

MEMOIRE

Présenté devant

**l'Institut National Polytechnique de Toulouse et
l'Université de Toulouse III**

pour obtenir

le Master Recherche
SCIENCES DE LA MODÉLISATION, DE L'INFORMATION ET DES SYSTÈMES
Spécialité SÛRETÉ DU LOGICIEL ET CALCUL À HAUTE PERFORMANCE

par

Benoît COMBEMALE

Laboratoire d'accueil : GRIMM - ISYCOM

Directeur : Bernard COULETTE

Titre du mémoire :

***Spécification et Vérification
de Modèles de Procédés de Développement***

soutenu le 23 juin 2005

Président :	Patrick	SALLE	<i>Professeur, ENSEEIHT (INP)</i>
Directeur de Recherche :	Bernard	COULETTE	<i>Professeur, Université Toulouse II</i>
Encadrants :	Xavier	CRÉGUT	<i>Maître de conférence, ENSEEIHT (INP)</i>
	Alain	CAPLAIN	<i>Doctorant, Université Toulouse II</i>

*Ce qui se conçoit bien s'énonce clairement
et les mots pour le dire arrivent aisément.*

par Boileau

Remerciements

Je tiens à exprimer ma profonde gratitude à tous ceux qui m'ont permis de réaliser ce Master Recherche dans les meilleures conditions. Leur soutien et leur aide m'ont permis de rendre cette année de spécialisation à la Recherche captivante et très enrichissante.

J'adresse tout d'abord ma reconnaissance à l'ensemble du Laboratoire GRIMM¹ et plus précisément à Xavier CRÉGUT, Maître de Conférence, et Alain CAPLAIN, Doctorant. Je les remercie tout particulièrement pour m'avoir offert la chance de réaliser ces travaux mais également de m'avoir encadré et soutenu tout au long de cette année.

D'autre part je tiens à remercier Jean-Michel INGLEBERT, Maître de Conférence au sein du Laboratoire GRIMM. En effet, il m'a toujours accordé de son temps pour toutes mes interrogations et j'ai pu compter sur son soutien constant au cours des diverses difficultés que j'ai dû surmonter.

Je remercie également l'ensemble des membres de l'équipe ISYCOM² et plus particulièrement son responsable Bernard COULETTE, Professeur à l'Université Toulouse II, pour m'avoir parfaitement intégré dans le Laboratoire de Recherche et pour avoir été constamment présent pour répondre à mes questions. Leur aide et leurs renseignements m'ont été très précieux pour la réalisation de mon stage ainsi que pour l'élaboration de ce mémoire.

Merci également à Jean-Bernard CRAMPES, Professeur à l'Université Toulouse II, pour les très bons rapports que nous avons eus et qui ont permis une collaboration de travail efficace.

Enfin, j'adresse tous mes remerciements à Laurence REDON, chef du département Informatique de l'I.U.T. B de Blagnac ainsi qu'à Marianne DEMICHIEL et Franck MOUSTAPHA, enseignants au sein de cet IUT, pour m'avoir permis d'avoir une première expérience de l'enseignement universitaire à travers un contrat de vacation.

Un grand merci également à Véronique, Jonathan, Patrick, Martine, Christian et tous les autres pour leurs soutiens tout au long de cette année et pour leurs précieuses relectures.

Ce mémoire ainsi que l'ensemble de mes résultats de Recherche sont le fruit de mon travail mais également de leur soutien. J'espère qu'ils ne seront pas déçus du résultat final.

¹Groupe de Recherche en Informatique et Mathématiques du Mirail

²Ingénierie des SYStèmes COMplexes; composante du Laboratoire GRIMM

Table des matières

Table des matières	1
Introduction	3
Préambule	3
Objectifs du stage	4
Plan du mémoire	5
1 L'ingénierie des modèles	7
1.1 Spécification par les modèles	7
1.2 <i>Model Driven Architecture</i> (MDA)	10
1.3 La transformation de modèles	10
1.4 La vérification de modèles	13
2 <i>Software Process Engineering Metamodel</i> (SPEM)	17
2.1 Présentation de SPEM	17
2.1.1 Origines et évolutions	17
2.1.2 SPEM : méta-modèle MOF ou profil UML ?	18
2.1.3 Les concepts de SPEM	19
2.1.4 Détails d'utilisation de certains stéréotypes	22
2.2 Outils pour la modélisation en SPEM	25
2.2.1 Objecteering/UML	25
2.2.2 Enterprise Architect	26
3 <i>Object Constraint Language</i> (OCL)	29
3.1 Historique et Évolution	29
3.2 Présentation du langage	30
3.3 Étude de l'expressivité	33
3.3.1 OCL comme langage relationnel	34
3.3.2 OCL comme langage de manipulation de données	36
3.3.3 L'expression de contraintes temporelles avec OCL	36
3.3.4 OCL vs. Machines de Turing	38
3.4 Les limites de la version 2.0	38
3.5 La vérification outillée de contraintes	39
3.6 OCL pour les modèles de procédés de développement	41

4	Spécification rigoureuse et cohérente de procédés	43
4.1	Spécialisation du méta-modèle SPEM	43
4.1.1	Les limites du méta-modèle d'origine	43
4.1.2	Présentation de notre spécialisation	44
4.1.3	Description de notre approche	46
4.2	Précisions sémantiques à l'aide d'OCL	48
4.3	Démarche pour la spécification d'un procédé de développement	50
4.3.1	Structuration du procédé	50
4.3.2	Description du procédé	51
4.3.3	Proposition de démarche de spécification	52
5	Spécification de MACAO	55
5.1	Présentation de la méthode	55
5.2	Notre approche pour la spécification de MACAO	58
5.3	Modélisation de MACAO selon SPEM	60
5.3.1	Point de vue structurel	60
5.3.2	Point de vue descriptif	64
	Conclusion	65
	La spécification de procédés par les modèles	65
	Perspectives de Recherche	66
	Annexe A : Modifications apportées par la version 2.0 d'OCL	67
.1	Modifications syntaxiques	67
.2	Nouveaux types	67
.3	Opérations supplémentaires sur les collections	68
.4	Nouvelle option dans les post-conditions	68
.5	Autres changements	68
	Bibliographie	74
	Table des figures	75

Introduction

Préambule

*L*e génie logiciel existe depuis plus de trente-cinq ans³ pour répondre à une problématique initialement bien définie : *les logiciels n'étaient pas fiables et il était incroyablement difficile de satisfaire les cahiers des charges dans des délais prévus.*

Pour cela, le génie logiciel considère le logiciel comme un objet manufacturé complexe et a pour but de définir des techniques de “fabrication” justifiées soit par la théorie, soit par la pratique [GMSB96]. Le génie logiciel est donc l’art de spécifier, de concevoir, de réaliser, et de faire évoluer, avec des moyens et dans des délais raisonnables, des programmes, des documentations et des procédures de qualité en vue d’utiliser un ordinateur pour résoudre certains problèmes.

Pour atteindre les objectifs de qualité et de productivité nécessaires à l’industrie du logiciel, de nombreuses évolutions ont été proposées. Parmi celles-ci, deux semblent particulièrement intéressantes. La première concerne la technique (aspects langages et méthodes) et semble aujourd’hui s’orienter vers les technologies à objets associées à l’ingénierie des modèles. La seconde concerne les outils supports au développement qui, bénéficiant d’une meilleure prise en compte des procédés de développement sous-jacents, apportent une assistance plus efficace aux développeurs puisqu’ils prennent en compte l’avancement du projet. Ces deux voies prometteuses sont bien entendu complémentaires.

Le processus d’industrialisation du développement logiciel, entamé avec l’apport des méthodes comme Merise et le RUP, s’est accéléré avec l’adoption à la fin des années 90 des propositions d’unification telles que UML⁴ et leurs outils associés (XML, XMI, etc.). Une phase importante, objet de nombreux travaux de recherche, de normalisation et d’application par le biais d’environnements professionnels libres (e.g. Eclipse) et propriétaires (e.g. Objecteering/UML et Enterprise Architect) tente d’aborder une étape ultime incarnée par l’adoption de l’ingénierie des modèles dans le processus d’exploitation du cycle de vie complet des *artefacts* logiciels.

³Le génie logiciel est né en Europe, très exactement du 7 au 11 octobre 1968, à Garmisch-Partenkirchen, sous le nom de *software engineering*, et sous le parrainage de l’OTAN

⁴UML (*Unified Modeling Language*), langage de modélisation pour la programmation orientée objet, est devenu un standard de fait suite à son succès dans l’ingénierie du logiciel. (*Cf.* <http://www.uml.org>).

Afin de mieux maîtriser des exigences toujours plus fortes, un logiciel est fabriqué ; comme tout produit manufacturé complexe ; en suivant un certain *procédé de développement*⁵. Il est important que le procédé permettant la fabrication du logiciel fasse une large place à l'analyse des besoins, à la conception, à la validation et à l'évolution.

L'ingénierie des modèles, apparue avec des standards comme le MOF ou MDA, tente d'unifier l'explicitation des processus dans l'optique industrielle indispensable de la réutilisation. Dans ce mémoire, qui s'inscrit dans cette démarche, nous proposons une synthèse des normes et des propositions actuelle de l'ingénierie des modèles et nous abordons la définition concrète d'un procédé de développement logiciel.

La problématique est identique pour l'élaboration du procédé lui-même. Il s'agit également d'une tâche complexe nécessitant une démarche rigoureuse et précise : un *méta-procédé*. Bien que les premiers procédés soient apparus à la fin des années 60, la notion de méta-procédé est bien plus récente. L'intérêt de spécifier de manière formelle les procédés afin de bien les maîtriser est né du besoin d'industrialiser les développements. En effet, cette industrialisation nécessite de bien contrôler les procédés de manière à apporter une certaine assistance aux acteurs du développement et ainsi assurer une production de qualité avec une productivité maximale.

Objectifs du stage

L'industrialisation des développements de logiciel passe de nos jours par l'établissement et le respect de procédés de développement rigoureux. Fort de sa réussite dans le domaine du logiciel, l'ingénierie des modèles tente plus récemment de répondre à cette problématique. L'OMG (*Object Management Group*)⁶ propose pour cela le langage de modélisation SPEM (*Software Process Engineering Metamodel*) permettant de spécifier un procédé par les modèles. Ces modèles peuvent également être complétés d'expressions OCL (*Object Constraint Language*) afin d'une part, d'apporter certains détails qui ne peuvent pas être spécifiés par les modèles et d'autre part, de préciser la sémantique de certains éléments du modèle. L'association de ces deux langages a pour objectif d'assurer la réalisation de modèles rigoureux, cohérents et permettant une vérification formelle.

Les travaux réalisés dans le cadre du stage de Master Recherche ont pour objectif d'utiliser les possibilités offertes par l'ingénierie des modèles dans le domaine des procédés de développement. Il s'agit donc d'une étude théorique et technique des moyens de

⁵Les termes de *procédé*, *processus* ou *démarche* sont également employés pour désigner un procédé de développement logiciel. Ces termes seront donc utilisés de manière indifférente tout au long de ce mémoire.

⁶L'OMG est un consortium à but non lucratif d'industriels et de chercheurs, dont l'objectif est d'établir des standards permettant de résoudre les problèmes d'interopérabilité des systèmes d'information (Cf. <http://www.omg.org>).

spécification et de vérification formelle pour les modèles de procédés décrit en SPEM et OCL.

Nous essayons donc d'apporter une assistance plus efficace pour la spécification et la vérification de modèles de procédés décrits en SPEM et complétés d'expressions formelles en OCL. Nous offrons pour cela une spécialisation par restriction du méta-modèle d'origine associée à une sémantique plus précise et à des conseils pour une formalisation rigoureuse et cohérente des modèles de procédé. Une application de ces travaux pour la spécification et la vérification de la méthode MACAO⁷[Cra02] termine ce mémoire.

Cette étude s'inscrit dans les travaux de Recherche de la composante "Ingénierie des procédés" de l'équipe ISYCOM⁸ sous la direction de Bernard Coulette, Professeur à l'Université Toulouse II, de Xavier Crégut, Maître de conférence à l'ENSEEIH et d'Alain Caplain, Doctorant à l'Université Toulouse II.

Plan du mémoire

Ce mémoire est organisé selon une étude théorique, au fur et à mesure des chapitres, des différentes techniques et outils disponible dans l'ingénierie des modèles pour la spécification des procédés de développement. Il reprend l'ensemble des travaux réalisés au cours de mon stage de Master Recherche.

- le chapitre 1 offre une présentation générale de l'ingénierie des modèles et des problématiques que soulève cette nouvelle approche,
- le chapitre 2 est consacré à la présentation du langage graphique et semi-formel SPEM, proposé par l'OMG pour la modélisation des procédés de développement. Nous proposons également à la fin de ce chapitre certaines précisions sémantiques sur le langage,
- le chapitre 3 présente une étude du langage formel proposé initialement par l'OMG pour l'expression de contraintes : OCL,
- le chapitre 4 expose des conseils et donne un cadre méthodologique pour la formalisation cohérente et rigoureuse d'un procédé de développement dans le contexte de l'ingénierie des modèles.
- Le chapitre 5 termine ce mémoire par une application de notre approche dans le cadre de la modélisation et de la vérification de la méthode MACAO.

Nous concluons enfin par un rappel de l'approche, en soulignant ses apports et ses faiblesses par rapport à la vérification de procédés de développement. Nous indiquons

⁷MACAO est une Méthode d'Analyse et de Conception d'Applications orientées-Objet élaborée par M. Crampes, Professeur des Universités au sein du laboratoire GRIMM-ISYCOM de l'Université de Toulouse II.

⁸L'équipe "Ingénierie des SYstèmes COMplexes" (ISYCOM) est une composante du Groupe de Recherche en Informatique et Mathématiques du Mirail (GRIMM) à l'Université Toulouse II.

également les voies d'études actuelles et futures dans ce domaine ainsi que nos perspectives de recherches.

Chapitre 1

L'ingénierie des modèles

1.1 Spécification par les modèles

Suite à l'approche objet des années 80 et de son principe du “tout est objet”, l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie des modèles et le principe du “tout est modèle”. Cette nouvelle approche peut être considérée soit en *continuité* soit en *rupture* des précédents travaux [Béz04] : tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des systèmes informatiques sous la forme d'objets communicants entre eux, il s'est posé la question de les classer en fonction de leurs différentes origines (objets métiers, techniques, etc.).

L'ingénierie des modèles vise donc, de manière plus radicale que pouvait l'être les approches des *patterns* et des *aspects*, à fournir un grand nombre de langages spécifiques de domaines (DSL) pour exprimer séparément chacune des préoccupations. C'est par ces principes de base fondamentalement différents que l'ingénierie des modèles peut être considérée en rupture par rapport aux travaux de l'approche objet.

Le consensus sur UML fût décisif dans cette transition vers des techniques de production basées sur les modèles. Après l'acceptation du concept clé de méta-modèle comme langage de description de modèles, de nombreux méta-modèles ont émergés afin d'apporter chacun leurs spécificités dans un domaine bien particulier (développement logiciel, entrepôt de données, procédé de développement, etc.). Devant le danger de voir émerger indépendamment et de manière incompatible cette grande variété de méta-modèles, il y avait un besoin urgent de donner un cadre général pour leurs descriptions. La réponse logique fût donc d'offrir un langage de définition de méta-modèles, qui prit lui-même la forme d'un modèle : ce fût le méta-méta-modèle MOF (*Meta-Object Facility*) [Obj02a, Obj03a]. Celui-ci se veut unique car sa description peut être faite à partir de lui-même (méta-circularité). C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale (fig. 1.1).

Le principe du “tout est objet” fût utile dans les années 80 afin de présenter sur la

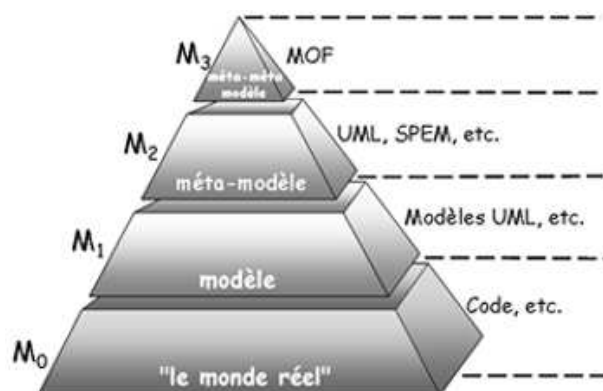


FIG. 1.1 – Pile de modélisation de l'OMG

scène industrielle la technologie des objets. De la même manière, le principe de base du "tout est modèle" possède, en ingénierie des modèles, plusieurs propriétés intéressantes qu'il est important de ne pas confondre avec les propriétés mises en avant dans la technologie à objet [Béz04].

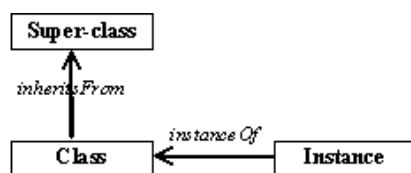


FIG. 1.2 – Notions de base en technologie des objets

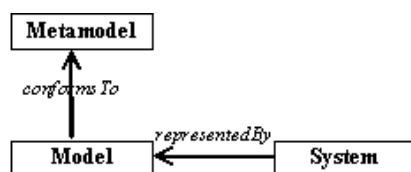


FIG. 1.3 – Notions de base en ingénierie des modèles

En effet, comme le montre les figures 1.2 et 1.3, les outils conceptuels qui étaient les plus utilisés dans les années 1980 sont en cours de renouvellement. En effet, les relations d'héritage et d'instanciation mises en avant dans les technologies objets sont différentes dans l'ingénierie des modèles. Chacune des vues particulières d'un système (aspects) est représentée par un modèle spécifique ; lui-même écrit selon le méta-modèle auquel il se conforme (en fonction de l'aspect étudié par le modèle) [Béz04].

Ainsi, l'organisation de la pile classique à quatre niveaux de l'OMG devrait plus précisément se nommer architecture 3+1 (fig. 1.4) comme le préconise J. Bézivin [Béz04].

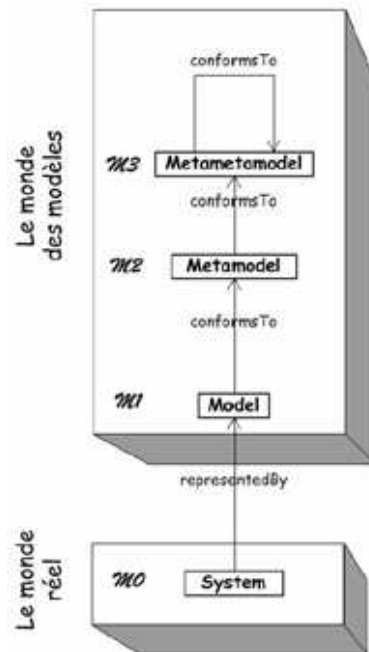


FIG. 1.4 – Organisation 3+1 de la pile de modélisation

Le danger consisterait à utiliser les anciennes relations de la technologie objet dans le nouveau contexte de l'ingénierie des modèles.

L'intérêt essentiel d'un méta-modèle est de faciliter la séparation des préoccupations. Lorsque l'on considère un système donné, on peut travailler avec différentes vues de ce système, chacune de celles-ci étant caractérisée de façon précise par un méta-modèle donné. Quand plusieurs modèles différents ont été extraits du même système à l'aide de méta-modèles différents, ils restent liés et pourront être recomposés par la suite. Pour que ceci puisse être largement appliqué, il est nécessaire de pouvoir disposer d'une organisation régulière des modèles composites [Béz04].

Le passage des objets aux modèles peut être vu en continuité ou en rupture. De la même façon que l'arrivée de la programmation par objets n'a pas invalidé les apports de la programmation structurée, le développement dirigé par les modèles ne contredit pas les apports de la technologie objet. Il est donc important de ne pas considérer ces solutions comme antagonistes mais comme complémentaires.

Toutefois un point de divergence entre ces deux approches est celui de l'intégration

de paradigmes. Initialement, la technologie objet se voulait aussi une technologie d'intégration car il était théoriquement possible de prendre en compte de façon uniforme les processus, les règles, les fonctions, etc. par des objets. Aujourd'hui, on en revient à une vision moins hégémonique où les différents paradigmes de programmation coexistent sans donner plus d'importance à l'un ou à l'autre [Béz04].

1.2 Model Driven Architecture (MDA)

L'OMG a défini le MDA en 2000 [MM03] pour acquérir de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles. Cette approche vise à mettre en valeur les qualités intrinsèques des modèles, telles que pérennité, productivité et prise en compte des plateformes d'exécution. MDA inclut pour cela la définition de plusieurs standards, notamment UML, MOF et XMI¹.

Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, le MDA préconise l'élaboration de modèles :

- d'exigences (*Computation Independent Model* - CIM),
- d'analyse et de conception (*Platform Independent Model* - PIM),
- de code (*Platform Specific Model* - PSM).

L'objectif majeur de MDA est l'élaboration de modèles pérennes (PIM), indépendants des détails techniques des plate-formes d'exécution (J2EE, .Net, PHP, etc.), afin de permettre la génération automatique de la totalité du modèle de code (PSM) et d'obtenir un gain significatif de productivité.

Le passage de PIM à PSM fait intervenir des mécanismes de transformation de modèle (*Cf.* paragraphe suivant) et un modèle de description de la plateforme (*Platform Description Model* - PDM).

Cette approche fait actuellement l'objet d'un grand intérêt dans la littérature spécialisée. Nous citons entre autre les ouvrages de X. Blanc [Bla05] et de A. Kleppe [KWB03] dont est inspiré ce paragraphe.

