HAL
open science

Spem4mde : un métamodèle et un environnement pour la modélisation et la mise en œuvre assistée de processus IDM

Samba Diaw

To cite this version:

Samba Diaw. Spem4mde : un métamodèle et un environnement pour la modélisation et la mise en œuvre assistée de processus IDM. Autre [cs.OH]. Université Toulouse le Mirail - Toulouse II; Université de Ziguinchor, 2011. Français. NNT : 2011TOU20148 . tel-00668956

HAL Id: tel-00668956

https://theses.hal.science/tel-00668956v1

Submitted on 10 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Toulouse

Université de Ziguinchor

THESE EN COTUTELLE

Soutenue en vue de l'obtention du titre de

DOCTEUR EN INFORMATIQUE

De l'Université de Toulouse
et de l'Université de Ziguinchor

SPEM4MDE : Un métamodèle et un environnement pour la modélisation et la mise en œuvre assistée de processus IDM

Présentée et soutenue publiquement par

SAMBA DIAW

Le 28 Septembre 2011 à l'Université de Toulouse

Devant le jury composé des membres suivants :

Rapporteurs :

Franck **BARBIER**	Professeur à l'Université de Pau et des Pays de l'Adour
Jacky **ESTUBLIER**	Directeur de Recherche CNRS à l'Unversité de Grenoble

Examinateurs :

Bernard **COULETTE** (Directeur)	Professeur à l'Université de Toulouse II-Le Mirail
Salomon **SAMBOU** (Co-directeur)	Professeur à l'Université de Ziguinchor, Sénégal
Redouane **LBATH** (Encadrant)	Maître de Conférences à l'Université de Toulouse I-Capitole
Moussa **LO**	Maître de Conférences à l'Université Gaston Berger, Sénégal
Amadou **COULIBALY**	Maître de Conférences à l'INSA de Strasbourg

École Doctorale MITT
« Mathématiques Informatique et Télécommunications de Toulouse »
Laboratoire : Institut de Recherche en Informatique de Toulouse (IRIT, UMR 5505)

A ma mère que la mort a arrachée trop tôt à notre affection.

A mon épouse pour sa patience.

A mon père qui m'a appris, par la parole et par l'exemple que le travail ennoblit l'honneur.

A mes frères et sœurs.

A mes collègues-enseignants de l'Université de Ziguinchor

A tous les amis qui pendant des moments d'indécision et de doute, allaient me prendre en haute mer et, vague par vague, me ramenaient au rivage de la confiance et de l'espoir.

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés dans le cadre d'une cotutelle entre l'Université de Ziguinchor au Sénégal et l'Université Toulouse II Le Mirail en France. Mes remerciements vont tout d'abord à la coopération française qui a financé cette thèse, contribuant ainsi à la réalisation de mes travaux de recherche dans des conditions très favorables. Je remercie également l'EGIDE pour une bonne gestion du financement qui m'a été octroyé.

Je remercie sincèrement les Professeurs Franck BARBIER et Jacky ESTUBLIER qui ont accepté de juger ce travail et d'en être les rapporteurs. Je les remercie pour leur lecture minutieuse et leurs commentaires enrichissants. Je remercie également les Professeurs Moussa LO et Amadou COULIBALY d'avoir accepté de participer à l'examen de cette thèse.

Mes remerciements vont aussi à l'endroit de mon directeur de thèse Bernard COULETTE pour son soutien, sa disponibilité, sa patience, la collaboration étroite dans laquelle nous avons travaillé et son aide qui m'ont permis de mener à bien ces travaux. Merci également pour ses relectures minutieuses. Je remercie également mon co-directeur de thèse Salomon SAMBOU pour son assistance, ses conseils précieux, et son soutien indéfectible.

Je tiens également à témoigner ma profonde reconnaissance à mon encadrant Redouane LBATH. Je tiens aussi à lui assurer ma profonde gratitude pour m'avoir initié à la recherche.

Mes remerciements vont également à tous les membres de l'équipe MACAO pour leur accueil et la bonne ambiance dans laquelle ils m'ont permis de travailler. Dans cette liste je ne peux m'empêcher de citer Youness, Eric, Rhama, Hakim, Adel pour leur affection et leur amitié dévouée qui m'ont profondément touché.

Ces quatre années resteront à jamais gravées dans ma mémoire et constituent sans nul doute une expérience très enrichissante de ma vie. Ce mémoire est l'aboutissement d'un long travail qui ne pourrait être possible sans l'aide et le soutien indéfectible de ma famille. Je remercie au passage mon épouse pour sa patience et sa compréhension.

Je ne saurais oublier dans ces remerciements le Recteur de l'Université de Ziguinchor, le Professeur Oumar Sock, et l'ancien Recteur et actuel Ministre de l'enseignement supérieur sénégalais, le Professeur Amadou Tidiane Ba qui à l'époque avait ratifié la convention de cotutelle.

Enfin merci à ceux et celles qui de près ou de loin m'ont permis d'arriver à bout dans ce long périple.

Résumé

L'ingénierie dirigée par les modèles (IDM) connue sous le terme MDE (Model-Driven Engineering) en anglais est une discipline récente du génie logiciel qui recommande l'utilisation intensive des modèles et des transformations de modèles au cœur du processus de développement logiciel. Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artéfacts de développement et doivent en contrepartie être suffisamment précis et riches afin de pouvoir être interprétés ou transformés par des outils. L'avènement de l'IDM a suscité beaucoup d'intérêt de la part des organisations qui de fait commencent à transformer leur processus de développement traditionnel en un processus de développement dirigé par les modèles – appelé aussi *processus IDM*, ce dernier étant vu comme un enchaînement de transformations de modèles, chacune consommant un ou plusieurs modèles sources et produisant un ou plusieurs modèles cibles.

Au moment où les processus IDM commencent à émerger, nous notons l'absence d'un langage dédié pour les modéliser et les mettre en œuvre. L'intérêt de la modélisation des processus en général et des processus IDM en particulier est d'utiliser une terminologie unifiée et cohérente, afin de permettre d'une part une communication plus efficace entre les développeurs et d'autre part un meilleur suivi d'un projet de développement. La modélisation de processus permet aussi de gérer, de faire évoluer et de réutiliser efficacement les modèles de processus. L'intérêt de la mise en œuvre des processus est de permettre un meilleur guidage du développement, une vérification des contraintes des activités/transformations et la gestion de la cohérence des artéfacts/modèles du développement.

Le standard SPEM 2.0 dédié à la modélisation des processus propose des concepts génériques qui sont supposés être capables de décrire tout type de processus logiciel ou système, incluant les processus IDM. Cependant, les concepts de SPEM ne capturent pas la nature exacte des processus IDM. Les concepts de modèles, de métamodèles, de transformation de modèles et de leurs diverses relations ne sont pas explicitement pris en compte. D'autre part, une autre insuffisance majeure de SPEM réside dans le fait qu'il n'intègre pas les concepts relatifs à la mise en œuvre des processus. En effet, la spécification de SPEM affirme clairement que cette préoccupation n'est pas dans le champ d'intérêt de SPEM et propose d'exécuter les modèles de processus dans un formalisme externe basé soit sur les diagrammes d'activités UML 2.0, soit sur les machines à états d'UML, soit sur la notation BPMN (Business Process Modeling Notation).

L'objectif de cette thèse est triple : (1) proposer une extension de SPEM dans laquelle les concepts centraux des processus IDM sont réifiés ; (2) proposer un langage dédié à la modélisation comportementale des processus IDM ; (3) proposer une architecture conceptuelle d'un environnement logiciel d'aide à la modélisation et à la mise en œuvre des processus IDM. L'intérêt de la réification des concepts IDM est de permettre d'une part aux concepteurs de processus d'expliciter les aspects spécifiques au développement IDM, d'autre part de mieux assurer la cohérence des modèles produits par les transformations. Par exemple pour vérifier la relation de conformité entre un modèle et son métamodèle, il est important de spécifier les modèles et les métamodèles qui participent à une transformation.

Pour répondre aux deux premiers objectifs, nous avons défini un langage de modélisation et de mise en œuvre des processus IDM spécifié formellement sous forme d'un métamodèle appelé

SPEM4MDE. Le métamodèle SPEM4MDE étend certains concepts de SPEM par des concepts relatifs au développement IDM (transformation, modèle, métamodèle, outil IDM). La réutilisation de SPEM 2.0 a pour principal intérêt de favoriser l'alignement de SPEM4MDE avec ce standard et de permettre ainsi une meilleure diffusion des concepts de SPEM4MDE. Pour réduire la complexité de SPEM4MDE, nous n'avons réutilisé que le paquetage *Process Structure* de SPEM 2.0 qui fournit les concepts basiques pour décrire la partie structurelle d'un processus. Pour décrire le comportement des processus IDM, SPEM4MDE réutilise le paquetage d'UML 2.2 Superstructure qui décrit les machines à états d'UML. Une machine à états est composée d'états définissant l'ensemble des états d'un élément d'un processus IDM et de transitions définissant les opérateurs de mise en œuvre spécifiques à cet élément. SPEM4MDE réutilise aussi le métamodèle QVT afin de spécifier le comportement d'une transformation, l'intérêt étant de tirer pleinement profit des outils d'implémentation et d'exécution liés à ce standard.

Pour répondre au troisième objectif, nous avons élaboré une architecture conceptuelle d'un environnement fondé sur une démarche en trois étapes.

La première étape a pour but de décrire le modèle structurel et le modèle comportemental d'un processus IDM, conformément au métamodèle SPEM4MDE. Par la suite, ces deux modèles sont validés sur la base des contraintes OCL spécifiées dans le métamodèle SPEM4MDE.

La deuxième étape a pour but de réutiliser et/ou d'adapter les modèles définis dans l'étape précédentes aux spécificités d'un projet de développement déterminé et d'assigner les ressources nécessaires à la mise en œuvre (développeurs, outils, espaces de travail, …).

La troisième et dernière étape de cette démarche est la mise en œuvre du modèle de processus adapté. Elle est réalisée par les développeurs du projet en utilisant les outils IDM mis à leur disposition. Les développeurs sont assistés dans leurs tâches par un environnement de mise en œuvre qui s'appuie sur les modèles comportementaux du processus. Cette étape produit les livrables du projet (code, modèles, documentation, etc.).

Pour valider notre approche, un prototype a été développé sous l'environnement TOPCASED. Ce prototype fournit d'une part un éditeur graphique pour la modélisation structurelle et comportementale des processus IDM et d'autre part un environnement de mise en œuvre s'appuyant sur les modèles comportementaux des processus. Nous avons également appliqué notre approche à une étude de cas significatif: le processus UWE (UML-based Web Engineering), qui est un processus IDM dédié au développement d'applications web.

Mots clés: Ingénierie Dirigée par les Modèles (IDM), Transformation de Modèles, Langage de Modélisation de Processus (LMP), Développement dirigé par les modèles, Processus IDM, Processus UWE, Atelier de Génie Logiciel centré-Processus (AGLP), TOPCASED

Abstract

MDE (Model-Driven Engineering) is a recent software engineering discipline that advocates the use of models and transformations in the heart of software development. In this new perspective, models take an important place among software development artifacts and must be formal in order to be understood or transformed by tools. With the emergence of MDE, many organizations have been starting to transform their traditional software development processes into model-driven processes. Kleppe and al. define a model-driven software development as "*a process of developing software using different models on different levels of abstraction with (automated) transformations between these models*".

