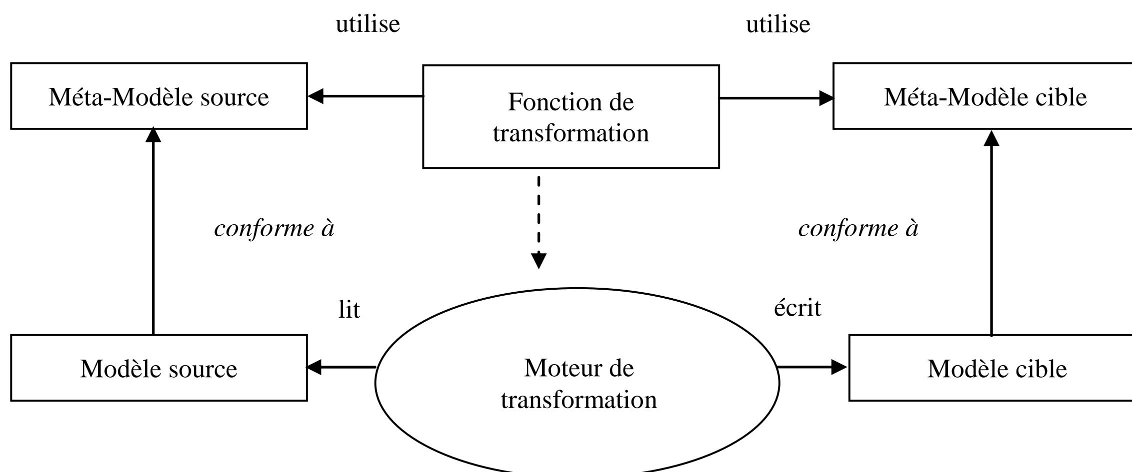


Figure 17 : ***Concepts de base de la transformation de modèles***

Cette figure montre qu'à partir de règles de transformation entre le Méta-Modèle source et le Méta-Modèle cible, on en déduit le programme de transformation (Moteur de transformation) qui pourra s'appliquer sur le modèle source, instance du Méta-Modèle source. Le résultat de la transformation donnera un modèle cible qui devra être conforme au Méta-Modèle cible.

La définition la plus générale et la plus largement acceptée par la communauté de l'Ingénierie du logiciel Dirigée par les Modèles (IDM) [B04] consiste à dire qu'une transformation de modèles est la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources, conformément à une définition de transformation. Son dénominateur commun fait référence à des modèles et méta-modèles, c'est-à-dire à des graphes orientés étiquetés.

Dans l'architecture de Méta-Modélisation à quatre niveaux de l'OMG [OMG], un Méta-Modèle est considéré comme un langage d'expression de modèles. Il permet de faciliter les opérations de manipulation de modèles, comme, entre autres, la transformation qui nous intéresse dans ce document. Les approches de Méta-Modélisation permettent de décrire les concepts, les propriétés et les relations utilisés par un ensemble de modèles.

Un modèle écrit dans un Méta-Modèle donné sera dit conforme à ce dernier. Cette relation entre modèle et Méta-Modèle (conforme à) s'apparente à la relation bien connue en programmation entre une variable et son type ou entre un objet et sa classe d'appartenance si l'on considère la programmation par objets. Pour traduire cette relation, certains auteurs emploient le terme d'instanciation (instance de) en indiquant qu'un modèle est une instance d'un Méta-Modèle, mais ce point de vue est source de controverses car le terme instance est souvent utilisé dans des contextes variés.

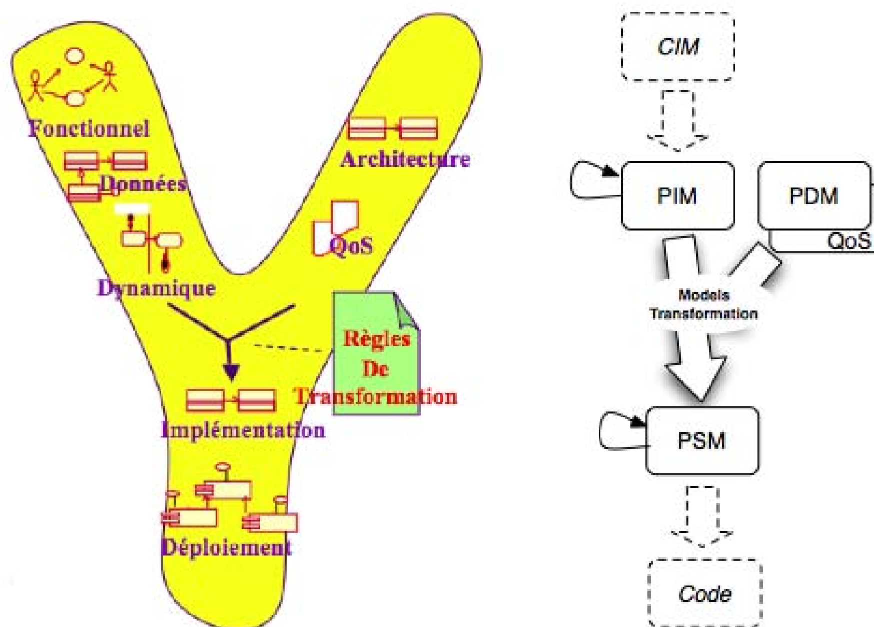
La problématique de la transformation de modèles basée sur cette technique de Méta-Modélisation peut s'énoncer dès lors de la façon suivante : « étant donné un modèle source m1 décrit par un méta-modèle MM1, il s'agit de définir un processus permettant d'obtenir un modèle m2 conforme à un méta-modèle MM2 ». Cette approche est illustrée par le schéma de la figure du paragraphe suivante.

De manière générale, un traducteur ou un transformateur de modèles est défini comme une fonction qui prend en entrée un ou plusieurs modèles sources, et crée en sortie un ou plusieurs modèles cibles, en se basant sur les informations des modèles sources et en tenant compte de la structure des modèles cibles. Une transformation d'éléments d'un modèle source met en jeu deux étapes :

- La première étape permet d'identifier les correspondances entre les concepts des modèles source et cible au niveau de leurs Méta-Modèles, ce qui induit l'existence « d'une fonction de transformation » applicable à tous les modèles conformes au Méta-Modèle source.
- La seconde étape consiste à appliquer la transformation du modèle source afin de générer automatiquement le modèle cible par un programme appelé « moteur de transformation » ou d'exécution.

II.2.b Architecture d'un Processus de développement de logiciels

La figure suivant montre une architecture du processus de développement proposé par [OMG], dans le cadre d'ingénierie dirigée par les modèles :



[WP5]

Figure 18 : ***Architecture d'un processus de développement proposé par l'OMG***

Avec l'aide des transformations entre différents niveaux de modélisation, ou entre différents formalismes de modélisation, le processus IDM définit une séparation claire et nette entre des abstractions (CIM, PIM, PSM…) et des formalismes (les DSMLs, les plateformes…). C'est en se basant sur la nature des Méta-Modèles source et le cible de la transformation que l'on désigne ces transformations comme endogènes ou exogènes. Les transformations appelées endogènes font référence à des modèles source et cible issus d'un même méta-modèle. Dans le cas contraire (les méta-modèles source et cible sont différents), elles sont exogènes.

II.2.c Différents types de transformation de modèles

D'autre part, comme cité ci-dessus, les transformations peuvent aussi enchaîner le changement du niveau d'abstraction ou non, d'où vient la catégorie des transformations de type verticale ou horizontale. Dans le cas où l'on passe d'une abstraction vers une autre, les transformations sont appelées verticales; sinon, elles sont dites horizontales.

Le tableau suivant montre les différents types de transformation que l'on peut trouver dans les processus :

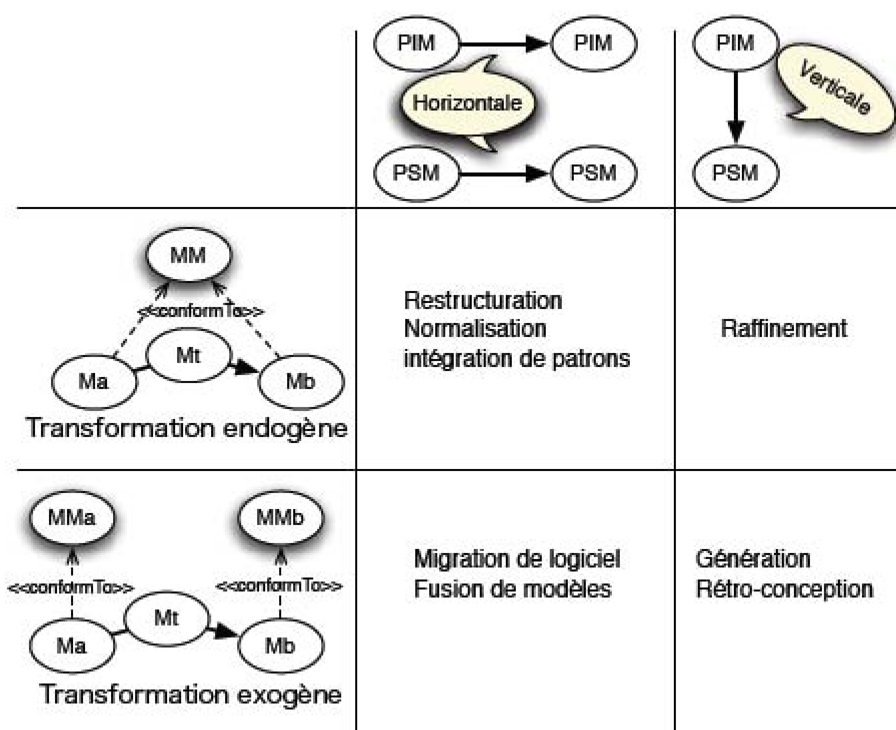


Figure 19 : ***Différents types de transformations pouvant apparaitre au cours d'un processus de développement***

III Mise en œuvre des Transformations de Modèles de l'activité de génération de codes

Il s'agit d'étudier dans ce paragraphe la mise en œuvre des transformations de modèles liées au processus IDM incrémental suggéré. Nous présentons tout d'abord la transformation d'un modèle de conception réalisé à l'aide d'un diagramme d'activité et se représentant sous la forme d'un graphe orienté composé de nœuds exécutables et de flots de contrôle, en un modèle arborescent composé de nœuds d'activité structurée et d'actions élémentaires.

Il s'agit principalement de faire en sorte que les entités relativement abstraites des modèles de conception, puissent prendre en compte les spécificités techniques liées aux langages des plates-formes cibles, et pouvant être différentes selon les exigences des applications, pour arriver par étapes de raffinage successives aux codes des applications. Les Concepteurs doivent pouvoir contrôler, voire guider, cette activité de génération de codes, en vérifiant le bon déroulement du processus à chacune de ses phases essentielles. Il s'agit, en particulier, de refaçonner les modèles décrivant les enchainements des actions que le système à développer doit réaliser, sous la forme de structures arborescentes de manière ensuite à faciliter la prise en compte des artefacts de programmation des langages des plates-formes cibles. Cette hiérarchisation des enchainements des différentes actions qui peut être remontée au niveau du diagramme d'activité dont les propriétés syntaxiques sont décrites au niveau du Méta-Modèle UML, reste donc indépendante des caractéristiques et des spécificités des langages cibles. On peut alors prendre en compte les spécificités du langage cible au niveau du Méta-Modèle du langage cible, à partir duquel la génération proprement dite des codes pourront se faire après avoir intégré les éléments de syntaxe concrète du langage cible, en l'occurrence le langage « L ».

III.1 Transformation d'un graphe de flots de contrôle en une arborescence de nœuds d'activité structurée

Cette transformation est délicate puisque, se situant dans le cadre d'un processus incrémental, il s'agit de vérifier les pré-conditions que doivent respecter les modèles sources et de s'assurer que les modèles cibles respectent les propriétés qui lui sont imposées.