1.3 La transformation de modèles

Dans le cadre de l'ingénierie des modèles, une grande question se pose quant à la transformation de modèle à modèle, point clé de l'approche MDA de l'OMG. En effet, l'intérêt de transformer un modèle Ma en un modèle Mb que les méta-modèles respectifs MMa et MMb soient identiques (transformation *endogène*) ou différents

¹XML Metadata Interchange est un format d'échange basé sur XML pour les modèles exprimés à partir d'un méta-modèle MOF.

(transformation *exogène*) apparaît comme primordial (refactoring, migration technologique, etc.) [Béz04]. D'autre part, l'approche MDA repose sur le principe de la création d'un modèle indépendant de toute plateforme (PIM) pouvant être "instancié" en un ou plusieurs modèle(s) spécifique(s) à une plateforme (PSM). Les méthodes de transformations sont là aussi indispensables pour le passage de PIM à PSM mais également de PSM à PIM et enfin de PIM à PIM [GLR⁺02].

On comprend donc pourquoi le succès de l'ingénierie des modèles, et donc de l'approche MDA, repose en grande partie sur la résolution du problème de la transformation de modèles.

Pour cela, de nombreux travaux ont été réalisés et peuvent être classés en plusieurs générations [Béz03a] :

– *Génération 1 : Transformation de structures séquentielles d'enregistrement :*

Dans ce cas un script spécifie de manière déclarative comment un fichier d'entrée est réécrit en un fichier de sortie (ex : scripts Unix, AWK ou Perl). Bien que ces systèmes soient plus lisibles et maintenables que d'autres systèmes de transformation, ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie [GLR⁺02, Béz03a].

– *Génération 2 : Transformation d'arbres :*

Ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie (ces méthodes se basent généralement sur des documents au format XML et l'utilisation de XSLT ou XQuery).

– *Génération 3 : Transformation de graphes :*

Avec ces méthodes, un modèle en entrée (graphe orienté étiqueté) est transformé en un modèle en sortie (ex : ATL, MIA). Ces approches visent à considérer l'"opération" de transformation comme un autre modèle conforme à son propre méta-modèle (lui-même défini par rapport au MOF dans le cas de l'approche MDA).

La transformation d'un modèle Ma (conforme à son méta-modèle MMa) en un modèle Mb (conforme à son méta-modèle MMb) par le modèle Mt peut donc être formalisée sous la forme fonctionnelle suivante [Béz03a] :

$$Mb \leftarrow f(MMa, MMb, Mt, Ma)$$

Les premiers travaux sur ce type de transformations ont permis (de manière non définitive) de faire ressortir les sept niveaux suivants de systèmes de transformation² [Béz03a] :

- Niveau 1 : $Ma \rightarrow Ma$
- Niveau 2 : $Ma \rightarrow Mb$; $MMa = MMb$
- Niveau 3 : $Ma \rightarrow Mb$; $MMb = MMa'$
- Niveau 4 : $Ma \rightarrow Mb$; $MMa \subseteq MMb \vee MMb \subseteq MMa$
- Niveau 5 : $Ma \rightarrow Mb$; $MMa \cap MMb \neq \emptyset$

²En considérant MMa & MMb les méta-modèles respectifs des modèles Ma & Mb et MMa' un méta-modèle "proche" du méta-modèle MMa (e.g. les profils).

- Niveau 6 : $Ma \rightarrow Mb$; $MMa \cap MMb = \emptyset$
- Niveau 7 : $Ma \rightarrow Mb$; $MMa = Mc$; $MMb = MMc$

Par ailleurs, les travaux de l'organisation DSTC³ sur les transformations de modèles de troisième génération ont permis d'établir une liste des exigences que doit respecter un langage de transformations [GLR⁺02] :

– *Exigences fonctionnelles* :

- Permettre de transformer un ensemble d'éléments d'un modèle source,
- Permettre de transformer les éléments par type,
- Permettre de modifier la transformation d'un élément de modèle en fonction des associations, des valeurs des attributs, etc.,
- Permettre d'établir les relations entre les éléments d'un modèle source et les éléments d'un modèle cible. Ces associations peuvent être implicites et peuvent être utilisées pour la traçabilité,
- Permettre de conserver une relation d'ordre au sein d'un ensemble d'éléments à transformer,
- Permettre de manipuler des structures récursives en se limitant à un niveau d'imbrication arbitraire.

– *Exigences non fonctionnelles* :

- Définir plusieurs modèles cibles à partir d'une seule règle,
- Groupement des règles pour la lisibilité et la modularité,
- Définition de transformations intermédiaires (i.e. transformation par étapes),
- Possibilité d'intégrer de manière explicite les conditions et expressions dans la transformation de modèles,
- Possibilité de traiter des attributs optionnels.

Le problème de la transformation de modèles est au cœur de l'approche MDA et fait à ce jour l'objet d'un appel à proposition industriel [Obj02b] afin d'établir un nouveau standard intégré au MOF 2.0 : Query/Views/Transformations (Cf. fig.1.5).

Cet appel, ou *Request For Proposal* (RFP), est le sixième dans la série du MOF 2.0 et jouera certainement un rôle clé dans l'approche MDA qu'il est indispensable d'enrichir d'un langage riche et puissant de transformation de modèles [GLR⁺02].

Cet RFP est un appel aux industriels afin d'établir un standard permettant la transformation de modèles mais également l'interrogation d'un modèle afin d'en faire ressortir différentes vues avant la transformation proprement dite [Béz03a]. “*Interroger*” (*Query*) un modèle demande à la fois de pouvoir filtrer ce modèle et d'autre part de pouvoir en sélectionner des éléments. La sélection d'un élément étant la base de la transformation (ceci est similaire au besoin de XPath⁴ dans XSLT⁵). La *création de vues* (*View*)

³ Cf. <http://www.dstc.edu.au/>

⁴ Langage permettant de localiser avec précision une partie donnée d'un document XML.

⁵ Langage de transformation de documents XML en d'autres documents XML.

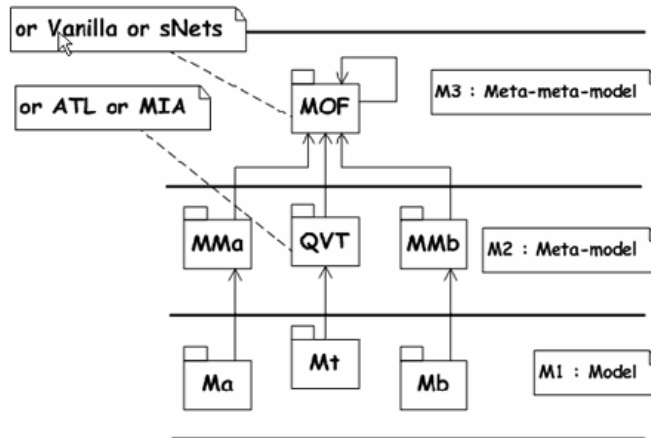


FIG. 1.5 – Transformation de modèles dans l’approche MDA

permet de faire ressortir d’un modèle différents aspects spécifiques. Une vue est alors un modèle spécifique dérivant du modèle d’origine. Enfin, le *langage de transformation* (*Transformation*) devra permettre la description des relations entre un élément d’un modèle source et un élément du modèle cible. Ce langage devra par ailleurs être de type déclaratif.

La syntaxe abstraite des langages de définition des transformations, des vues et des interrogations devra être conforme au méta-méta-modèle MOF (version 2.0).

Une des réponses à cet RFP est le langage ATL (*Atlas Transformation Language*) proposé par l’équipe Atlas⁶. ATL, publié dans un rapport de Recherche de 2003 [BBDV03], est un langage de transformation de modèles conforme aux normes MDA dans lequel les modèles et les transformations de modèles sont traités en tant que citoyens de première classe. En particulier, ATL permet les transformations évoluées qui s’avèrent très utiles pour la réutilisation des transformations.

D’autres réponses ont également été proposées à l’OMG comme QVT-Partner [QP03] et QVT Merge [Gro02].

Par ailleurs, d’autres types de transformation sont actuellement en cours de réflexion au sein de l’OMG comme la transformation de modèles en une représentation textuelle [Obj04a].

1.4 La vérification de modèles

Les diagrammes UML définissent différents points de vue d’une même modélisation. Ces points de vue s’expriment au travers de différents types de modèles (neuf dans la version 1.5 et treize dans la version 2.0 d’UML) ayant des finalités différentes : aspects fonctionnels, aspects structurels, aspects comportementaux [MG03]. Un même élément

⁶INRIA/IRIN - Université de Nantes.

peut apparaître dans différents diagrammes pour modéliser une même abstraction selon différents points de vue pouvant ainsi introduire des redondances, des imprécisions et des incomplétudes [RJB04].

De par l'existence de ces différents points de vue, il est nécessaire d'assurer la cohérence de l'élément de modélisation. La cohérence de modèles UML peut être définie selon trois axes : la cohérence par rapport à la norme, la cohérence par rapport à son contexte et la cohérence par rapport à une démarche d'analyse et de conception [BBLC⁺03]. Ces axes peuvent être regroupés au sein de deux types d'analyse [CL04] :

- L'analyse syntaxique et sémantique (contraintes au niveau du méta-modèle) qui concerne le contrôle de l'écriture du modèle relativement aux règles imposées par l'OMG au sein du méta-modèle de référence et aux normes du projet (e.g. règles de nommages, etc.).
- L'analyse métier (contraintes au niveau modèle) qui permet de s'assurer de la bonne compréhension des processus métier. Elle permet de comprendre et de certifier l'expression des besoins, de s'assurer de l'exhaustivité des solutions proposées et de poser une expertise sur les solutions avancées.

Il existe à ce jour une multitude d'approches possibles visant à détecter des anomalies dans une conception UML [TPC] ; elles font appel à des vérifications discrètes de systèmes réactifs. L'une des principales approches est le "model-checking" [SKM01]. Notons que dans le domaine de l'ingénierie du logiciel, les nouvelles normes de l'OMG concernant UML et OCL offrent de plus en plus la possibilité de vérifier l'intégrité et la cohérence de modèles construits à partir du MOF. Une meilleure cohérence des différents points de vue permet d'assurer une transition semi-automatique vers l'étape de mise en oeuvre et s'apparente à un processus de transformation de modèle.

D'autre part, d'une façon quasi générale, une modélisation UML d'un système réel comporte des aspects statiques et des interactions dynamiques. En général, les diagrammes statiques en UML, comme le diagramme de classe, sont utilisés pour identifier les aspects statiques du système. La représentation des aspects dynamiques d'une modélisation UML peut s'effectuer au travers des diagrammes d'activité ou des diagrammes de machine état (version UML < 2.0). Pour la nouvelle version d'UML (2.0), des diagrammes nouveaux peuvent intervenir dans la description de la dynamique, particulièrement ceux mettant en oeuvre les notions de timing (explorer les comportements d'un ou plusieurs objets pendant une période de temps), de port (connexions inter-diagrammes) et d'échange inter-diagramme.

Le contrôle d'une modélisation UML examine des aspects statiques (comme ci-dessus) mais également des aspects dynamiques, la solution "idéale" étant fournie par un même outil, apportant des aides au concepteur relativement au méta-modèle dans le cadre du contrôle des aspects statiques [SCH02]. L'existence de *points de variation sémantique* au sein d'UML (i.e. différentes interprétations possibles d'un jeu de concepts) [CJV04] nécessite toutefois, pour une validation dynamique, une réification⁷ du com-

⁷La réification est un moyen de représenter une entité abstraite en objet physique.

portement des modèles UML en systèmes de transitions étiquetées. La vérification dynamique permet alors la simulation de scénarii pour la validation d'invariants du modèle.

*L'*effervescence que l'on retrouve autour de l'ingénierie des modèles et plus particulièrement sur la problématique de la transformation de modèles devrait aboutir dans le futur à la normalisation d'un ensemble de standards garantissant la pérennité et le caractère formel des modèles.

C'est pourquoi notre travail s'appuie sur ces principes pour la spécification, la vérification et l'outillage de l'ingénierie des procédés.

Chapitre 2

Software Process Engineering Metamodel (SPEM)

L'ingénierie des modèles tente elle aussi de répondre à la problématique des procédés de développement et l'OMG a pour cela proposé le langage de modélisation de procédé SPEM [Obj05].

Suite à une étude détaillée de la dernière version de SPEM, nous proposons au sein de ce chapitre certaines précisions sur la sémantique des principaux concepts indispensables pour une bonne utilisation du langage.

2.1 Présentation de SPEM

2.1.1 Origines et évolutions

SPEM est un langage de modélisation utilisé pour la spécification de procédés de développement concrets ou d'une famille de procédés de développement. Il offre une approche orientée objet et utilise la notation UML. Issu des travaux de l'OMG, SPEM est un méta-modèle qui s'inscrit dans l'organisation de la pile de modélisation du MDA. Nous emploierons donc indifféremment les termes de *langage* (implicitement *de modélisation*) et de *méta-modèle*.

La spécification de SPEM actuellement disponible est la version 1.1 de janvier 2005 [Obj05] et fait suite à la toute première version (1.0) publiée en novembre 2002 [Obj02c].

D'autre part, SPEM s'inscrivant dans l'organisation des méta-modèles de l'OMG, il fait actuellement l'objet d'un "appel à proposition" (RFP) lancé en novembre 2004 pour la mise au point de la version 2.0 [Obj04b]. Cette nouvelle version devra adapter le méta-modèle en fonction des évolutions d'UML 2.0 et du MOF 2.0. Les propositions doivent être rendues pour la fin du mois de mai 2005.

2.1.2 SPEM : méta-modèle MOF ou profil UML ?

Dans le cadre de l'ingénierie des modèles et plus particulièrement au sein de l'approche MDA, deux techniques existent pour la description d'une nouvelle syntaxe :

- la définition d'un modèle conforme au MOF et donnant une "grammaire" d'un nouveau langage : il s'agit d'un *méta-modèle*.
- l'extension d'un méta-modèle existant au travers d'annotations (stéréotypes, notes, contraintes, etc.) permettant de spécialiser (principalement pour un domaine métier) un élément quelconque du méta-modèle étendu : il s'agit du principe des *profils*¹ (e.g. les profils Temps Réel pour le développement logiciel).

La sémantique sera dans les deux cas exprimée, soit de manière formelle à l'aide de langage comme OCL, soit de manière informelle avec des explications en langage naturel.

De manière conceptuelle et théorique, le choix entre les deux se fait par rapport au domaine d'application du futur langage. En effet, pour la création d'un langage dont le domaine d'application est le même qu'un méta-modèle existant et dont la sémantique a besoin d'être étendue, il est nécessaire d'avoir recourt au principe des *profils*. Les nouveaux langages apportant une syntaxe et une sémantique dans un nouveau domaine d'application donneront lieu à la création d'un nouveau *méta-modèle*. Ce principe nécessite toutefois que les concepts partagés entre les différents méta-modèles soient centralisés au sein du MOF afin qu'ils puissent tous en hériter.

SPEM permettant la modélisation des procédés de développement, il devrait donc naturellement s'exprimer sous la forme d'un nouveau méta-modèle [BB01b] (fig. 2.1). Toutefois, les aspects historiques et commerciaux de l'approche MDA, et plus globale-

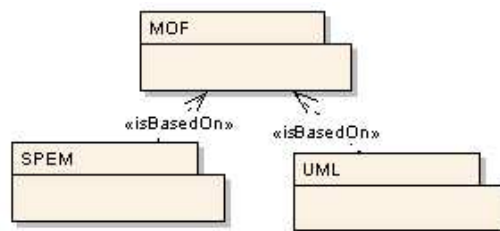


FIG. 2.1 – SPEM comme méta-modèle MOF

ment de l'OMG, ont fait que SPEM fut défini comme un profil UML, i.e. comme un modèle UML (fig. 2.2) :

- d'une part *historique* car des concepts fondamentaux (e.g. certains diagrammes) ont été intégrés, à l'origine, au sein d'UML et non du MOF. Afin d'utiliser UML comme notation concrète sans réécrire ces concepts, SPEM a dû étendre UML et

¹Les profils UML sont standardisés par l'OMG depuis 2001 afin de permettre d'assurer entre autre la transformation d'un PIM vers un PSM. Ils constituent la solution clé en main pour piloter un procédé de développement avec l'approche MDA.

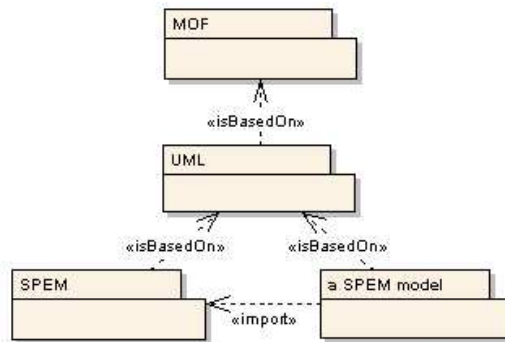


FIG. 2.2 – SPEM comme profil UML

- non le MOF [BB01a],
- d'autre part *commercial* car cette approche facilite l'échange avec les outils aussi bien basés sur UML que sur le MOF [Obj05]. Elle est par ailleurs compatible avec l'ensemble des outils déjà disponibles pour UML.

SPEM est donc défini comme un profil UML et, transitivement, comme un méta-modèle conforme au MOF. Cette organisation particulière des méta-modèles (*Cf.* [Obj05], §11) apporte également certaines facilités dans leurs évolutions ; comme la compatibilité ascendante avec les versions précédentes et l'implémentation d'outils par extension d'outils UML [Obj04b].

Les sociétés Softeam² et Sparx Systems³ ont développé un profil SPEM d'UML afin de pouvoir implanter ce langage au sein de leurs outils respectifs objecteering/UML et Enterprise Architect. Par ailleurs, le méta-modèle SPEM est manipulé au cœur du logiciel RUP BUILDER d'IBM⁴ et permet la personnalisation du processus RUP conformément au méta-modèle SPEM.

Une présentation plus détaillée de l'organisation des méta-modèles de procédés est proposée dans [BB01a].

2.1.3 Les concepts de SPEM

2.1.3.1 L'idée centrale

Le méta-modèle SPEM repose sur l'idée qu'un procédé de développement de logiciel est une collaboration entre des entités actives et abstraites appelées les *rôles* qui effectuent des opérations appelées les *activités* sur des entités concrètes et réelles appelées

² *Cf.* <http://www.softteam.fr/>

³ *Cf.* <http://www.sparxsystems.com.au/>

⁴ *Cf.* <http://www-306.ibm.com/software/awdtools/rup/>

les *produits*. Les figures 2.3 et 2.4, reprises de la spécification de SPEM (Cf. [Obj05], §3), traduisent ce modèle conceptuel fondamental en utilisant la notation UML. Ces figures ne font pas partie de la formalisation de SPEM et sont données pour des raisons uniquement explicatives. Elles sont intentionnellement inachevées.

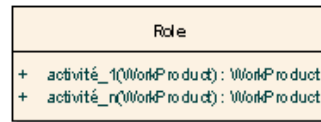


FIG. 2.3 – Modèle conceptuel de SPEM

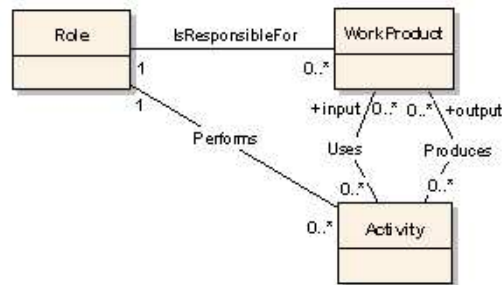


FIG. 2.4 – Réification du modèle conceptuel de SPEM

Les différents rôles agissent les uns sur les autres ou collaborent en échangeant des produits et en déclenchant l'exécution de certaines activités. L'objectif d'un procédé est de mener un ensemble de produits à un état bien défini.

La réification de *rôle*, *activité* et *produit* de la figure 2.3 mène au modèle simple de la figure 2.4.

Un autre point de vue clé de SPEM est la catégorisation en fonction de “thèmes communs” des tâches (et donc les rôles) selon des *disciplines*.

2.1.3.2 Le méta-modèle

Le méta-modèle, tel qu'il est proposé dans sa version 1.1, est découpé en deux paquetages principaux⁵ (fig. 2.5) :

- Le paquetage *SPEM_Extensions* qui décrit l'ensemble minimum des éléments nécessaires à la modélisation des procédés de développement logiciel sans ajouter

⁵Le détail de la structure des packages du méta-modèle SPEM est présenté à la figure 4-1 de la spécification [Obj05]. Nous citons également le rapport [Zyt02], en français, qui offre une présentation du méta-modèle.

de modèles ou de contraintes spécifiques à ce domaine (approche souhaitée par l'OMG [Obj05]).

- D'autre part, UML est défini par un méta-modèle qui est lui-même défini comme une instance du méta-méta-modèle MOF. Un sous-ensemble de la notation graphique d'UML est utilisé pour définir le méta-modèle SPEM. Ce méta-modèle, décrit similairement, est formellement défini comme une extension d'un sous-ensemble d'UML appelé *SPEM_Foundation*.

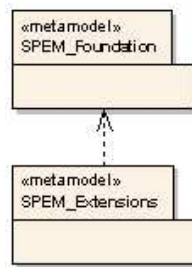


FIG. 2.5 – Packages SPEM

SPEM reprend ainsi d'UML une grande partie des diagrammes (packages, cas d'utilisation, classes, activités, séquences, états-transitions) mais exclut toutefois la sémantique de certains éléments et ne permet donc pas l'établissement de diagrammes d'implémentations et de composants. De nombreux stéréotypes ont par ailleurs été ajoutés pour affiner la sémantique de certains éléments UML (*Cf.* [Obj05], §11.4).