While model-driven development processes – called MDE processes – have started to appear, a tool-supported Process Modeling Language (PML) for describing and enacting such processes is still lacking. One of the major advantages of software process modeling is to help developers using a unified and consistent terminology in order to communicate around the process. Software process modeling should also make possible understanding, reuse, evolution, management, and standardization of processes. One of the major advantages of software process enactment is to allow best guidance of development, checking of activities' and transformations' constraints, and managing consistency of development artifacts/models.

The concepts of SPEM 2.0 are quite generic since they are supposed to allow describing any kind of software (and even system) process including MDE ones. However, SPEM 2.0 concepts do not succeed in capturing the exact nature of most activities and artifacts of model-driven development. Indeed, most of model-driven development activities are model transformations, and most of model-driven development artifacts are models. The concepts of model, metamodel, model transformations, and theirs diverse relationships are not explicitly taken into account. In addition, SPEM 2.0 does not fulfill executability. Indeed, the specification of SPEM 2.0 claims that this concern is out of the scope of SPEM, and advocates to enact SPEM process models by using an external behavior such as Activity diagrams, UML state-machines, or BPMN (Business Process Modeling Notation).

The objective of this thesis is threefold: (1) provide an extension of SPEM that reifies the MDE concepts; (2) provide a language dedicated to behavioral modeling of MDE processes; (3) provide a conceptual architecture of a PSEE (Process-centered Software Engineering Environment) that guides process designer at modeling phase and developers at enactment time. One of the major advantages of reification of MDE concepts is to allow in one hand process designers to explicitly describe specific aspects of MDE development, in the other hand to ensure the consistency between models produced by transformations. For instance, to make it possible to check the conformance relationship, it is important to specify models and metamodels as input/output parameters of transformations.

To fulfill the first two objectives, we have defined a MDE software processes modeling and enactment language formally specified as a metamodel called SPEM4MDE. SPEM4MDE extends SPEM 2.0 by adding concepts related to MDE development (transformation, model, metamodel, MDE tool. To reduce complexity of SPEM4MDE, we only reuse in SPEM 2.0 the *Process Structure* package, which provides basic concepts for describing structural aspects of processes. To describe the behavior of MDE processes, we reuse UML state-machines. SPEM4MDE also reuses the QVT standard in order to specify the behavior of model transformations; the underlying objective is to benefit from existing tools that implement this standard.

To fulfill the third objective, we have designed a conceptual architecture of a PSEE based on a methodology for modeling and enactment that comprises three stages. The first stage consists in describing structural and behavioral models, which are compliant with SPEM4MDE. Afterwards, these models are validated with respect to the OCL constraints defined in the SPEM4MDE metamodel.

The second stage consists in reusing and/or adapting models defined in the precedent stage to a specific project and assigning necessary resources for enactment (developers, tools, workspaces, etc.).

The third and last stage of this methodology is the enactment of the adapted process model. This stage is realized by developers that use MDE tools. Developers are assisted in their tasks by the PSEE on the basis of process behavior models. This stage produces the deliverables of the project (code, models, documentation, etc.).

To validate our approach, a prototype of this PSEE is developed under the TOPCASED environment. This prototype provides a graphical editor for structural and behavioral modeling of MDE processes, and a process enactment engine based on process behavior models. We have also applied our approach to a significant case study: the UWE (UML-based Web Engineering) process, which is a MDE process dedicated to web applications development.

Keywords: Model-Driven Engineering (MDE), Model Transformations, Process Modeling Language (PML), Model-Driven Development (MDD), MDE Software Process, UWE Process, Process-centered Software Engineering Environment (PSEE), TOPCASED

Table des Matières

CHAPITRE I. Introduction générale 18

I.1. Problématique 18
I.2. Objectifs de la thèse 20
I.3. Approche adoptée 20
I.4. Contenu du Mémoire 21

CHAPITRE II. État de l'art 23

II.1. Introduction 23
II.2. L'Ingénierie des Processus Logiciels (IPL) 24
II.2.1. Concepts et Définitions 24
II.2.2. Objectifs de la modélisation des processus logiciels 25
II.2.3. Les Langages de Modélisation de Processus logiciels (LMP) 26
II.2.4. Mise en œuvre des processus logiciels 27
II.2.5. Les Ateliers de Génie logiciel centrés-Processus (AGL-P) 27
II.3. L'Ingénierie Dirigée par les Modèles (IDM) 29
II.3.1. Genèse de l'IDM 30
II.3.1.1. Model-Integrated Computing 30
II.3.1.2. Les usines logicielles (Software Factories) 31
II.3.2. Les bases de l'IDM 31
II.3.2.1. Concept et Définitions 32
II.3.2.2. Les approches de l'OMG 33
II.3.2.2.1. Meta-Object Faclity (MOF) 34
II.3.2.2.2. Object Constraint Language (OCL) 34
II.3.2.2.3. Software & Systems Process Engineering Metamodel (SPEM 2.0) 34
II.3.2.2.4. L'approche MDA (Model-Driven Architecture) 36
II.3.3. Transformation de modèles 38
II.3.3.1. Principales approches de transformations de modèles 38
II.3.3.1.1. L'approche par programmation 39
II.3.3.1.2. L'approche par template 39
II.3.3.1.3. L'approche par modélisation 39
II.3.3.1.4. Synthèse des approches de transformations de modèles 40
II.3.3.2. Typologie et propriétés des transformations 40
II.3.3.3. Les langages/outils dédiés aux transformations de modèles 42
II.3.3.3.1. Le standard QVT (Query, Views, Transformation) 42
II.3.3.3.2. Le Langage ATL 44
II.3.3.4. Les outils génériques 45
II.3.3.5. Les outils intégrés aux AGL 46
II.3.3.5.1. Objecteering MDA Modeler 46
II.3.3.5.2. IBM Rational Software Modeler (RSM) 47
II.3.3.5.3. FUJABA 47
II.3.3.5.4. Synthèse 48
II.3.3.6. Les outils de métamodélisation 48
II.3.3.6.1. Kermeta 48
II.3.3.6.2. XMF-Mosaic 49
II.3.3.6.3. EMF 49
II.3.3.6.4. TOPCASED 50
II.3.3.6.5. Synthèse 50
II.3.3.7. Traçabilité des transformations 50
II.3.4. Discussion 51
II.4. Exemples de Processus IDM 52
II.4.1. Le Processus UWE dédié aux applications web 52
II.4.2. Processus IDM dédié aux services d'architectures (Middleware Services) 53

II.4.3. Processus IDM pour le développement de systèmes d'apprentissage en ligne 55
II.4.4. Processus VUML pour la composition de modèles structurels 55
II.4.5. Le Processus MoPCoM 57
II.5. Les Langages de description de processus IDM 58
II.5.1. L'approche de Maciel 58
II.5.2. L'approche de Porres 60
II.5.3. Synthèse 62
II.6. Bilan des approches étudiées 62
II.7. Conclusion 64

CHAPITRE III. Le métamodèle SPEM4MDE 65

III.1. Introduction 65
III.2. Approche de construction du métamodèle SPEM4MDE 65
III.3. Structuration en paquetages du métamodèle SPEM4MDE 66
III.4. Concepts de base de SPEM4MDE 67
III.5. Volet structurel de SPEM4MDE 68
III.5.1. Le paquetage Process Structure de SPEM 2.0 68
III.5.2. Le paquetage MDE Process Structure 69
III.5.2.1. BreakdownElement (from SPEM 2.0 Process Structure) 70
III.5.2.2. Activity (from SPEM 2.0 Process Structure) 70
III.5.2.3. InitialActivity 71
III.5.2.4. FinalActivity 71
III.5.2.5. Model 72
III.5.2.6. ModelParameter 73
III.5.2.7. TransformationDefinition 73
III.5.2.8. InformalRule 75
III.5.3. Le paquetage Model Relationship 75
III.5.3.1. Model 75
III.5.3.2. ModelRelationship 76
III.5.3.3. Composition 76
III.5.3.4. Refinement 76
III.5.3.5. Trace 77
III.5.3.6. TraceElement 77
III.5.3.7. ModelElement 78
III.5.4. Exemple illustratif du volet structurel 78
III.6. Volet Comportemental de SPEM4MDE 79
III.6.1. Le paquetage BehaviorStateMachines d'UML 2. 79
III.6.2. Les paquetages de MOF 2.0 QVT 79
III.6.3. Le paquetage MDE Process Behavior 81
III.6.3.1. BreakdownElement (from SPEM 2.0 Process Structure) 82
III.6.3.2. ImplementationParameter 82
III.6.3.3. TransformationImpl 82
III.6.3.4. MDETool 84
III.6.3.5. Transformation (from QVTBase) 84
III.6.3.6. ProgramBasedTransformation 85
III.6.3.7. TemplateBasedTransformation 85
III.6.3.8. HumanActor 85
III.6.4. Comportements par défaut fournis par SPEM4MDE 86
III.6.4.1. Comportement d'une transformation 86
III.6.4.2. Comportement d'une activité (Activity) 87
III.6.4.3. Comportement d'un modèle créé par une activité/transformation 89
III.6.4.4. Comportement d'un modèle modifié par une activité/transformation 90
III.6.4.5. Comportement d'un outil MDE requis pour une activité/ transformation 91
III.6.4.6. Comportement d'un rôle participant à une activité/transformation 92
III.6.5. Exemple illustratif du volet comportemental 93
III.6.5.1. Description de l'implémentation de la transformation 93
III.6.5.2. Description des modèles comportementaux 96
III.7. Conclusion 96

CHAPITRE IV. SPEM4MDE-PSEE : Une implémentation de SPEM4MDE..... 98

IV.1. Introduction.....98
IV.2. La plate-forme TOPCASED.....98
IV.2.1. Vue d'ensemble de TOPCASED.....98
IV.2.2. Fonctionnalités utilisées dans TOPCASED.....99
IV.3. Architecture de SPEM4MDE-PSEE.....100
IV.4. Description fonctionnelle de SPEM4MDE-PSEE.....101
IV.4.1. Description fonctionnelle de « SPEM4MDE Process Editor » 101
IV.4.2. Description fonctionnelle de « SPEM4MDE Process Enactment Engine » 102
IV.5. Implémentation du prototype SPEM4MDE-PSEE 104
IV.5.1. Implémentation du module « SPEM4MDE Process Editor ».....104
IV.5.1.1. Les diagrammes de l'éditeur graphique de SPEM4MDE 104
IV.5.1.2. Processus de réalisation de l'éditeur graphique de SPEM4MDE 105
IV.5.1.2.1. L'activité « Configure GenModel » 106
IV.5.1.2.2. L'activité « Generate EMF Tree Structure Editor ».....106
IV.5.1.2.3. L'activité « Configure SPEM4MDE Graphical Editor » 106
IV.5.1.2.4. L'activité « Generate SPEM4MDE Graphical Editor ».....108
IV.5.1.2.5. L'activité « Refine SPEM4MDE Process Editor Code ».....111
IV.5.1.2.6. L'activité « Run SPEM4MDE Process Editor Code » 111
IV.5.2. Implémentation du module « SPEM4MDE Process Enactment Engine » 114
IV.5.2.1. Illustration du module « SPEM4MDE Process Enactment Engine ».....114
IV.5.2.2. Mécanisme d'écoute et de traitement des changements d'états 116
IV.6. Conclusion 117