De plus, il s'agit de montrer que les différentes transformations proposées sont correctes par rapport aux propriétés comportementales. D'autre part, cette transformation est lourde puisque le modèle cible est obtenu par applications successives de règles élémentaires déduites des correspondances sémantiques qui ont été faites préalablement. Il n'est pas sûr que le modèle de graphe puisse se transformer en une structure arborescente. La transformation peut donc ne pas réussir. Comme en base de données, on se situe dans un contexte transactionnel puisque le modèle source qui va être élaboré passe par différentes étapes successives de transformation, chacune devant être conforme au Méta-Modèle UML, c'est-à-dire que le modèle qui est mis à jour à chaque étape doit vérifier toutes les propriétés qui ont été rajoutées dans le Méta-Modèle.

Enfin, la mise à jour d'un modèle implique la création et l'élimination d'objets, d'associations et de compositions. Il est donc nécessaire de prévoir dans les algorithmes une fonction de « ramasse-miettes » éliminant les objets et les liens devenus inaccessibles, polluant donc la représentation graphique des modèles. Dans ces conditions, les fonctions de transformation ne peuvent plus s'écrire sous la forme d'expressions suivant une syntaxe OCL.

Nous en décrivons tout d'abord le principe de l'algorithme, puis nous montrerons comment on peut le mettre en œuvre. D'autre part, nous montrerons la pré-condition que l'on pourrait rajouter à cette transformation assurant que la transformation réussirait, ce qui correspond, en fait, à une autre mise en œuvre de l'algorithme.

III.1.a Transformations élémentaires et enchainement des transformations

Il s'agit de voir, donc, comment un enchainement d'actions décrit à l'aide de flots de contrôle (ControlFlow) peut se transformer en structure hiérarchisée de nœuds d'activité structurée, telles que la séquence (SequenceNode), le test (ConditionnalNode) et la boucle (LoopNode). C'est une transformation d'un graphe en un arbre, elle ne peut donc se faire que par itérations successives de transformations élémentaires.

III.1.b Mise en œuvre de la règle 1

La figure suivante rappelle les différents éléments à prendre en compte pour effectuer la transformation de cette règle :

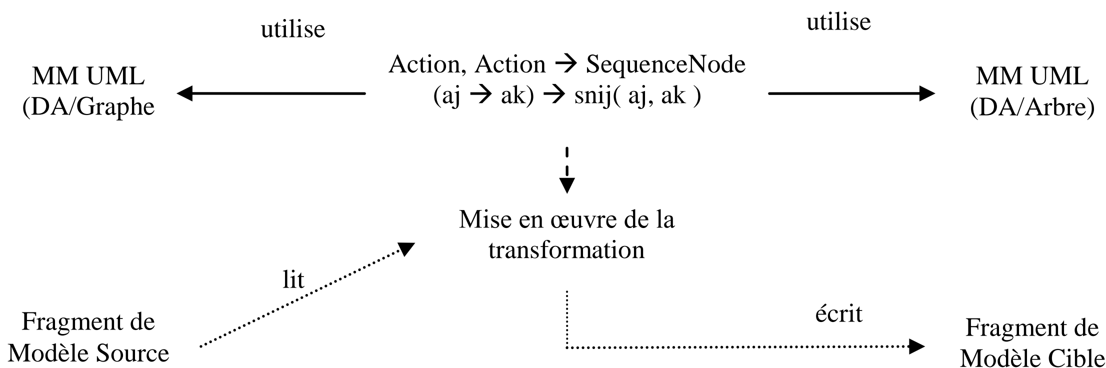


Figure 20 : ***Mise en œuvre de la première règle***

Il s'agit de transformer un fragment de modèle composé de deux actions qui se succèdent en un fragment de modèle rassemblant les deux actions rentrant dans la composition d'un nœud d'activité structurée de type SequenceNode

-- Un exemple de fragment de modèle

La figure suivante montre la représentation graphique (externe) d'un fragment de modèle composé de deux actions se succédant et la représentation graphique (externe) d'un fragment de modèle regroupant ces deux actions au niveau d'un nœud d'activité structurée de type SequenceNode :

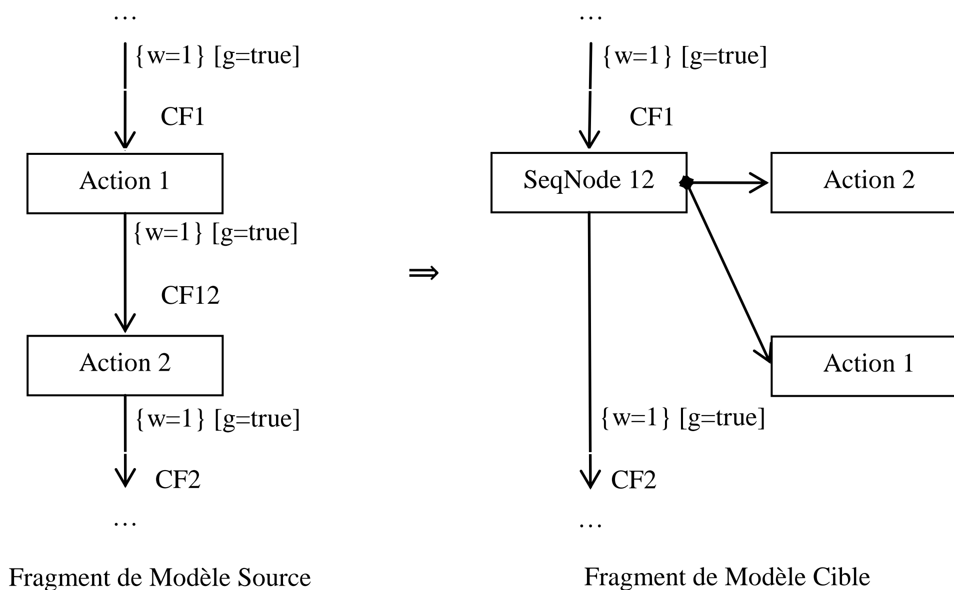


Figure 21 : ***Regroupement de deux actions consécutives en un nœud d'activité structurée de type SequenceNode***

Dans le fragment de modèle cible, les deux actions n'apparaissent pas directement dans le modèle.

-- La représentation interne du fragment de modèle

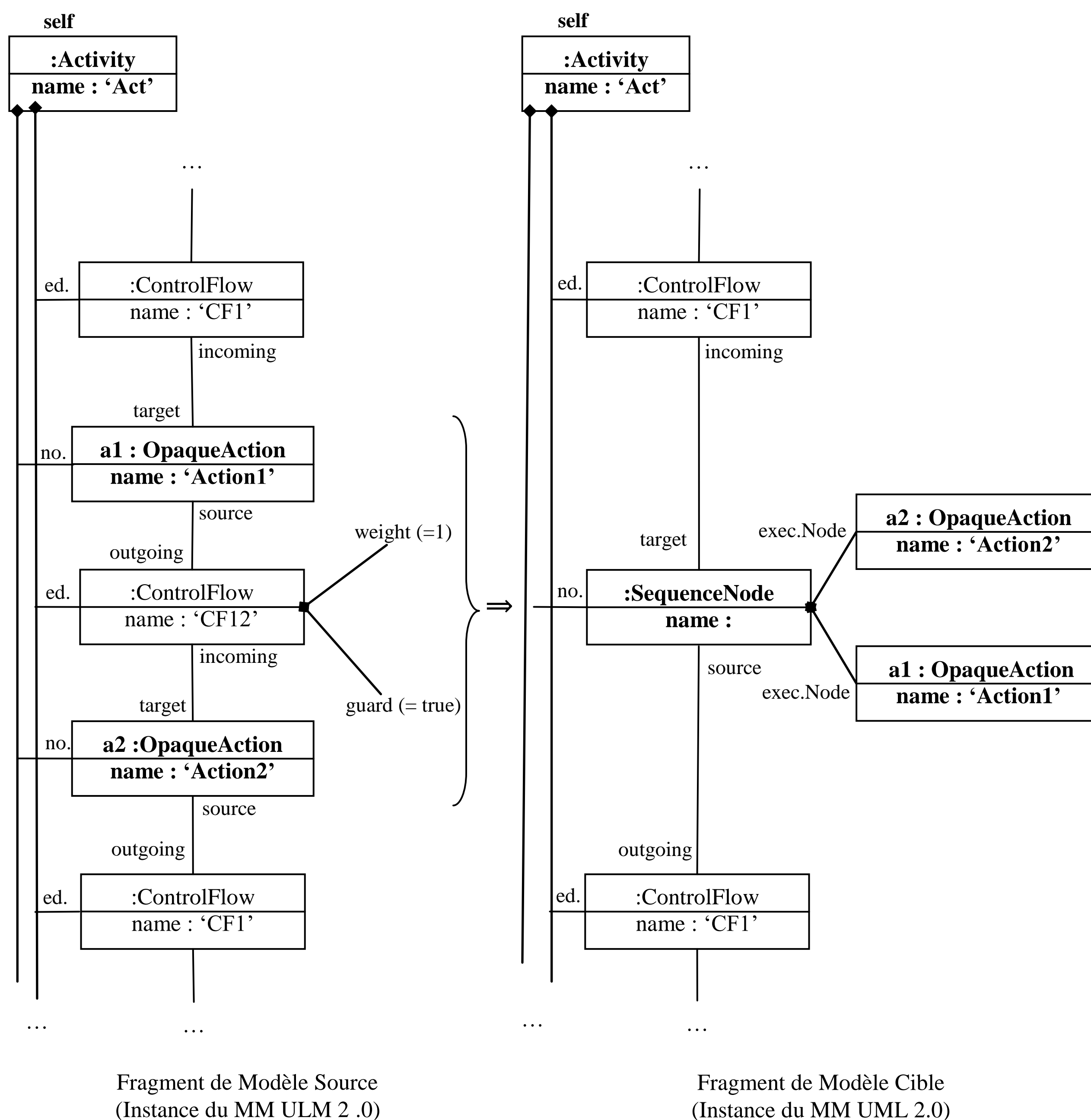


Figure 22 : ***Représentation interne (Instance du Méta-Modèle UML 2 .0) des deux fragments de modèle***

-- <u>Fonction recherchant un objet de type Action qui a est suivi par un objet de type Action</u> :

```
Activity ::pre_CondActionAction2SequenceNode() : Action =

    let         ens_act1 : set( Action )  = self.node->select( act |
                                                  act.oclIsKindOf( Action )    and
                                                  act.outgoing.target.oclIsKindOf( Action )
                                                )->asSet()                                    -- 1
    in  if      ens_act1->size() > 0
        then    ens_act1->asSequence()->at( 1)
        else    oclUndefined( Action )
```

```
endif
```

-- Fonction construisant un objet de type SequenceNode composée des deux actions dont la première est a1 :

```
Activity ::SequenceNode( a1 : Action ) : SequenceNode
begin
  var        a2 :        Action
  set        a2 := a1.outgoing.target->asSequence()->first().oclAsType( Action )
  var      sn :            SequenceNode
  create   sn := new       SequenceNode
  insert ( self, sn ) into Activity_ActivityNode
  set      sn.name := a1.name.concat( '_' ).concat( a2.name )
  insert (   sn, a1                    ) into SequenceNode_ExecutableNode
  insert (   sn, a2                    ) into SequenceNode_ExecutableNode
  set result := sn
end
```

-- Transformation de l'action a1 et de sa suivante en un nœud d'activité structurée de type SequenceNode :

```
Activity::actionAction2SequenceNode( a1 : Action  )
Begin
    var cfa1 : Control                                                    -- cf
    set cfa1 := a1.incoming->asSequence()->first().oclAsTypeOf( ControlFlow )
                                                                          -- a1
    var cf12 : ControlFlow                                                -- cf12
    set cf12 := a1.outgoing->asSequence()->first().oclAsTypeOf( ControlFlow )
    var a2 : Action                                                       -- a2
    set a2 := cf12.target.oclAsType( Action )
    var cfa2 : ControlFlow                                                -- cfa2
    set cfa2 := a2.outgoing->asSequence()->first().oclAsTypeOf( ControlFlow )
    var g : ValueSpecification
    set g := cf12.guard
    var w : ValueSpecification
    set w := cf12.weight

--                      elimination des liens entre self et les noeuds executables  des 2 actions
--                                     et self et le  noeud  de controle les 2 actions
    delete ( self, a1 )         from Activity_ActivityNode
    delete ( self, a2 )         from Activity_ActivityNode
    delete ( self,    cf12 )    from Activity_ActivityEdge
--                      elimination des liens entre le CF en debut  des 2 actions
--                                     et le CF terminant les 2 actions
    delete (    a1,  cf )        from ActivityNode_ActivityEdge_2
    delete (    a1,  cf12 )      from ActivityNode_ActivityEdge_1
    delete (    a2,  cf12 )      from ActivityNode_ActivityEdge_2
    delete (    a2,  cfa2 )      from ActivityNode_ActivityEdge_1
--                      elimination du lien entre le flow  de controle liant a1 et a2, et la garde
    delete (        cf12, g ) from ActivityEdge_ValueSpecification_1
    delete (        cf12, w ) from ActivityEdge_ValueSpecification_2

--                      destruction du flot de controle et de sa garde
    destroy g
    destroy w
    destroy cf12
--                      creation du noeud de type SequenceNode avec les 2 actions
    var    sn :          SequenceNode
    set    sn :=     self.SequenceNode( a1 )
```

```
--                                        rattachement du nœud de seq. au diagramme d activite
     insert (  sn, cfa2 ) into ActivityNode_ActivityEdge_1
     insert (  sn, cf    ) into ActivityNode_ActivityEdge_2

  end
```