2.1.3.3 Vers une version 2.0

Les exigences de l'OMG pour la future version 2.0 de SPEM sont recensées au sein d'un document appelé *Request For Proposal* (RFP) [Obj04b]. Cet RFP recense et classe, en fonction de leur importance, les exigences en deux catégories : spécifiques et optionnelles.

- *Exigences spécifiques :*

La sémantique de la version 1.1 était ambiguë et difficile à comprendre. La nouvelle version devra proposer des solutions qui amélioreront les techniques de modélisation et les possibilités offertes (processus d'automatisation, processus de technicien système, modélisation basée sur les composants, spécification de contraintes, notion de "point de décision" automatisé ou basé sur une action humaine explicite, etc.).

Pour cela la version 2.0 de SPEM devra s'intégrer au nouveau méta-modèle UML⁶ (particulièrement au niveau de la redéfinition des activités) et fournir un nouveau schéma XML basé sur le MOF 2.0. SPEM devra également s'aligner avec les autres standards : BPDM & BPRI.

Afin de garantir une parfaite compatibilité ascendante, la nouvelle version devra fournir des conseils pour la migration de SPEM 1.1 à 2.0. Idéalement, la solution proposée pourrait offrir des règles permettant d'automatiser la migration.

– *Exigences optionnelles :*

La solution proposée pourra également prédéfinir des produits, permettre d'annoter un procédé (peut-être en OCL) afin de pouvoir calculer des métriques, supporter l'analyse de risque et l'évaluation de coûts et offrir des dispositifs pour l'extensibilité et la variabilité des procédés.

2.1.4 Détails d'utilisation de certains stéréotypes

La sémantique d'un grand nombre d'éléments de SPEM est donnée au sein de la spécification en langage naturel. Cette sémantique, n'étant pas dans ce cas formalisée, peut donner lieu à diverses interprétations et ainsi fausser l'utilisation du langage SPEM.

Nous proposons ci-dessous des précisions venant en complément des définitions données au sein de la spécification officielle [Obj05].

2.1.4.1 Procédé et discipline

Un Procédé (*Process*) correspond à la racine du modèle de procédé à partir duquel un outil peut faire la fermeture transitive du procédé complet.

Une Discipline (*Discipline*) permet au sein d'un procédé de partitionner les activités selon un thème commun. Les produits de sortie de chacune des activités d'une discipline doivent être similairement catégorisées sous ce même thème.

Les procédés et les disciplines sont des *composants de procédé*⁷ (i.e. héritent de la méta-classe *ProcessComponent*), partie fondamentale de la description et de l'assemblage d'un procédé complet. Un *ProcessComponent* importe un ensemble d'éléments de procédé, modélisé en SPEM. Un tel ensemble doit être autonome (i.e. d'un seul bloc) ; ceci signifie qu'il doit n'y avoir aucune dépendance (de type "RefersTo") d'un élément du package vers un autre élément extérieur au package.

⁶SPEM devra toutefois garder sa double représentation : sous forme de profil UML et d'un méta-modèle MOF. Il devra également utiliser une partie d'UML mais pas l'inverse.

⁷Un composant processus est un élément de modèle dont la structure interne est suffisamment consistante pour être réutilisée avec ou à l'intérieur d'un autre composant de processus. Ce mécanisme est possible dès lors que le composant de processus possède une certaine "autonomie" envers les contraintes et les autres éléments du modèle [Béz03b].

2.1.4.2 Cycle de vie, phase, itération et activité

Chacun de ces éléments hérite de *WorkDefinition* (*Définitions de Travail*), entité abstraite définissant de manière générique un travail au sein d'un procédé (fig.2.6 et fig.2.7). Plus précisément :

- Le cycle de vie (*Lifecycle*) représente une séquence de phases permettant d'atteindre un but spécifique. Le cycle de vie définit le comportement d'un processus.
- Une phase (*Phase*) est un ensemble de tâches ayant un objectif commun. La précondition d'une phase définit son critère d'entrée, et son but, son critère de sortie. Les phases sont ordonnées dans le temps et peuvent être composées d'itérations ou d'activités.

La notion de phase se retrouve au sein du procédé RUP sous les noms de *inception*, *élaboration*, *construction* et *transition* [Lar05].

- Une itération (*Iteration*) est définie comme la description d'un travail composite, c'est-à-dire qu'elle est structurée selon un certain nombre d'activités ordonnées permettant la réalisation d'une étape précise. Notons qu'une itération n'a pas pour sémantique le fait d'être répétée plusieurs fois au sein du procédé de développement. Cette notion devra être exprimée au sein d'un diagramme d'activité par une boucle et une garde explicite.
- Une activité (*Activity*) est une définition de travail élémentaire. D'autres rôles (que celui qui en a la responsabilité) peuvent intervenir en tant qu'assistants. Une activité utilise certains produits et permet la réalisation d'autres produits. Notons qu'une activité est le plus bas niveau d'une Définition de Travail mais peut être décomposée en actions (*Step*). Les actions ne sont pas des Définitions de Travail et n'ont, par conséquent, pas de produits d'entrée et de sortie. Elles sont également toutes réalisées par le rôle ayant la responsabilité de l'activité complète.

Chaque *WorkDefinition* est associée à un rôle (*ProcessPerformer*) qui est responsable de sa réalisation.

2.1.4.3 Les rôles

Chaque *WorkDefinition* est sous la responsabilité d'un unique rôle. Un ensemble d'autres rôles peuvent également assister le rôle principal dans la réalisation de cette activité. Dans la spécification de SPEM, un rôle est une entité abstraite qui correspond à un ensemble de compétences. Un acteur concret (mais pas forcément humain) d'un procédé pourra donc jouer un ou plusieurs rôles en fonction de ses compétences. Réciproquement, un rôle peut être joué par un ou plusieurs acteurs.

SPEM introduit au sein du méta-modèle deux types de rôles (fig.2.6) :

- Les *ProcessPerformer* qui réalisent des tâches de haut niveau d'agrégation qui ne peuvent pas être attribuées à un rôle plus particulier.
- Les *ProcessRole* sont liés directement aux activités et aux produits.

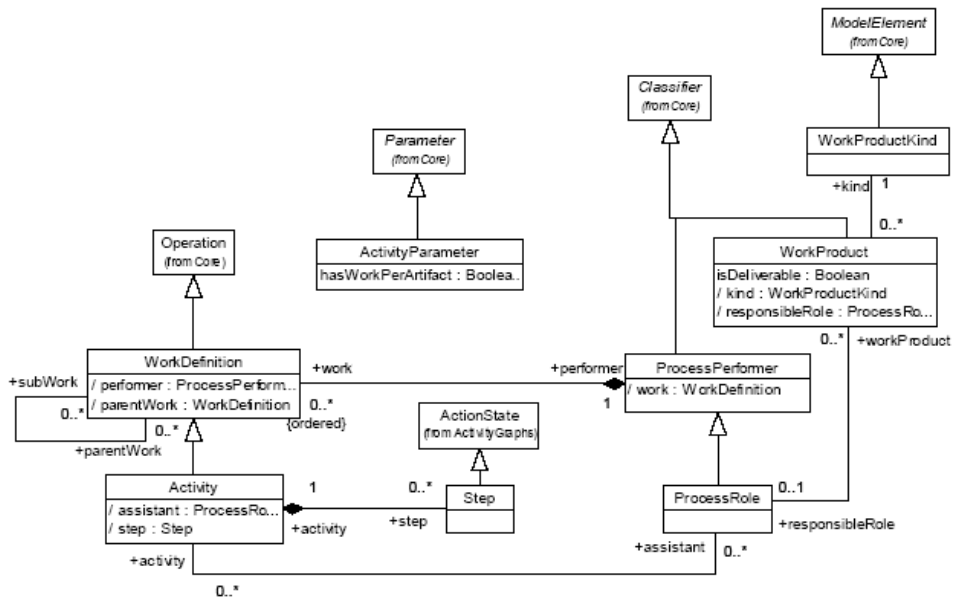


FIG. 2.6 – Package Process Structure

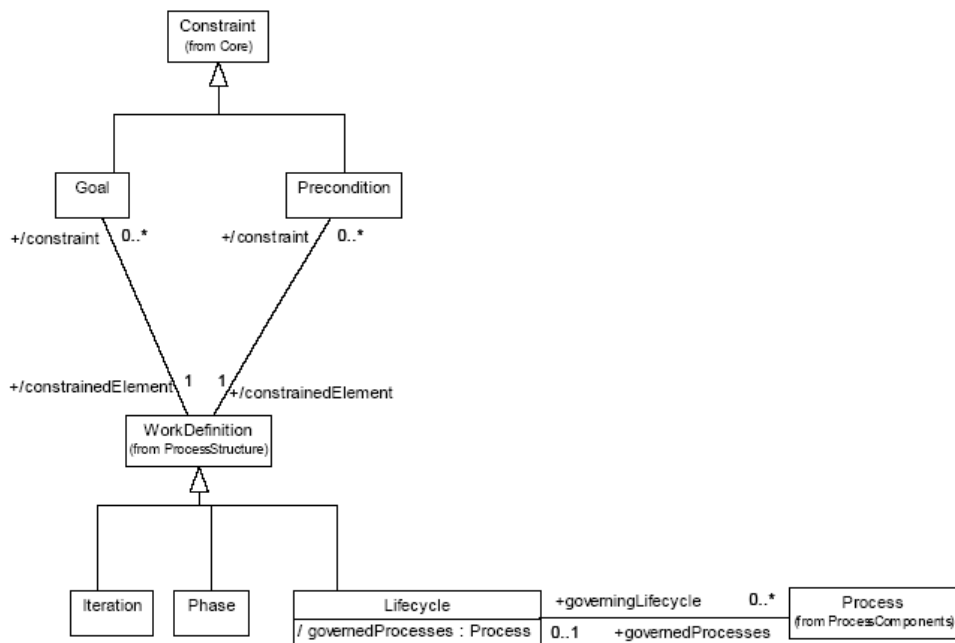


FIG. 2.7 – Package Process Lifecycle

2.2 Outils pour la modélisation en SPEM

2.2.1 Objecteering/UML

2.2.1.1 Présentation de l'outil

Objecteering/UML⁸ est un outil simple et complet pour l'utilisation du langage UML et permet ainsi d'analyser, de concevoir, de mettre en application, d'examiner et de déployer des applications logicielles. Cet outil s'inscrit également dans l'approche innovante du MDA.

Objecteering/UML assure l'uniformité entre tous les éléments UML et une productivité accrue pour Java, C++, Corba IDL, la génération de code SQL et la génération automatisée de documentation. Pour cela, il s'intègre avec les principaux gestionnaires de configuration, IDE et serveur d'application tel que EJB.

Enfin, grâce à l'outil de création de profil UML (Profile Builder), Objecteering/UML offre aux équipes la possibilité d'établir leurs propres profils UML.

L'architecture générale de l'outil Objecteering/UML est présentée sur la figure 2.8.

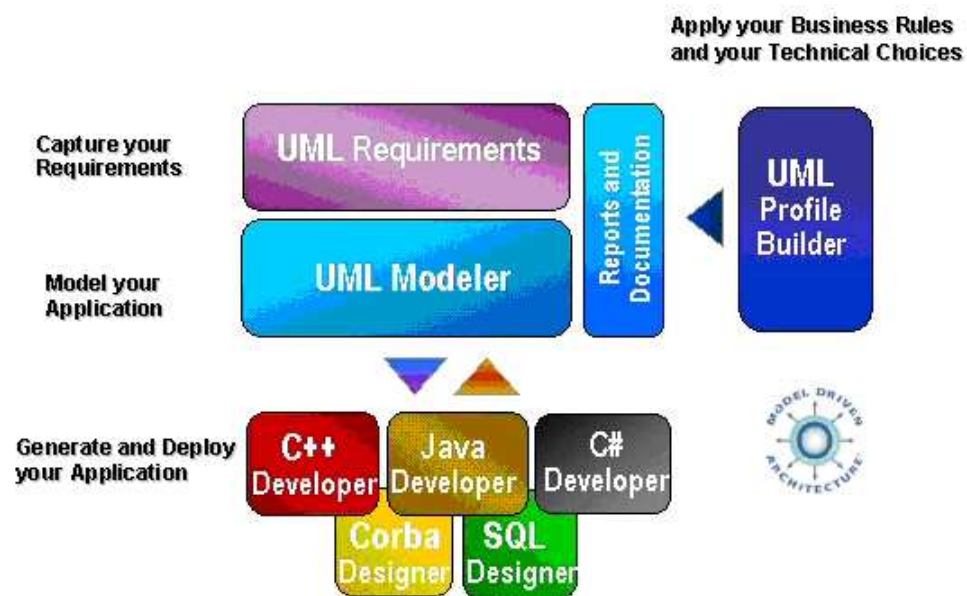


FIG. 2.8 – Architecture générale de l'outil Objecteering/UML

2.2.1.2 Utilisation du module SPEM

L'équipe R&D de la société Objecteering, sous la direction de P. Desfray, est actuellement en train de mettre au point un profil SPEM permettant de prendre en compte

⁸Outil proposé par la société Objecteering, Cf. <http://www.objecteering.com>

au sein de leur outil de modélisation, Objecteering Modeler, la sémantique de SPEM ainsi que l'ensemble des stéréotypes propres à ce langage. Ce profil a été implanté directement à partir de leur générateur de profil (Objecteering/Profile Builder) et permet la réalisation de modèles SPEM. Ce profil est actuellement dans sa phase d'implémentation et ne fait donc pas partie du package commercial de l'outil. Objecteering n'offre également pas de support pour ce profil.

L'utilisation de ce profil se fait simplement en ajoutant le "module" SPEM à la liste des modules pris en compte dans le projet en cours. Les concepts de SPEM sont alors disponibles pour la réalisation de modèles de procédés de développement.

Objecteering a pour objectif d'être directif et de faire des vérifications par rapport au méta-modèle et à la sémantique. Un certain nombre de manques réside toutefois au sein du logiciel, limitant ainsi les possibilités actuelles de modélisation. L'utilisation de ce logiciel pour la spécification complète d'un procédé reste donc délicate mais bénéficie d'améliorations régulières de la part du service R&D.

2.2.2 Enterprise Architect

2.2.2.1 Présentation de l'outil

Enterprise Architect⁹ (EA) est un logiciel de modélisation UML supportant entre autre la version 2.0 d'UML et la notion de Profil. A l'image d'Objecteering, il s'agit d'un outil évolué et ergonomique permettant l'intégration partielle des modèles au sein de l'approche MDA. La nouvelle version 5.0 propose une intégration encore plus importante ainsi qu'une amélioration de l'interprétation de la sémantique. EA n'offre toutefois pas d'outil complémentaire comme Objecteering permettant la réalisation complète d'un projet centré sur les modèles.

2.2.2.2 Utilisation du profil SPEM

Le profil SPEM est fourni sous forme de fichier XML (XMI) qui doit être importé au sein du projet. L'import de ce profil permet à l'outil de fournir l'ensemble des stéréotypes de la spécification (ou presque) mais ne fait aucune vérification sémantique. EA n'a donc pas de restriction dans l'utilisation mais ne garantit pas la création de modèles cohérents, même par rapport à la norme.

Objecteering et EA constituent donc principalement des outils de modélisation avancée supportant les profils UML et particulièrement le langage SPEM. Ils permettent de manière conviviale, de faire la modélisation d'un procédé. Les deux outils ont toutefois une approche différente : le premier essaye de faire une interprétation de la sémantique du profil UML, dans notre cas SPEM. Cette vérification n'étant pas toutefois finalisée, certaines restrictions résiduelles limitent l'utilisation possible de l'outil. EA, pour sa

⁹Outil proposé par la société Sparx Systems, Cf. <http://www.sparxsystems.com.au> .

part, ne fait pas de vérification sémantique et permet donc d'organiser notre projet de la manière dont on le souhaite sans en garantir la cohérence.

*L'*OMG offre la possibilité, avec SPEM, d'appliquer l'ingénierie des modèles pour la problématique des procédés de développement. Bien que cette approche soit naturelle, elle n'apparaît pas toujours triviale. Le langage SPEM connaît encore à ce jour de nombreux manques au niveau de son pouvoir expressif et recèle certaines incohérences. Les précisions apportées au sein de ce chapitre sur la sémantique des concepts principaux de SPEM demande à être développées et les outils manipulant ce méta-modèle ne prennent toujours pas en compte l'ensemble de la sémantique. Son utilisation est donc très délicate et nécessite beaucoup de rigueur.

Chapitre 3

Object Constraint Language (OCL)

La communauté d'où est issue l'ingénierie des modèles a proposé dans un premier temps des langages de spécification graphique permettant dans de nombreux domaines la spécification abstraite par les modèles (logiciels, bases de données, procédés, etc.). Les langages de modélisation graphique se trouvent toutefois limités dans leur pouvoir d'expressivité et dans la définition formelle de leur sémantique [WK99]. Les manques et les ambiguïtés que l'on peut trouver au sein des modèles ne permettent donc pas l'interprétation et la vérification formelle nécessaires pour envisager une validation et une génération de code à partir de ces modèles. Comme tels, ils ne peuvent donc pas être considérés comme "exécutables".

3.1 Historique et Évolution

fin de formaliser la sémantique des modèles, Jos Warmer (IBM) a proposé en 1997 le langage OCL. Ce langage, développé sur la base du langage IBEL (*Integrated Business Engineering Language*), est un langage formel pour l'expression de contraintes et de requêtes appliquées à des diagrammes initialement en UML. UML et OCL s'utilisent donc conjointement.

OCL s'inspire de Syntropy, méthode basée sur une combinaison d'OMT (*Object Modeling Technique*) et d'un sous ensemble de Z, et fut formellement intégré à UML 1.1 en 1999 [Obj97].

Ce langage a subi des améliorations au fur et à mesure des évolutions qu'ont subi les standards de l'OMG, principalement UML. La version 2.0 actuelle fut adoptée par l'OMG en octobre 2003 [Obj03b] en restant toujours liée au méta-modèle UML 1.4¹.

L'ensemble des évolutions dont a pu profiter le langage depuis sa création s'intègre dans le cadre des travaux de l'OMG et plus récemment du MDA. OCL tend ainsi à améliorer aussi bien son expressivité que son caractère formel afin de permettre l'établissement de modèles "exécutables", i.e. modèles considérés comme un produit (i.e.

¹Les liens entre OCL et le méta-modèle UML 2.0 ne seront défini que pour la version 2.1 d'OCL.

artefact) au même titre que du code source. Le modèle pourra alors jouer un rôle central dans l'ensemble du procédé de développement.

Tout au long de la présentation de ce langage nous nous attacherons à mettre en exergue les nouveautés de la version 2.0. La liste exhaustive des changements apportés par la nouvelle version est également disponible en annexe A.

3.2 Présentation du langage

Le langage OCL a deux zones d'applications principales au sein de l'ingénierie des modèles :

- La définition des standards proposés par l'OMG à un niveau méta (M2),
- La modélisation de l'application (M1) où il est alors utilisé à la place d'un langage plus formel (comme Z, VDM, OBJ, etc.).

Le caractère formel de ce langage offre la possibilité d'accompagner les modèles de descriptions précises et non ambiguës (i.e. en se basant sur une grammaire rigoureuse) tout en évitant les désavantages des langages formels traditionnels car il reste facile à écrire et à lire [HZ04].

OCL permet, au sein des diagrammes, l'expression des contraintes suivantes [MG03] :

- Les *invariants* au sein d'une classe ou d'un type,
- Les *contraintes* au sein d'une opération,
- Les *pré- et post-conditions* d'une opération,
- Les *expressions de navigation*,
- Les *gardes* qui conditionnent la modification de l'état d'un objet.

Avec la version 2.0, OCL est passé d'un langage de contraintes à un langage de requêtes [HZ04] et d'expressions de valeurs (valeurs initiales, corps d'opération et règles de dérivation)[WK03] pour des modèles orientés objet. Un des témoins de ce changement est l'intégration du type *tuple* (Cf. [Obj03b], §7.5.15 p.25). Ce type permet, comme les types *record* ou *struct* des langages de programmation structurés, de combiner plusieurs valeurs de type différent au sein d'une même entité (i.e. structure). Ils peuvent ainsi être inclus au sein des expressions d'itération et offrent une gamme complète d'expressions de requêtes (Cf. [Obj03b], §12).

Au niveau du méta-modèle, la nouvelle version d'OCL apporte également un certain nombre de modifications afin de mieux s'intégrer dans l'approche MDA de l'OMG. Les concepts ont donc été en grande partie intégrés au MOF afin de rendre son intégration à UML et aux autres méta-modèles plus naturelle. Cela permet également un meilleur *mapping* de domaine sémantique que les versions précédentes qui n'avaient pas de représentation à un niveau méta [HZ04].

Les modifications au niveau du méta-modèle apportent donc une amélioration du caractère formel du langage. De plus, les vendeurs d'outils ont maintenant toutes les informations nécessaires à la construction d'outils d'analyse et de preuve sur UML et OCL [HZ04].

Concrètement, on retrouve deux définitions de la syntaxe au sein de la spécification :

- la première utilise une approche basée sur le méta-méta-modèle MOF (règles *wellformedness*), c'est la *syntaxe abstraite* (Cf. [Obj03b], §8),
 - la deuxième se base sur l'utilisation d'un ensemble de théories et propose une grammaire, c'est la *syntaxe concrète* (Cf. [Obj03b], §9).
- Ces deux définitions sont équivalentes mais seule la première a un caractère normatif.