CHAPITRE V. Étude de cas : Modélisation et Mise en œuvre du Processus UWE . 119

V.1. Introduction.....119
V.2. Démarche de modélisation et de mise en œuvre de processus avec SPEM4MDE 119
V.2.1. Description informelle de la démarche 119
V.2.2. Formalisation de cette démarche avec SPEM4MDE.....121
V.3. Description informelle du processus UWE.....122
V.3.1. Les modèles du processus UWE.....122
V.3.2. Les transformations du processus UWE 123
V.4. Application de la démarche au processus UWE 124
V.4.1. Description structurelle du processus UWE 124
V.4.1.1. La transformation « Requirements 2 Architecture » 126
V.4.1.2. Les transformations entre les modèles fonctionnels 126
V.4.1.2.1. La transformation « Requirements 2 Content » 127
V.4.1.2.2. La transformation « Content 2 Navigation » 128
V.4.1.2.3. La transformation « Requirements 2 Navigation ».....129
V.4.1.2.4. Raffiner le modèle de navigation (« Navigation Refinement »).....130
V.4.1.2.5. La transformation « Navigation 2 Presentation » 130
V.4.1.2.6. Application des styles sur le modèle de présentation (« Style Adjustment »).....131
V.4.1.3. La transformation « Functional 2 BigPicture » 131
V.4.1.4. La transformation « Architecture Integration » 132
V.4.1.5. La génération des PSM (« Integration 2 J2EE » et « Integration 2 .NET ») 132
V.4.2. Description comportementale du processus UWE 133
V.4.2.1. Comportement de l'activité « Describe Requirements Model ».....133
V.4.2.2. Comportement d'une transformation du processus UWE 134
V.4.2.3. Comportement d'un modèle créé ou modifié par une activité/transformation du processus UWE 135
V.4.2.4. Comportements d'un outil et d'un rôle du processus UWE 135
V.4.3. Adaptation du modèle de processus UWE.....136
V.4.3.1. Adapter l'activité « Describe Requirements Model » 136
V.4.3.2. Adapter la transformation « *Requirements 2 Content* » 137
V.4.3.3. Adapter la transformation « *Content 2 Navigation* ».....137
V.4.3.4. Adapter la transformation « *Navigation 2 Presentation* ».....138

V.4.4. Mise en œuvre du modèle de processus UWE 139
V.4.4.1. Mise en œuvre de l'activité « *Describe Requirements Model* » 139
V.4.4.2. Mise en œuvre de la transformation « *Requirements 2 Content* » 144
V.4.4.3. Mise en œuvre de la transformation « Content 2 Navigation » 148
V.4.4.4. Mise en œuvre de la transformation « *Navigation 2 Presentation* » 149
V.5. Conclusion 151

CHAPITRE VI. Conclusion générale et perspectives 152

VI.1. Rappel de la problématique et des objectifs de la thèse 152
VI.2. Contribution du travail de thèse 152
VI.3. Limites de notre approche et perspectives 153

Liste des publications 155

Bibliographie 156

Annexe A : Le paquetage Process Structure de SPEM 2.0 162

Annexe B : Le paquetage BehaviorStateMachines d'UML 171

Annexe C: Le Paquetage QVTBase de MOF 2.0 QVT 178

Annexe D: Les Métamodèles du Processus UWE 182

Table des figures

Figure II.1 Principes de fonctionnement d'un AGL-P [Dowson et al, 1994]..... 28
Figure II.2 Notions de base en ingénierie des modèles 33
Figure II.3 Pyramide de modélisation de l'OMG [Bézivin, 2003a] 33
Figure II.4 Structure du métamodèle de SPEM 2.0 [OMG-SPEM 2.0, 2008] 36
Figure II.5 Principes du processus MDA [Combemale, 2008] 38
Figure II.6 Exemple d'application de l'approche par modélisation [Blanc, 2005] 40
Figure II.7 Typologie des transformations de modèles [Combemale, 2008] 41
Figure II.8 Architecture du standard QVT [OMG-QVT, 2008] 43
Figure II.9 Extrait du métamodèle ATL [Bézivin, 2003b] 45
Figure II.10 Principes de MDA Modeler et UML Modeler [Softeam, 2008] 47
Figure II.11 Vue d'ensemble du processus UWE [Koch, 2006] 53
Figure II.12 Processus dédié aux services d'architectures [Maciel et al., 2006]. 54
Figure II.13 Le processus IDM dédié à la plate-forme E-learning [Cong et al., 2010]..... 55
Figure II.14 Processus de composition structurelle dans VUML [Anwar et al., 2010] 56
Figure II.15 Vue d'ensemble du processus MoPCoM [Koudri, 2010]..... 58
Figure II.16 Le métamodèle de l'approche de Maciel [Maciel et al., 2009] 59
Figure II.17 Vue d'ensemble de l'environnement Transforms [Maciel et al., 2009]..... 60
Figure II.18 Le métamodèle de l'approche de Porres [Porres et al, 2006] 61
Figure III.1. Organisation en paquetages du métamodèle SPEM4MDE..... 67
Figure III.2. Modèle conceptuel de l'approche adoptée dans SPEM4MDE 68
Figure III.3. Paquetage MDE Process Structure..... 70
Figure III.4. Paquetage Model Relationship..... 75
Figure III.5. Exemple d'une définition de transformation en SPEM4MDE..... 78
Figure III.6 La structuration en paquetages de MOF 2.0 QVT [OMG-QVT, 2008]..... 80
Figure III.7. Paquetage MDE Process Behavior..... 81
Figure III.8. Comportement par défaut d'une transformation 87
Figure III.9. Comportement par défaut d'une activité..... 88
Figure III.10. Comportement par défaut d'un modèle créé par une activité/transformation 89
Figure III.11. Comportement par défaut d'un modèle modifié par une activité/transformation..... 91
Figure III.12. Comportement par défaut d'un outil MDE 92
Figure III.13. Comportement par défaut d'un rôle 93
Figure III.14 Exemple d'une implémentation de transformation en SPEM4MDE 94
Figure III.15 Version simplifiée du métamodèle « Class Diag Metamodel » 94
Figure III.16 Version simplifiée du métamodèle « Data Base Metamodel » 95
Figure IV.1 Architecture de la plate-forme TOPCASED [Farail et al., 2005] 99
Figure IV.2. Architecture de SPEM4MDE-PSEE 101
Figure IV.3 Diagramme de cas d'utilisation du module « *SPEM4MDE Process Editor* »..... 102
Figure IV.4 Diagramme de cas d'utilisation relatif au Chef de Projet 103
Figure IV.5 Diagramme de cas d'utilisation relatif au développeur 103
Figure IV.6 Relations entre les diagrammes de l'éditeur et les paquetages de SPEM4MDE 104
Figure IV.7. Diagramme structurel décrivant le processus de réalisation de l'éditeur graphique de SPEM4MDE 105
Figure IV.8 Les modèles de configuration de l'éditeur « *SPEM4MDE Process Editor* » 107
Figure IV.9 Les paramètres de « *SPEM4MDE.*editorconfigurator»..... 107
Figure IV.10 Modèle de configuration du diagramme « *Model Relationship* »..... 108
Figure IV.11 Architecture logicielle de « *SPEM4MDE Process Editor* » 109
Figure IV.12 Fichiers contenant les classes Java du composant « *Model Relationship »*..... 110
Figure IV.13 Fichiers contenant les classes Java du composant « *Properties view* » 110
Figure IV.14 Vue graphique du composant « MDE Process Structure » 112
Figure IV.15 Interface graphique de spécification des contraintes d'une activité/transformation..... 112
Figure IV.16 Vue graphique du composant « Model Relationship » 113
Figure IV.17 Vue graphique du composant « MDE Process Behavior » 113

Figure IV.18 Les opérateurs de mise en œuvre d'une transformation 114
Figure IV.19 Etats des opérateurs de mise en œuvre après l'exécution de l'opérateur « *instantiate MDE Process* » 115
Figure IV.20 Etats des opérateurs de mise en œuvre après l'exécution de l'opérateur « *run* » 115
Figure IV.21 Mécanisme d'écoute et de traitement d'un événement de changement d'état..... 116
Figure V.1 Démarche générale de modélisation et de mise en œuvre dans SPEM4MDE..... 120
Figure V.2 Démarche de modélisation et de mise en œuvre avec SPEM4MDE 121
Figure V.3 Description structurelle du Processus UWE 125
Figure V.4 Les transformations des modèles fonctionnels du processus UWE..... 125
Figure V.5 Relations entre la transformation « *Requirements 2 Architecture* » et sa définition..... 126
Figure V.6 Enchaînement des transformations des modèles fonctionnels 127
Figure V.7 Relations entre la transformation « *Requirements 2 Content* » et sa définition 128
Figure V.8 La transformation « *Content 2 Navigation* » et sa définition 129
Figure V.9 La transformation « *Requirements 2 Navigation* » et sa définition..... 129
Figure V.10 La transformation « *Navigation Refinement* » et sa définition..... 130
Figure V.11 La transformation « *Navigation* 2 *Presentation* » et sa définition 130
Figure V.12 La transformation « *Style Adjustment* » et sa définition..... 131
Figure V.13 La transformation « *Functional2BigPicture* » et sa définition..... 131
Figure V.14 La transformation « *Architecture Integration* » et sa définition 132
Figure V.15 La transformation « *Integration 2 J2EE* » et sa définition..... 132
Figure V.16 La transformation « *Integration 2 .NET* » et sa définition 133
Figure V.17. Rappel du comportement de l'activité « *Describe Requirements Model* »..... 134
Figure V.18. Rappel du comportement d'une transformation du processus UWE 134
Figure V.19 Adaptation de l'activité « *Describe Requirements Model* »..... 136
Figure V.20 Adaptation de la transformation « *Requirements 2 Content* » 137
Figure V.21 Adaptation de la transformation « *Content 2 Navigation* »..... 138
Figure V.22 Adaptation de la transformation « *Navigation 2 Presentation* »..... 138
Figure V.23 Opérateurs de mise en œuvre de l'activité « *Describe Requirements Model* »..... 140
Figure V.24 État des opérateurs de mise en œuvre si l'outil *ArgoUWE* est ouvert..... 141
Figure V.25 État des opérateurs de mise en œuvre de l'activité « *Describe Requirements Model* » après son lancement 141
Figure V.26 Modèle des exigences produit par l'activité « *Describe Requirements Model* » 142
Figure V.27 État des opérateurs de mise en œuvre après le passage du modèle « *Mail Portal Requirements Model* » à l'état « *InitialVersion* » 142
Figure V.28 État des opérateurs de mise en œuvre après le passage de l'outil *ArgoUWE* à l'état « *Used* »..... 143
Figure V.29 État des opérateurs après le passage du modèle produit à l'état « *FinalVersion* » 143
Figure V.30 Opérateurs de mise en œuvre de la transformation « *Requirements 2 Content* » 144
Figure V.31 Opérateurs de mise en œuvre de l'outil *MediniQVT*..... 144
Figure V.32 État des opérateurs de mise en œuvre de « *Requirements 2 Content* » après son passage à l'état « *Startable* »..... 145
Figure V.33 État des opérateurs de l'outil « *MediniQVT »* après son passage à l'état « *Opened* » 145
Figure V.34 État des opérateurs de mise en œuvre après l'exécution de l'opérateur « *run* ». 146
Figure V.35 Modèle de contenu produit par la transformation « *Requirements 2 Content* »..... 146
Figure V.36 État des opérateurs du modèle du contenu après son passage à l'état « *InitialVersion* » 147
Figure V.37 État des opérateurs après le passage de l'outil à l'état « *Used* ». 147
Figure V.38 État des opérateurs du modèle du contenu après son passage à l'état « *FinalVersion »*. 148
Figure V.39 Modèle de navigation produit par la transformation « *Content 2 Navigation* »..... 149
Figure V.40 Modèle de présentation produit par la transformation « *Navigation 2 Presentation* » ... 150
Figure V.41 Modèle de présentation de la classe « Utilisateur » 151

CHAPITRE I. INTRODUCTION GENERALE

I.1. Problématique

L'ingénierie dirigée par les modèles (IDM) [France et al., 2007 ; Bézivin et al., 2004 ; Favre et al., 2006] connue sous le terme MDE (Model-Driven Engineering) en anglais est une discipline récente du génie logiciel qui recommande l'utilisation intensive des modèles et des transformations au cœur du processus de développement logiciel. Le terme IDM a été proposé pour la première fois par [Kent, 2002] et est dérivé de l'initiative MDA de l'OMG [Soley et al., 2000 ; Bézivin et al., 2001]. L'IDM a pour principal objectif de favoriser la réutilisation afin d'améliorer significativement le développement des systèmes complexes en fournissant des moyens permettant de passer d'un niveau d'abstraction à un autre ou d'un espace de modélisation à un autre. La finalité de l'IDM est de décrire un problème et sa solution en utilisant des modèles, et en s'appuyant sur une méthodologie qui permet de passer du problème à sa solution en utilisant les transformations. L'IDM est donc une forme d'ingénierie générative, par laquelle tout ou partie d'une application est engendrée à partir des modèles. Dans cette nouvelle perspective, les modèles occupent une place de premier plan (citoyens de première classe) parmi les artéfacts de développement et doivent en contrepartie être suffisamment précis et riches afin de pouvoir être interprétés ou transformés par des outils.