Figure 23 : ***Mise en œuvre d'une transformation élémentaire hiérarchisant un diagramme d'activité***

-- Post-condition de la transformation

Le fragment du modèle cible est composé d'un nœud d'activité structurée de type SequenceNode composé désormais des deux actions du modèle source. En fait, ce fragment de modèle cible doit vérifier la propriété correspondante à la pré-condition de la transformation qui aboutit au modèle du langage, instance du Méta-Modèle du langage. Cette post-condition est donc la suivante :

```
SequenceNode ::post_CondActionAction2SequenceNode( a : Action ) : Boolean
        if        self.oclIsTypeOf( SequenceNode )
        then      self.executableNode->asSequence()->at( 1 ).oclIsKindOf( Action ) and
                  self.executableNode->asSequence()->at( 2 ).oclIsKindOf( Action )
        else false
        endif
```

-- Vérification de la transformation

Les vérifications de la transformation sont déjà prévues en tant que pré-conditions et post-conditions de la transformation. Ce sont donc des vérifications dynamiques. Les méthodes et techniques des traducteurs, comme on l'a vu précédemment, sont basées sur des vérifications statiques s'appliquant directement sur les codes. On ne peut donc pas affirmer que la transformation écrite précédemment soit correcte, bien que ce soit une transformation élémentaire. Il est donc nécessaire de prolonger cette étude dans ce sens.

Nous avons, en fait, deux types de vérification pour une telle transformation : une vérification structurelle et une vérification portant sur la sémantique comportementale des fragments de modèles source et cible.

La vérification structurelle est assurée dynamiquement par la pré-condition et la post-condition de la transformation. Il faudrait donc, comme cela a été fait dans le cadre des triplets de Hoare vérifiant des fragments de modèles de code du langage « L », écrire le programme simulant cette vérification sous la forme d'un triplet de Hoare et traduire ce programme B qui se chargera de vérifier si cette transformation est correcte. Ecrire le programme simulant un tel triplet de Hoare est possible. Cependant, la transformation étant écrite en langage OCL/LA et les propriétés en OCL portant sur le diagramme d'activité, la traduction en B relève des techniques de compilation qui n'ont par été abordées dans cette étude. Les fonctions calculant la Plus Faible Pré-condition des instructions du langage d'actions UML, qui est un langage objet, et de la substitution de variables dans une expression OCL n'ont pas été réalisé. La gestion des associations et des compositions auraient du être définies préalablement. Les Concepteurs peuvent cependant faire des tests dynamiques en choisissant des modèles sources pertinents.

Les deux fragments de modèle source et cible doivent avoir des comportements identiques sur un même environnement. Ce qui signifie que l'exécution d'un fragment de modèle cible issu de la transformation d'un fragment de modèle source doit avoir le même effet que l'exécution du fragment source sur des environnements identiques. Cette propriété peut s'exprimer à l'aide d'une fonction OCL/LA que les Concepteurs peuvent vérifier sur des exemples significatifs. En présentant l'algorithme de la transformation d'un tel fragment de modèle, nous avons déjà défini en début du chapitre le sens que l'on donne à cette équivalence comportementale à l'aide de deux environnements d'exécution (*Figure 2*) et la propriété qui en découle (*Figure 3*). De cette propriété, on peut en déduire sa transformation en B. Sans aller jusqu'à une validation statique en B, les Concepteurs peuvent vérifier cette propriété dynamiquement.

III.1.c Mise en œuvre de la règle 2, 3 et 4

La mise en œuvre des trois autres règles se base sur le même principe que la mise en œuvre de la règle 1.

Nous montrons seulement la représentation graphique des fragments de modèle impliqués dans ces transformations. Ces différentes représentations externes des modèles montrent toutes les informations à prendre en compte pour réaliser ces transformations, en tenant compte, bien sûr, de leur représentation donnée par le Méta-Modèle du langage UML. La figure suivante rappelle les propriétés et montre les représentations externes des fragments de modèle de la deuxième et troisième règles des transformations élémentaires :

-- Règle 2 : MergeNode, Action, DecisionNode → LoopNode

mj → dnl ; dnl(glk) → ak ; ak → mj ; dnl(glm) → … ⇒ LoopNode(glk, ak)

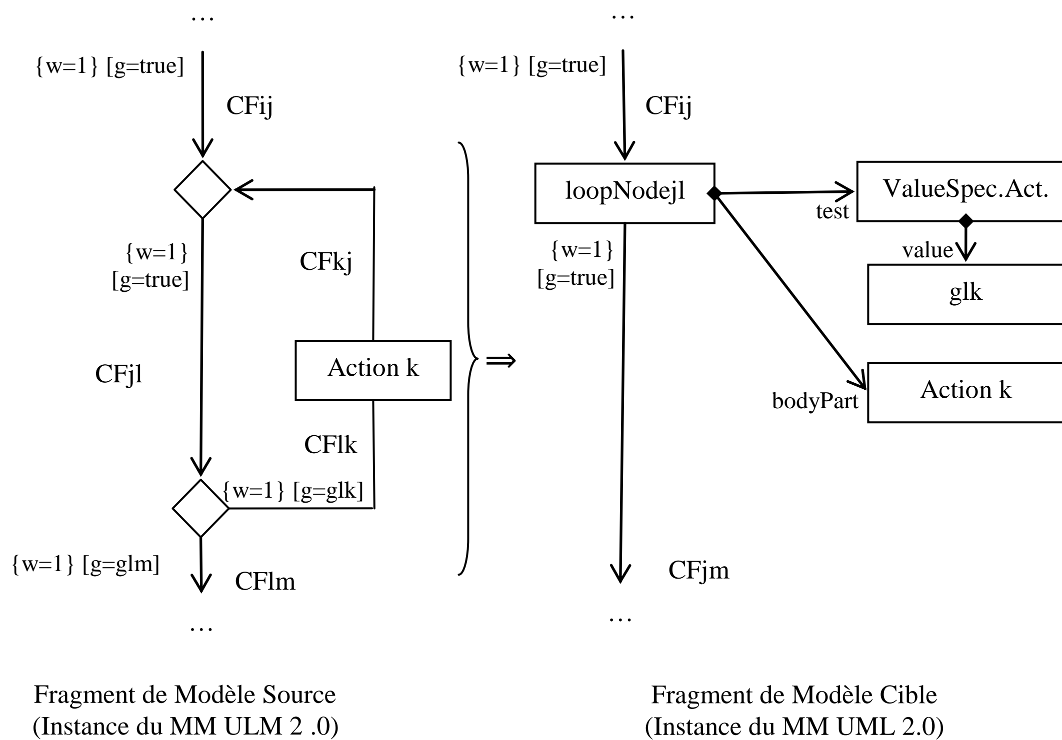


Figure 24 : ***Fragment de modèle transformable en un nœud d'activité de type LoopNode***

R3 : DecisionNode, Action, MergeNode → ConditionalNode

dnj → ml ; dnj(gjk) → ak ; ak → ml ⟹ ConditionalNode(gjk, ak))

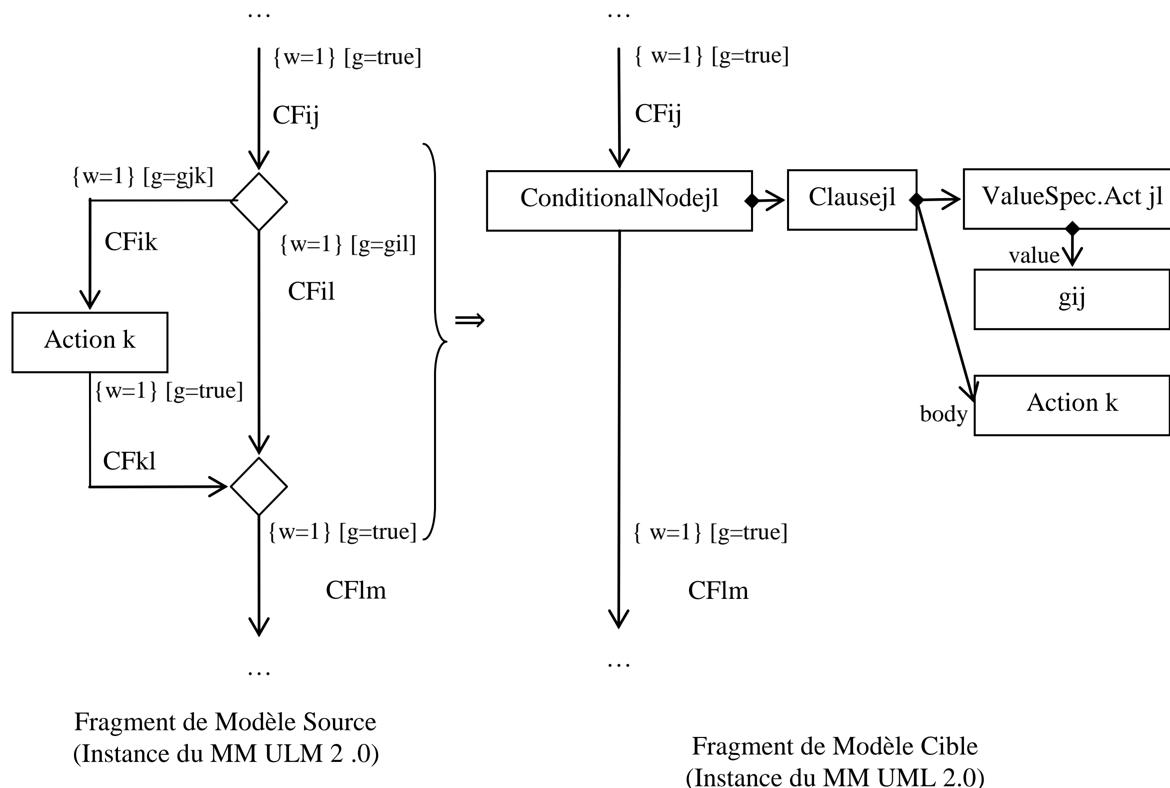


Figure 25 : ***Fragment de modèle transformable en un nœud d'activité structurée de type ConditionalNode***

III.1.d Pré-condition à la hiérarchisation : Algorithme basé sur la trace du graphe

L'algorithme de transformation du modèle de graphe en un modèle d'arbre peut être relativement lourd si le modèle source contient un grand nombre d'actions. Il s'agit préalablement de faire une copie du modèle, de procéder ensuite par itérations successives de transformations élémentaires exigeant à chaque étape une mise à jour importante du modèle. De plus, la transformation peut ne pas aboutir, et dans ce cas, il faut effacer tout l'environnement mis en place pour réaliser cette opération.

Il est donc nécessaire de voir si l'on peut trouver une solution pour vérifier que le modèle source est « hiérarchisable », avant d'effectuer cette transformation.

En fait, il en existe plusieurs solutions. Nous avons retenues celle qui consiste à tracer les éléments de modélisation du graphe en fonction de leur appartenance à un sous-arbre que l'on peut définir dans le graphe. Cette solution a été choisie parce que d'une part, on reprend le même algorithme qui transforme le modèle de graphe en un modèle d'arbre, et parce que d'autre part, on peut exécuter cet algorithme de marquage comme on a exécuté le modèle d'un programme à l'aide d'un environnement d'exécution qui enregistre ce marquage sans modification du modèle source.