Le méta-modèle OCL se décompose en deux paquetages principaux² :

- ***package-type*** qui définit les types reconnus par OCL et les liens avec les concepts du noyau UML,
- ***package-expression*** qui définit les types d'expressions. Les nouveaux types de la version 2.0 sont *oclMessage* et *tuples*.

Le méta-modèle OCL (i.e. syntaxe abstraite) permet de représenter n'importe quelle expression OCL sous forme de modèle. Cette représentation permet de rendre les expressions OCL pérennes et productives. De plus, le lien fort qui unit les modèles UML et les expressions OCL représentées sous forme de modèles permet d'exploiter pleinement les modèles UML dans une approche MDA. OCL contribue de la sorte à faire des modèles UML des PIM de MDA. Un effort important a été fourni pour définir la façon de passer automatiquement de la forme textuelle à la forme modèle car cette traduction n'est pas triviale.

OCL est un langage *déclaratif, typé et sans effet de bord*. Il permet donc de comparer uniquement des instances de même type sans les modifier. Chaque contrainte OCL est liée à un *contexte* définissant le type auquel la contrainte se rapporte (*typeName*). Dans le cas d'un diagramme de classe, une contrainte OCL s'exprime par rapport à une classe (ou à un de ses membres, e.g. attributs ou méthodes) mais s'applique à chacun des objets instances de cette classe.

La syntaxe générale d'une expression OCL est donc :

context *typeName* *stereotype* : *OCLexpression*
avec *stereotype* : := *inv* | *pre* | *post* | *body*³

Sur le diagramme de classe de la fig. 3.1, qui modélise l'arborescence d'un système de fichier, l'invariant suivant permet de spécifier que le nom d'un élément (fichier ou répertoire) ne doit jamais être vide⁴ :

context Element *inv* :
self.name <> ""

Une contrainte peut être également écrite au sein d'un diagramme, placée dans une note attachée au contexte de la contrainte (fig. 3.2).

²Pour plus de détails, Cf. p.162 de l'ouvrage de Jos Warmer et Anneke Kleppe [WK03].

³OCL permet d'exprimer le corp d'opération uniquement si il n'y a que des sélections de données au sein du modèle; OCL étant un langage sans effet de bord.

⁴Le mot-clé *self* permet de désigner l'instance de la classe spécifiée comme contexte (ici *Elements*).

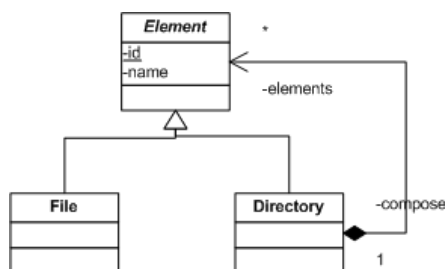


FIG. 3.1 – Diagramme de classe de l’arborescence d’un système de fichier

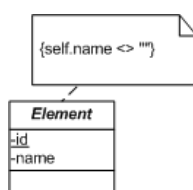


FIG. 3.2 – Contrainte OCL sur un diagramme de classe

La navigation à travers les attributs, les méthodes et les associations se fait à partir du contexte grâce à l’opérateur point (`.`). Dans le cas d’une association, l’expression retourne une collection d’objets dont le nombre est borné par la multiplicité du rôle. Lorsque des collections d’objets sont désignées, le point (`.`) est remplacé par une flèche (`→`).

OCL définit cinq types de collections (fig. 3.3) :

- **Collection** : type abstrait définissant des opérations de manipulation pour les sous-types concrets.
- **Set** : permet de définir un ensemble au sens mathématique du terme . Il s’agit du type des collections obtenues par navigation au travers d’une association car UML impose aux relations une sémantique d’ensemble.
- **OrderedSet** : définit un ensemble où les éléments sont ordonnés. Ces collections, intégrées à OCL à partir de la version 2.0, sont obtenues par navigation sur une association dont la contrainte `{ordered}` est appliquée (i.e. éléments ordonnés).
- **Bag** : Il s’agit d’un “panier à provision” (i.e. multi-ensemble) où un même élément peut être présent plusieurs fois. Ces collections sont obtenues par navigation au travers de plusieurs associations.
- **Sequence** : Il s’agit d’un “panier à provision” où les éléments sont ordonnés.

De nombreuses opérations sont incluses nativement au sein du langage OCL afin de pouvoir manipuler les collections. De nouvelles furent ajoutées dans la version 2.0 d’OCL comme `collectNested`, `flatten`, `excludes`, `excludesAll`, `insertAt`, `indexOf`, `any`, `one`, `isUnique` et `sortedBy`. La liste exhaustive des opérations fournies pour les collections se trouve dans le paragraphe 11.7 de la spécification d’OCL 2.0 [Obj03b].

A partir de l’exemple de la figure 3.1, la contrainte suivante permet de préciser qu’un

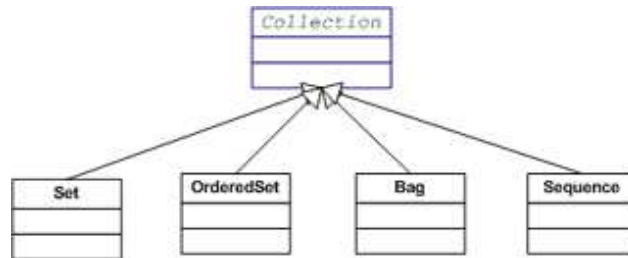


FIG. 3.3 – Collections en OCL 2.0

répertoire ne peut pas se contenir lui même :

```
context Directory inv :
    self.elements → forAll(e : Element | e.id <> self.id)
```

Enfin, on trouve au sein de la version 2.0 d'OCL la possibilité d'exprimer l'envoi de messages⁵ au sein des diagrammes dynamiques. Pour cela, la nouvelle spécification introduit les opérateurs *sent* (\wedge) et *message* ($\wedge\wedge$) :

- *sent* permet d'exprimer l'envoi d'un message sur un objet comme le montre l'exemple suivant :

```
context Subject :: hasChanged()
post observer  $\wedge$  update(12, 14)
```

Cet exemple, basé sur le patron de conception "observateur" [GHJ95], permet de préciser que la méthode *update* de l'objet *observer* devra être appelée avec les paramètres 12 et 14 au cours de l'exécution de l'opération *hasChanged*.

- *message* permet de capturer les différentes occurrences d'un même message qui ont été envoyées au cours de l'exécution d'une opération :

```
context Subject :: hasChanged()
post observer  $\wedge\wedge$  update(12, 14) → notEmpty()
```

Cet exemple, basé sur le patron de conception "observateur" [GHJ95], permet de préciser que la méthode *update* de l'objet *observer* devra être appelée avec les paramètres 12 et 14 au moins une fois au cours de l'exécution de l'opération *hasChanged*.

Les messages capturés peuvent être manipulés à l'aide du nouveau type *OclMessage* et des opérations : *isSignalSent*, *isOperationCall*, *hasReturned* et *result* (Cf. [Obj03b], §11.2.5).

3.3 Étude de l'expressivité

Cette partie fait une analyse du pouvoir expressif d'OCL par comparaison avec les langages relationnels, l'expression de contraintes temporelles et les machines de Tu-

⁵Pour plus de détails, Cf. [WK03], p.156-159 et [Obj03b], §7.7.

ring.

3.3.1 OCL comme langage relationnel

Il existe trois langages de requêtes abstraits : l'algèbre relationnelle, le calcul relationnel de tuples et le calcul relationnel de domaines. Ces langages abstraits ont une expressivité équivalente et sont proposés par Codd comme représentant le minimum d'un langage de requêtes utilisant le modèle relationnel [Cod72]. Ce langage est alors dit complet [GSUW94].

Les langages de requêtes sont décomposés en deux grandes classes [MC99] :

- *langages algébriques* : requêtes exprimées en appliquant les opérateurs spécialisés aux relations,
- *Langages de calcul de prédicats* : langages où les requêtes décrivent l'ensemble de tuples désirés par un prédicat que les tuples doivent satisfaire.

OCL se présente comme un langage mélangeant ces deux types : les requêtes sont définies d'une part comme un ensemble d'éléments satisfaisant la contrainte et d'autre part des opérateurs spéciaux peuvent être appliqués afin d'interagir avec la relation.

Le calcul relationnel propose un certain nombre d'opérations applicables sur les collections. Nous proposons dans la suite de cette partie une étude de l'expressivité d'OCL par rapport à chacune de ces opérations. Pour cela, nous nous appuyons principalement sur les travaux réalisés sur la version 1.x d'OCL par l'Institut Informatique de l'Université de Munich [MC99] en faisant une réactualisation suite à la nouvelle version du langage :

- *union* : il s'agit d'une opération primitive au sein d'OCL pour des collections de même type (ainsi qu'entre les *bag* et les *set*).
- *différence* : il s'agit également d'une opération primitive dans OCL mais uniquement pour les collections de type *set*. On peut néanmoins l'implémenter pour le type *bag* (i.e. $bag - (bag2 : Bag(T)) : Bag(T)$) à partir de la fonction *including* (fig. 3.4)[MC99].

```

{bag -> iterate(
  e : T;
  acc : Bag(T) = Bag{}
  |
    if bag2->includes(e)
      then acc
    else acc->including(e)
    endif
  })

```

FIG. 3.4 – Différence en OCL avec des bag

- *produit cartésien* : Cette opération n'était pas primitive au sein du langage OCL jusqu'à la version 1.5. Cependant différentes implémentations étaient possibles

pour la mettre en place : une solution proposée consistait à mettre en place une nouvelle classe au sein du diagramme stockant l'ensemble des couples entre les deux classes dont on souhaitait calculer le produit cartésien (fig. 3.5). Le produit cartésien était alors garanti par l'ajout d'un invariant sur cette classe (fig. 3.6) [MC99].

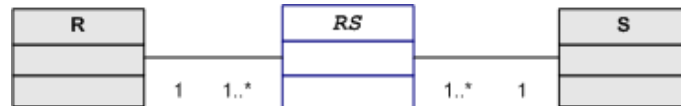


FIG. 3.5 – Classe temporaire pour le Produit Cartésien

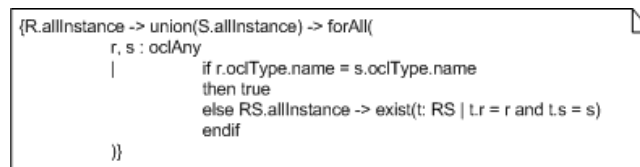


FIG. 3.6 – Produit Cartésien en OCL 1.x

Les concepts de *Tuple* et de *collections de collections*, intégrés dans la version 2.0 d'OCL, ont beaucoup simplifiés l'implémentation du produit cartésien. Cette opération a donc pu être intégrée de manière native par l'opérateur *product* qui s'applique sur tout type de collection (Cf. [Obj03b], §11.7.1). L'exemple précédent peut alors être exprimé de la manière suivante :

$$R.allInstance() \rightarrow product(R.s) : Set(Tuple(first : R, second : S))$$

Notons que la classe "virtuelle"⁶ *RS* n'est plus indispensable pour la navigation.

- **projection** : cette opération n'est possible directement que pour un attribut au moyen de l'opération *collect*. La projection de plusieurs attributs de classe instantiés nécessitera un algorithme plus complexe mais réalisable.
- **sélection** : cette opération est nativement possible en OCL par la fonction *select*.
- **intersection** : il s'agit d'une opération primitive pour les collections de type *bag* et *set*.
- **quotient** : cette opération n'est pas primitive dans OCL mais se base sur le produit cartésien et la projection.
- **jointure** : cette opération n'est pas primitive au sein d'OCL mais repose sur le produit cartésien.

OCL n'est donc pas un langage relationnel complet car la complétude relationnelle est présentée par [Ozk86] comme devant également offrir :

⁶Il s'agit d'une classe temporaire nécessaire comme contexte pour l'expression de la contrainte.

- la capacité de représenter des affectations,
- la capacité de calculer la fermeture transitive et/ou l’emboîtement d’opérations d’algèbre relationnelle.

Codd [Cod72] ajoute le besoin de fournir des fonctions de manipulation de tuples et d’agrégats.

3.3.2 OCL comme langage de manipulation de données

En plus du calcul relationnel, un langage de manipulation de données doit offrir des opérateurs arithmétiques, des commandes d’affectations et des fonctions globales (min, max, etc.). OCL inclut des fonctions globales et arithmétiques mais n’inclut pas des fonctions d’affectations.

Toutefois, la fermeture transitive qui n’est pas exprimable en algèbre et calcul relationnel est exprimable en OCL. Dès les premières versions du langage, l’algorithme de Warshall (93) permettait déjà son implémentation [MC99]. De plus, la notion de tuple intégrée dans la version 2.0 simplifie grandement son implémentation au même titre que le produit cartésien. L’expression de la fermeture transitive n’est toutefois pas nativement incluse dans OCL sous la forme d’un opérateur comme cela est le cas pour les autres langages de navigation comme le langage Alloy.

Au sein de l’extension OCL+ [BMP⁺02] mise au point dans le cadre du projet Neptune⁷, la fermeture transitive est introduite sous la forme d’un opérateur *closure* qui s’applique sur les collections. Le résultat est alors un *set* ou un *bag*.

Cet opérateur permet donc, sur l’exemple de la figure 3.1, d’exprimer *de manière récursive* qu’un répertoire ne peut se contenir lui même. Pour cela, l’invariant sur la classe *Directory* est le suivant :

```
context Directory inv :
    Set{self} → closure(e :Element | e.elements
        → select(e :Element |
            e.oclIsKindOf(Directory)).oclAsType(Directory))
        → excludes(self)
```

3.3.3 L’expression de contraintes temporelles avec OCL

OCL est similaire au calcul des prédicats du premier ordre. Des expressions booléennes peuvent être construites et combinées par des connecteurs logiques. Les quantificateurs universels et existentiels sont bien sûr disponibles.

Comme pour le calcul de prédicats du premier ordre, une expression OCL est évaluée dans un seul état du système. De plus les pré- et post-conditions caractérisent des opérations en considérant des transitions d’état, i.e. une paire d’état.

⁷Nice Environment with a Process and Tools Using Norms (UML, XML and XMI) and Example, Cf. <http://neptune.irit.fr/>.

La logique temporelle, extension de la logique du premier ordre, est souvent utilisée dans le développement de logiciels. Il existe différentes logiques temporelles : linéaire (LTL), arborescente (CTL), d'action (TLA), etc.. Par ailleurs, des combinaisons de chacune sont possibles [Mon00]. L'idée de base de la logique temporelle est de considérer non pas un état mais un ensemble d'états. Il est alors possible de caractériser le comportement d'un système ainsi que les états de son cycle de vie.

Il fût donc logique de voir très rapidement apparaître de nombreuses extensions d'OCL offrant des opérateurs temporels afin de spécifier des contraintes par rapport à un ou plusieurs enchainement(s) d'états souhaités ou interdits.

3.3.3.1 TOCL (Université de Brême, Allemagne)

Cette extension repose sur la logique temporelle linéaire et introduit l'ensemble des opérateurs temporels futurs et passés (*always, previous, next, sometime, before, until, etc.*) [ZG03a]. TOCL (i.e. *Temporal OCL*) peut être utilisé au sein d'invariants mais aussi au sein des pré- et post-conditions.

L'exemple suivant, tiré de l'article [ZG02], permet d'exprimer qu'un programme (*Program*) doit obligatoirement passer par un premier mode (*mode*) d'initialisation (*initialization*) :

```
context Program inv :
    previous false implies mode = 'initialization'
```

Au niveau de l'outillage, TOCL peut être manipulé au sein du logiciel USE⁸ mais reste très dur à mettre en place manuellement [ZG03a]. Les futurs travaux devraient permettre une production automatique des contraintes TOCL au sein de USE.

3.3.3.2 OCL+ (Consortium Neptune, France)

Le fait qu'UML introduise au sein des diagrammes de séquences un ordre partiel oriente le choix vers la logique temporelle arborescente (CTL). OCL+ [BMP⁺02], extension d'OCL mise en place dans le cadre du projet Neptune, offre donc les opérateurs de la CTL (e.g. E, A, AF, AX, EX, EF) applicables sur les diagrammes dynamiques (particulièrement les diagrammes de séquences et d'états-transitions).

L'invariant suivant, donné comme exemple par le Consortium Neptune, permet d'exprimer la propriété d'équité sur une machine de vente qui doit infiniment souvent (*globally finally*) être dans l'état disponible (*Machine : :idle*) :

```
context VendingMachine inv :
    globallyfinally self.oclInState(Machine : :Idle)
```

OCL+ met en application les opérateurs de la logique temporelle par un calcul de point fixe sur un modèle de Kripke. Le vérificateur OCL est actuellement en cours de validation sur cet aspect.

⁸Voir description, partie 3.5.

De nombreuses autres extensions sont également disponibles et reprennent les mêmes concepts que ceux présentés ci-dessus. Une présentation et un comparatif sont proposés au sein de [Fla03].

3.3.4 OCL vs. Machines de Turing

OCL n'est pas, sur de nombreux points, un langage Turing complet⁹. En effet, une des premières limites d'OCL face aux machines de Turing¹⁰ est le fait que l'on ne puisse pas manipuler des boucles dont le nombre d'itérations n'est pas initialement déterminé [MC99].

3.4 Les limites de la version 2.0

Les nouveaux concepts de la version 2.0 sont d'un intérêt incontestable mais ont certaines lacunes et apportent également un certain nombre de problèmes [HZ04] :

- ***oclMessage***, très nouveau, recèle encore des problèmes dont deux fondamentaux :
 - o OCL n'ayant pas de moyen de déclarer des variables libres, l'expression de l'envoi d'un message (i.e. l'appel d'une méthode) intégrée dans la nouvelle version du langage ne permet pas d'exprimer des contraintes sur la valeur ou l'état des paramètres du message¹¹.
 - o Il n'est pas possible d'exprimer l'ordre de l'envoi des messages. En effet, OCL permet d'exprimer un seul message de même nom dans une expression et on ne peut pas dater chaque occurrence pour les messages sélectionnés dans plusieurs expressions. D'autre part la sémantique d'OCL 2.0 ne couvre pas l'expression de messages.
- ***CommonSuperTypes*** est une opération définie en OCL 2.0 sur les *classifier* et qui est utilisée dans un processus appelé "*l'inférence de type*". Il s'agit de la tentative d'identifier le type d'une expression OCL sans donner d'informations explicites.

$$\text{Ex : } \text{Set}\{0,1,3.5\} \rightarrow \text{Set}\{\text{Real}\}$$

Selon l'invariant décrit dans la spécification sur cette opération (Cf. [Obj03b], §8.3.7 p.52), elle n'apporte pas de précision dans le cas où les types ne sont pas directement dépendants par une relation d'héritage. Par exemple, dans le cas de la fig. 3.7, l'ensemble sera de type *Set{oclAny}*.

- Pour la première fois OCL est spécifié par une syntaxe abstraite (i.e. dans la forme du MOF). Un mapping de la syntaxe concrète vers la syntaxe abstraite a donc besoin d'être écrit. La grammaire fournie a été intensivement retouchée

⁹Le terme Turing-complet désigne en informatique un système formel ayant au moins le pouvoir des machines de Turing

¹⁰Une machine de Turing est un modèle abstrait du fonctionnement d'un ordinateur et de sa mémoire, créé par Alan Turing en vue de donner une définition précise du concept d'algorithme ou "procédure mécanique"

¹¹Une alternative pour les types supportant *allInstance* serait : *Real.allInstance(r | op ∧ (r) and r < 10)*

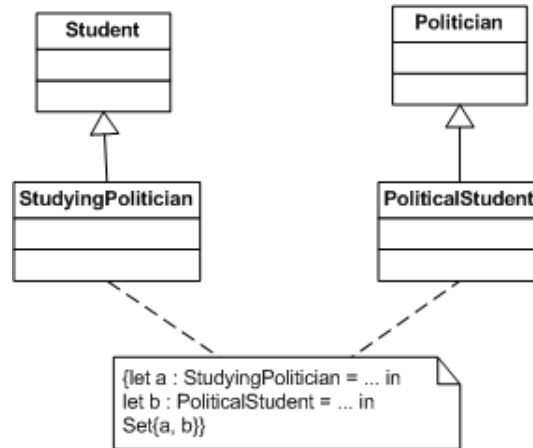


FIG. 3.7 – Super type commun

et exprimée dans un format qui a été inventé spécifiquement à cette fin. Cette approche pose toutefois deux problèmes :

- difficultés de voir les liens entre cette grammaire et celle des versions précédentes,
- impossibilité de dériver des *parser* tant que les ambiguïtés ne seront pas supprimées en transformant la grammaire sous forme LR/LALR (*Left to Right / Look Ahead Left to Right*).

3.5 La vérification outillée de contraintes

Un diagramme de classes peut être complété avec des contraintes OCL. Toutefois, il doit être impossible d’instaurer des contraintes ne respectant pas les standards (i.e. les méta-modèles décrivant les langages), contredisant le diagramme (c’est le cas par exemple des multiplicités) ou n’étant pas respectées au cours d’une des exécutions possibles du modèle (i.e. validation dynamique de modèles).

Bien qu’OCL soit un langage plus facile à lire que les langages formels traditionnels, la vérification de contraintes n’est pas toujours triviale de manière manuelle. C’est pourquoi il est indispensable d’avoir recours à des outils qui permettent une vérification automatique et formelle des contraintes et de leur respect. C’est dans cette optique d’automatisation de la vérification des contraintes que le langage tend vers une meilleure expressivité et une description formelle exhaustive.

Par ailleurs, dans le cas de contraintes qui ne sont pas vérifiables statiquement au cours de l’analyse et de la conception, il est nécessaire de faire des simulations et d’utiliser la technique peu disponible consistant à traduire les contraintes OCL en expressions booléennes et à être insérées dans le code produit de l’utilisateur [CBMC03].