L'avènement de l'IDM a suscité beaucoup d'intérêt de la part des organisations qui de fait commencent à transformer leur processus de développement traditionnel en un processus de développement dirigé par les modèles appelé aussi – *processus IDM* – [Rios al, 2006 ; Stahl et al., 2006 ; Fondement et al., 2004 *;* Larrucea et al., 2004]. Dans [Kleppe et al., 2003] un processus IDM est défini comme suit: « a process of developing software using different models on different levels of abstraction with (automated) transformations between these models ». Un *processus IDM* est donc vu comme un enchaînement de transformations (automatiques) de modèles, chaque transformation consommant un ou plusieurs modèles sources et produisant un ou plusieurs modèles cibles [Bézivin, 2004]. Un processus IDM est moins centré-humain qu'un processus traditionnel dans la mesure où la majeure partie des activités du développement sont des transformations (automatiques) de modèles. Au moment où les processus IDM commencent à émerger [Koch, 2006 ; Maciel et al., 2006 ; Cong et al., 2010 ; Garcia et al., 2008 ; Koudri, 2010 ; OpenUP/MDD-website ; Anwar et al., 2010], nous notons l'absence d'un langage outillé reconnu par la communauté IDM pour modéliser et mettre en œuvre ces processus.

L'intérêt de la modélisation des processus en général et des processus IDM en particulier est d'utiliser une terminologie unifiée et cohérente afin de permettre d'une part une communication plus efficace entre les développeurs et d'autre part un meilleur suivi d'un projet de développement. La modélisation de processus permet aussi de gérer, de faire évoluer et de réutiliser efficacement les

processus [Humphrey et al., 1989]. Un modèle de processus est mis en œuvre lorsqu'une équipe de développement suit le modèle de processus. L'intérêt de la mise en œuvre des processus est de permettre un meilleur guidage du développement, une vérification des contraintes des activités/transformations, et la gestion de la cohérence des artéfacts/modèles du développement.

Le standard SPEM 2.0 [OMG-SPEM 2.0, 2008] dédié à la modélisation des processus propose des concepts génériques qui sont supposés être capables de décrire tout type de processus logiciel ou système incluant les processus IDM. Cependant, les concepts de SPEM ne capturent pas la nature exacte des processus IDM. Les concepts de modèles, de métamodèles, de transformation de modèles et de leurs diverses relations ne sont pas explicitement pris en compte. Face à la non prise en compte explicite des concepts centraux de l'IDM, la solution devrait se traduire par une réification des concepts IDM (modèle, métamodèle, transformation, outil IDM). L'intérêt de la réification des concepts IDM est de permettre d'une part aux concepteurs de processus d'expliciter les aspects spécifiques au développement IDM, d'autre part de mieux assurer la cohérence des modèles produits par les transformations. Par exemple pour vérifier la relation de conformité entre un modèle et son métamodèle, il est important de spécifier les modèles et les métamodèles qui participent à une transformation.

En outre, SPEM ne satisfait pas le critère d'exécutabilité dans sa dernière version, critère pourtant indispensable pour mettre un œuvre un modèle de processus. Néanmoins, SPEM propose d'exécuter les modèles de processus dans un formalisme externe basé soit sur les diagrammes d'activités UML 2.0, soit sur les machines à états d'UML, soit sur la notation BPMN (Business Process Modeling Notation).

Pour exécuter une transformation spécifiée dans un processus, il faut la décrire dans un formalisme exécutable basé soit sur des règles, un Template ou un programme. Le standard QVT permet de spécifier dan un formalisme exécutable une transformation à base de règles déclaratives ou impératives. L'intérêt de la réutilisation de QVT est de tirer pleinement profit des outils d'implémentation et d'exécution liés à ce standard.

En examinant les approches qui prennent en compte explicitement les concepts IDM, nous en avons identifié que deux : celles de Maciel [Maciel et al., 2009] et Porres [Porres et al, 2006]. La première (*celle de Maciel*) propose un langage de modélisation et de mise en œuvre des processus IDM. Ce langage est basé sur le standard SPEM et l'initiative MDA de l'OMG. Il est supporté par un environnement intégré appelé *Transforms*. Cette approche décrit les modèles UML d'entrée et de sortie d'une transformation, et ses métamodèles source et cible sous forme de profils UML. Cependant, elle ne spécifie pas la relation de conformité entre un modèle et son métamodèle. En outre, cette approche ne prend pas en compte les transformations basées sur un programme ou un Template. L'assistance aux développeurs dans la réalisation des activités autre que les transformations est aussi absente dans cette approche.

La deuxième approche (*celle de Porres*) propose un langage basé sur MDA, les diagrammes d'activités d'UML 2.2, le standard SPEM, et les réseaux de Petri. Cette approche permet de décrire un modèle de processus avec le métamodèle proposé et propose d'exécuter ce modèle de processus dans le formalisme des réseaux de Petri. Cependant la définition de la transformation (transformation avec ses métamodèles source et cible) et le formalisme d'implémentation de la transformation (règles, programme, ou Template) ne sont pas spécifiés dans ce métamodèle.

I.2. Objectifs de la thèse

L'objectif général de cette thèse est d'élaborer un langage et un environnement support pour modéliser et mettre en œuvre les processus IDM. Pour atteindre cet objectif, nous proposons une étude centrée autour de trois stratégies:

(1) proposer une extension de SPEM dans laquelle les concepts centraux des processus IDM sont réifiés. L'intérêt de la réification des concepts IDM est de permettre d'une part aux concepteurs de processus d'expliciter les aspects spécifiques au développement IDM, d'autre part de mieux assurer la cohérence des modèles produits par les transformations. Par exemple pour vérifier la relation de conformité entre un modèle et son métamodèle, il est important de spécifier les modèles et les métamodèles qui participent à une transformation.

(2) proposer un langage dédié à la modélisation comportementale des processus IDM. L'intérêt de la modélisation comportementale des processus IDM est d'apporter une assistance aux développeurs en leur fournissant à tout instant l'état de leurs activités /transformations et les opérateurs de mise en œuvre spécifiques.

(3) proposer une architecture conceptuelle d'un environnement logiciel d'aide à la modélisation et à la mise en œuvre des processus IDM. Cette architecture a pour principal intérêt de guider les concepteurs de processus dans la modélisation et les développeurs dans la mise en œuvre.

I.3. Approche adoptée

Pour atteindre l'objectif général décrit dans la section précédente, nous proposons un langage de modélisation et de mise en œuvre des processus IDM spécifié formellement sous forme d'un métamodèle appelé SPEM4MDE, et supporté par un environnement. Le métamodèle SPEM4MDE étend certains concepts de SPEM par des concepts relatifs au développement IDM (transformation, modèle, métamodèle, outil IDM). La réutilisation de SPEM 2.0 a pour principal intérêt de favoriser l'alignement de SPEM4MDE avec les autres LMP et de permettre une meilleure diffusion des concepts de SPEM4MDE. Pour réduire la complexité de SPEM4MDE, nous n'avons réutilisé que le paquetage *Process Structure* de SPEM 2.0 qui fournit les concepts basiques pour décrire la partie structurelle d'un processus.

Dans sa dernière version, SPEM ne spécifie aucun formalisme pour la description comportementale d'un processus. Pour surmonter cette limitation de SPEM et permettre aux concepteurs de processus de décrire le comportement des processus IDM, SPEM4MDE réutilise le paquetage d'UML 2.2 Superstructure qui décrit les machines à états d'UML. Une machines à états est composée d'états spécifiant l'ensemble des états d'un élément d'un processus IDM et de transitions spécifiant les opérateurs de mise en œuvre.

Nous réutilisons aussi le standard QVT afin de spécifier une transformation de SPEM4MDE dans un formalisme exécutable. La réutilisation du standard QVT a pour principal intérêt de tirer pleinement profit de ses outils d'implémentation et d'exécution liés à ce standard (SmartQVT, QVT-Eclipse, Medini QVT, OptimalJ, etc.). Le métamodèle SPEM4MDE prend aussi en compte les autres approches de transformations basées soit sur un programme, soit sur un Template afin de réutiliser les outils qui implémentent ces types de transformations (Modelio, IBM Rational Software Modeler, etc.).

Pour modéliser et mettre en œuvre un processus IDM, nous proposons une architecture conceptuelle d'un environnement logiciel fondé sur une démarche en trois étapes:

- La *première* étape a pour objectif de décrire le modèle structurel d'un processus IDM (*Structural MDE Process Model*) et le modèle comportemental du processus (*Behavioral MDE Process Model*). Ces modèles sont conformes au métamodèle SPEM4MDE. Par la suite, ces deux modèles sont validés sur la base des contraintes OCL spécifiées dans le métamodèle SPEM4MDE.

- La *deuxième* étape est l'assignation des ressources nécessaires (développeurs, outils, espaces de travail, etc.) à la mise en œuvre du modèle structurel. A ce stade, le modèle comportemental décrit dans l'étape ci-dessus peut être réutilisé ou adapté en tenant compte du projet visé.

- La *dernière* étape de cette démarche est la mise en œuvre du modèle de processus adapté. Elle est réalisée par les développeurs du projet en utilisant les outils IDM mis à leur disposition. Les développeurs sont assistés dans leurs tâches par un environnement de mise en œuvre qui s'appuie sur les modèles comportementaux du processus. Cette étape produit les livrables du projet (code, modèles, documentation, etc.).

Pour illustrer l'approche développée dans cette thèse, nous l'avons d'abord appliquée sur des exemples simples. Pour valider notre approche, nous l'avons appliquée à l'étude d'un processus IDM dédié au développement d'applications web.

Pour l'implantation de l'approche, un prototype a été développé sous l'environnement TOPCASED. Ce prototype fournit d'une part un environnement graphique pour la modélisation structurelle et comportementale des processus IDM et d'autre part un environnement de mise en œuvre qui offre aux développeurs les opérateurs de mise en œuvre spécifiques à chaque élément du processus.