Il suffit de procéder par itérations successives comme on l'a fait précédemment, mais au lieu d'effectuer la transformation élémentaire correspondante, on identifie les éléments correspondants comme faisant partie d'un sous-arbre du graphe, sous-arbre qui prend le nom du nœud qui est à la racine du sous-arbre. Si on reprend la première règle, tel que le montre la

figure suivante, il suffit d'enregistrer que les actions j et k font partie d'un sous-arbre constitué des actions j et k, et dont la racine est l'action j :

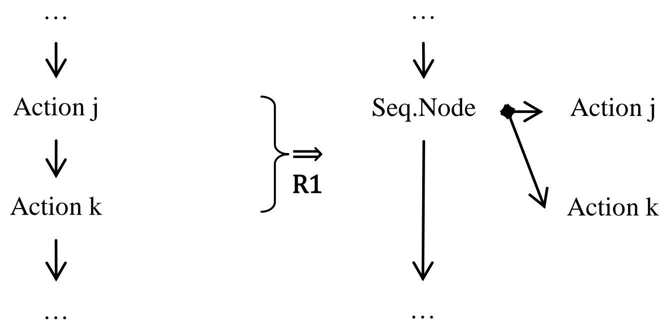
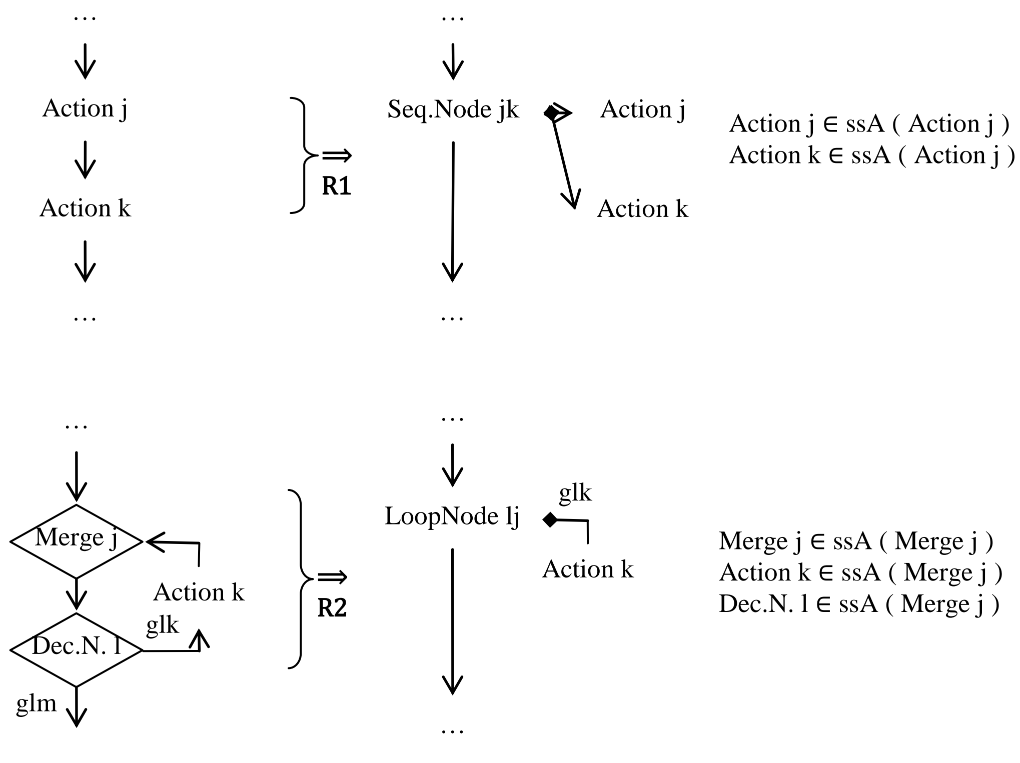


Figure 26 : ***Segment de modèle permettant d'identifier un sous-arbre, et les nœuds qui le composent***

Cette règle (1) montre qu'au lieu de faire la transformation, on enregistre les éléments appartenant à un sous-arbre.

Des règles de transformation explicitées plus haut, on en déduit les règles indiquant pour chaque nœud, repéré par son nom, à quel sous-arbre il appartient. Ces règles sont les suivantes :

Action i ∈ ssA(Action i)
Action j ∈ ssA (Action i) et Action i ∈ ssA (Action j) ⇒ Action i ∈ ssA (Action j)



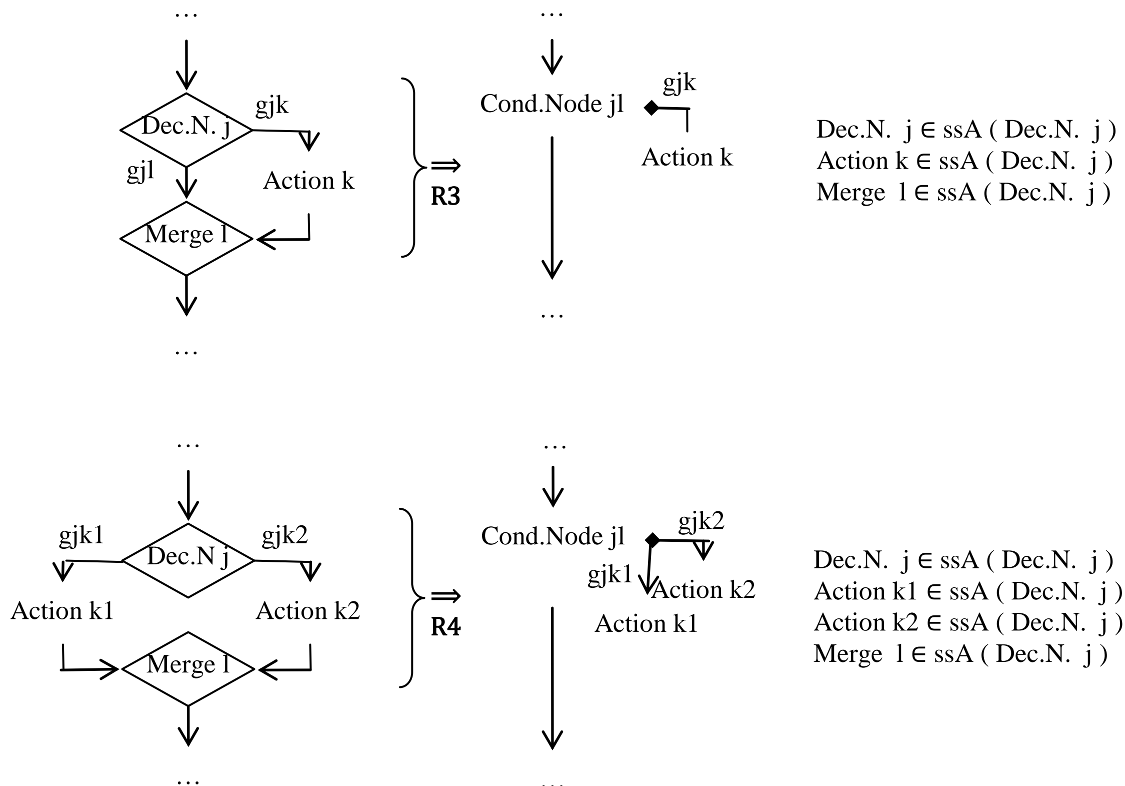


Figure 27 : ***Règles définissant pour chaque nœud d'activité à quel sous-arbre il appartient***

La première règle rappelle que chaque nœud d'activité appartient à un sous-arbre (ssA), qui porte son nom. La deuxième règle montre que l'appartenance d'un nœud à un sous-arbre se déduit par transitivité. Les autres règles montrent que l'on peut déduire des règles de transformations énoncées précédemment à quel sous-arbre appartient chaque nœud du fragment correspondant.

En reprenant l'exemple du diagramme d'activité (*Figure 11*) montrant comment le modèle de graphe se transforme en un modèle arborescent, on peut suivre sur les quatre tableaux suivants comment l'on peut déterminer et en déduire, par itérations successives, le sous-arbre auquel appartient chaque nœud d'activité :

Nœud d'activité	**Sous-Arbre**
Act. i	Act. i
Dec.N. j	Dec.N. j
Act. k1	Act. k1
Act. k2	Act. k2
Merge l	Merge l
Act. M	Act. m

(1)

Nœud d'activité	**Sous-Arbre**
Act. i	Act. i
Dec..N. j	Cond.N. jl
Act. k1	Cond.N. jl
Act. k2	Cond.N. jl
Merge l	Cond.N. jl
Act. m	Act. l

(2)

Nœud d'activité	**Sous-Arbre**
Act. i	Seq.N. ijl
Dec..N. j	Seq.N. ijl
Act. k1	Seq.N. ijl
Act. k2	Seq.N. ijl
Merge l	Seq.N. ijl
Act. m	Act. m

(3)

Nœud d'activité	Sous-Arbre
Act. i	Act. i
Dec..N. j	Act. i
Act. k1	Act. i
Act. k2	Act. i
Act. j1	Act. i
Merge l	Act. i
Act. m	Act. i

(4)

Figure 28 : ***Exemple de recherche d'une composante fortement connexe dans un graphe de flots de contrôle***

Chaque ligne d'un tableau donne le nom d'un nœud d'activité et le nom du sous-arbre auquel il appartient.

Au début de l'algorithme, le tableau est initialisé : dans la colonne de gauche on retrouve tous les nœuds d'activité du graphe, et pour chacun d'entre eux le même nœud puisque chaque nœud appartient à son propre sous-arbre. Il suffit donc d'itérer comme précédemment. En particulier, le fragment de modèle transformable en un nœud de type ConditionalNode, montre que les nœuds DecisionNode j, Act. k1, Act. k2 et MergeNode l appartiennent au sous-arbre DecisionNode.

Le tableau (4) montre que tous les nœuds appartiennent au même sous-arbre Act. i. Donc, le modèle de ce diagramme d'activité est hiérarchisable. La transformation peut alors se faire, et elle pourra aboutir. On en déduit que ce diagramme d'activité pourra s'exécuter, soit sous la forme de graphe, soit sous la forme d'arbre ! si bien sûr les calculs qui y sont décrits sont corrects.

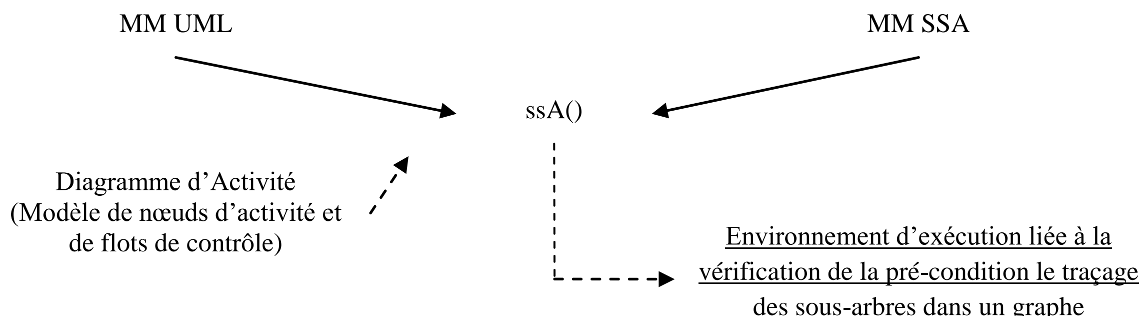


Figure 29 : ***Environnement d'exécution pour le calcul de la pré-condition de hiérarchisation d'un modèle***

Cette figure reprend celle que nous avions définie pour montrer peuvent s'exécuter les propriétés comportementales du langage « L ». Le principe est ici le même, le diagramme d'activité est une instance du Méta-Modèle UML et on peut considérer que les tableaux traçant les sous-arbres dans le modèle source sont assimilés à un environnement d'exécution dont la structure définissant un ensemble d'associations de noms est décrite par un Méta-Modèle appelé MM SSA.