D’un point de vue support, de nombreux outils commerciaux existent pour UML toutefois rares sont ceux qui supportent OCL (pourtant partie intégrante d’UML). Le

premier outil supportant OCL fût USE (UML Specification Environnement) [RG00]. La tâche principale de cet outil est de valider et de vérifier la consistance d'un diagramme avec des invariants et des pré-, post-conditions en OCL. Les composants principaux de cet outil sont un animateur de diagrammes permettant de simuler un modèle UML et un interpréteur OCL pour la validation des contraintes. Pour cela, USE utilise le méta-modèle UML proposé par l'OMG et introduit son propre méta-modèle pour le langage OCL [RG99] qui était dépourvu d'une formalisation par rapport au MOF jusqu'à la version 2.0.

La validation de modèles UML et de contraintes OCL passe ensuite par la génération d'instantanés (i.e. *snapshots*) représentant l'état du système en un point particulier puis par une comparaison avec la spécification initiale. Ces *snapshots* étaient dans un premier temps construits explicitement par une séquence de commandes. Les travaux actuels tentent d'automatiser la génération de *snapshot* par une description déclarative de ses propriétés [GBR03].

L'outil intègre également ses propres commandes pour la manipulation avec effet de bord des objets.

USE a permis, dans le cadre de nombreuses expérimentations, de soulever différents types d'erreurs au sein d'expressions OCL [ZG03b] :

- erreurs de syntaxe,
- erreurs de type,
- contradictions mineures,
- erreurs générales de sémantique,

Ce même outil a également permis de soulever de nombreuses erreurs au sein de la super-structure de la version 2.0 d'UML [BGG04]. Ces erreurs au sein de la description du langage participe à son ambiguïté.

L'outil Neptune¹² [CBMC03], développé au cours du projet de même nom, offre la possibilité d'écrire des contraintes au niveau du méta-modèle (M2). Le vérificateur de Neptune intervient à deux niveaux :

- Vérification de la cohérence des contraintes introduites au niveau applicatif (M1) :
 - invariants, pré- et post-conditions,
 - clauses d'actions proposées par la version 2.0 d'OCL,
 - contraintes temporelles au sein des diagrammes dynamiques.
- Contrôle du respect des contraintes du niveau méta (M2) :
 - règles de cohérence intra-diagramme (*WellFormed Rules*),
 - règles de cohérence inter-diagramme,
 - règles portant sur le langage cible,
 - règles de génie logiciel,
 - règles du procédé métier.

Cet outil, actuellement disponible pour la vérification de modèles issus d'UML

¹²L'outil est disponible gratuitement à l'URL : <http://neptune.irit.fr/Public/Anglais/SoftwareDownload.html>

(v1.4), est maintenant en cours d'extension sous le nom de "Neptune 2" pour supporter les méta-modèles de différents domaines (e.g. la modélisation des procédés des organisations). Cette nouvelle version de l'outil s'appuyera sur le nouveau vérificateur OCL KMF (*Kent Modelling Framework*) proposé par l'université de Kent¹³.

Le vérificateur OCL créé initialement a pour sa part suivi une autre évolution au sein du laboratoire LCI¹⁴ sous le nom de OCLE. Cet outil propose une très bonne ergonomie et offre la saisie de contraintes aussi bien au niveau méta-modèle (M2) qu'au niveau modèle (M1).

D'autres outils ont également été développés pour le support d'OCL comme *Key Tool* [ABB⁺02], l'ancien outil *BoldSoft*, intégré depuis son rachat par Borland à *Together Designer*¹⁵ et *OCL Compiler*¹⁶ [Fin00]. Un interpréteur de contraintes OCL est également disponible sur le site Internet du projet ATL (*Atlas Transformation Language*) du Laboratoire LINA (*Laboratoire d'Informatique de Nantes Atlantique*)¹⁷.

La vérification de modèles basée sur OCL connaît toutefois un certain nombre de limites [BMP⁺02] :

- les règles de la méta-classe *Constraint* sont spécifiées uniquement en langage naturel,
- la contrainte de la méta-classe *Guards* spécifie que celle-ci ne doit pas avoir d'effet de bord. La vérification de cette contrainte ne peut être faite qu'en exécutant une analyse syntaxique et sémantique (ceci n'étant pas le but d'OCL),
- les contraintes sur les flots d'objets ont besoin d'opérations spécifiques au sein du méta-modèle dont la sémantique est exprimée uniquement en langage naturel.

3.6 OCL pour les modèles de procédés de développement

L'approche que nous suivons par la suite propose d'utiliser OCL pour compléter les modèles de procédés de développement exprimés en SPEM. Cette démarche s'intègre dans celle de l'OMG qui s'efforce de fournir des langages graphiques spécifiques pour chacun des domaines et propose OCL comme langage formel commun pour l'expression de contraintes.

Cette approche pose toutefois un certain nombre de problèmes. En effet, OCL est très lié au méta-modèle UML car au niveau de sa syntaxe abstraite il est relié au méta-modèle UML par la notion d'objet et d'instance. Il est donc possible d'appliquer des règles OCL sur des modèles SPEM de la même manière que cela est fait sur des modèles UML uniquement s'il reste défini comme un profil UML héritant ainsi des concepts nécessaires. Cette application d'OCL doit d'ailleurs être traitée dans les réponses au

¹³ Cf. <http://www.cs.kent.ac.uk/projects/kmf/>

¹⁴ Laboratorul Cercetare in Informatica, Cf. <http://lci.cs.ubbcluj.ro/ocle/index.htm>

¹⁵ Cf. <http://www.borland.com/together/>

¹⁶ Cf. <http://dresden-ocl.sourceforge.net/>

¹⁷ Cf. <http://www.sciences.univ-nantes.fr/lina/atl/atldemo/>

RFP lancé par l'OMG pour la future version 2.0 de SPEM [Obj04b]. Notons que les contraintes apparaissent dans la définition de la structure du fichier XMI (DTD¹⁸) associée à une modélisation SPEM.

D'autre part, d'un point de vue outillage, cette approche apparaît d'autant plus naturelle que les éditeurs de logiciels considèrent SPEM comme un profil UML et non un méta-modèle MOF. On retrouve donc au sein d'Objecteering et Enterprise Architect la possibilité de saisir des contraintes OCL en application aux modèles UML (ou SPEM). Plus précisément, Objecteering permet d'appliquer des contraintes à des modèles et permet par la suite de les exporter au sein du fichier XMI du modèle. EA pour sa part permet la saisie des contraintes mais ne les exportent que sous la forme de "tagged values". C'est donc à l'outil exploitant le fichier XMI d'interpréter ces champs comme des contraintes afin de les prendre en compte dans le modèle. Dans les deux cas ces logiciels ne réalisent pas de vérifications statiques ou dynamiques des contraintes.

¹⁸Document Type Definition.

Chapitre 4

Spécification rigoureuse et cohérente de procédés

SPEM est un méta-modèle pour la définition des procédés et de leurs composants. Un outil basé sur SPEM devra donc permettre de décrire et d'adapter un procédé. Pour cela le langage de modélisation SPEM offre une syntaxe et formalise partiellement la sémantique mais ne donne pas de cadre méthodologique pour l'élaboration d'un procédé de développement. Par ailleurs, le méta-modèle proposé par l'OMG se veut très généraliste et par conséquent n'offre pas de directive sur le découpage d'un procédé ainsi que sur les détails d'utilisation de certains stéréotypes propres à SPEM.

Fort du constat que nous avons fait sur la complexité d'utilisation et les incohérences du langage SPEM, nous proposons au sein de ce chapitre des précisions sur la sémantique et l'utilisation du méta-modèle SPEM et proposons pour cela une spécialisation du méta-modèle d'origine. Une démarche pour la formalisation cohérente d'un procédé de développement est également proposée à la fin du chapitre.

Ces travaux s'appuient sur les conclusions de l'élaboration du langage de description de procédé PBOOL [Cré97] et plus récemment sur l'analyse de sa complémentarité avec le langage SPEM [Cap04].

4.1 Spécialisation du méta-modèle SPEM

4.1.1 Les limites du méta-modèle d'origine

Le méta-modèle SPEM proposé par l'OMG, généraliste et très peu directif, permet d'engendrer certains modèles de procédés incohérents (par rapport, entre autres, aux explications complémentaires données en langage naturel).

Par exemple, la relation de composition entre *Namespace* et *ModelElement* du paquetage *Model_Management* (fig. 4.1) n'a pas de contrainte de restriction et permet certaines compositions incohérentes. En effet, on peut aussi bien composer un procédé de

plusieurs disciplines (ce qui est attendu) que l'inverse. Toutefois, composer une discipline de plusieurs procédés vient en contradiction avec les explications de la spécification :

“ Une discipline [...] divise les activités dans un procédé selon un thème commun ” (Cf. [Obj05], §8.4).

Ces explications restent toutefois exprimées en langage naturel et ne peuvent être considérées de manière formelle au sein de la sémantique du méta-modèle.

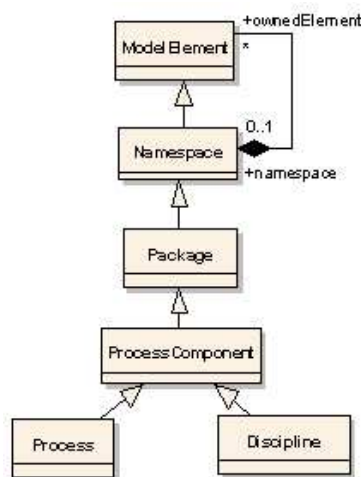


FIG. 4.1 – Packages Model_Management de SPEM

4.1.2 Présentation de notre spécialisation

Les manques de précision, de formalisation et de cohérence au sein du méta-modèle d'origine induisent de grandes difficultés dans l'utilisation du langage SPEM et dans son intégration au sein d'outils de modélisation. Ces manques limitent également les possibilités de vérifications des modèles engendrés à partir de ce méta-modèle.

C'est pourquoi nous proposons une spécialisation du méta-modèle original de SPEM au sein de laquelle sont simplifiés un certain nombre de concepts. En étant plus directive, notre proposition apporte plus d'assistance dans la construction d'un procédé et plus de facilité dans l'utilisation du langage SPEM.

Les restrictions faites au sein de ce langage permettent également d'assurer la cohérence des modèles engendrés et ainsi de permettre leurs vérifications formelles.

Enfin, notre proposition s'ouvre aux évolutions futures du langage au fur et à mesure de leurs formalisations et de leurs exploitation cohérente au sein d'un procédé.

Le méta-modèle proposé offre une modélisation du procédé selon deux vues principales : *structurelle et descriptive* :

- *La vue structurelle* permet de découper le procédé de manière hiérarchique. Pour

cela le procédé est associé à un cycle de vie découpé en phases, elles-mêmes composées d'activités (ou d'itération).

- La *vue descriptive*, quant à elle, permet d'apporter tous les détails nécessaires à la réalisation des définitions de travail. Elle permet ainsi de classifier et d'ordonner (en fonction des pré-conditions et des objectifs de chacune) les définitions de travail selon les rôles qui en sont responsables ; eux-mêmes associés à des disciplines. Cette vue apporte également des détails sur l'utilisation et la réalisation des produits.

Le méta-modèle ainsi obtenu est partiellement présenté sur la figure 4.2. Nous proposons au sein de celui-ci d'affiner à la fois la syntaxe et la sémantique du langage SPEM tel qu'il est proposé par l'OMG [Obj05].

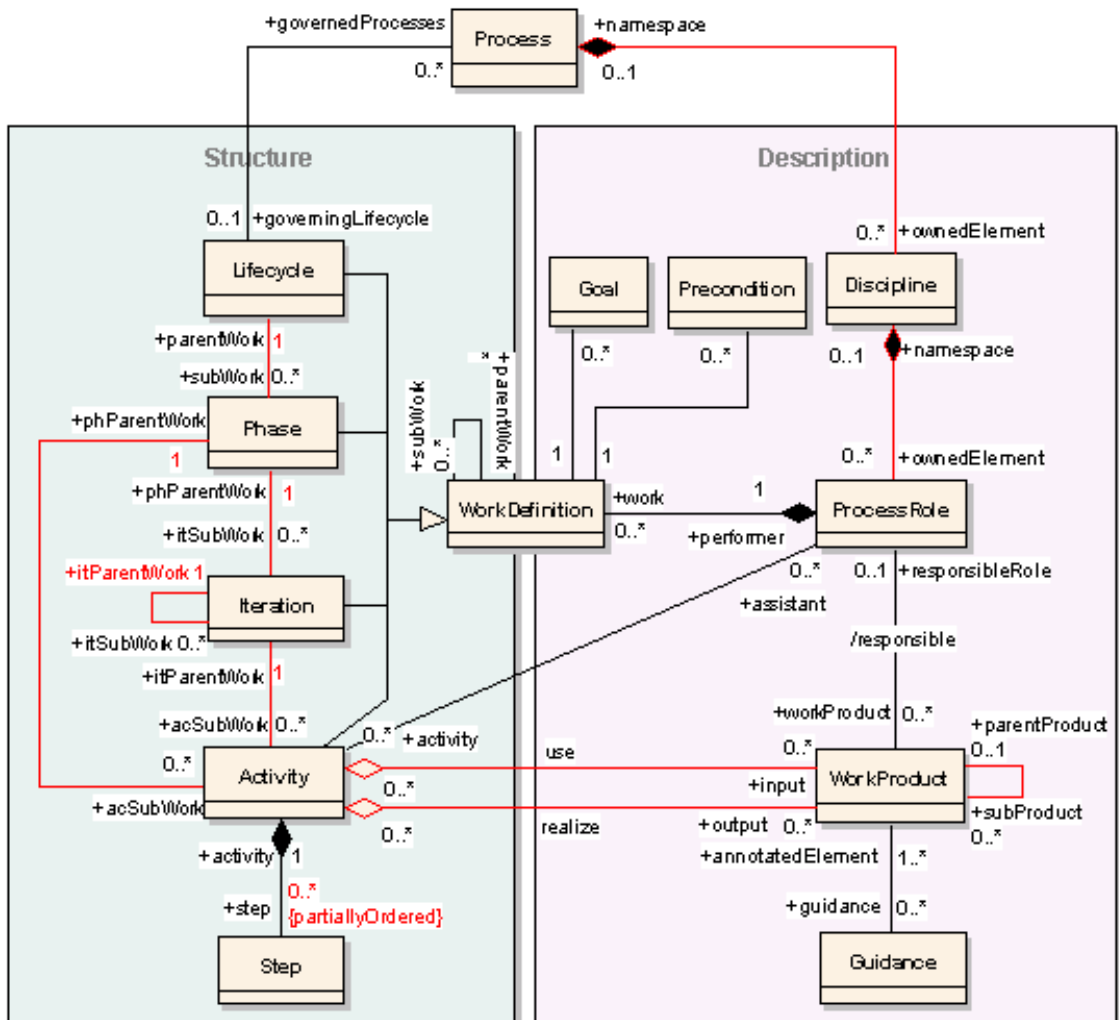


FIG. 4.2 – Spécialisation du méta-modèle SPEM

4.1.3 Description de notre approche

Observons au sein de notre spécialisation que la méta-classe *WorkDefinition* est, comme au sein de la spécification initiale, une classe concrète afin de permettre de l’instancier par des définitions de travail qui ne seront pas encore typées et donc pas encore sémantiquement définies au sein du procédé (e.g. période d’analyse). Par ailleurs, les *WorkDefinition* sont associées au rôle sans contrainte d’ordonnancement car celle-ci est déduite implicitement des pré-conditions et des objectifs de chacun d’eux. Un vérificateur de cohérence de modèle devra toutefois garantir qu’au sein de l’enchaînement des *WorkDefinition*, toute pré-condition est satisfaite par une post-condition et que le procédé ne pourra pas ce trouver en position d’inter-blocage.

Les *Step* pour leur part, n’héritant pas de *WorkDefinition* et n’ayant donc pas de pré-conditions et d’objectifs précis, sont soumis à une contrainte d’ordre que nous avons ajoutée au sein de notre proposition. En effet, il nous semble indispensable d’ordonner les actions au sein d’une activité pour garantir une réelle assistance auprès des acteurs d’un développement. Notons que cet ordre reste toutefois partiel (stéréotype *{partiallyOrdered}*) afin de permettre la réalisation d’actions en parallèle, influant éventuellement l’une sur l’autre.

La notion de *ProcessPerformer* (fig. 2.6), n’ayant pas une sémantique claire au sein du méta-modèle initial de SPEM, a été fusionnée au sein de notre proposition avec celle de *ProcessRole*. Cette fusion nous permet, entre autre, d’apporter une sémantique précise à la notion de *rôle*. Elle permet également de pouvoir attribuer la responsabilité d’un *WorkProduct* à un rôle réalisant n’importe quel type de *WorkDefinition*. La spécification officielle ne permettant que de donner la responsabilité d’un produit à un rôle réalisant une activité.

Par ailleurs, une grande partie des relations que l’on retrouve au sein de notre proposition est reprise du méta-modèle d’origine. Elles sont reprises soit de manière identique soit par spécialisation.

Ainsi, les associations entre les méta-classes *Lifecycle*, *Phase*, *Iteration* et *Activity* correspondent à la redéfinition de la relation réflexive présente sur la méta-classe *WorkDefinition* du méta-modèle d’origine (fig. 2.6). Nous avons contraint la multiplicité de la source de “0..*” par “1” (donnant ainsi la sémantique de la composition). Nous avons toutefois gardé la relation réflexive sur *WorkDefinition* afin de permettre l’instanciation de tout type de composition au moment de la définition du procédé où les Définitions de Travail ne sont pas encore typées. Celle-ci sera toutefois restreinte à l’aide du langage OCL dans la section suivante.

Les relations qui ne sont pas reprises du méta-modèle d’origine de SPEM sont reprises d’UML 1.4 (et plus particulièrement du package Core, Cf. [Obj01], p.22) sur lequel SPEM s’appuie dans sa version 1.1.

Ainsi, les relations de composition que l’on retrouve entre *Process* et *Discipline* et

entre *Discipline* et *ProcessRole* sont un héritage du package “Model_Management” repris au sein de SPEM mais initialement intégré au Core d’UML 1.4.

De la même manière, la relation réflexive sur *WorkProduct* est elle aussi une redéfinition d’une relation héritée de la méta-classe *Classifier* du package *Core* d’UML.

D’autre part les relations d’agrégations entre les méta-classes *Activity* et *WorkProduct* expriment les relations d’utilisation et de réalisation que l’on retrouve au sein du modèle conceptuel de SPEM présenté au chapitre 2 (fig. 2.4). Il s’agit, de manière formelle, d’un héritage du Package *Core* d’UML 1.4 repris de manière simplifiée au sein de notre proposition afin d’assister l’utilisateur dans l’utilisation du langage SPEM. Les relations d’héritage vérifiant notre spécialisation sont mises en exergues au sein de la figure 4.3. Les cardinalités, pour leur part, ont été reprises du méta-modèle initial et permettent de prendre en compte toutes les particularités : produits qui ne sont pas réalisés au cours d’une activité pour les documents initiaux (e.g. le cahier des charges), produits jamais utilisés au sein d’une activité pour les produits finaux ou encore les activités n’utilisant pas ou ne produisant pas de produit.

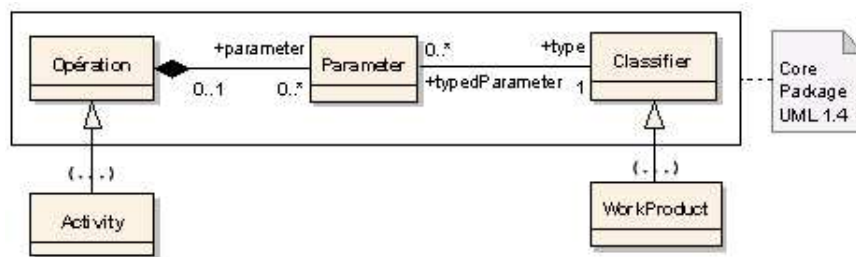


FIG. 4.3 – Héritage du Package Core UML 1.4

Certains problèmes ont toutefois été rencontrés dans l’établissement de ce méta-modèle et particulièrement sur la relation de composition entre *ProcessRole* et *Activity* (issue de la relation de composition entre *ProcessPerformer* et *WorkDefinition* au sein du méta-modèle d’origine et qui exprime la notion de rôle assistant) qui est initialement annotée par une contrainte d’ordre (`{ordered}`). Nous avons décidé de supprimer cette contrainte car nous ne lui trouvons pas de sens précis sachant que les activités sont ordonnées de manière implicite par les préconditions et les objectifs de chacune.

Par ailleurs, le méta-modèle présenté ici n’est que partiel et n’a pas fait l’objet d’une implémentation en tant que profil. Ces points constituent les prochaines perspectives de recherche et sont envisageables avec le logiciel Objecteering/Profile Builder. Ces travaux doivent également faire l’objet d’évolutions suite à diverses applications concrètes pour la spécification de différents procédés de développement.