I.4. Contenu du Mémoire

La suite de ce mémoire de thèse est organisée en cinq chapitres :

- **Chapitre II –** État de l'art**.** Ce chapitre est focalisé sur les deux domaines concernés prioritairement par notre travail de recherche : l'Ingénierie des Processus Logiciels (IPL) et l'Ingénierie Dirigée par les Modèles (IDM). Il s'agit de faire d'une part une synthèse sur les langages de modélisation de processus proposés dans le domaine de l'IPL et d'autre part un tour d'horizon sur les travaux récents de l'IDM qui ont pour point focal la notion de transformation de modèles. Pour illustrer cette étude, nous décrivons quelques exemples représentatifs de processus IDM et les langages proposés pour modéliser puis mettre en œuvre ces processus. L'étude des langages de description des processus IDM est conclue par un tableau comparatif qui montre les limites de ces langages selon des critères que nous avons définis.

- **Chapitre III –** Le métamodèle SPEM4MDE**.** Ce chapitre présente notre langage de description de processus IDM spécifié sous forme d'un métamodèle appelé SPEM4MDE. Le métamodèle SPEM4MDE réutilise SPEM pour la description structurelle des processus IDM, UML 2 pour la description comportementale des processus IDM, et QVT pour la description des transformations dans un formalisme exécutable.

- **Chapitre IV –** SPEM4MDE-PSEE : Une implémentation de SPEM4MDE**.** Ce chapitre présente un prototype d'implémentation de notre approche sous la forme d'un PSEE (Process-centered

Software Engineering Environment). Ce prototype développé sous la plate-forme TOPCASED offre en premier lieu un support aux concepteurs de processus pour décrire graphiquement les processus IDM à travers l'éditeur « *SPEM4MDE Process Editor* ». En second lieu, il fournit un environnement de mise en œuvre des processus IDM à travers le module « *SPEM4MDE Process Enactment Engine* ».

- **Chapitre V –** Étude de cas : Modélisation et Mise en œuvre du Processus UWE**.** Ce chapitre décrit en premier lieu une démarche générale de modélisation et de mise en œuvre d'un processus avec SPEM4MDE. En second lieu, il décrit la validation de la spécification et de l'implémentation du métamodèle SPEM4MDE à travers une étude de cas qui porte sur le processus UWE (*UML-Based Web Engineering*). Cette étude de cas, menée selon la démarche proposée dans SPEM4MDE, permet de modéliser le processus UWE et de le mettre en œuvre à travers un exemple simple de projet dédié à une application web de messagerie.

- **Chapitre VI –** Conclusion générale et perspectives**.** Ce chapitre conclut le mémoire de thèse en rappelant la problématique et les objectifs de cette thèse, en dressant un bilan de notre approche et en ouvrant des perspectives du point de vue conceptuel et pratique.

- Une série d'annexes est présentée à la fin de la conclusion. **L'annexe A** présente le paquetage *Process Structure* de SPEM 2.0 qui est réutilisé par SPEM4MDE dans la modélisation de la partie structurelle des processus IDM. **L'annexe B** présente le paquetage *BehaviorStateMachines* d'UML qui permet de spécifier le comportement d'un processus IDM. **L'annexe C** présente le standard QVT qui permet de spécifier dans un formalisme exécutable les transformations définies dans SPEM4MDE. **L'annexe D** présente les métamodèles et les profils UML du processus UWE utilisé pour notre cas d'étude. Les spécifications des transformations du processus UWE sont entièrement basées sur ces métamodèles et profils.

CHAPITRE II. ÉTAT DE L'ART

II.1. Introduction

La problématique adressée dans cette thèse est la modélisation et la mise en œuvre des processus IDM. Cet objectif ne saurait être atteint sans une étude bibliographique centrée autour de l'Ingénierie Dirigée par les Modèles (IDM) et de l'Ingénierie des Processus Logiciels (IPL) connue aussi sous le terme de SPE (Software Process Engineering) en anglais.

L'IDM [France et al., 2007 ; Bézivin et al., 2004 ; Favre et al., 2006], connue sous le terme MDE (Model-Driven Engineering) en anglais, est une discipline récente du génie logiciel qui promeut les modèles comme entités de première classe dans le développement logiciel. Le terme IDM a été proposé pour la première fois par [Kent, 2002].

L'IDM a été impulsée par l'initiative MDA (Model-Driven Architecture) de l'OMG [Soley et al., 2000 ; Bézivin et al., 2001 ; OMG-MDA, 2001] et fait depuis l'objet d'un grand intérêt aussi bien de la part des équipes de recherche académiques [ActionIDM-website] que des laboratoires industriels [Compuware-website ; Microsoft-website ; Softeam-website ; Xactium-website].

L'IPL est une discipline du génie logiciel qui vise à maîtriser les projets de développement logiciel en fournissant les moyens de modéliser les processus logiciels qui supportent ces projets [Humphrey, 1988b ; Finkelstein, 1994 ; Derniame, 1999]. Un processus logiciel est l'ensemble des activités qui transforment les exigences d'un client en un logiciel [Humphrey, 1988b]. Il inclut la spécification des exigences, la conception, l'implémentation, la vérification, l'installation, le support opérationnel et la documentation. Il inclut aussi parfois la maintenance afin de satisfaire les besoins futurs qui seront exprimés par le client. Un processus logiciel est exprimé à travers une notation ou un langage appelé LMP (Langage de Modélisation de Processus).

Dans ce chapitre, nous proposons un tour d'horizon sur les travaux relatifs à l'IDM et l'IPL. Pour l'IPL il s'agit de mettre l'accent sur l'étude des langages de modélisation des processus en général, et des processus IDM en particulier. Quant à l'IDM, c'est un domaine particulièrement vaste et évolutif, c'est la raison pour laquelle nous avons opté d'en donner une vision limitée en mettant l'accent sur la notion de transformation de modèles avec ses langages et outils associés.

Pour mener à bien l'étude bibliographique de notre travail de recherche, nous définissons les critères d'étude suivants qui nous permettront de comparer et d'évaluer par la suite les approches étudiées dans cette thèse.

- *La convivialité* : Un LMP est convivial lorsqu'il permet de décrire aisément un processus. Nous considérons que les LMP proposant une notation graphique sont plus conviviaux que ceux proposant une notation textuelle.
- *L'exécutabilité : Un LMP exécutable fournit des concepts ou un formalisme pour décrire le comportement ou l'exécution d'un processus.*
- *Prise en compte concepts de l'IDM. Ce critère permet de déterminer les LMP qui explicitent les concepts centraux de l'IDM (modèle, métamodèle, transformation, outil IDM).*
- *Traçabilité : Ce critère permet de déterminer les langages qui supportent la traçabilité des transformations, c'est-à-dire la possibilité de garder les traces d'exécution de chaque règle d'une transformation.*

Le chapitre est organisé comme suit. Nous présentons d'abord les concepts et les définitions des termes utilisés dans l'IPL, puis nous faisons une synthèse des LMP d'avant l'IDM (section II.2). La section II.3 donne une vue d'ensemble de l'IDM en mettant l'accent sur la notion de transformation et langages/outils associés. La section II.4 présente des exemples représentatifs de processus IDM tandis que la section II.5 aborde les langages permettant de décrire ces processus. La section II.6 présente un bilan comparatif des approches étudiées dans cette thèse.

II.2. L'Ingénierie des Processus Logiciels (IPL)

II.2.1. Concepts et Définitions

Dans cette section nous donnons les définitions de certains concepts [Feiler et al., 1993 ; Humphrey, 1989] couramment utilisés dans le domaine de l'IPL, notamment dans la modélisation et la mise en œuvre des processus logiciels.

- Un *processus logiciel* est un ensemble d'activités qui transforment des besoins en un logiciel.
- Un *modèle de processus* est la définition ou la représentation d'un processus. Il est utilisé pour définir le processus à suivre pour atteindre un objectif : la réalisation du produit logiciel dans le cas des processus logiciels. De plus, le modèle de processus peut être analysé, validé. Les modèles de processus sont décrits par le biais des AGL-P (Ateliers de Génie Logiciel centrés Processus) connus en anglais sous le terme PSEE (Process-centered Software Engineering Environment).
- *L'adaptation d'un modèle de processus* est l'acte qui consiste à modifier (ou « customiser ») un modèle général (ou « générique ») à un projet particulier et à assigner les ressources nécessaires pour supporter sa mise en œuvre. On parle parfois *d'instanciation d'un modèle de processus*.
- Un *modèle de processus exécutable* (enactable process model en anglais) est le résultat d'une instancitation des éléments génériques d'un modèle de processus et d'une assignation des ressources. Un modèle de processus exécutable est composé de la définition d'un processus, de ses entrées requises pour sa mise en œuvre, des agents et ressources assignés pour sa mise en œuvre.

- *Un agent de processus* est une entité qui participe à l'exécution d'un modèle de processus. Cette entité peut être une personne qui suit le modèle de processus ou un outil qui exécute une activité du processus.
- *La mise en œuvre d'un processus (Process enactment)* est l'exécution d'un modèle de processus par les agents du processus. La mise en œuvre d'un processus est surveillée et guidée par un AGL-P.
- Un AGL-P est un environnement de génie logiciel qui d'une part fournit une assistance aux concepteurs de processus et aux développeurs, d'autre part exécute les activités automatiques du processus. Un AGL-P utilise un modèle de processus pour coordonner les activités des agents du processus et automatiser certaines tâches telles que la vérification des préconditions et postconditions de chaque activité, l'affectation des rôles aux agents, le lancement automatique d'un outil pour réaliser une activité, etc.

II.2.2. Objectifs de la modélisation des processus logiciels

La modélisation des processus logiciels vise à relever certains défis du développement logiciel tels que la réduction du coût du développement, le respect des délais, et l'amélioration de la qualité des logiciels. L'objectif de la modélisation de processus n'est pas seulement de capitaliser et de pérenniser un savoir-faire en matière de développement logiciel sous forme de modèle de processus mais de s'en servir pour comprendre, analyser et exécuter le processus. La modélisation de processus a une double finalité : une finalité descriptive et une finalité prescriptive [Crégut, 1998].

Dans sa finalité descriptive, le processus est étudié pour savoir comment a été réellement développé le logiciel. Les objectifs de l'approche descriptive sont la documentation et l'analyse du processus. La documentation permet d'une part de garder les traces des procédures suivies lors de l'élaboration du produit logiciel et d'autre part les bonnes pratiques qui constitueront une bonne base pour les développements futurs. L'analyse du processus permet de mesurer le niveau de maturité du processus afin de l'améliorer. Il existe cinq niveaux de maturité successifs définis dans CMMI (*Capability Maturity Model Integration*) [SEI, 2010]. Les modèles du *CMMI* sont des collections de bonnes pratiques qui aident les organisations à améliorer leur processus. Les niveaux de maturité constituent les différentes étapes pour améliorer un processus logiciel :

- *Le niveau 1 ou niveau initial (Initial)* : il n'y a pas de processus bien défini, ni de consensus sur la base desquels les logiciels seraient construits. Le processus de développement n'est pas maîtrisé ni géré. Le succès dépend essentiellement des efforts individuels et héroïques des développeurs. Les exigences de qualité, les plannings et les budgets ne sont pas en général respectés.
- *Le niveau 2 ou niveau de gestion (Managed):* il y a un consensus dans l'organisme sur la manière dont le développement doit être géré, mais ce consensus n'est ni formalisé ni écrit. Un management de projet basé sur la réussite des projets précédents permet de contrôler rigoureusement les coûts et les délais. Le processus de développement est stabilisé (discipliné).
- *Le niveau 3 ou niveau défini (Defined)* : le processus de développement est formalisé, documenté, standardisé et approuvé. Les revues sont menées avec rigueur et les configurations sont convenablement gérées. Les projets utilisent une version approuvée et adaptable de ce

processus pour le développement et la maintenance des logiciels. La grande différence entre le niveau 2 et le niveau 3 se trouve sur les standards utilisés, la description du processus ou les procédures suivies. Au niveau 2, les standards utilisés, la description du processus, et les procédures sont très différents pour chaque instance du processus appliqué à un projet particulier. Au niveau 3, un processus standardisé est utilisé pour toutes les instances d'un processus. Ce processus est adapté suivant le projet visé.