III.1.e Hiérarchisation d'un diagramme d'activité

La figure suivante montre le processus réalisant la transformation d'un modèle d'un diagramme d'activité décrivant l'enchainement des actions à l'aide de flots de contrôle en un modèle décrivant cet enchainement à l'aide d'une structure hiérarchisée de nœuds d'activité structurée :

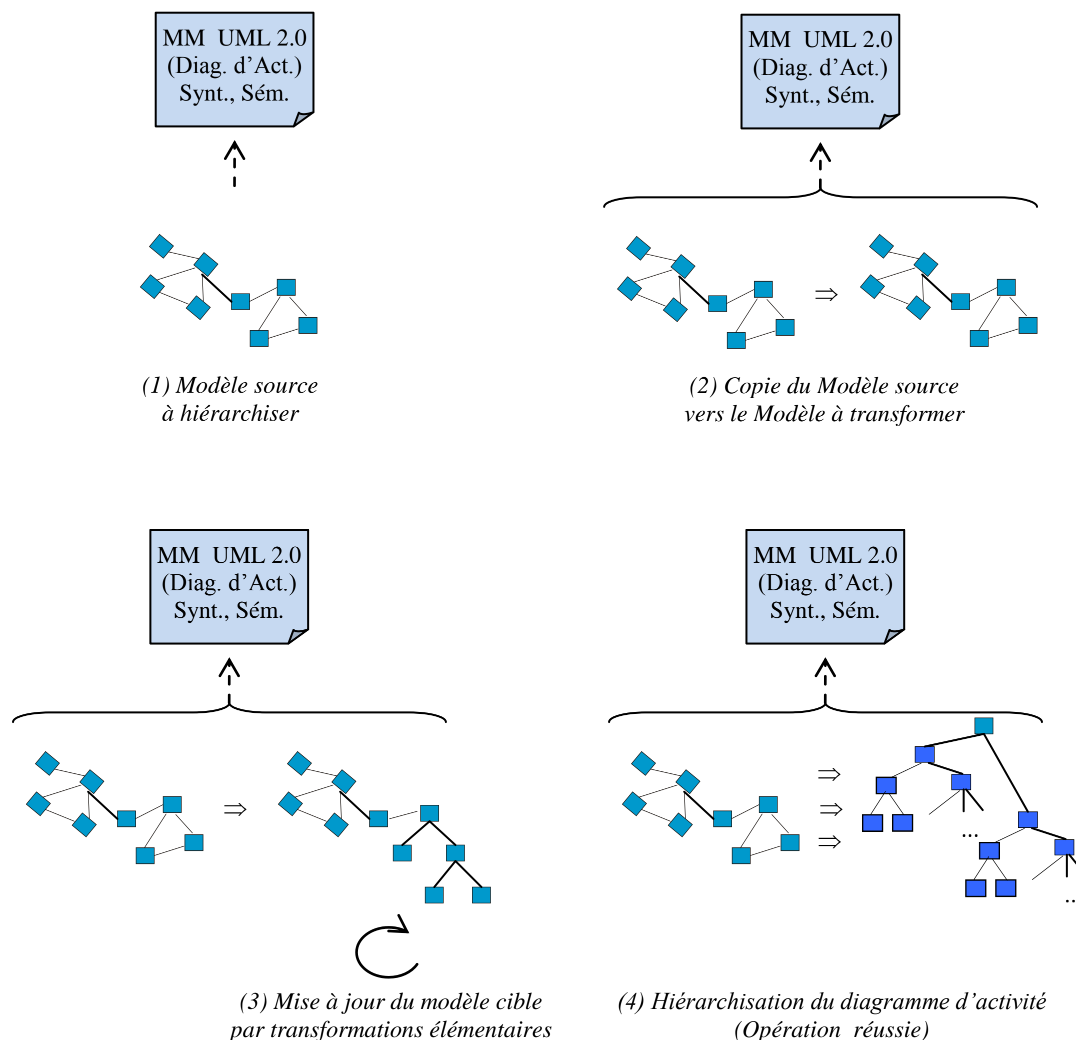


Figure 30 : ***Processus de hiérarchisation d'un modèle d'un diagramme d'activité***

En partant d'un modèle source (1), il s'agit de procéder tout d'abord par recopie (2) du fragment de modèle source à transformer. C'est donc sur cette copie que la transformation se fera, par itérations successives de transformations élémentaires (3), jusqu'à ce que l'on ne trouve plus de fragment de modèle à hiérarchiser, si l'opération aboutit (4).

Lors de la mise à jour du modèle (3), l'enchainement des différentes actions est décrit à l'aide de nœuds de contrôle et de nœuds d'activité structurée. Ce modèle en cours de restructuration doit donc être conforme au Méta-Modèle UML.

L'opération de hiérarchisation d'un modèle d'un diagramme d'activité est donc la suivante, en supposant que le modèle de nœuds exécutable et de flots de contrôle a déjà été recopié, faisant donc l'objet d'un autre diagramme d'activité :

```
Activity ::graphe2Arbre() : Activity
begin
```

```
  var arbre : Activity
  if self.pre_condSSA() and self.pre_Typage()
  then var arbre : Activity
        set arbre := self.copy()
        var b : Boolean
        set b := true
        while b
        do
        set b:= false
                                                          → SequenceNode
        if arbre.pre_CondActionAction2ActionAction() <> oclUndefinedAction( Action )
        then execute arbre.actionAction2SequenceNode(self.pre_CondActionAction() )
             set b := true
        endif
                                                          → LoopNode

        if arbre.pre_CondMergeDecisionNode2LoopNode() <> oclUndefinedAction( Action )
        then execute arbre.mergeDecisionNode2LoopNode(
                                              arbre. pre_CondMergeDecisionNode () )
             set b := true
        endif
        …
                                                          → ConditionalNode (simple)
        …
                                                          → ConditionalNode (double)

        wend
        set result := arbre
  else set result := oclUndefined( Activity )
  endif
end
```

III.1.f Validation de la hiérarchisation d'un diagramme d'activité contenant un modèle de graphei

La vérification de la transformation a déjà été évoquée lors de l'une des transformations élémentaires. On retrouve donc, ici aussi, les pré-conditions et les post-conditions à la transformation du modèle, avec, en plus, une pré-condition portant sur la propriété assurant que la transformation réussira ou pas. Au niveau de la transformation globale, on a donc plusieurs types de propriétés qui sont les suivantes :

- Des propriétés de typage portant sur le modèle source qui est composé de nœuds d'activité et de flots de contrôle et sur le modèle cible qui doit être composé de nœuds d'activité structurée et d'actions élémentaires.
- Des propriétés liées au fait que l'on transforme un graphe en un arbre.
- Des propriétés sémantiques traduisant des comportements équivalents, entre le modèle source et le modèle cible.

On peut donc reprendre au niveau de la transformation les remarques qui ont été faites à l'occasion des transformations élémentaires.

Par contre, cette transformation du graphe en arbre est intéressante pour vérifier qu'un modèle d'algorithme réalisé à l'aide d'un diagramme d'activité est correct et de pouvoir intégrer au niveau de la structure arborescente des triplets de Hoare vérifiant que des

fragments de modèles d'un diagramme d'activité sont corrects, en reprenant exactement ce qui a été fait au niveau du modèle du langage « L ». Vérifier qu'un fragment de modèle est correct peut donc se faire à un niveau d'un diagramme d'activité, avec la possibilité d'en suivre l'évolution à la suite de ses diverses transformations.

III.2 Transformation d'un diagramme d'activité hiérarchisé en un modèle de code

Enfin, il reste à présenter la transformation d'un modèle de nœuds d'activité structurée et d'actions élémentaires d'un diagramme d'activité, instance du Méta-Modèle UML en un modèle de programme d'un langage de programmation. C'est une transformation relativement classique qui correspond à du raffinement de modèle de programme prenant en compte les spécificités du langage cible, en l'occurrence le langage « L », dont le Méta-Modèle a été défini à partir de la grammaire abstraite du langage. La correspondance entre un modèle de nœuds d'activité structurée et d'actions élémentaires d'un diagramme d'activité et un modèle du langage « L » est très proche dans la mesure où l'on s'est limité aux mêmes types d'instructions de contrôle, que ce soit au niveau du Méta-Modèle du langage ou au niveau du Méta-Modèle UML.

On décrit donc, dans ce paragraphe, les principales caractéristiques de la transformation pour ces types de modèles. Il aurait cependant été intéressant d'introduire au niveau du diagramme d'activité des instructions de manipulation d'ensembles de données, et au niveau de la grammaire du langage « L », la notion de tableau de données et la notion de pointeur, de manière à laisser aux Analystes/Programmeurs le choix entre différentes implantations possibles lors de cette transformation de modèles.

III.2.a Environnement d'exécution de la transformation

Une telle transformation de modèles fait référence à deux Méta-Modèles différents. Le Méta-Modèle UML pour le modèle source, et le Méta-Modèle du langage « L » pour le modèle cible. Sur chacun des Méta-Modèles, des propriétés comportementales ont été définies.

Il faut définir les propriétés qui permettent de vérifier que la transformation est correcte. Comme précédemment, on peut définir les propriétés vérifiant le typage du modèle source (nœuds de type StructuredActivityNode et nœuds de type Action) et le typage du modèle cible, instances du Méta-Modèles du langage cible.

Par contre, en ce qui concerne la vérification des propriétés comportementales de modèle source et du modèle cible de la transformation de modèles, on ne peut plus parler d'équivalence de propriétés comportementales puisque le typage du modèle source et du modèle cible peuvent ne plus être les mêmes. On retrouvait le même problème quand il s'agissait de définir la sémantique opérationnelle des différentes constructions syntaxiques du langage « L » sur l'environnement d'exécution, ou lorsqu'il s'agissait de transformer un modèle UML du langage « L » en B, où les types prédéfinis ne sont pas les mêmes (Rel et Bool, pour le langage L, et NAT et BOOL pour le langage B). Par contre, vu les restrictions faites pour le langage « L », ces types se correspondent deux à deux, puisqu'ils font référence aux mêmes types primitifs (les nombres entiers positifs, négatifs et nuls pour les types Rel et Nat).

La figure suivante montre les environnements de modélisation nécessaires vérifiant qu'une transformation est correcte par rapport aux propriétés comportementales des modèles source et cible, instances de Méta-Modèles différents :

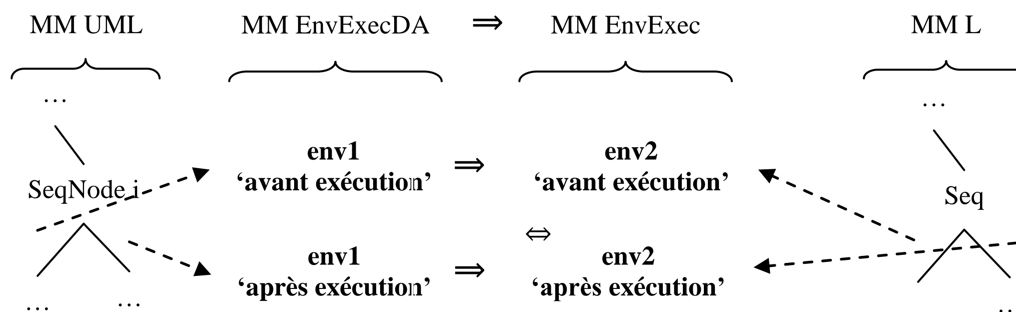


*Figure 31 : **Les environnements d'exécution pour une transformation dont les modèles source et cible ne sont pas définis par le même Méta-Modèle***

Chaque environnement est défini par son Méta-Modèle, EnvExecDA pour le modèle source, et EnvExec pour le modèle cible. Il s'agit, d'autre part, de montrer la correspondance comportementale entre les propriétés sémantiques des différentes constructions syntaxiques du modèle source et du modèle cible.

En fait, ce schéma ne fait que reprendre celui qui a été déjà mis en place, en début de ce chapitre, pour les transformations élémentaires d'un graphe en un arbre.

Nous donnons dans les paragraphes suivants le principe général de transformation des différentes constructions syntaxiques du fragment du Méta-Modèle UML concernant les nœuds d'activité structurée et les actions élémentaires.

III.2.b Déclaration des variables

Il s'agit préalablement de définir au niveau du modèle cible les variables et leur type. Dans le Méta-Modèle UML, nous avons rappelé le lien qui existait entre le diagramme d'activité et l'opération de la classe dont on trouverait le modèle de l'algorithme dans le diagramme d'activité. On peut donc effectuer les déclarations de variables dans le modèle cible ainsi que leur type dans le modèle du langage à partir de la correspondance qui en a été faite entre les deux Méta-Modèles source et cible.