4.2 Précisions sémantiques à l'aide d'OCL

Pour garantir une certaine cohérence du méta-modèle que nous proposons ci-dessus (fig. 4.2) et préciser la sémantique du langage d'origine, nous avons ajouté des contraintes limitant les instanciations possibles des modèles de procédés. Comme le préconise l'OMG, nous avons utilisé pour cela le langage OCL qui nous a permis d'ajouter au méta-modèle que nous proposons des contraintes dont en voici un extrait¹ :

- Une activité doit être obligatoirement associée à une phase ou une itération mais pas les deux en même temps.

```
context Activity inv :
  self.itParentWork → notEmpty()
  xor
  self.phParentWork → notEmpty()
```

- Une itération doit être obligatoirement associée à une phase ou une itération mais pas les deux en même temps.

```
context Iteration inv :
  self.itParentWork → notEmpty()
  xor
  self.phParentWork → notEmpty()
```

- La réalisation d'une activité ne peut pas être assistée par le rôle qui en a déjà la responsabilité.

```
context Activity inv :
  self.assistant → excludes(self.performer)
```

- Une activité doit utiliser ou réaliser au moins un produit

```
context Activity inv :
  self.input → notEmpty() or self.output → notEmpty()
```

- Un rôle doit être responsable de l'ensemble des produits réalisés par les activités dont il a la charge et réciproquement.

```
context ProcessRole inv :
  let productsActivities : Set{WorkProduct} =
    – Définition de l'ensemble des produits réalisés par les activités
    – dont le rôle (self) est responsable.
    – Pour cela on sélectionne les Activity parmi l'ensemble des WorkDefinition
    – Attention : le cast (oclAsType) est obligatoire pour pouvoir ensuite
    – naviguer à partir de la méta-classe Activity.
```

¹La liste donnée ici n'est pas exhaustive.

```

self.work → select(a : WorkDefinition |
a.oclIsTypeOf(Activity)).oclAsType(Activity).output
→ asSet() – Suppression des doublons
in
self.workProduct = productsActivities

```

- Restriction des compositions possibles des WorkDefinition à partir de la relation reflexive de cette méta-classe. Seules les compositions explicitement décrites au sein du méta-modèle sont possibles.

```

context WorkDefinition inv :
if self.oclIsTypeOf(WorkDefinition) then
– Un WorkDefinition peut être composé que de WorkDefinition.
– Notons que oclIsTypeOf teste le type stricte et retournera faux
– si il s'agit d'un sous-type
self.subWork → forAll(sw : WorkDefinition |
sw.oclIsTypeOf(WorkDefinition))
else
if self.oclIsTypeOf(Lifecycle) then
– Un Lifecycle peut être composé que de Phase.
self.subWork → forAll(sw : WorkDefinition |
sw.oclIsTypeOf(Phase))
else
if self.oclIsTypeOf(Phase) then
– Une Phase peut être composé que d'Iteration ou d'Activity.
self.subWork → forAll(sw : WorkDefinition |
sw.oclIsTypeOf(Iteration) or sw.oclIsTypeOf(Activity))
else
if self.oclIsTypeOf(Iteration) then
– Une Iteration peut être composé que d'Iteration ou d'Activity.
self.subWork → forAll(sw : WorkDefinition |
sw.oclIsTypeOf(Iteration) or sw.oclIsTypeOf(Activity))
else
– Une Activity ne peut pas être composé de WorkDefinition.
self.subWork → isEmpty()
endif
endif
endif
endif
endif

```

L'ensemble des contraintes appliquées au méta-modèle ont été vérifiées et validées statiquement à l'aide de l'outil OCLE. Notre choix s'est tourné vers cet outil car il offre une ergonomie très avancée et permet la vérification de contraintes aussi bien au niveau modèle que méta-modèle.

4.3 Démarche pour la spécification d'un procédé de développement

La création d'un procédé de développement est une tâche complexe et nécessite une méthodologie précise. L'utilisation d'un langage doit donc s'intégrer dans une démarche rigoureuse permettant d'assurer le respect des exigences initiales. On retrouve la même problématique dans l'ingénierie du logiciel où UML doit nécessairement être accompagné d'une démarche comme, par exemple, RUP [KK03] ou MACAO [Cra02].

Le langage SPEM proposé par l'OMG doit donc être associé à une démarche proposant une manière rigoureuse d'établir des procédés formels (dans la limite de l'expressivité du méta-modèle). Il s'agit de la notion de *méta-procédé*. Nous citons par exemple l'environnement RHODES [Cré97] qui est un Atelier de Génie Logiciel centrés Procédé (AGL-P) mis au point au laboratoire de l'ENSEEIH-IRIT. Cet outil permet de décrire et de formaliser les procédés de développement de logiciel selon une méthode précise [TCC⁺03].

L'objectif principal d'un méta-procédé est d'obtenir une formalisation d'un procédé ; formalisation établie au fur et à mesure des étapes de ce méta-procédé. Cette formalisation exprime un procédé de développement complet, rigoureux et offrant la possibilité d'être vérifié. Un tel procédé permet alors la réalisation de logiciels garantissant l'application des règles de génie logiciel (i.e. celles qui sont formalisées dans le procédé de développement).

Les travaux réalisés nous ont permis d'établir un certain nombre de principes, point de départ pour l'établissement futur d'un méta-procédé complet s'appuyant sur SPEM et OCL. Sans offrir un méta-procédé complet, nous proposons donc une démarche de formalisation associée à la spécialisation du méta-modèle SPEM proposée ci-dessus. Cette démarche est dans un premier temps décrite de manière intuitive et sera, dans le chapitre suivant, mise en application pour la formalisation de la méthode MACAO.

Nous proposons de formaliser un procédé selon deux étapes correspondantes à la formalisation des deux vues proposées dans le méta-modèle ci-dessus, i.e. *structurelle* et *descriptive*. Ces deux étapes doivent être réalisées en parallèle de manière à influencer l'une sur l'autre. Par ailleurs, chaque vue nécessite une formalisation à travers deux aspects : *statique* et *dynamique*.

Nous exposons donc dans un premier temps les diagrammes nécessaires à la formalisation des deux vues d'un procédé puis nous proposons une démarche permettant l'établissement progressif de l'ensemble de ces diagrammes.

4.3.1 Structuration du procédé

La structuration du procédé de développement, correspondant à la formalisation de la vue structurelle du procédé, permet de donner une vue d'ensemble du procédé et de le découper de manière hiérarchique. Ce point de vue repose sur le partitionnement des définitions de travail selon des phases, des itérations et des activités rassemblées au sein

d'un cycle de vie associé au procédé.

Il est important d'établir en premier lieu la hiérarchie du procédé de développement. Pour cela, nous proposons l'établissement de diagrammes de package montrant les relations de composition suivantes :

- les phases au sein du cycle de vie,
- les activités (ou itérations) au sein de chaque phase,
- les activités au sein de chaque itération,
- les actions élémentaires au sein de chaque activité.

Cet aspect statique du procédé de développement peut être complété par une vue dynamique. Ce deuxième aspect est optionnel car il sera repris et complété dans la formalisation de la vue descriptive du procédé.

4.3.2 Description du procédé

Cette description, qui correspond à la formalisation de la vue descriptive du procédé de développement, apporte des précisions sur la réalisation des activités d'un procédé de développement : détails des définitions de travail, des rôles et des produits, enchaînements des activités et attributions des responsabilités, utilisations et réalisations des produits, etc.

Ce point de vue repose sur un partitionnement des activités selon des disciplines qui, comme le décrit la spécification, "catégorisent les activités dans un procédé selon un thème commun".

La formalisation de cette vue doit dans un premier temps établir, à travers des diagrammes de package, les relations de composition entre :

- le procédé de développement et les différentes disciplines,
- chaque discipline et les différents rôles.

Les rôles sont par la suite décrits en fonction de leurs responsabilités par rapports aux activités et aux différents produits ; eux même associés à des conseils (guidance). Cette description est faite à l'aide de diagrammes de classe ou de package montrant :

- les relations de composition et les dépendances entre produits et/ou sous-produits ainsi que leurs dépendances avec les conseils,
- les relations (i.e. *perform* ou *assist*) du rôle avec les définitions de travail. Les relations d'assistance et de réalisation peuvent être exprimées dans des diagrammes distincts. On doit également définir les pré-conditions et les objectifs des *WorkDefinition*,
- les relations de dépendances (i.e. utilisation ou production) entre les activités et les produits.

Cet aspect statique doit être complété par un aspect dynamique décrit à la fois par des diagrammes d'activité et des diagrammes d'état-transition.

Les diagrammes d'activité montrent :

- L'enchaînement des définitions de travail qu'un rôle réalise ou assiste,

- L’enchaînement des actions élémentaires au sein d’une activité.

Nous proposons ici d’exprimer explicitement la séquentialité des définitions de travail (*WorkProduct*) qui est pourtant implicitement déduite de leurs préconditions et de leurs objectifs. En effet, ces diagrammes permettent une meilleure compréhension de la part des utilisateurs et permettent également de valider les préconditions et objectifs de chacune des définitions de travail. Un vérificateur de modèles de procédé devra donc offrir une validation de la cohérence de la séquentialité des définitions de travail entre celle qui est exprimée au sein des diagrammes d’activité et celle déduite implicitement.

Les diagrammes d’état-transition montrent les différents états par lesquels passent les produits tout au long de leur élaboration à travers les activités.

4.3.3 Proposition de démarche de spécification

Les précédents conseils sur l’utilisation des différents diagrammes SPEM pour la modélisation d’un procédé n’apportent toutefois pas de démarche méthodologique donnant un cadre rigoureux pour l’utilisation du langage. Nous proposons pour cela un enchaînement de tâches présenté par un diagramme d’activité (Cf. fig. 4.4) qui permet d’établir progressivement les deux vues que nous proposons pour la spécification d’un procédé.

La description d’un procédé se fait à partir d’un ensemble de *WorkDefinition* très général qui a pour objectif de décrire très “grossièrement” les tâches à réaliser au cours du procédé de développement et qui sont définies à partir d’une analyse de l’existant et des besoins. La démarche propose ensuite d’affiner par un processus itératif la structure et la description du procédé de développement. Chaque *WorkDefinition* suit une succession d’étapes au cours desquelles le modèle va se compléter et se préciser :

- *Sélectionner un WorkDefinition* à partir des définitions de travail déterminées initialement ou issues d’une décomposition précédente. Celui-ci va alors suivre séquentiellement les trois étapes suivantes dans un ordre restant toutefois partiel :
 - *Typier le WorkDefinition* consiste à déterminer s’il s’agit d’un cycle de vie (*Lifecycle*), d’une phase (*Phase*), d’une itération (*Iteration*) ou d’une activité (*Activity*) et ainsi placer hiérarchiquement la définition de travail au sein du procédé de développement complet. Le typage est une des tâches les plus importantes de la démarche qui nécessite la maîtrise parfaite de la sémantique de chacun des *WorkDefinition*.
 - *Déterminer le rôle puis la discipline associés au WorkDefinition* afin de l’associer à un domaine de compétence et un thème du procédé de développement. Cette tâche permet d’apporter une description précise pour la réalisation du *WorkDefinition*.
 - *Déterminer les pré-conditions (Precondition) et les objectifs (Goal)* afin de préciser le placement chronologique du *WorkDefinition* au sein du procédé

- complet. La vérification d'un modèle de procédé devra s'attacher à vérifier qu'un cycle ne s'est pas immiscé dans la séquentialité du procédé.
- Ces premières étapes permettent d'apporter une description précise de la définition de travail. Les étapes suivantes dépendent du type défini précédemment :
 - s'il ne s'agit pas d'une Activité, il faut *décomposer la définition de travail* en sous *WorkDefinition*. Les possibilités de décompositions sont restreintes par le méta-modèle proposé ci-dessus.
 - s'il s'agit d'une activité, il faut *déterminer les produits d'entrée et de sortie, les rôles assistant sa réalisation et les étapes élémentaires (Step) qui la composent*. Les produits de sortie sont alors attribués sous la responsabilité du rôle ayant en charge l'activité et l'ensemble des tâches élémentaires sont sous la responsabilité de ce même rôle.

L'ensemble de ce processus est répété pour l'ensemble des *WorkDefinition* déterminés initialement ou au cours des précédentes décompositions.

Nous exploiterons la démarche proposée dans ce travail pour la spécification de la méthode MACAO présentée dans la deuxième partie de ce mémoire. Nous n'avons pas approfondi l'analyse d'un méta-procédé complet car nous avons exploité dans nos travaux la description initiale de la méthode donnée par son créateur [Cra02]. Cette démarche pourrait toutefois servir de point de départ pour l'établissement d'un méta-procédé formel et rigoureux. Un tel méta-procédé pourrait alors être intégré dans un outil d'assistance pour la création de procédé de développement.

À partir du constat que nous avons fait sur la complexité d'utilisation du langage ASPeM, nous avons proposé au sein de cette partie certaines simplifications. Nous apportons tout d'abord une spécialisation du méta-modèle d'origine. Cette simplification, plus directive, apporte plus d'assistance dans la création d'un procédé de développement et garantit la cohérence des modèles engendrés. Nous proposons enfin une formalisation permettant de spécifier de manière cohérente et rigoureuse un procédé de développement complet.

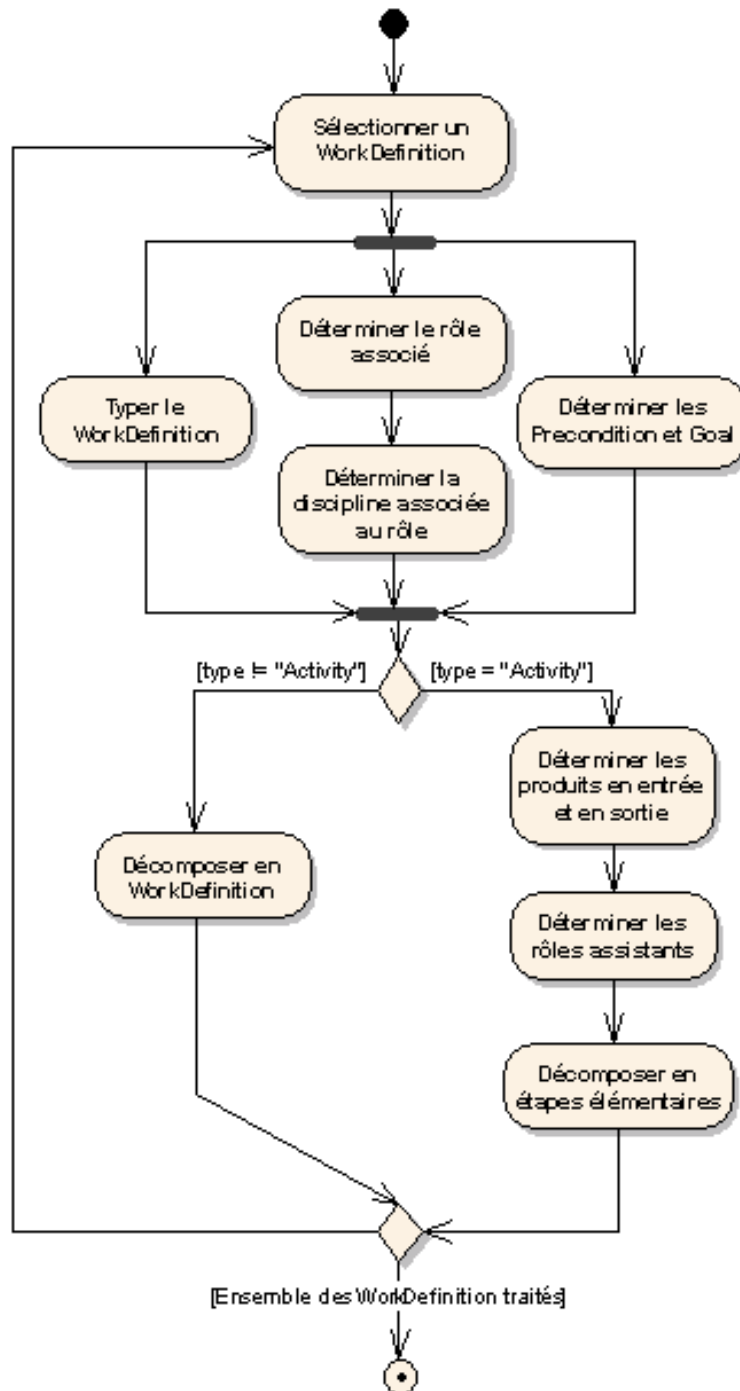


FIG. 4.4 – Démarche pour la spécification d'un procédé

Chapitre 5

Spécification de MACAO

Dans le but de modéliser les logiciels pour leur programmation orientée objet, UML est aujourd'hui devenu un standard en la matière. Cependant, une méthode de conception ne se satisfait pas de simples modèles, mais a besoin d'une démarche qui donne un cadre pour l'analyse et la conception des projets logiciels modernes. La *Méthode d'Analyse et de Conception d'Applications orientées-Objet* (MACAO) [Cra02], que nous avons choisie pour évaluer l'ensemble de nos travaux, est une de ces démarches .

Son interprétation dans le cadre de l'ingénierie des modèles n'est toutefois pas immédiate et a nécessité, avec son créateur M. Crampes, de réaliser des modifications et des ajouts permettant de la rendre compatible avec les directions prises par l'OMG au sein du langage SPEM. Ce chapitre introduit dans un premier temps cette méthode telle qu'elle est décrite au sein du livre [Cra02] puis décrit les choix que nous avons fait pour permettre une modélisation cohérente en SPEM.

5.1 Présentation de la méthode

MACAO est une méthode complète de génie logiciel qui s'appuie sur la notation et les modèles UML. Elle se base sur UP (*Unified Process*) et se place ainsi au même niveau que le RUP (*Rational Unified Process*) proposé par IBM [KK03]. MACAO s'en différencie cependant sur de nombreux points et propose :

- une approche utilisateur proche des méthodes systémiques telle que MERISE, et basée sur les Cas d'Utilisation d'UML,
- une démarche itérative de conception-développement, basée sur la réalisation de prototypes soumis aux utilisateurs pour validation (limitation des risques),
- une règle de non-régression des prototypes afin de limiter les propagations de bugs liés à des modifications intempestives de prototypes déjà recettés,
- une documentation type pour chaque étape du déroulement du projet,
- trois modèles d'IHM¹ : le Schéma Navigationnel d'Interfaces (SNI), le Schéma d'Enchaînement des Fenêtres (SEF) pour les IHM de type Windows et le Schéma

¹Interface Homme-Machine

- d’Enchaînement des Pages (SEP) pour les IHM de type WEB,
- des patrons de conception de l’IHM s’appuyant sur le diagramme des classes métiers.

Les deux derniers points cités, qui concernent la modélisation des IHM, sont étrangement ignorés des langages de modélisation universellement reconnus, tel que UML. Ils revêtent pourtant une importance capitale, car tous les logiciels interactifs imposent la réalisation d’une IHM qui se doit d’être ergonomique et intuitive et qui dicte pratiquement la structure et le comportement du logiciel. La programmation “artisanale” des IHM telle qu’elle est pratiquée aujourd’hui est inacceptable pour le développement d’un logiciel complexe. Les modèles d’IHM proposés par MACAO permettent, d’une part de disposer d’une notation graphique simple pour présenter des solutions embryonnaires aux utilisateurs (en mode esquisse), d’autre part de représenter la totalité de l’IHM, tant sur le plan conceptuel que sur celui de la réalisation. Notons que ces notations ont récemment été formalisées sous la forme de profil UML [CF05] et sont actuellement en cours d’intégration au sein de la plateforme d’accueil *Eclipse*². Leur formalisation permettra d’envisager des générations de code à partir de patron (*pattern*).

Par ailleurs, MACAO offre une méthode de modélisation qui applique le principe “objet-action” permettant, à partir du diagramme de classes métiers et de divers patrons de conception, de produire des structures d’IHM types.

L’ensemble de ces concepts sont définis au sein d’un procédé de développement complet composé de quatre phases successives : l’analyse globale, la conception globale, le développement et la finalisation. Celles-ci sont décrites au sein de la figure 5.1. Notons que la phase de développement est elle-même composée d’un cycle de vie en spirale (fig. 5.2) composé des phases de définition, de conception détaillée, de codage et d’intégration. La phase de *béta*-test se fait pour sa part en parallèle de la phase de définition du prototype suivant. Chaque itération au sein de ce cycle de vie donne lieu à la création d’un prototype opérationnel recetté par le client. Cette démarche permet d’avoir très rapidement un retour de la part des futurs utilisateurs et de pouvoir ainsi rectifier très tôt les erreurs de conception.

La méthode MACAO est actuellement enseignée dans plusieurs IUT³ et diverses formations d’ingénieurs, et commence à être adoptée par certaines SSII⁴. C’est en particulier le cas de Capgemini, qui l’utilise pour mener à bien le projet CARINS⁵, dont le maître d’ouvrage est la Direction des Lanceurs du Centre National d’Etudes Spaciales.