- *Le niveau 4 ou niveau de gestion plus quantitative (Quantitatively Managed)* : l'organisme a institué un processus formel pour collecter et analyser des métriques qui seront utilisées pour suivre et contrôler le processus de développement. Des indicateurs contrôlent le bon déroulement des projets et le respect des délais et des objectifs de qualité. La réussite du projet est prédictible.
- *Le niveau 5 ou niveau optimisé (Optimizing)* : l'organisme exploite les métriques et les technologies innovatrices pour optimiser et améliorer en permanence son processus de développement.

La deuxième finalité est d'ordre prescriptif c'est-à-dire que le processus est formalisé pour répondre à la question « *comment le logiciel sera développé* ». L'approche prescriptive consiste à contrôler le développement et fournir une assistance aux développeurs. Cette assistance permet aux développeurs de connaître l'état du processus, les activités qui sont éligibles et celles qui sont inéligibles. De plus un AGL-P interprète le modèle de processus, réalise les activités automatiques et demande aux intervenants de réaliser celles qui sont manuelles.

II.2.3. Les Langages de Modélisation de Processus logiciels (LMP)

Depuis qu'il a été établi que la qualité des processus logiciels pouvait avoir un impact considérable sur la qualité des produits logiciels [Montangero et al., 1999 ; Armenise et al., 1993 ; Humphrey, 1988a], la communauté du génie logiciel ne cesse de voir émerger des langages de modélisation de processus. Rapidement les éditeurs de logiciels comprennent l'enjeu et commencent à décrire leur processus de développement sous forme de modèle afin de pérenniser leurs meilleures pratiques.

Les premiers Langage de Modélisation de Processus (LMP) ont vu le jour avec les notations informelles pour décrire les modèles de cycle de vie [Boehm, 1988 ; Royce, 1987]. Mais depuis la proposition de Lee Osterweil (« Software Processes are Software too » [Osterweil, 1987]), deux grandes familles de LMP se dessinent: les LMP de première génération et les LMP de deuxième génération. Beaucoup de ces LMP et les AGL-P qui leur sont associés sont discutés dans [Acuña et al., 2001].

Les LMP de première de génération sont des langages exécutables basés sur des langages de programmation (par exemple, *APPL-A* [Sutton et al., 1995] basé sur Ada, *MERLIN* [Junkerman et al., 1994] basé sur Prolog, *PBOOL* [Cregut et al., 1997 ; Coulette et al., 2000] basé sur Eiffel), d'autres sont similaires à des formalismes tels que les réseaux de Petri (par exemple *Slang* [Bandinelli et al., 1994]) ou bien sont basés sur des règles déclaratives (par exemple, *MSL* [Kaiser et al., 1990]), d'autres combinent le paradigme procédural (impératif) et le paradigme déclaratif (par exemple Little JIL [Sutton et al., 1997]). Les LMP de première génération décrivent un processus logiciel comme un programme informatique mais du fait de leur niveau d'abstraction très bas ils n'ont pas eu les faveurs

de la communauté du génie logiciel. Pour élever le niveau d'abstraction, une deuxième famille de LMP dite de deuxième génération fut proposée.

L'objectif des LMP de deuxième génération est de concilier la compréhensibilité et l'exécutabilité des LMP [Curtis et al., 1992], critères parfois antagonistes. En effet en élevant le niveau d'abstraction et en utilisant des notations graphiques au lieu du code, les modèles de processus sont devenus plus compréhensibles par les acteurs du développement. Dans cette catégorie de langage nous citons en premier lieu le standard *SPEM* (Software Process Engineering Metamodel) [OMG-SPEM 2.0, 2008] de l'OMG (Open Management Group). Cependant, SPEM ne satisfait pas le critère d'exécutabilité dans sa dernière version. Néanmoins, il propose deux approches pour exécuter les modèles de processus SPEM 2.0 : faire un mapping des modèles de processus SPEM 2.0 vers des plans de projets, ou bien associer un comportement externe aux éléments de processus de SPEM 2.0. En second lieu, nous citons les langages basés sur le standard SPEM (xSPEM [Bendraou et al., 2007], *eSPEM* [Elner et al., 2010], *MODAL* [Koudri et al. 2010], etc.). Enfin, nous terminons par les langages basés sur le standard UML (*UML4SPM* [Bendraou et al, 2005], *l'approche de Di Nitto* [Di Nitto et al., 2002], *l'approche de Chou* [Chou, 2002], *PROMENADE* (*PROcess-oriented Modelling and ENactment of software DEvelopment)* [Ribbo et al., 2000], etc.).

II.2.4. Mise en œuvre des processus logiciels

Pour mettre en œuvre un modèle de processus résultant d'une approche générique, il faut d'abord l'adapter à un projet spécifique. L'adaptation inclut l'assignation des ressources (espace de travail, outil, acteurs humains) nécessaires à la mise en œuvre du modèle de processus. La mise en œuvre a pour objectif d'assister les développeurs et de coordonner l'exécution d'un projet de développement.

Plusieurs LMP on été proposés pour mettre en œuvre les processus de développement. Certains LMP proposent de passer dans un autre espace de modélisation pour mettre en œuvre leurs modèles de processus (SPEM par exemple), d'autres proposent un formalisme fortement couplé au LMP (*UML4SPM,* xSPEM, eSPEM). Les premiers offrent plus de flexibilité mais supportent difficilement l'évolution du modèle de processus.

Les LMP de première génération supportent l'exécutabilité du fait qu'ils sont basés sur des langages de programmation ou sur des approches formelles telles que les réseaux de Petri. Ces langages proposent généralement de réprésenter le modèle de processus sous forme d'un programme informatique ; ce qui implique une parfaite connaisssance du langage de la part des concepteurs de processus. Du fait de leur bas niveau d'abstracation, ces langagages n'ont pas eu les faveurs de la communauté du génie logiciel pour modéliser les processus. Néanmoins, ils conviennnet pour mettre en œuvre un processus.

II.2.5. Les Ateliers de Génie logiciel centrés-Processus (AGL-P)

Les ateliers de génie logiciel centrés-processus ont pour objectif de fournir un support permettant de modéliser, d'analyser et d'exécuter (mettre en œuvre) un processus. Un AGL-P assiste les développeurs dans la réalisation des tâches dont ils son responsables, automatise les tâches routinières du développement, invoque et contrôle les outils de développement. La Figure II.1 ci-

dessous décrit les principes de fonctionnement d'un AGL-P. Dans cette figure, le modèle de processus représente le point d'entrée d'un AGL-P qui l'utilise pour coordonner l'exécution d'un projet basé sur ce modèle. L'AGL-P fournit une aide aux développeurs qui à leur tour lui soumettent les résultats de leurs activités. L'AGL-P fournit également et à tout instant l'état du développement.

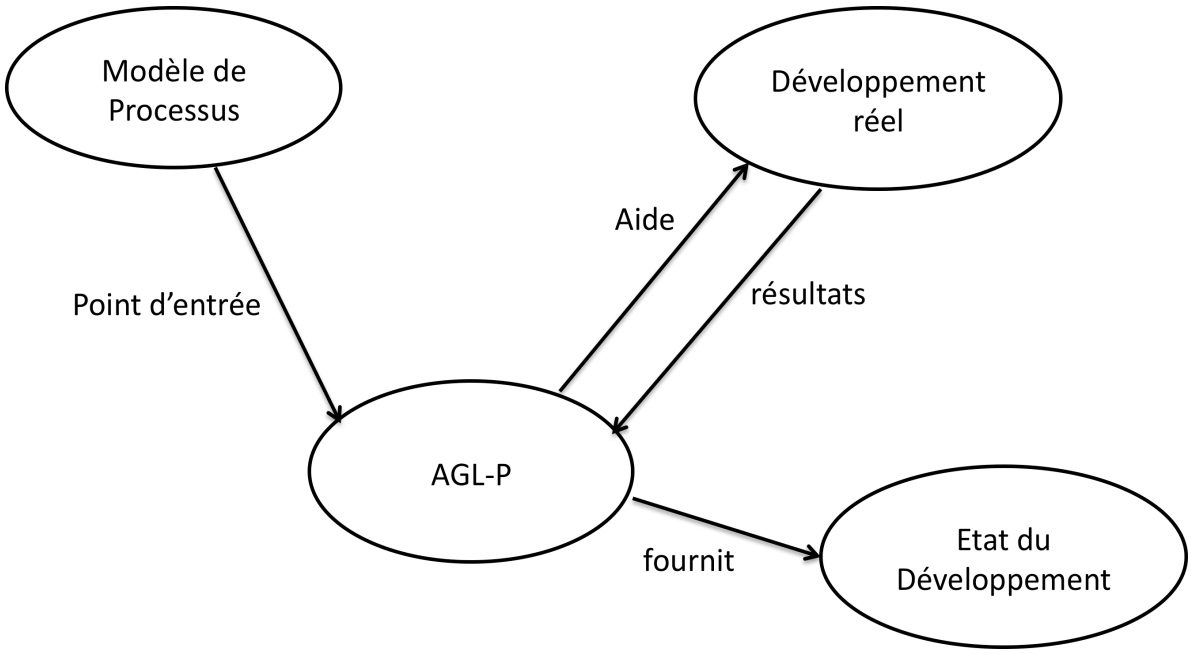


Figure II.1 Principes de fonctionnement d'un AGL-P [Dowson et al, 1994]

Nous distinguons plusieurs types d'assistance au niveau des AGL-P : passive, active, directive et automatique.

- **L'assistance passive** consiste à apporter une aide aux développeurs suite à leur demande.
- **L'assistance active** consiste à apporter systématiquement une aide aux développeurs sans qu'ils en aient formulé expressément la demande. Ce peut être le rappel d'une activité non encore réalisée, le signalement d'un produit qui se trouve dans un état incohérent ou un message spécifiant qu'une activité n'est pas encore éligible. Dans ce type d'assistance, le développeur peut effectuer ou ignorer les instructions de l'AGL-P.
- **L'assistance directive** consiste à forcer les développeurs à suivre le processus formalisé et à prendre en compte les instructions de l'AGL-P.
- **L'assistance automatique** (automatisation) consiste à réaliser toutes les activités d'un développement par le biais d'un AGL-P sans intervention humaine. Ce type d'assistance n'est pas réaliste car les processus de développement requièrent une grande part de créativité qui implique des activités manuelles.

Plusieurs AGL-P on été proposés dans la littérature. Les plus représentatifs sont *Acradia* qui implémente le langage *APPL-A* [Sutton et al., 1995], *MERLIN* qui implémente la langage qui porte le même nom [Junkerman et al., 1994], *Rhodes* qui implémente le langage *PBOOL* [Cregut et al., 1997 ; Coulette et al., 2000], *SPAD* qui implémente le langage *Slang* [Bandinelli et al. ,1994], *MARVEL qui*

implémente le langage *MSL* [Kaiser et al., 1990], *Julia* qui implémente le langage *Little JIL* [Sutton et al., 1997], et *Endeavors* [Bolcer atlm., 1996].