III.2.c Transformation des nœuds de l'arbre : les structures de contrôle

Après avoir mis en place la partie déclarative des variables du modèle de programme, il s'agit d'écrire les opérations élémentaires transformant les nœuds d'activité structurée du diagramme d'activité en structures de contrôle du langage « L ». En fait, ces transformations concernent d'une manière générale les nœuds de l'arbre, pour lesquels on doit donc décrire pour chaque type de nœud d'activité structurée (SequenceNode, DecisionNode et LoopNode) comment ils sont modélisés en langage « L ».

III.2.d Transformation des feuilles de l'arbre : Les actions élémentaires

Il suffit ensuite de décrire les opérations élémentaires transformant les actions élémentaires du diagramme d'activité en instructions élémentaires du langage « L », en l'occurrence l'instruction d'affectation et l'instruction vide. Ces transformations concernent d'une manière générale les feuilles de l'arbre, pour lesquelles on doit donc décrire, en

particulier, la transformation en langage « L » de l'affectation comprenant d'une part la modélisation des expressions et d'autre part la variable qui recevra la valeur résultant du calcul de l'expression.

Des propriétés syntaxiques et sémantiques injectées au niveau du fragment du Méta-Modèle UML concernant les éléments de modélisation du diagramme d'activité permettent de mieux préciser leurs rôles dans un processus de développement, et donc d'en définir leurs positions. Ces propriétés, pouvant prendre en compte des spécificités des applications d'un domaine métier, peuvent être utilisées pour vérifier la cohérence des modèles et des codes des applications. Il s'en suit une meilleure définition d'un processus qui pourra aider les Analystes/Concepteurs tout un long du processus.

Bilan - Perspectives

Le domaine de recherche qui a été abordé dans cette étude porte à la fois sur les modèles et sur les grammaires. Le point de départ concerne la modélisation en UML/OCL des langages de programmation et de leurs propriétés syntaxiques et sémantiques. Il s'agit d'intégrer lors d'un processus de développement logiciels un niveau de modélisation des codes entre les activités de modélisation des exigences et de génération des codes de manière à vérifier la cohérence entre les modèles et les codes.

C'est la raison pour laquelle ce domaine de recherche concourt au rapprochement des technologies de Modèles (ModelWare) et des Grammaires (GrammarWare). Il s'agit tout d'abord de s'inspirer des propriétés des langages de programmation basées sur des approches formelles pour décrire, à l'aide du langage de contraintes OCL/LA, les propriétés des langages de modélisation, comme UML. Il s'agit ensuite de s'inspirer des techniques des traducteurs en vue de les appliquer sur les transformations de modèles. En effet les traducteurs vérifient les propriétés formelles des programmes avant de les traduire ou de les interpréter.

Il existe, dans ce domaine, bon nombre de travaux de recherche principalement sur les aspects structurels des modèles. Par contre, il existe peu de travaux sur les propriétés sémantiques de UML, en OCL/LA. Les preuves de programmes se font à l'aide de prouveurs, comme l'atelier B par exemple. En fait, nous avons cherché à rester le plus possible dans le formalisme du langage de modélisation UML, devenu un standard de fait, et du langage de contraintes OCL, partie intégrante de UML complété par un langage d'actions UML. Nous avons utilisé les environnements d'exécution USE, pour tester et valider toutes les opérations qui ont été mises en œuvre. Les plates-formes KerMéta et TopCased ont aussi été utilisés assurant que les programmes écrits sont bien compatibles avec le Méta-Modèle UML. Le passage à une programmation Java/EMF a été étudié. La prise en compte des aspects structurels se fait sans problème, par contre, le langage Java ne prend pas en compte les expressions OCL.

Les différents domaines qui ont pu être traités directement sont les suivants :

<u>Modélisation des langages de programmation et de leurs propriétés</u> :

La modélisation en UML de la grammaire abstraite des langages de programmation permet d'insérer un niveau de modélisation des codes à la charnière des activités de modélisation des exigences d'une application et de génération des codes d'un processus IDM.

La modélisation des propriétés syntaxiques en OCL des langages de programmation permet de vérifier que les modèles de codes sont bien formés, pré-conditions importantes avant la génération des codes.

La modélisation des propriétés comportementales en OCL/LA des langages de programmation et leur exécution à l'aide d'un environnement d'exécution du langage OCL et d'un langage d'actions UML permettent aux Analystes/Concepteurs de vérifier, au niveau des modèles, le comportement des codes à un certain niveau d'abstraction.

<u>Environnement d'exécution des modèles</u> :

La modélisation des triplets de Hoare en UML/OCL/LA permet aux Analystes/Concepteurs de spécifier les propriétés vérifiant que les fragments de modèles de codes sont corrects, en tenant compte des propriétés sémantiques qui sont définies au niveau d'abstraction correspondant. Apporter la preuve formelle que tout triplet de Hoare se fait à l'aide d'un Atelier B. Il aurait été préférable de traiter cet aspect directement au niveau UML pour éviter aux Analystes/Concepteurs de changer d'environnement de modélisation. Pour des raisons évidentes, il n'était pas possible de réaliser cette preuve en UML/OCL. Cependant, l'assertion déduite de la définition du triplet de Hoare est modélisée en UML/OCL/LA. Elle peut donc servir de programmes de test exécutables sur des données bien choisies, à l'aide de l'environnement d'exécution. Les Analystes/Concepteurs peuvent donc vérifier dynamiquement que les modèles de codes ont des comportements « corrects », anticipant ainsi le bon fonctionnement de processus avant la génération des codes. La modélisation en UML/OCL des algorithmes de la plus faible pré-condition et de la substitution d'une variable dans une expression permettende de simplifier la preuve de la validité d'un triplet de Hoare. Ainsi, l'appel à l'Atelier B s'en trouve réduit au strict minimum, ainsi que les programmes de test que les Concepteurs seraient à même à demander au niveau UML pour en avoir confirmation sur des exemples concrets.

<u>Intégration de propriétés syntaxiques et sémantiques sur le fragment de Méta-Modèle UML concernant le diagramme d'activité</u>

L'étude approfondie du diagramme d'activité a permis de montrer comment on peut appliquer au modèle de conception tout ce qui a été défini pour les modèles de code. Nous avons montré, en particulier, les conditions que doivent vérifier les modèles des diagrammes d'activité pour les exécuter, au même titre que les modèles de codes, et pour vérifier que des fragments de modèles de diagramme d'activité sont corrects.

<u>Processus IDM pour l'activité de génération de codes, vérification des transformations</u> :

L'avancement de ces travaux nous a permis de déduire un processus IDM pour l'activité de génération de codes, à partir d'un diagramme d'activité. A chaque niveau de modélisation, les experts d'un domaine d'applications peuvent définir les propriétés que les modèles devront respecter, en particulier les propriétés assurant la cohérence entre les codes et les modèles de conception.

De la même manière que les méthodes et les techniques des traducteurs ont pu être appliquées sur des modèles de codes, nous avons pu les appliquer au niveau de l'activité de conception des modèles pour vérifier l'équivalence comportementales entre des modèles et entre des fragments de modèles et pour vérifier que les transformations de modèles et de fragments de modèles sont correctes. Bien sûr, nous en sommes restés à des vérifications dynamiques, sur des exemples ponctuels. Les preuves complètes ne sont donc pas faites sur les codes, puisque les programmes de transformation sont écrits en OCL et en Langage d'actions UML. La transformation en B nécessite le passage à un niveau Méta-Modèle du langage OCL et du langage d'actions UML, comme on l'a fait pour modéliser les propriétés syntaxiques et sémantiques du langage pris en exemple, le langage « L ». Il en est de même pour le calcul de la plus faible pré-condition et de l'opération de substitution qui ne pose pas de problème au niveau des propriétés du langage « L », compte tenu de ses limitations.

Si nous avons apporté des réponses aux différentes questions qui ont été posées au départ et qui ont permis en particulier de proposer un processus IDM pour l'activité de génération de codes, cette étude n'est cependant par close. En effet, certaines parties de cette recherche pourraient être complétées. En particulier, nous pourrions suggérer les différents points suivants.

Les propriétés syntaxiques et sémantiques applicables aux diagrammes d'activité ont été définies au niveau du Méta-Modèle UML. Si, techniquement, nous avons donc pu apporter des solutions aux différents problèmes posés, il aurait été plus méthodologique de définir un profil pour les modèles de graphe du diagramme d'activité et pour les modèles d'arbre du même diagramme. Cependant, dans les études bibliographiques que nous avons faites[1,2], les profils abordent les aspects structurels à partir d'extension d'éléments de modélisation du diagramme de classes. L'étude approfondie du diagramme d'activité montre qu'il faudrait définir les profils des modèles des langages à partir des éléments du diagramme d'activité. On pourrait de cette manière définir le Méta-Modèle UML/OCL des langages de programmation et de leurs propriétés syntaxiques et sémantiques directement à partir du diagramme d'activité. Le processus IDM suggéré s'en trouverait donc plus simple, et surtout, chaque niveau d'abstraction serait défini par son propre Méta-Modèle, sans risque de conflits entre les différentes propriétés sémantiques qui y sont définies.

De la même manière, les règles de transformation que nous avons mises en place devraient permettre de rechercher dans un diagramme d'activité les fragments de modèles à transformer en nœuds d'activité structurée à l'aide de patrons. Ceci aurait pu être effectivement fait, puisque ces règles définissent une grammaire d'un langage applicable aux modèles de graphe et aux modèles d'arbre du diagramme d'activité. En fait, l'expérience a montré que les patrons n'abordent que les aspects syntaxiques et que la spécification de chaque transformation doit être complétée par des pré-conditions et des pos-conditions.

Le diagramme d'activité a été utilisé pour modéliser des organigrammes et des algorithmes. On s'est donc intéressé au couplage du diagramme d'activité et du diagramme de classes. Il serait intéressant de reprendre l'ensemble des méthodes et des techniques que nous avons développées pour les étendre à d'autres diagrammes, en l'occurrence aux diagrammes d'états/ transitions qui définissent différentes évolutions d'un système et aux diagrammes de séquence qui permettraient de construire des programmes de tests des algorithmes ainsi modélisés.

---

[1] Integration of domain-specific modeling languages and UML through UML profile extension mechanism
G. Giachetti, B. Beatriz, O. Pastor
International Journal Of Computer Sciences an Application
Vol. 6, No. 5, pp 145-174, 2009

[2] Mise en œuvre d'un atelier d'aide à la conception de profil
F. Lagarde, F. Terrier, C. Nadré, S. Gérard
CEA, LIST, Gif-sur-Yvette, F-91191, France
I3S Laboratory, BP 121, 06903 Sophia Antiplois Cédex

Ce qui nous paraitrait, par contre, le plus intéressant du point de vue recherche, c'est de reprendre le calcul de la plus faible pré-condition et de la substitution de variable dans une instruction d'affectation du langage OCL et du langage d'actions UML. La vérification des transformations ne se limiterait plus à des programmes de tests sur des exemples pertinents. Mais, il faudrait préalablement définir la sémantique de ces opérations pour le langage OCL, et le langage d'actions UML, en particulier la sémantique des opérations gérant les associations et les compositions (création ou élimination). C'est une recherche qui demanderait une implication importante des techniques de compilation.

D'autres thèmes pourraient aussi faire l'objet d'une suite à ce travail. En particulier, les transformations que nous avons étudiées sont des transformations sans interventions des Concepteurs. Il arrive souvent, dans un processus, que l'on puisse être amené à vérifier des transformations où les Analystes/Concepteurs aient besoin d'intervenir.
C'est le cas, par exemple, des transformations liées aux opérations de raffinage. Par exemple, on peut définir un modèle très général à partir d'actions de type OpaqueAction, sans donc définir avec précision ces opérations. Si la transformation en arbre d'un tel modèle peut être validée, alors les Analystes/Concepteurs peuvent raffiner un tel modèle, en précisant la classe où se trouve l'opération que l'on cherche à raffiner, et pour chaque action opaque le raffinage souhaité.

Bibliographie

[ISO96] ISO/IEC 14977: 1996 (E), Information technology - Syntactic metalanguage - Extended BNF.