²Eclipse est une plate-forme de développement Java, lancé par IBM. Il permet de télécharger et installer de nombreux plug-ins disponibles sur l’Internet, et de développer facilement les programmes en C#, C++ etc. Eclipse offre une plate-forme de développement extensible, riche, efficace et adaptable. Cf. <http://www.eclipse.org/>

³Institut Universitaire Technologique

⁴Société de Service en Ingénierie Informatique

⁵Calcul de Réseaux en régime INStationnaire

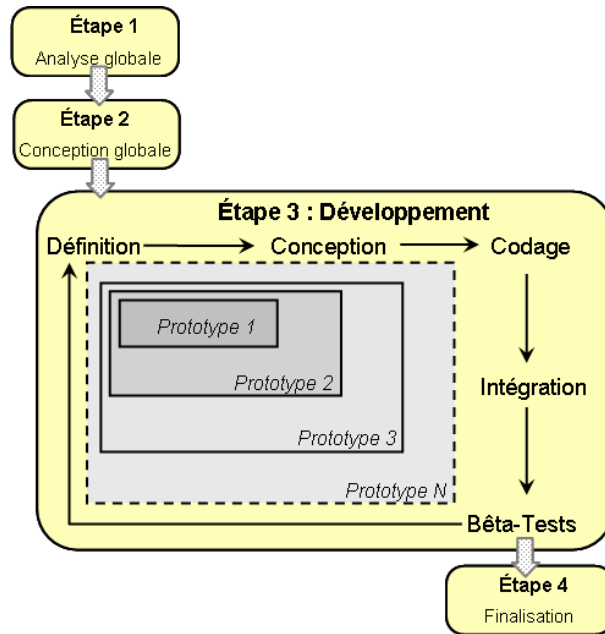


FIG. 5.1 – Procédé de développement MACAO

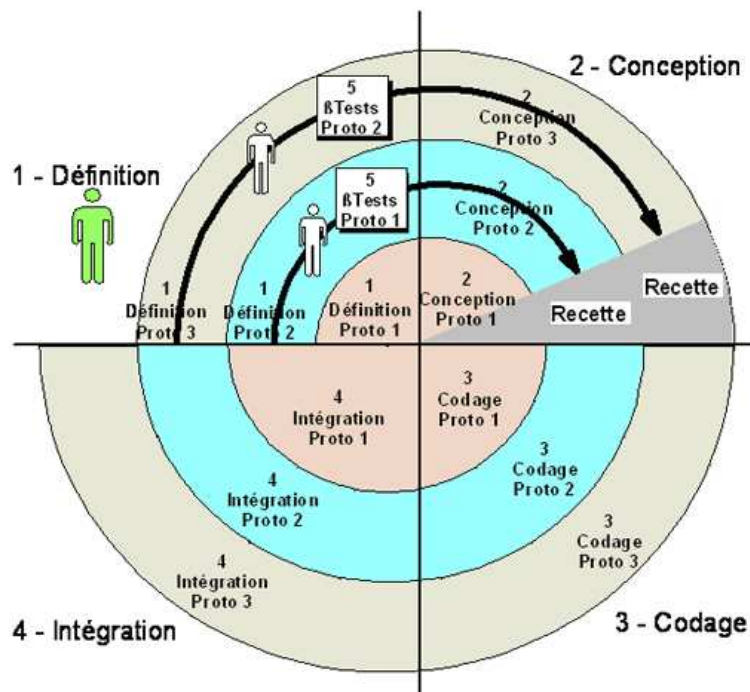


FIG. 5.2 – Phase de développement de MACAO

5.2 Notre approche pour la spécification de MACAO

L'ouvrage de M. Crampes [Cra02] fait explicitement ressortir la structure hiérarchique du procédé. A la simple lecture détaillée du livre, il est donc facile de déterminer l'ensemble des phases, des itérations, des activités et des actions élémentaires qui composent la méthode. Cet ouvrage explicite également l'ensemble des produits utilisés ou réalisés au sein du procédé et fournit de nombreux conseils (*Guidance*) pour la réalisation des activités et des produits.

Toutefois la méthode, parue à la fin de l'année 2002, ne précise pas un certain nombre de concepts nécessaires pour une modélisation cohérente en SPEM. Par exemple les disciplines, utilisées au sein de SPEM pour catégoriser les activités selon un thème commun, ne sont pas présentes au sein de la méthode initiale.

Les travaux de spécification que nous avons entrepris sur la méthode MACAO nous ont donc emmené, en collaboration avec son créateur, à ajouter un certain nombre de détails nécessaires pour pouvoir faire une modélisation rigoureuse, exhaustive et offrant la possibilité d'être intégrée au coeur d'un outil supportant l'approche MDA.

Afin d'explicitier la notion de discipline dans la méthode MACAO, point de départ de la vue descriptive du procédé, nous nous sommes appuyés sur des travaux réalisés au sein du laboratoire ICAR CNR⁶ [CSS03]. Ces travaux proposent d'associer chaque "Phase" à une "Discipline" de même nom.

Ceci ne permet pas une réelle classification des activités en fonction de "thèmes communs" tel que cela est décrit dans la sémantique de SPEM mais permet d'intégrer la notion de discipline au sein d'un procédé ne la possédant pas et ainsi permettre sa modélisation à partir de SPEM. Il s'agit donc plutôt d'un contournement permettant d'utiliser le langage SPEM pour la spécification. Une classification précise en fonction de différents thèmes communs transversaux aux phases pourrait (et devrait) être envisagée pour affiner la méthode MACAO telle qu'elle est décrite initialement.

Le choix que nous avons actuellement fait nous permet donc de modifier le cycle de vie tel qu'il est proposé initialement (fig. 5.1) vers un formalisme plus proche de celui utilisé pour la méthode RUP (fig. 5.3). Celui-ci a bien entendu l'énorme intérêt d'être compatible avec la sémantique et la syntaxe du langage SPEM et ainsi de permettre son utilisation pour la spécification de la méthode.

Ce choix implique un certain nombre de restrictions par rapport à l'établissement de la vue descriptive de la future modélisation du procédé. Tout d'abord, dans le cadre de la simplification que nous proposons du méta-modèle SPEM, les rôles (*Process Role*) composent les disciplines. Le choix que nous avons fait pour la constitution des disciplines nous amène donc à déterminer des rôles obligatoirement intra-phases. Les rôles présents initialement au sein de plusieurs phases devront être décomposés en différents rôles pour chacune des phases, propre à chaque fois à l'espace de nom que constitue la

⁶Istituto di Calcolo e Reti ad Alte Prestazioni del Consiglio Nazionale delle Ricerche (ICAR-CNR), <http://kms.icar.cnr.it/icarkms/>.

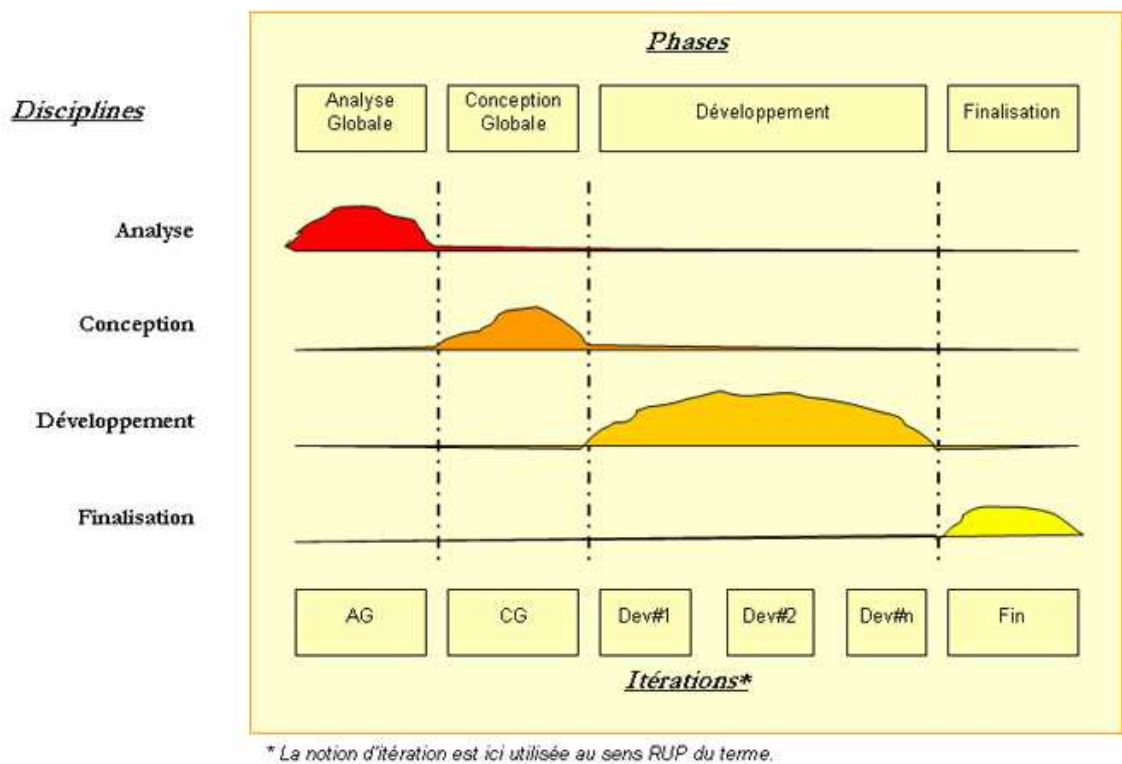


FIG. 5.3 – Représentation de MACAO compatible SPEM

discipline de même nom (*Discipline* héritant de *Namespace* définit sémantiquement un espace de nom ; fig. 4.1).

Un exemple très répandu de rôle inter-phase est celui du chef de projet qui a la responsabilité de valider chacune des phases. Ce rôle va donc être décomposé pour chaque espace de nom (*Discipline*) et nous obtiendrons donc les rôles suivants : *AnalyseGlobale.valideur* ; *ConceptionGlobale.valideur*, etc...

Notons que cette décomposition apparaît d'autant plus naturelle qu'il ne s'agit généralement pas des mêmes compétences qui sont utilisées pour la validation de la phase d'analyse et celle de conception. La différence est encore plus nette pour la validation de la phase de développement qui nécessite des compétences beaucoup plus techniques que pour l'analyse qui nécessite principalement des compétences métiers.

5.3 Modélisation de MACAO selon SPEM

Les travaux de modélisation que nous avons réalisés sur le procédé de développement MACAO ont été faits à l'aide du logiciel Enterprise Architect. Bien qu'il n'offre pas de vérifications sémantiques, nous nous sommes tournés vers cet outil car il nous a permis de respecter la sémantique précise que nous avons déterminé.

L'établissement des modèles s'est déroulé selon la démarche que nous avons proposée dans le chapitre 4 en s'appuyant sur le livre [Cra02] et les choix décrits dans la partie précédente. L'ensemble des travaux de spécifications réalisés sur la méthode MACAO fait actuellement l'objet de la rédaction d'un rapport de recherche complet. Celui-ci n'est pas à ce jour finalisé et nous présentons ci-dessous les grandes orientations que nous avons prises.

5.3.1 Point de vue structurel

Le point de vue structurel de MACAO ressort explicitement de la description initiale de la méthode. Celle-ci se compose d'un cycle de vie qui est découpé chronologiquement en quatre phases (fig. 5.4).

Nous nous sommes plus particulièrement attachés à la phase de développement qui, comme nous l'avons vue sur la figure 5.2, se compose d'une itération répétée autant de fois que de prototypes déterminés dans la phase de conception globale (fig. 5.5).

L'itération *RealisationPrototype* est elle-même composée d'itération en séquence dont la conception détaillée qui regroupe une grande partie des spécificités de la méthode : conception des IHM à différents niveaux d'abstraction, règle de non régression (fig. 5.6), etc.

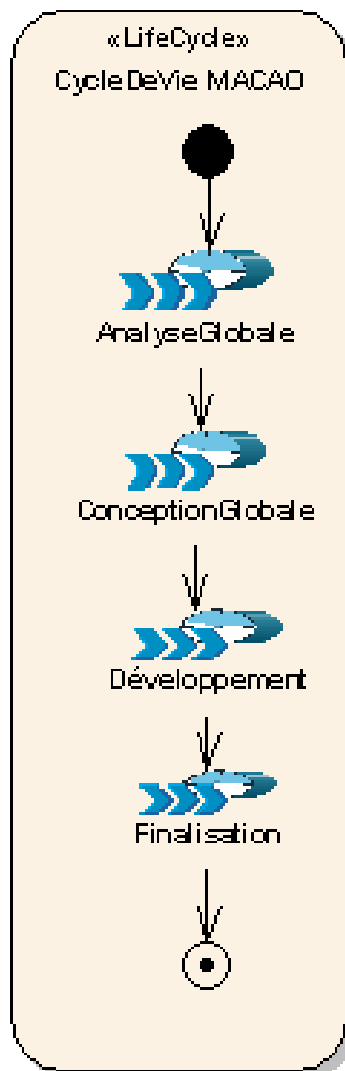


FIG. 5.4 – Cycle de vie de MACAO

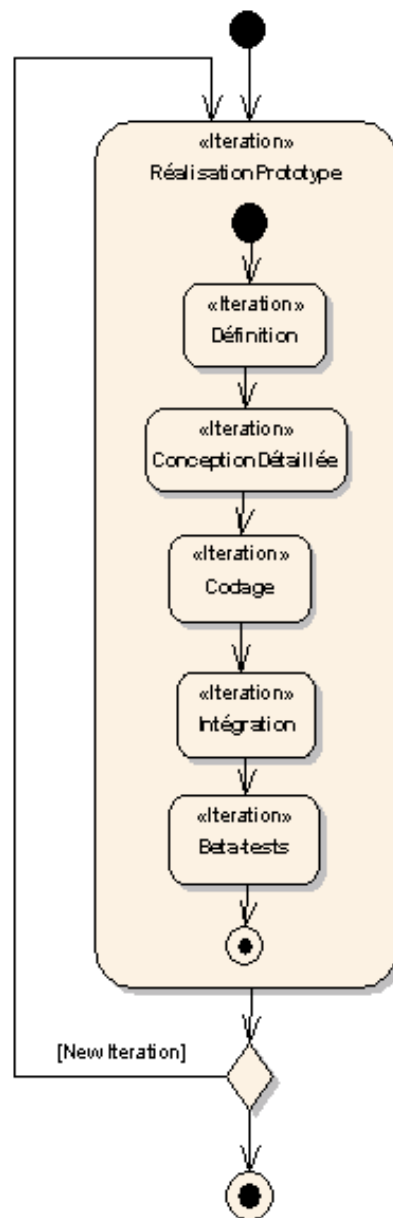


FIG. 5.5 – Phase de développement de MACAO

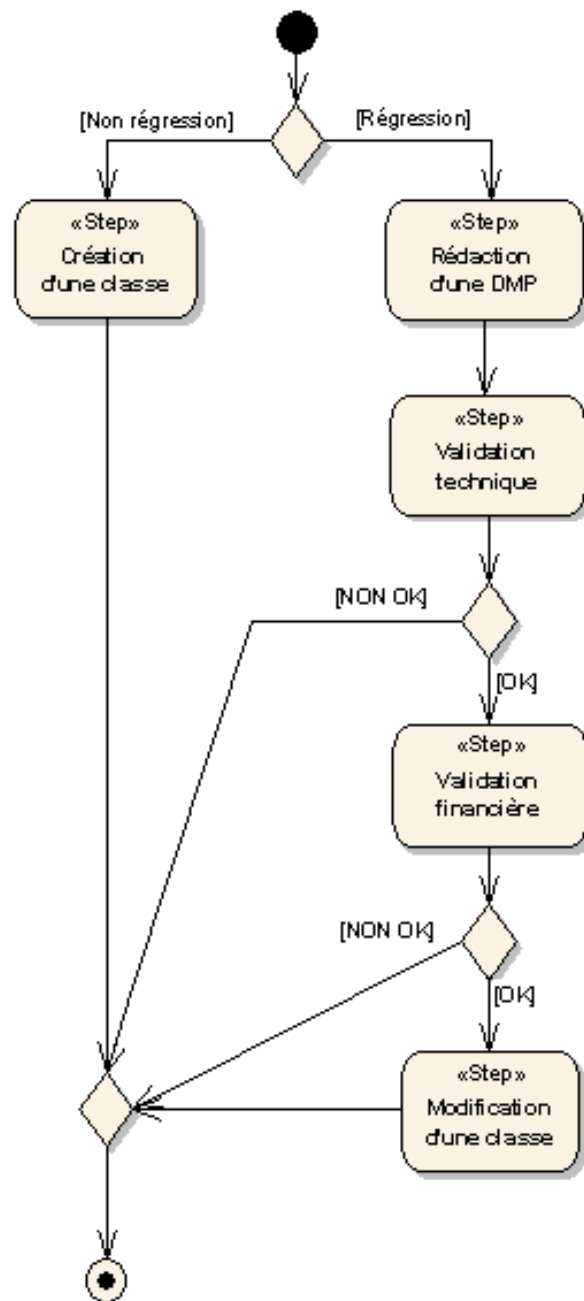


FIG. 5.6 – Règle de non régression de MACAO

5.3.2 Point de vue descriptif

Le point de vue descriptif actuellement modélisé repose sur les choix explicités précédemment : nous avons décrit une discipline pour chacune des phases du cycle de vie (fig. 5.7). Les rôles ont pour leur part été détaillés afin qu'ils soient tous intra-disciplines (Cf. l'exemple de la discipline *Développement*).

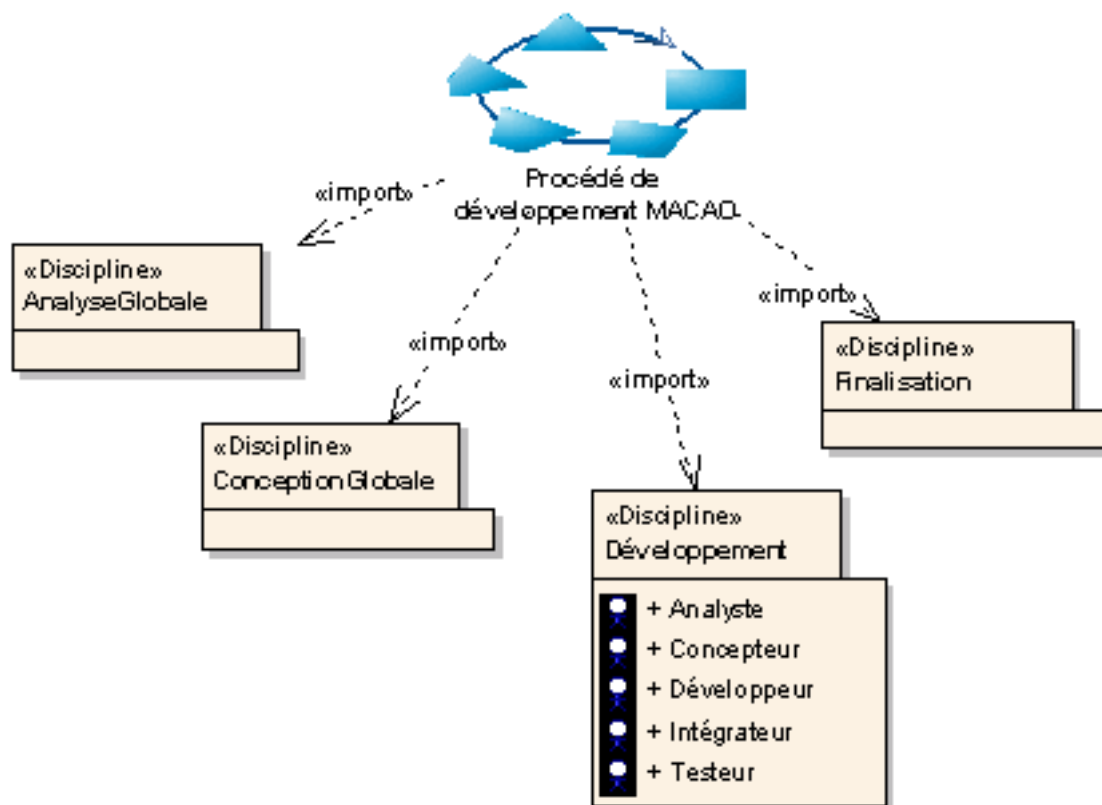


FIG. 5.7 – Disciplines de MACAO

Les précisions sémantiques réalisées avant ces travaux de spécifications nous ont permis d'aborder facilement la méthode MACAO. Nous avons donc réalisé l'ensemble des modèles nécessaires pour une spécification cohérente et rigoureuse. Les futurs travaux s'attacheront à appliquer des règles au niveau des modèles afin de retranscrire précisément les détails présentés informellement dans la description originale de la méthode.

Conclusion

La spécification de procédés par les modèles

Le besoin d'industrialiser les développements de logiciel a montré l'intérêt d'avoir un procédé de développement rigoureux et cohérent. Pour cela, l'ingénierie des modèles, tentant de centraliser la fabrication d'un produit autour de modèles, propose des langages appropriés à ce domaine. Ce mémoire a tenté de montrer le pouvoir expressif à la fois de SPEM, langage de modélisation fourni par l'OMG pour le domaine des procédés, et de celui d'OCL, langage formel d'expression de contraintes. OCL permet de compléter les modèles pour en préciser la sémantique.

Dans l'optique de réaliser des spécifications formelles, pouvant être soumises à des outils de vérification automatique, la complémentarité de ces deux langages apparaît comme indispensable. OCL, langage ne permettant pas en soi de réaliser des spécifications, est indispensable pour donner une sémantique formelle aux modèles et s'orienter de plus en plus vers un fort pouvoir expressif.

Cette complémentarité permet donc de s'approcher de modèles dit "exécutables" permettant à la fois d'être soumis à des vérifications formelles et d'augmenter considérablement la valeur ajoutée des modèles de conception.

L'utilisation de ces langages n'est toutefois pas toujours triviale et la sémantique fournie pour le langage SPEM recèle encore un certain nombre d'incohérences. Nous proposons pour cela une spécialisation du méta-modèle SPEM, plus directive, associée à une sémantique précise et formalisée en OCL. Ces travaux permettent de donner une assistance plus efficace dans l'utilisation du langage mais aussi d'assurer la cohérence (plus particulièrement sémantique) des modèles engendrés.

L'utilisation de SPEM requiert également une démarche, comme dans le domaine du logiciel, pour offrir un cadre méthodologique et rigoureux. Nous proposons donc des conseils pour la formalisation d'un procédé par les modèles.

L'ensemble de ces travaux a été appliqué à la méthode MACAO et a ainsi permis d'en affiner la démarche et d'en fournir une spécification rigoureuse en SPEM et OCL.