II.3. L'Ingénierie Dirigée par les Modèles (IDM)

L'IDM est une discipline du génie logiciel qui préconise l'utilisation intensive des modèles et des transformations au cœur du processus de développement. Elle a apporté des améliorations significatives dans le développement des systèmes complexes en fournissant les moyens permettant de passer d'un niveau d'abstraction à un autre ou d'un espace de modélisation à un autre. La finalité est de décrire à la fois un problème et sa solution en utilisant les modèles, et en s'appuyant sur une méthodologie qui permet de passer du problème à sa solution en utilisant les transformations de modèles. Avec l'avènement de l'IDM, le modèle est devenu le paradigme majeur par lequel l'industrie du logiciel pourrait lever le verrou de l'automatisation du développement. C'est ainsi que des organismes tels que l'OMG (Object Management Group) et les chercheurs en génie logiciel, cherchent à définir de nouveaux langages dédiés par le biais des modèles, à faciliter la création de nouveaux espaces technologiques plus adaptés aux besoins des utilisateurs et à faciliter les différentes étapes de modélisation nécessaires à l'élaboration d'un produit fini.

L'IDM est une forme d'ingénierie générative par laquelle tout ou partie d'une application informatique est générée à partir de modèles. Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artéfacts de développement des systèmes et doivent en contrepartie être suffisamment précis et riches afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme un enchaînement de transformations de modèles, chaque transformation prenant un ou des modèles en entrée et produisant un ou des modèles en sortie, jusqu'à l'obtention d'artéfacts exécutables. Cette transformation des modèles n'est bien sûr pas une tâche aisée. Il est donc nécessaire de disposer d'outils pour la gestion des modèles et de langages dédiés pour leurs transformations et leur manipulation tout au long de leur cycle de vie. Pour donner aux modèles cette dimension opérationnelle, il est essentiel de spécifier leur sémantique et donc aussi de décrire de manière précise les langages utilisés pour les représenter. On parle alors de métamodélisation. La métamodélisation a permis l'émergence de plusieurs DSMLs (*Domain Specific Modeling Language*), littéralement *langages de modélisation spécifiques à un domaine*. A l'image des langages de programmation, un DSML est défini par :

– *sa syntaxe abstraite* qui décrit les constructions du langage et leurs relations ;

– *une syntaxe concrète* qui décrit le formalisme permettant à l'utilisateur de manipuler les constructions du langage ;

– *une sémantique* statique qui permet d'avoir un DSML clairement défini donc «outillable ».

Cette section est organisée comme suit. Dans la sous-section II.3.1, nous présentons les travaux ayant contribué à la genèse de l'IDM. Dans la sous-section II.3.2, nous présentons les bases de l'IDM, principalement certains standards de l'OMG et l'approche MDA. La sous-section II.3.3 est dédiée à la transformation de modèles avec son panorama d'outils et de langages dédiés. Nous terminons par la sous-section II.3.4 qui discute les acquis et les limites de l'IDM.

II.3.1. Genèse de l'IDM

L'ingénierie logicielle fournit sans cesse de nouvelles technologies facilitant la mise en œuvre des systèmes informatiques [Blanc, 2005]. Quelle que soit l'approche utilisée (fonctionnelle, objet, composants, Middleware ou services, etc.), le but est toujours le même : l'accroissement de la qualité des systèmes informatiques tout en facilitant leur mise en œuvre sur une plate-forme d'exécution donnée. L'inconvénient majeur de ces approches traditionnelles réside dans le fait qu'il faille réécrire beaucoup de code quand on passe d'une plate-forme à l'autre ou que l'on veut adapter une plate-forme à l'évolution technologique. Face à cette situation, les chercheurs et les industriels se sont mis d'accord sur le fait que la solution à ce problème devait se traduire par une montée en puissance des modèles, et une séparation plus nette entre le métier et la technique. Cette décision a permis de faire passer les modèles de leur stade contemplatif qui se réduisait à la représentation et à la documentation des systèmes informatiques, à un stade plus productif qui envisage l'utilisation des modèles au cœur du cycle de développement de ces systèmes. Avec cette approche, le code final exécutable n'est plus considéré comme l'élément central dans le processus de développement mais comme un élément – naturellement important – qui résulte d'une transformation de modèles. Dans [Bézivin, 2004] une transformation de modèles est définie comme la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources. Pour mieux comprendre la philosophie de l'IDM, nous présentons dans cette section quelques approches à base de modèles qui ont fortement contribué à l'émergence de l'IDM : le Model-Integrated Computing (MIC) [MIC-website ; Sztipanovits et al., 1995] et les usines logicielles de Microsoft (Software factories) [Greenfield et al., 2004].

II.3.1.1. Model-Integrated Computing

Le Model-Integrated Computing (MIC) [MIC-website ; Sztipanovits et al., 1995] a été développé depuis plus de deux décennies au laboratoire ISIS (Institute for Software Integrated Systems) de l'université Vanderbilt aux USA. Il propose une vision du génie logiciel dans laquelle les artéfacts de base sont des modèles. Initialement, le MIC est conçu pour le développement des systèmes embarqués complexes mais aujourd'hui il est utilisé aussi dans plusieurs systèmes logiciels.

Le MIC met particulièrement l'accent sur la représentation formelle, la composition, l'analyse, et la manipulation des modèles durant le processus de conception. Il place les modèles au centre du cycle de vie des systèmes incluant la spécification, la conception, le développement, la vérification, l'intégration, et la maintenance. Le MIC facilite le développement logiciel basé sur les modèles en fournissant une technologie pour la spécification et l'utilisation des DSML (Domain-Specific Modeling Language), des outils de méta-programmation, des techniques de vérification et de transformations de modèles.

Le MIC repose en fait sur une architecture à trois niveaux :

– le niveau *Meta* fournit des langages de métamodélisation, des environnements de métamodélisation et des métagénérateurs pour créer des outils spécifiques à un domaine qui seront utilisés dans le niveau MIPS (Model-Integrated Program Synthesis) ;

– le niveau *Modèle* ou *MIPS* est constitué de langages de modélisation spécifiques à un domaine, et de chaînes d'outils pour la construction, l'analyse de modèles et la synthèse d'applications ;

– le niveau *Application* représente les applications logicielles adaptables. Dans ce niveau les programmes exécutables sont spécifiés en termes de composition de plates-formes.

II.3.1.2. Les usines logicielles (Software Factories)

Les usines logicielles (Software Factories) [Greenfield et al., 2004] sont la vision par Microsoft de l'ingénierie des modèles. Le terme d'usine logicielle fait allusion à une industrialisation du développement logiciel et s'explique par une analogie entre le processus de développement proposé et une chaîne de montage. Si l'on considère par exemple l'industrie automobile :

– Une chaîne de montage fabrique en général un seul type de voiture avec différentes combinaisons d'options ;

– Les ouvriers sont spécialisés. Un ouvrier peut avoir des compétences à différents niveaux de la chaîne de montage ; mais il ne peut pas avoir des compétences sur tous les postes de la chaîne c'est-à-dire de l'assemblage à la peinture des véhicules ;

– Les outils utilisés sont très spécialisés, fortement automatisés, et sont conçus uniquement pour cette chaîne de montage, ce qui permet d'atteindre des degrés élevés d'automatisation ;

– Une chaîne de montage automobile fait un assemblage de pièces normalisées ou préfabriquées dans une autre usine.

L'idée des usines logicielles est d'adapter ces caractéristiques au développement de logiciels. Les deux premiers points correspondent à la spécialisation des éditeurs de logiciels et des développeurs à l'intérieur des équipes de développement. Le troisième correspond à l'utilisation d'outils de génie logiciel spécifiques au domaine d'application, c'est-à-dire de langages et de transformateurs spécifiques. Le dernier point correspond à la réutilisation de composants logiciels. La plate-forme IDM *Microsoft DSL Tools* [Microsoft, 2005] est conçue autour de ces idées. Elle a pour vocation de créer des éditeurs graphiques et des générateurs de code personnalisés pour Visual Studio .Net 2005 [Microsoft, 2005], permettant de manipuler des modèles exprimés dans un langage proche des experts métiers, et d'en déduire le code source.

II.3.2. Les bases de l'IDM

L'IDM a pour principal objectif de relever un certain nombre de défis du génie logiciel (pérennité, prise en compte des plates-formes d'exécution, qualité, productivité, sûreté, séparation des préoccupations, coût, etc.) en suivant une approche à base de modèles dite générative. En focalisant le raisonnement sur les modèles, l'IDM permet de travailler à un niveau d'abstraction supérieur et de vérifier sur une maquette numérique (ensemble de modèles qui traitent divers aspects d'un système) un ensemble de propriétés que l'on devait vérifier auparavant sur le système final. L'un des apports supplémentaires du travail sur maquette numérique en lieu et place du code logiciel est de fournir un référentiel central permettant à plusieurs acteurs de s'intéresser aux différents aspects du système (sécurité, performance, consommation, etc.). Avec cette approche, l'IDM est en passe de lever le verrou qui consistait à ne pouvoir tester un système que tardivement dans le cycle de vie.

Dans la suite de cette section, nous présentons d'abord les définitions des concepts de base de l'IDM. Par la suite, nous décrivons les approches de l'OMG, particulièrement l'approche MDA qui

constitue une base indéniable de l'IDM même si cette architecture n'est pas l'unique manière d'aborder le développement dirigé par les modèles [Favre et al., 2006].

II.3.2.1. Concept et Définitions

Dans [Bézivin et al., 2001], un *modèle* est défini comme une représentation d'un (d'une partie) d'un *système* construit pour un objectif précis. Le modèle doit répondre aux questions que les utilisateurs se posent sur le système qu'il représente. Dans le contexte de l'IDM un modèle est défini dans [Kleppe et al., 2003] comme une description d'un (d'une partie) d'un *système* dans un langage bien défini. Un langage bien défini est un langage qui a une syntaxe et une sémantique bien définie et qui est interprétable par un outil.

Par analogie avec le monde de la programmation classique (Pascal, C, etc.), un *programme exécutable* représente le *système* alors que le *code source de ce programme* représente le *modèle*. De cette première définition découle une nouvelle relation appelée *represents* [Bézivin, 2004] reliant le *modèle* et le *système* modélisé. En poursuivant l'analogie avec le monde de la programmation, le *code source d'un programme* est exprimé ou écrit dans un *langage de programmation*. Il résulte de cette deuxième définition deux relations appelées « *conforms-to* » et « is *expressed in*» qui respectivement lient le *modèle* et son *métamodèle et le modèle* et la *syntaxe concrète* du *DSML* associé au *métamodèle.* On dira donc qu'un *modèle* est conforme à un *métamodèle* et est exprimé dans la *syntaxe concrète* d'un *DSML (Domain Specific Modeling Language)*. Le *modèle* respecte aussi la *sémantique statique* de son *métamodèle*. Dans le contexte de l'IDM, un *métamodèle* modélise un *DSML,* on dit aussi que le métamodèle représente le modèle du DSML. Un métamodèle est composé d'une *syntaxe abstraite* et d'une *sémantique statique*. Un *métamodèle* décrit les concepts pertinents d'un *domaine*. Un *domaine* c'est l'ensemble des concepts qui décrivent une connaissance. UML Superstructure, par exemple, est un *métamodèle* d'UML qui offre des concepts qui permettent de décrire les différents modèles UML (modèle de classes, modèle de cas d'utilisation, etc.).

Toujours par analogie avec le monde de la programmation, on dira qu'un langage de programmation respecte la grammaire BNF. Il résulte de cette définition une relation, également appelée « *conforms*-to », qui lie le métamodèle à son méta-métamodèle. Le méta-métamodèle est un langage d'expression de métamodèles. MOF [OMG-MOF 2.0, 2006], Ecore [Eclipse-EMF-website], Kermeta [Muller et al., 2005 ; Jezequel et al., 2005 ; Muller, 2006], et KM3 [ATLAS, 2005] sont des exemples de méta-métamodèles. Le modèle conceptuel ci-après (voir Figure II.2) résume ces notions de base en IDM, dont certaines sont tirées de [Stahl et al., 2006].