[Ach09] Acher M., et al., 2009. Modeling Context and Dynamic Adaptations with Feature Models.

[Agra02] Agrawal R., et al., 2002. Hippocratic databases. Dans *Proceedings of the 28th international conference on Very Large Data Bases*. Hong Kong, China: VLDB Endowment, p. 143-154.

[Alan03] Alanen M., & Porres I., 2003. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. *Turku Centre for Computer Science TUCS Technical Report No*, 606.

[Alan08] Alanen M. , & Porres I., 2008. A metamodeling language supporting subset and union properties. *Software and Systems Modeling*, 7(1), 103-124.

[Alva04] Alvaro A., et al., 2004. Towards an Incremental Process Model Based on AOP for Distributed Component-Based Software Development. Dans *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*. p. 38-39.

[At.B] Atelier B http://www.atelierb.eu/index-en.php

[Audi] Audibert L., Cours-UML
http://laurent-audibert.developpez.com/Cours-UML/html/index.html

[Bare07] Baresi L., Ehrig K., & Heckel R., 2007. Verification of Model Transformations: A Case Study with BPEL. Dans *Trustworthy Global Computing*. p. 183-199.

[Basi09] Basin D., et al., 2009. Automated analysis of security-design models. *Information and Software Technology*, 51(5), 815-831.

[Baud10] Baudry B., et al., 2010. Trust in MDE components : the DOMINO experiment. In *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems (S&D4RCES '10),* ISBN: 978-1-4503-0368-2

[Bern04] Bernstein A., Burton M., & Ghenassia F., How to bridge the abstraction gap in system level modeling and design. Dans *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* ICCAD 2004. International Conference on Computer Aided Design. San Jose, CA, USA, p. 910-914.

[Bhat05] Bhattacharjee A., & Shyamasundar R., 2005. Validated Code Generation for Activity Diagrams. Dans *Distributed Computing and Internet Technology*. p. 508-521.

[Bhat09] Bhattacharjee A., & Shyamasundar R., 2009. Activity Diagrams : A formal framework to model business processes and code generation. *Journal of Object Technology*, Vol. 8, N°1, p.189-220.

[Björ04] Björklund D., Lilius J. & Porres I., 2004. A Unified Approach to Code Generation from Behavioral Diagrams. Dans *Languages for System Specification*. p. 20-34.

[Bruc07] Brucker A.-D., & Doser J., 2007. Metamodel-based UML Notations for Domain-specific Languages. *4th International Workshop on Software Language Engineering (ATEM 2007).*

[Bürg10] Bürger C. & Karol S., 2010. *Towards Attribute Grammars for Metamodel Semantics*, Available at: ftp://ftp.inf.tu-dresden.de/pub/berichte/tud10-03.pdf

[Cabo07] Cabot J., & Teniente E., 2007. Transformation techniques for OCL constraints. *Science of Computer Programming*, 68(3), 179-195.

[Cado09] Cadoret F. & Kerboeuf M., 2009. Expérimentation d'un modèle abstrait de syntaxe abstraite. Dans MajecSTIC 2009. Avignon, France.

[Clar04] Clark T. et al., 2004. Applied Metamodelling : A Foundation for Language Driven Development.

[Clav06] Clavel M. & Egea M., 2006. ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. Dans *Algebraic Methodology and Software Technology*. p. 368-373.

[Comb07] Combemale B. et al. Expériences pour décrire la sémantique en ingénierie des modèles. *Actes des 2 èmes journées sur l'Ingénierie Dirigée par les Modèles*.

[Comb08] Combemale B., 2008. Approche de méta-modélisation pour la simulation et la vérification de modèle.

[Cucc09] Cuccuru A. et al., 2009. Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers. Dans *Model Driven Engineering Languages and Systems*. p. 644-649.

[Cupp00] Cuppens F., 2000. Modélisation formelle de la sécurité des systèmes d'informations. *Habilitation à Diriger les Recherches, Université Paul Sabatier*.

[Dela08] Delaitre S., Giboin A. & Moisan S., 2008. THE AEX METHOD AND ITS INSTRUMENTATION.

[Dem99] Demuth B. & Hussmann H., 1999. Using UML/OCL Constraints for Relational Database Design. Dans *«UML»'99 — The Unified Modeling Language*. p. 751.

[Dem01] Demuth B., Hussmann,H. & Loecher S., 2001. OCL as a Specification Language for Business Rules in Database Applications. Dans *«UML» 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. p. 104-117.

[DOMI07] Projet RNTL DOMINO. http://www.domino-rntl.org/scripts/home/publigen/content/templates/show.asp?L=FR&P=55

[Drah07] Draheim D. & Weber G. éd., 2007. *Trends in Enterprise Application Architecture*, Berlin, Heidelberg: Springer Berlin Heidelberg.

[Duch] Duchien L., Ledru Y. Défis pour le Génie de la Programmation et du Logiciel GDR CNRS GPL, Une réflexion coordonnées par L. Duchien.

[Enge10] Engelen L. & Van Den Brand M., 2010. Integrating Textual and Graphical Modelling Languages. *Electronic Notes in Theoretical Computer Science*, 253(7), 105-120.

[Fall08] Falleri J. et al. 2008. Metamodel Matching for Automatic Model Transformation Generation. Dans *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. Toulouse, France: Springer-Verlag, p. 326-340.

[Fall09] Falleri J. et al. 2009. Metamodel Matching for Automatic Model Transformation Generation. Dans *Model Driven Engineering Languages and Systems*. p. 326-340.

[Feki06] Fekih H., Aye L.-J.-B. & Merz, S., 2006. Transformation of B specifications into UML class diagrams and state machines. Dans *Proceedings of the 2006 ACM symposium on Applied computing*. Dijon, France: ACM, p. 1840-1844.

[Ferr07] Ferraiolo D.F. & Ferraiolo D.-F., Kuhn D.R. and Sandhu R., 2007. RBAC Standard Rationale: comments on a Critique of the ANSI Standard on Role Based Access Control. *IEEE Security & Privacy*, 5(6), 51–53.

[Fomb06] Fombelle G. et al. 2006. Finding a Path to Model Consistency. Dans *Model Driven Architecture – Foundations and Applications*. p. 101-112.

[Fomb07] Fombelle G. 2007. Gestion incrémentale des propriétés de cohérence structurelle dans l'ingénierie dirigée par les modèles.

[Giac08] Giachetti G., Valverde F. & Pastor O., 2008. Improving Automatic UML2 Profile Generation for MDA Industrial Development. Dans *Proceedings of the ER 2008 Workshops (CMLSA, ECDM, FP-UML, M2AS, RIGiM, SeCoGIS, WISM) on Advances in Conceptual Modeling: Challenges and Opportunities*. ER '08. Berlin, Heidelberg: Springer-Verlag, p. 113–122.

[Giac09] Giachetti G., MarÃn B. & Pastor O., 2009. Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles. Dans *Advanced Information Systems Engineering*. Springer Berlin / Heidelberg, p. 110-124-124

[Giac09] Giachetti G., Marin B. & Pastor O., 2009. Integration of Domain-Specific Modeling Languages and UML through UML profile extension mechanism. *International Journal of Computer Science & Applications*, 6(5), p.145–174.

[Gies06] Giese H. et al. 2006. Towards Verified Model Transformations. Dans *Proc. of the 3 rdInternational Workshop on Model Development, Validation and Verification (MoDeV 2 a), Genova, Italy*. p. 78–93.

[Gila04] Gilad B., 2004. Generics in the Java Programming Language. Available at: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[Gogo99] Gogolla M. & Richters M., 1999. Transformation Rules for UML Class Diagrams. Dans *The Unified Modeling Language. «UML»'98: Beyond the Notation*. p. 514.

[Gogo02] Gogolla M. & Richters M., 2002. Expressing UML Class Diagrams Properties with OCL. Dans *Object Modeling with the OCL*. p. 423-426.

[Gogo03] Gogolla M. & Lindow A., 2003. Transforming data models with UML. *Knowledge Transformation for the Semantic Web*, p. 18–33.

[Gree03] Greenfield J. & Short K., 2003. Software factories: assembling applications with patterns, models, frameworks and tools. Dans *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Anaheim, CA, USA: ACM, p. 16-27.

[Grus07] Gruschko B. et al., 2007. Towards Synchronizing Models with Evolving Metamodels. Dans *Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*.

[Henri08] Henriksson J. et al., 2008. Extending grammars and metamodels for reuse: the Reuseware approach. *IET Software*, 2(3), 165-184.

[Huan04] Huang Y. et al., 2004. Securing web application code by static analysis and runtime protection. Dans *Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM, p. 40-52.

[Hutt06] Hutter D. & Volkamer M., 2006. Information Flow Control to Secure Dynamic Web Service Composition. *LECTURE NOTES IN COMPUTER SCIENCE*, 3934, 196.

[Idani] Idani A., 2006. B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B. Thèse soutenue en novembre 2006, Université de Grenoble 1.

[Ivko04] Ivkovic I. & Kontogiannis K., 2004. Tracing evolution changes of software artifacts through model synchronization. Dans *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. p. 252-261.

[Jaeg07] Jaeger É. & Dubois C., 2007. Why Would You Trust B? Dans *Logic for Programming, Artificial Intelligence, and Reasoning*. p. 288-302.

[Jaeg09] Jaeger E. & Hardin T., 2009. Yet Another Deep Embedding of B:Extending de Bruijn Notations.

[Joua06a] Jouault F. & Bezivin J., 2006. KM3: A DSL for Metamodel Specification. *LECTURE NOTES IN COMPUTER SCIENCE*, 4037, 171.

[Joua06b] Jouault F., Bézivin J. & Kurtev I., 2006. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. Dans *Proceedings of the 5th international conference on Generative programming and component engineering*. Portland, Oregon, USA: ACM, p. 249-254.

[KeJi07] Ke Jiang, Lei Zhang & Miyake S., 2007. OCL4X: An Action Semantics Language for UML Model Execution. Dans *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*. Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International. p. 633-636.

[Kels08] Kelsen P. & Ma Q., 2008. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. Dans *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. Toulouse, France: Springer-Verlag, p. 690-704.

[Kerbo09] Kerboeuf M. & Cadoret F., 2009. *Modèle abstrait de syntaxe abstraite pour l'outillage réutilisable de DSL*,

[Kerm] Kermeta - Breathe life into your metamodels — Kermeta. http://www.kermeta.org/

[Laga] Lagarde F. et al., 2007. Improving uml profile design practices by leveraging conceptual domain models. Dans *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE '07. New York, NY, USA: ACM, p. 445–448.

[Lale00] Laleau R. & Mammar A., 2000. An overview of a method and its support tool for generating B specifications from UML notations. Dans *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*. Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on. p. 269-272.

[Lano96] Lano K., 1996. *The B Language and Method: A Guide to Practical Formal Development (Formal Approaches to Computing and Information Technology*, Springer.

Lano08] Lano K. & Clark D., 2008. Model Transformation Specification and Verification. Dans *Quality Software, 2008. QSIC '08. The Eighth International Conference on*. Quality Software, 2008. QSIC '08. The Eighth International Conference on. p. 45-54.

[Leda01] Ledang H., 2001. Automatic translation from UML specifications to B. Dans *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, p. 436.

[LeDe] Le Delliou R. et al., 2004. A Model-Driven Approach For Information System Migration. Dans *Workshop on ODP for Enterprise Computing*, Monterey Califormie US.

[LeFe04] LeFevre K. et al., 2004. Limiting disclosure in hippocratic databases. Dans *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*. Toronto, Canada: VLDB Endowment, p. 108-119.