Perspectives de Recherche

La formalisation des procédés de développement apparaît comme indispensable pour maîtriser l'industrialisation des développements logiciels et fait actuellement l'objet d'un grand effort au sein de la recherche. Les travaux qui ont été réalisés dans le cadre du stage de Master Recherche s'intègrent dans "l'état de l'art" des techniques et outils actuellement disponibles pour la spécification des procédés et leurs vérifications formelles. Dans un domaine aussi vaste que celui de l'ingénierie des modèles, ce mémoire n'a donc pas pour ambition de fournir des résultats scientifiques définitifs. Ces travaux s'ouvrent vers des perspectives plus larges qui visent à intégrer un procédé de développement au coeur même du fonctionnement d'un Atelier de Génie Logiciel (AGL) et ainsi assister les acteurs du développement tout au long du processus. Cette assistance doit permettre de diriger, d'aider et de vérifier les activités de chaque acteur.

Nous proposons pour cela d'utiliser les résultats de l'ingénierie des modèles et plus particulièrement le langage SPEM dédié à ce domaine et le langage OCL pour l'expression d'une sémantique précise.

Notre étude doit toutefois être complétée d'une classification des différents types de contraintes que l'on peut avoir besoin de spécifier sur un procédé de développement. D'autre part, le nouveau langage "Action Semantic" proposé par l'OMG, qui permet la spécification de traitements évolués (i.e. avec effet de bord), nécessiterait d'être analysé pour évaluer son utilité dans la spécification de procédés.

Par ailleurs, la spécialisation du méta-modèle SPEM proposée au sein de ce mémoire pourrait être implantée, sous forme de profil, sur un outil du type Objecteering/Profile Builder et être complétée afin d'affiner la sémantique du langage. Cette sémantique pourrait, entre autre, être exprimée sous la forme d'expressions OCL. L'implémentation sous forme de profil UML permettrait dans un premier temps d'être plus directif dans l'utilisation du langage SPEM et de donner des précisions sur la sémantique de ses éléments.

Cette assistance ne suffit toutefois pas et il est important d'intégrer le procédé au coeur même des outils de développement. Pour cela, en dehors des problèmes techniques que cela suppose, il est nécessaire d'avoir des procédés formels et cohérents. Nous donnons au cours de ce mémoire quelques pistes sur l'élaboration de tels procédés mais cela nécessiterait la mise au point d'une démarche intégrale venant en complément du langage SPEM : un méta-procédé. Ce méta-procédé devra alors être formalisé à partir des mêmes techniques de spécifications que pour les procédés. L'intégration d'un tel procédé au sein d'un outil permettrait alors de guider de manière efficace les acteurs tout au long du développement.

Ces travaux complémentaires pourront, bien entendu, se poursuivre pour la démarche MACAO afin d'affiner formellement la méthode et de fournir un AGL prenant en compte l'ensemble de la démarche et permettant ainsi d'assister les développeurs dans la réalisation de leurs projets.

Annexe A : Modifications apportées par la version 2.0 d'OCL

La nouvelle version du langage OCL apporte un certain nombre d'évolutions. Nous reprenons ci-dessous les modifications relevées principalement au sein de l'oeuvre de Jos Warmer et de Anneke Kleppe[WK03].

.1 Modifications syntaxiques

- Les anciennes versions permettaient de naviguer au sein des associations en utilisant les rôles ou, dans le cas où ils étaient absents, le nom des classes mais avec une minuscule pour la première lettre. A cause des caractères non-ASCII, il n'est maintenant plus nécessaire d'utiliser une minuscule mais le nom de la classe pleinement qualifié.
- Dans la version 1.1 d'OCL, il n'y avait pas de syntaxe pour déclarer le contexte d'une expression. Ce manque a été comblé dans la version 2.0 par le mot-clé *context*. D'autres mot-clés indiquent comment l'expression doit être évaluée, e.g. *inv* et *derive*,
- La syntaxe `/ * ... * /` est disponible pour l'écriture de commentaires,
- L'utilisation des attributs et opérations de classe doit être préfixée du nom de la classe selon la syntaxe `className :: staticAttribut`,
- Changement de syntaxe pour la spécification des énumérations.

.2 Nouveaux types

- *orderedSet* : Ensemble (au sens mathématiques) dont le contenu est trié,
- *tuples* : Structure pouvant être composée de plusieurs valeurs de types différents (Cf. [Obj03b], §7.5.15),
- *oclUndefined* : Type conforme avec tous les autres types et qui peut avoir comme seule valeur `oclVoid` (Cf. [Obj03b], §11.2.3).

.3 Opérations supplémentaires sur les collections

De nombreuses opérations ont été ajoutées au sein de la nouvelle version du langage : *collectNested*, *flatten*, *excludes*, *excludesAll*, *insertAt*, *indexOf*, *any*, *one*, *isUnique* et *sortedBy*.

.4 Nouvelle option dans les post-conditions

Possibilité de spécifier au sein des post-conditions le fait qu'un objet doit envoyer un message à un autre objet. Différentes variantes existent pour cela : *isSignalSent*, *isOperationCall*, *hasReturned* et *result*.

Syntaxe : *calledobject* \wedge *calledOperations*(*params*)

.5 Autres changements

- En plus de la spécification des contraintes, OCL a été étendu à la spécification de valeurs en général (valeur initiale, corps d'opération et règle de dérivation),
- Contrairement à la version 1.x ; tout aspect prédéfini est considéré comme un comportement sous forme d'opérations et ne peut être décrit en langage naturel en utilisant des parenthèses. Ceci permet de renforcer le caractère formel du langage et les possibilités de vérification automatique et exhaustive des contraintes,
- L'état d'un objet peut être exprimé et testé à partir de l'opération *oclInState()*,
- On peut explicitement énoncer le type des éléments d'une collection,
- Ajout du constructeur *let* permettant récursivement de définir des variables locales dans les contraintes,
- L'opération *oclType* pour *oclAny* a été supprimée car il peut y avoir plusieurs types,
- Possibilité de faire des packages d'expressions,
- Possibilité d'utiliser des opérateurs infixés,
- Changement et extension des règles de précédences.

Bibliographie

- [ABB⁺02] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system : Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.
- [BB01a] E Breton and J Bézivin. Process-centered model engineering. In *5th IEEE International Enterprise Distributed Object Computing Conference*. IEEE Computer Society, Seattle, Washington, USA, 2001. URL <http://www.sciences.univ-nantes.fr/lina/atl/www/papers/edoc.pdf>.
- [BB01b] Erwan Breton and Jean Bézivin. Using meta-model technologies to organize functionalities for active system schemes. In *Workshop of Ontologies in Agent Systems (OAS 2001), 5th International Conference on Autonomous Agents*. Montreal, Canada, 2001. URL <http://www.sciences.univ-nantes.fr/lina/atl/www/papers/oas.pdf>.
- [BBDV03] Jean Bézivin, Erwan Breton, Grégoire Dup, and Patrick Valduriez. *The ATL Transformation-based Model Management Framework*. IRIN, Université de Nantes, 2003. URL <http://www.sciences.univ-nantes.fr/info/recherche/Vie/RR/RR-IRIN2003-0%8.pdf>.
- [BBLC⁺03] Pierre Bazex, Jean-Paul Bodeveix, Christophe Le Camus, Thierry Millan, and Christian Percebois. Vérification de modèles UML fondée sur OCL. In *INFORSID, Nancy*, pages 185–200. Inforsid, 20, rue Axel Duboul - 31000 TOULOUSE, 3-6 juin 2003.
- [Béz03a] Jean Bézivin. *La transformation de modèles*. Equipe ATLAS (INRIA & IRIN), Nantes, 2003. URL <http://aristote1.aristote.asso.fr/Presentations/CEA-EDF-2003/Cours/Jea%nBezivin/Cours06.v1-01.ppt>. Ecole d’Eté d’Informatique CEA EDF INRIA 2003, cours #6.
- [Béz03b] Jean Bézivin. Les modèles de processus. Technical report, CEA EDF INRIA, juin 2003. URL <http://aristote1.aristote.asso.fr/Presentations/CEA-EDF-2003/Cours/Jea%nBezivin/IndexJeanBezivin.html>. Ecole d’Eté d’Informatique - Présentation.

- [Béz04] Jean Bézivin. Sur les principes de base de l'ingénierie des modèles. *Où sont les objets ?*, pages 145–156, octobre 2004. URL <http://www.sciences.univ-nantes.fr/lina/at1/www/papers/L0bjet2004.pdf>.
- [BGG04] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL traps in the UML 2.0 superstructure an experience report. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 188–196. Springer, 2004.
- [Bla05] Xavier Blanc. *MDA en action*. Eyrolles edition, mars 2005. ISBN 2-212-11539-3. 269 p.
- [BMP⁺02] Jean-Paul Bodeveix, Thierry Millan, Christian Percebois, Christophe Le Camus, Pierre Bazex, and Louis Feraud. Extending OCL for verifying UML models consistency. In Ludwik Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar, editors, *Blekinge Institute of Technology, Research Report 2002 :06. UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop Materials*, pages 75–90. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2002.
- [Cap04] Alain Caplain. Checking software development processes. In *IFIP Student Forum*, pages 113–122, 2004.
- [CBMC03] Juan-Carlos Cruellas, Jean-Paul Bodeveix, Thierry Millan, and Agusti Canals. The NEPTUNE Technology to verify and to Document Software components. In Franck Barbier, editor, *Business Component-Based Software Engineering*, pages 101–118. Kluwer Academic Publishers, Post Office Box 322 - 330 AH Dordrecht - THE NETHERLANDS, 2003.
- [CF05] Jean Bernard Crampes and Nicolas Ferry. Modélisation des ihm : le SNI, un diagramme d'activités stéréotypées. *Actes du congrès IHM'05*, juin 2005.
- [CJV04] F. Chauvel, J.-M. Jézéquel, and D. Vojtisek. Validation dynamique de modèles uml avec points de variation sémantique. *Génie Logiciel*, 69 :24–30, juin 2004.
- [CL04] Agusti Canals and Michel Lagreca. La vérification des modèles : Où en est-on? *Génie Logiciel*, 69 :2–8, juin 2004.
- [Cod72] E. F. Codd. Relational completeness of data base sublanguages. In : R. Rustin (ed.) : *Database Systems : 65-98*, Prentice Hall and IBM Research Report RJ 987, San Jose, California, 1972.
- [Cra02] Jean-Bernard Crampes. *Méthode orientée-objet intégrale MACAO, Démarche participative pour l'analyse, la conception et la réalisation de logiciels*. Génie logiciel. Ellipse edition, décembre 2002. ISBN 2-7298-1424-8. 308 p.
- [Cré97] Xavier Crégut. *Un environnement d'assistance rigoureuse pour la description et l'exécution de processus de conception - Application à l'approche*

- objet*. Thèse de Doctorat, Institut National Polytechnique de Toulouse, juillet 1997. 245 p.
- [CSS03] Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. SPEM description of the PASSI process rel. 0.3.6. Technical report, ICAR CNR, décembre 2003. 55 p.
- [Fin00] Franck Finger. *Design and Implementation of Modular OCL Compiler*. Thèse de Doctorat, Dresden University of Technology, mars 2000. URL <http://www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf>. 121 p.
- [Fla03] Stephan Flake. Temporal OCL extensions for specification of real-time constraints (position paper). In *UML 2003 Workshop "Specification and Validation of UML Models for Real Time and Embedded Systems (SVERTS'03)"*, octobre 2003.
- [GBR03] Martin Gogolla, Jörn Bohling, and Mark Richters. Validation of UML and OCL models by automatic snapshot generation. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 265–279. Springer, 2003.
- [GHJ95] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
- [GLR⁺02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation : The missing link of mda. DSTC, octobre 2002. URL <http://www.dstc.edu.au/Research/Projects/Pegamento/publications/icgt20%02.pdf>.
- [GMSB96] Marie-Claude Gaudel, Bruno Marre, Francoise Schlienger, and Gilles Bernot. *Précis de génie logiciel*. Masson edition, 1996. ISBN 2-225-85189-1. 135 p.
- [Gro02] QVT-Merge Group. Revised submission for MOF 2.0 Query/Views/Transformations. Technical report, avril 2002. URL <http://www.omg.org/cgi-bin/apps/doc?ad/04-04-01.pdf>.
- [GSUW94] Ashish Gupta, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint checking with partial information. In *PODS '94 : Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 45–55. ACM Press, New York, NY, USA, 1994. ISBN 0-89791-642-5.
- [HZ04] Heinrich Hussmann and Steffen Zschaler. The object constraint language for UML 2.0 - overview and assessment. *UPGRADE*, V(2) :25–28, avril 2004. URL <http://www.upgrade-cepis.org/issues/2004/2/up5-2Hussmann.pdf>.
- [KK03] Per Kroll and Philippe Kruchten. *Guide pratique du RUP*. Pearson education edition, 2003. ISBN 2-7440-1629-2.

- [KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003. ISBN 0-321-19442-X.
- [Lar05] Craig Larman. *UML 2 et les design patterns*. Pearson education edition, mars 2005. ISBN 2-7440-7090-4. 850 p.
- [MC99] Luis Mandel and María Victoria Cengarle. On the expressive power of OCL. pages 854–874, février 1999. ISBN 3-540-66587-0.
- [MG03] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML, 2^e ed.* Eyrolles edition, décembre 2003. ISBN 2-212-11397-8. 514 p.
- [MM03] Joaquin Miller and Jishnu Mukerji. *Model Driven Architecture (MDA) 1.0.1 Guide*. Object Management Group, Inc., juin 2003. URL <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [Mon00] Jean-François Monin. *Introduction aux méthodes formelles, 2^e ed.* Hermès science publications edition, juin 2000. ISBN 2-7462-0140-2. 351 p.
- [Obj97] Object Management Group, Inc. *Object Constraint Language (OCL) 1.1 Specification*, septembre 1997. URL <http://www.omg.org/docs/ad/97-08-08.pdf>. 32 p. - formal/97-08-08.
- [Obj01] Object Management Group, Inc. *Unified Modeling Language (UML) 1.4 Specification*, septembre 2001. URL <http://www.omg.org/docs/formal/01-09-67.pdf>. Final Adopted Specification.
- [Obj02a] Object Management Group, Inc. *Meta Object Facility (MOF) 1.4 Specification*, avril 2002. URL <http://www.omg.org/docs/formal/02-04-03.pdf>. Final Adopted Specification.
- [Obj02b] Object Management Group, Inc. *Request for Proposal : MOF 2.0 Query / Views / Transformations RFP*, avril 2002. URL <http://www.omg.org/docs/ad/02-04-10.pdf>.
- [Obj02c] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 1.0 Specification*, novembre 2002. URL <http://www.omg.org/docs/formal/03-12-02.pdf>. formal/02-11-14.
- [Obj03a] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, octobre 2003. URL <http://www.omg.org/docs/formal/02-04-03.pdf>. Final Adopted Specification.
- [Obj03b] Object Management Group, Inc. *UML Object Constraint Language (OCL) 2.0 Specification*, octobre 2003. URL <http://www.omg.org/docs/ptc/03-10-14.pdf>. Final Adopted Specification.
- [Obj04a] Object Management Group, Inc. *Request for Proposal : MOF Model to Text Transformation Language RFP*, avril 2004. URL <http://www.omg.org/docs/ad/04-04-07.pdf>.
- [Obj04b] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 2.0 RFP*, novembre 2004. URL <http://www.omg.org/docs/ad/04-11-04.pdf>. ad/04-11-04.

- [Obj05] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 1.1 Specification*, janvier 2005. URL <http://www.omg.org/docs/formal/05-01-06.pdf>. formal/05-01-06.
- [Ozk86] Esen Ozkarahan. *Database machines and database management*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-196031-8.
- [QP03] QVT-Partner. Revised submission for MOF 2.0 Query/Views/Transformations v1.1. Technical report, août 2003. URL <http://qvtp.org/downloads/1.1/qvtppartners1.1.pdf>.
- [RG99] Mark Richters and Martin Gogolla. A metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *UML '99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *LNCS*, pages 156–171. Springer, 1999.
- [RG00] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.
- [RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *UML 2.0 - Manuel de Référence*. Génie logiciel. Paris, campuspress référence edition, décembre 2004. ISBN 2-7440-1820-1. 800 p.
- [SCH02] Wuwei Shen, Kevin Compton, and James Huggins. A toolset for supporting uml static and dynamic model checking. In *COMPSAC '02 : Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life : Development and Redevelopment*, pages 147–152. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1727-7.
- [SKM01] Timm Schafer, Alexander Knapp, and Stephan Merz. Model checking uml state machines and collaborations. *Electr. Notes Theor. Comput. Sci.*, 55 (3), 2001.
- [TCC+03] Hanh Nhi Tran, Bernard Coulette, Xavier Crégut, Dong Thi Bich Thuy, and Tran Dan Thu. Modélisation du méta-procédé RHODES avec SPEM. In *RIVF*, pages 239–246, 2003. URL <http://e-ifi.org/rivf/2003/proceedings/p239-246.pdf>.
- [TPC] François Taiani, Mario Paludetto, and Thierry Cros. Model checking and object orientation : A tool overview.
- [WK99] Jos Warmer and Anneke Kleppe. *OCL : The Constraint Language of the UML*. mai 1999.
- [WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0-321-17936-6.

- [ZG02] Paul Ziemann and Martin Gogolla. An extension of OCL with temporal logic. pages 53–62, 2002. URL www4.in.tum.de/~csdum102/22.ps.
- [ZG03a] Paul Ziemann and Martin Gogolla. An OCL extension for formulating temporal constraints. (1/03), 2003.
- [ZG03b] Paul Ziemann and Martin Gogolla. Validating OCL specifications with the USE tool : An example based on the bart case study. *Electr. Notes Theor. Comput. Sci.*, 80, 2003. URL <http://www1.elsevier.com/gej-ng/31/29/23/137/23/show/Products/notes/in%dex.htm#012>.
- [Zyt02] Matthias Zytnecki. *Présentation de SPEM*. Rapport de stage, août 2002. 48 p.

Table des figures

1.1	Pile de modélisation de l'OMG	8
1.2	Notions de base en technologie des objets	8
1.3	Notions de base en ingénierie des modèles	8
1.4	Organisation 3+1 de la pile de modélisation	9
1.5	Transformation de modèles dans l'approche MDA	13
2.1	SPEM comme méta-modèle MOF	18
2.2	SPEM comme profil UML	19
2.3	Modèle conceptuel de SPEM	20
2.4	Réification du modèle conceptuel de SPEM	20
2.5	Packages SPEM	21
2.6	<i>Package Process Structure</i>	24
2.7	<i>Package Process Lifecycle</i>	24
2.8	Architecture générale de l'outil Objecteering/UML	25
3.1	Diagramme de classe de l'arborescence d'un système de fichier	32
3.2	Contrainte OCL sur un diagramme de classe	32
3.3	Collections en OCL 2.0	33
3.4	Différence en OCL avec des bag	34
3.5	Classe temporaire pour le Produit Cartésien	35
3.6	Produit Cartésien en OCL 1.x	35
3.7	Super type commun	39
4.1	Packages Model_Management de SPEM	44
4.2	Spécialisation du méta-modèle SPEM	45
4.3	Héritage du Package Core UML 1.4	47
4.4	Démarche pour la spécification d'un procédé	54
5.1	Procédé de développement MACAO	57
5.2	Phase de développement de MACAO	57
5.3	Représentation de MACAO compatible SPEM	59
5.4	Cycle de vie de MACAO	61
5.5	Phase de développement de MACAO	62
5.6	Règle de non regression de MACAO	63
5.7	Disciplines de MACAO	64

Résumé

Le besoin de plus en plus présent d'industrialiser les développements de logiciels a fait ressortir la nécessité d'avoir un procédé rigoureux et outillé afin d'assister les acteurs du développement. Pour outiller un procédé et réaliser des vérifications à la fois statiques et dynamiques, il est nécessaire d'en établir une formalisation à travers des standards établis à cet effet.

C'est à cette problématique que tente de répondre l'ingénierie des modèles en proposant le langage de modélisation SPEM. Ces travaux n'ont toutefois pas réussi à établir une sémantique précise à ce langage et aucune démarche n'a été proposée pour l'établissement d'un procédé (i.e. *méta-procédé*).

Nous essayons donc d'apporter une assistance efficace dans la spécification et la vérification de modèles de procédé décrits en SPEM et complétés d'expressions formelles en OCL. Nous offrons pour cela une spécialisation plus directive du méta-modèle d'origine associée à une sémantique plus précise et à des conseils pour une formalisation rigoureuse et cohérente. Une application de ces travaux pour la spécification et la vérification de la méthode MACAO termine ce mémoire.

Mots-clés : Ingénierie des modèles, Méta-modélisation, Procédé de développement, Méta-procédé, Vérification de modèle, SPEM, OCL, MACAO.

Abstract

The increasingly present need to industrialize software developments has emphasized the need for having a rigorous and equipped process in order to assist the actors of development. To provide aids for performing a process and to carry out static and dynamic checking at the same time, it is necessary to establish its formalization through standards established for this purpose.

It is with these problems that model engineering tries to answer by proposing the SPEM modeling language. This work however has not succeeded in establishing a precise semantics with this language and no step has been proposed to set up a process (i.e. metaprocess).

We try to bring an effective assistance in the specification and the checking of process models described in SPEM and supplemented with formal expression in OCL. To do so, we offer a more directing specialization of the original metamodel associated with a more precise semantics and advice for a rigorous and coherent formalization. An application of this work for the specification and checking of the MACAO method finishes this report.

Keywords : Model engineering, Metamodel, Software Process, Metaprocess, Model Checking, SPEM, OCL, MACAO.