[LeTh09a] Le Thi T.-T, 2009. Modeling of programming languages in UML/OCL and application in a MDE process. In *IADIS International Conference Applied Computing,* Vol. I, p. 234-242, Rome, Italy, novembre 2009

[LeTh09b] Le Thi T.-T., Millan T., Percebois C., Bazex P., 2009. *Modélisation des langages en UML et OCL et Processus IDM*. Journées Neptune'2009, Paris Tech Telecom, mai 2009

[LeTh09c] Le Thi T.-T, Bazex P., Millan T., 2009. Droits d'accès à des Services et des Données pour des Environnements Collaboratifs. Dans l'atelier SDEC'09, INFORSID 2009, Toulouse, France, mai 2009

[LeTh09d] Le Thi T.-T., 2009. Modélisation de la syntaxe et de la sémantique des langages et processus IDM. Forum Jeunes Chercheurs – Congres INFORSID 2009, Toulouse, France, mai 2009

[LeTh09e] Le Thi T.-T., Bazex P., Millan T., 2009. Modeling of languages' grammars in UML/OCL: Applying to a model driven software development process. In *International Conference on Theories and Applications of Computer Science (ICTACS 2009)*, Vol. 46, Vietnam Academy of Science and Technology, p. 25-46, février 2009

[LeTh09f] Le Thi T.-T., Millan T., Poupart E., Sabatier T., Dalbin J.-C., 2009. Modélisation des langages de programmation et Processus de développement de logiciels. Journée nationale du GDR GPL, Toulouse, France, janvier 2009

[LeTh10a] Le Thi T.-T, Bazex P., 2010. Language modeling in the context of a MDE process. In *IADIS International Conference Applied Computing,* Timisoara, Romania, octobre 2010

[LeTh10b] Le Thi T.-T, Bazex P., 2010. Vérification statique des droits d'accès sur les données. Dans l'atelier Sécurité, VSST 2010, Toulouse, France, octobre 2010

[LeTh10c] Le Thi T.-T., 2010. L'Activité de Génération de Codes Dirigée par les Modèles. Dans Journées nationales du GDR GPL, p. 308, Pau, France, mars 2010

[LeTh10d] Le Thi T-T., F. Pontico, O. Nicolas. Modéliser pour sécuriser des données personnelles au cours des processus. Dans Colloque STIC, organisé par ANR, Paris, France, 5-7 janvier, 2010

[LeTh11] Le Thi T.-T, 2011. Modélisation en UML/OCL des propriétés des langages de programmation et Processus IDM. DEVLOG, Toulouse, septembre 2011.

[Li05] Li P. & Zdancewic S., 2005. Practical Information-flow Control in Web-based Information Systems. Dans *Proceedings of 18th IEEE Computer Security Foundations Workshop. IEEE Computer Society Press*.

[Mank] Mankai M., 2005. Vérification et Analyse des politiques de contrôles d'accès : Application au langage XACML

[Male02] Malenfant J., 2002. Modélisation de la sémantique formelle des langages de programmation en UML et OCL. *Rapport de recherche*.

[Male07] Malenfant J., 2007. Du modèle au programme: aspects conceptuels et applications. *Génie logiciel(1995)*, (81), 8-13.

[Marc01] Marcano R. & Lévy N., 2001. Transformation d'annotations OCL en expressions B. Dans Journées AFADL'2001 : Approches formelles dans l'Assistance au Développement de Logiciels. Nancy.

[Marc02] Marcano R. & Lévy N., 2002. Transformation rules of OCL Constraints into B Formal Expressions. Dans CSDUML2002 Workshop on critical systems development with UML 5th International Conference on the Unified Modeling Language.

[Marcs03] Marcos E., Vela B. & Cavero J.M., 2003. A methodological approach for object-relational database design using UML. *Software and Systems Modeling*, 2(1), 59-72.

[McLe85] McLean J., 1985. A Comment on the `Basic Security Theorem' of Bell and LaPadula. *Information Processing Letters*, 20(2), 67-70.

[McLe87] McLean J., 1987. *Reasoning about security models*, Storming Media.

[McLe94] McLean J., 1994. Security models. *Encyclopedia of Software Engineering. Wiley & Sons*.

[MDA] MDA Guide Version 1.1 http://www.omg.org/cgi-bin/doc?omg/03-06-01

[Mill09] Millan T., Sabatier L., Le Thi T.-T., Bazex P., Percebois C., 2009. An OCL extension for checking and transforming UML Models. In *WSEAS – International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS 2009)*, Cambridge, United Kingdom, février 2009

[Moha08] Mohamed M., Romdhani M. & Ghedira K., 2008. Classification des approches de refactorisation des modèles. Dans IDM'2008 5-6 juin Mulhouse. Mulhouse.

[Moh] Mohring C.P. et al., Assistants de preuve.(Cours)
http://www.lri.fr/~paulin/MPRI/notes/index-2-7-2.html
[Mois03] Moisan S., Ressouche A. & Rigault, J., 2003. A Behavioral Model of Component Frameworks.

[Mois10] Moisan S. et al., 2010. Synchronous Formalism and Behavioral Substitutability in Component Frameworks.

[MullA05] Muller, A. et al., 2005. On Some Properties of Parameterized Model Application. Dans *Model Driven Architecture – Foundations and Applications*. p. 130-144.

[MullA06] Muller A., 2006. Construction de systèmes par application de modèles paramétrés.

[MullP04] Muller P.A. & Gaertner, N., 2004. *Modélisation objet avec UML*, Eyrolles.

[MullP05a] Muller P.A. et al., 2005. On Executable Meta-Languages applied to Model Transformations. Dans *Model Transformations In Practice Workshop*.

[MullP05b] Muller P.A. & Hassenforder M., 2005. HUTN as a Bridge between ModelWare and GrammarWare–An Experience Report. *WISME Workshop, MODELS/UML.*

[MullP06] Muller P. et al., 2006. Model-Driven Analysis and Synthesis of Concrete Syntax. Dans *Model Driven Engineering Languages and Systems*. p. 98-110.

[MullP08] Muller, P.A. et al., 2008. Model-driven analysis and synthesis of textual concrete syntax. *Software and Systems Modeling*, 7(4), 423-441.

[Mura06] Murata M. et al., 2006. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3), 292-324.

[MyerB92] Myer B., 1992. *Introduction à la theorie des langages de programmation*, InterEditions.

[MyerE01] Myer E., 2001. *Développements formels par objets : utilisation conjointe de B et d'UML.* Université Nancy 2.

[Nara08a] Narayanan A. & Karsai G., 2008. Verifying Model Transformations by Structural Correspondence. Dans *Proc. GT-VMT*. p. 14.

[Nara08b] Narayanan A. & Karsai G., 2008. Towards Verifying Model Transformations. *Electron. Notes Theor. Comput. Sci.*, 211, 191-200.

[Naum04] Naumovich G. & Centonze P., 2004. Static analysis of role-based access control in J2EE applications. *SIGSOFT Software Engineering Notes*, Vol. 29, Issue 5, p. 1-10.

[Occel08] Occello A., Dery-Pinna A. & Riveill M., 2008. Validation and Verification of an UML/OCL Model with USE and B: Case Study and Lessons Learnt. Dans *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE Computer Society, p. 113-120.

[Onab06] Onabajo A. & Jahnke J., 2006. Modeling and reasoning for confidentiality requirements in software development. Dans *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*. Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on. p. 8 pp.-467.

[Onab08] Onabajo A. & Weber-Jahnke J., 2008. Stratified Modelling and Analysis of Confidentiality Requirements. Dans *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*. Hawaii International Conference on System Sciences, Proceedings of the 41st Annual. p. 232.

[Peig08] Peiguang Lin et al., 2008. An Ontology-Based & Distributed Service Model for EG. Dans *Internet Computing in Science and Engineering, 2008. ICICSE '08. International Conference on*. Internet Computing in Science and Engineering, 2008. ICICSE '08. International Conference on. p. 485-491.

[Pist07] Pistoia M. et al., 2007. When Role Models Have Flaws: Static Validation of Enterprise Security Policies. Dans *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, p. 478-488.

[Pons08] Pons C. & Garcia D., 2008. A Lightweight Approach for the Semantic Validation of Model Refinements. *Electronic Notes in Theoretical Computer Science*, 220(1), 43-61.

[Rich78] Richy H., 1978. Confidentialité, bases de données et réseaux d'ordinateurs.

[Roqu] Roques P., Vallée F., Présentation du processus 2TUP Eyrolles

[Royc99] Roychoudhury S. & Gray J., Towards Language-Independent Weaving Using Grammar Adapters. *Tourwe et al.[1399]*.

[Sabel03] Sabelfeld A. & Myers A.C., 2003. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1), 5-19.

[Sand96] Sandhu, R., Coyne E.J., Feinstein H.L. and Youman C.E., 1996. Role-Based Access Control Models. *IEEE Computer*, 29(2), 38–47.

[Sand94] Sandhu R. & Samarati P., 1994. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9), 40-48.

[Schm07] Schmidt M., 2007. Transformations of UML 2 Models Using Concrete Syntax Patterns. Dans *Rapid Integration of Software Engineering Techniques*. p. 130-143.

[Seli07] Selic B., 2007. A Systematic Approach to Domain-Specific Language Design Using UML. Dans *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE Computer Society, p. 2-9.

[Sen08] Sen S., Baudry B. & Mottu J., 2008. On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing. Dans *Software Testing, Verification, and Validation, 2008 1st International Conference on*. Software Testing, Verification, and Validation, 2008 1st International Conference on. p. 328-337.

[Sen09] Sen S. et al., 2009. Meta-model Pruning. Dans *Model Driven Engineering Languages and Systems*. p. 32-46. Available at: http://dx.doi.org/10.1007/978-3-642-04425-0_4

[Sero08] Sérot J. & Falcou J., Métaprogrammation fonctionnelle appliquée à la génération d'un DSL dédié à la programmation parallèle. *Journées Francophones des Langages Applicatifs 2008*.

[Siik06a] Siikarla M. & Systa T., 2006. Transformational pattern system-some assembly required. *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, 57–68.

[Siik06b] Siikarla M., 2006. Slide : GraphTransformations , KK seminar, srping 2006. http://www.cs.tut.fi/~kk/webstuff/GraphTransformations.pdf

[Sintak] http://kermeta.org/sintaks/

[Snoo92] Snook C. & Butler M., 2006. UML-B. *ACM Transactions on Software Engineering and Methodology*, 15(1), 92-122.

[Suny02] Sunyé G., Le Guennec A. & Jézéquel J., 2002. Using UML Action Semantics for model execution and transformation. *Information Systems*, 27(6), 445-457.

[Terr01] Terrasse M.N., 2001. A Metamodeling Approach to Evolution. *LECTURE NOTES IN COMPUTER SCIENCE*, 202-219.

[TopC] TopCased - The Open-Source Toolkit for Critical Systems. http://www.topcased.org/index.php

[UML] Unified Modeling Language : Superstructure - version 2.1.1. http://www.omg.org/spec/UML/2.1.1/

[Vale08] Vale S. & Hammoudi S., 2008. Context-aware Model Driven Development by Parameterized Transformation. *MDISIS held with CAISE*, 8, 121–133.

[Weis08] Weisemöller I. & Schürr A., 2008. A Comparison of Standard Compliant Ways to Define Domain Specific Languages. Dans *Models in Software Engineering*. p. 47-58.

[Wile97] Wile D.S., 1997. Abstract syntax from concrete syntax. Dans *Proceedings of the 19th international conference on Software engineering*. Boston, Massachusetts, United States: ACM, p. 472-480.

[Wimm06] Wimmer M. & Kramler G., 2006. Bridging Grammarware and Modelware. Dans *Satellite Events at the MoDELS 2005 Conference*. p. 159-168.

[Wimm07] Wimmer M. et al., 2007. Towards Model Transformation Generation By-Example. Dans *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, p. 285b. Available at: http://portal.acm.org/citation.cfm?id=1255775

[Xia03] Xia Y. & Glinz M., 2003. Rigorous EBNF-based definition for a graphic modeling language. Dans *Software Engineering Conference, 2003. Tenth Asia-Pacific*. Software Engineering Conference, 2003. Tenth Asia-Pacific. p. 186-196.

[Xion07] Xiong Y. et al., 2007. Towards automatic model synchronization from model transformations. Dans *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM New York, NY, USA, p. 164-173.

[xText] http://www.eclipse.org/Xtext/

[Yagi05] Yagi I., Takata Y. & Seki H., 2005. A Static Analysis Using Tree Automata for XML Access Control. Dans *Automated Technology for Verification and Analysis*. p. 234-247.

[Zdan04] Zdancewic S., 2004. Challenges for information-flow security. Dans *Proc. Programming Language Interference and Dependence (PLID).*

[Zhao07] Zhao Jia et al., 2007. Multi-layer Secure Model of Task-based Information Systems. Dans *Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference on*. Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference on. p. 2-296-2-299.