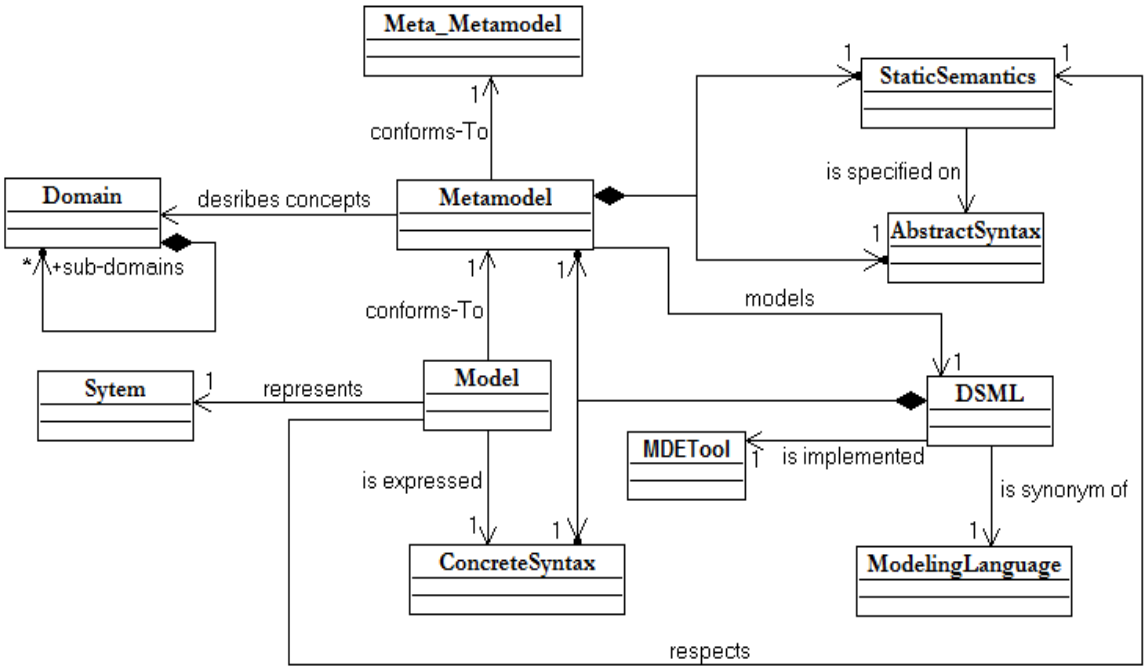


Figure II.2 Notions de base en ingénierie des modèles

II.3.2.2. Les approches de l'OMG

L'OMG est un consortium regroupant des industriels, des fournisseurs d'outils, et des académiques dont l'objectif principal est de développer des standards pour normaliser les différentes approches d'un domaine. Ces standards sur lesquels repose l'IDM sont centrés sur les notions de métamodèles et de méta-métamodèles et se trouvent sur différents niveaux dans l'architecture à quatre niveaux de l'OMG. Dans cette architecture (voir Figure II.3), le monde réel est représenté à la base de la pyramide (niveau M0). Les modèles représentant cette réalité constituent le niveau M1. Les métamodèles permettant la définition de ces modèles (UML, SPEM, etc.) constituent le niveau M2. Enfin, le méta-métamodèle (MOF), unique et métacirculaire, est représenté au sommet de la pyramide (niveau M3).

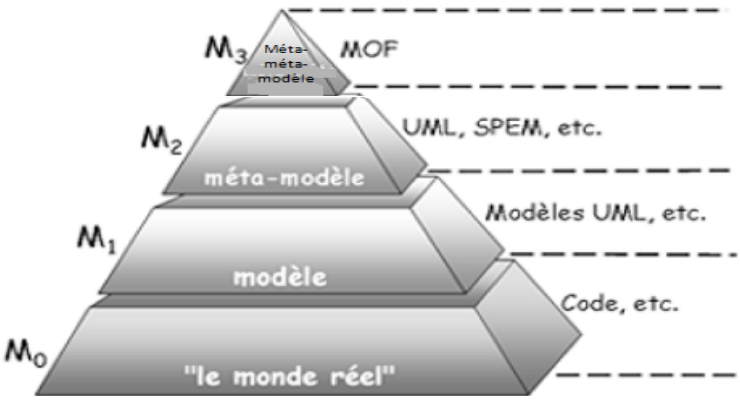


Figure II.3 Pyramide de modélisation de l'OMG [Bézivin, 2003a]

Dans cette section, nous présentons essentiellement les approches de l'OMG couramment utilisés dans l'IDM : *MOF* (Meta-Object Facility) : le standard pour établir de nouveaux langages de modélisation ; *UML* (Unified Modeling Language) : le standard de modélisation des systèmes logiciels ; *OCL* (Object Constraint Language) : le standard pour exprimer des contraintes (invariant, précondition, postconsidition) sur des modèles ; *XMI* (XML Metadata Interchange) qui permet de représenter les modèles sous forme de documents XML pour des besoins d'interopérabilité ; *DI* (Diagram Interchange) qui permet la représentation au format XML des parties graphiques d'un modèle ; *SPEM* (Software & Systems Process Engineering Metamodel) : le standard dédié à la modélisation des processus logiciels et systèmes ; et l'approche MDA.

II.3.2.2.1. Meta-Object Faclity (MOF)

Le standard MOF [OMG-MOF 2.0, 2006] est un méta-formalisme c'est-à-dire un formalisme pour décrire des langages de modélisation (par exemple UML). Dans sa version 2.0 le métamodèle MOF est constitué de deux parties : *EMOF* (Essential MOF), pour l'élaboration des métamodèles sans association, et *CMOF* (Complete MOF) pour les métamodèles avec associations. Il faut souligner que MOF s'auto-décrit pour pouvoir limiter l'architecture pyramidale de l'OMG à quatre niveaux.

II.3.2.2.2. Object Constraint Language (OCL)

Le but du langage OCL [OMG-OCL, 2005] est de permettre l'ajout de contraintes pour exprimer la sémantique statique des modèles et donc des métamodèles. Un métamodèle structurel n'a pas toujours l'expressivité suffisante pour décrire toutes les relations entre les méta-éléments qu'il contient. Dans le contexte des normes de l'OMG, OCL est utilisé pour décrire des contraintes non capturées dans le métamodèle structurel, sous forme d'invariants. L'ajout de ces contraintes est fondamental pour obtenir des métamodèles clairement définis et donc facilement outillables. En outre, OCL permet de définir des pré et postconditions sur les opérations d'un modèle pour que le système modélisé reste dans un état cohérent quand on exécute ces opérations. Les constructions d'OCL ne permettent pas de créer, détruire ou modifier les objets d'un modèle : la vérification des contraintes se fait sans effet de bord. En plus de la définition de contraintes, OCL peut être utilisé pour spécifier de manière déclarative le comportement de propriétés dérivées et d'opérations, sans toutefois pouvoir exprimer impérativement des modifications de modèles. OCL fournit donc une bonne solution pour exprimer des contraintes sur les modèles mais pas pour décrire le comportement ou la sémantique d'exécution des modèles. Cette limitation d'OCL a amené l'OMG à définir le standard AS (*Action Semantics*) qui permet de modifier l'état d'un modèle. Depuis la deuxième version d'UML, le standard AS est intégré dans UML Superstructure. Cette limitation d'OCL ne remet pas en cause ses vertus, d'autant plus qu'il peut être appliqué sur tout type de modèle (MOF, Ecore, UML, etc.) et est utilisé dans plusieurs langages ou normes de transformation (QVT, ATL, Kermeta, etc.).

II.3.2.2.3. Software & Systems Process Engineering Metamodel (SPEM 2.0)

Le but du standard SPEM est de proposer des concepts partagés par la communauté du génie logiciel pour décrire les processus de développement logiciel. La première version de SPEM (SPEM 1.1) présentait plusieurs points faibles parmi lesquels, une sémantique confuse de certains de ses

concepts, le manque de flexibilité, la non prise en compte des processus systèmes et l'impossibilité d'exécuter les modèles de processus. Pour palier ces insuffisances de SPEM, et en même temps aligner SPEM et la deuxième version d'UML, une nouvelle version de SPEM [OMG-SPEM 2.0, 2008] fut adoptée le 01 Avril 2008. Cette nouvelle version est dédiée à la modélisation des processus logiciels et systèmes. SPEM 2.0 est décrit à la fois comme un modèle MOF et un profil UML. Comme profil UML, il bénéficie de l'outillage d'UML existant ainsi que des principaux diagrammes d'UML.

L'objectif général de SPEM 2.0 est de fournir un ensemble de concepts pour définir les processus logiciels et les processus systèmes sans ajouter de caractéristiques spécifiques à la gestion de projet. Pour cela, SPEM 2 .0 propose un métamodèle générique et flexible qui couvre divers types de processus. Les nouveautés de SPEM 2.0 par rapport à SPEM 1.1 sont les suivantes :

- *Réutilisation de savoir-faire :* SPEM 2.0 sépare nettement le contenu réutilisable d'une méthode de développement de son application dans un processus spécifique. Une méthode de développement décrit les activités d'un développement, leurs produits en entrée et en sortie, leurs rôles sans spécifier leur ordre d'exécution alors qu'un processus réutilise ces activités et les lie selon un ordre donné.

- *Généricité* : SPEM 2.0 propose un mécanisme d'extension qui permet de décrire et de documenter divers types de processus (ajout du concept *Kind* qui permet de typer tous les concepts de SPEM 2.0). Par exemple, une activité peut être typée en une phase ou une itération.

- *Flexibilité* : Le mécanisme de plug-in de SPEM 2.0 introduit des concepts pour la gestion et la maintenance des librairies d'une méthode de développement ou d'un processus. Avec le mécanisme du « plug-in », un concepteur de processus pourra enrichir ou utiliser tout ou partie des processus définis.

- *Modularité* : SPEM 2.0 propose plusieurs mécanismes de réutilisation des processus (*ProcessPatterrn*, *ProcessComponent*, mécanisme *ActivityUseKind*, etc.) Les composants de processus sont des paquetages spéciaux qui appliquent le principe de l'encapsulation ; le concept de patron de processus permet de modéliser un savoir-faire et de l'appliquer dans un contexte ; le concept *ActivityUseKind* permet de définir les différentes manières de réutiliser une activité dans un processus.

- *Exécutabilité* : SPEM 2.0 ne spécifie aucun formalisme pour la description comportementale d'un processus. Cependant, il propose un mécanisme permettant de lier un modèle de processus à un ou plusieurs modèles comportementaux basés soit sur un diagramme d'activité UML 2.0, soit sur une machine à états d'UML, soit sur la notation BPMN (Business Process Modeling Notation). Cependant, SPEM ne fournit aucune information sur comment la liaison entre les éléments d'un processus et les modèles comportementaux sera réalisée.

Le métamodèle de SPEM 2.0 est constitué de sept paquetages (voir Figure II.4) :

- *Core* : Il définit le noyau de SPEM 2.0 en fournissant les métaclasses de base pour les autres paquetages. Ce paquetage réutilise une partie de l'infrastructure d'UML 2.0.

- *ProcessStructure* : Il fournit les concepts de base permettant la description statique des processus sans réutiliser une méthode de développement. Ce paquetage « merge » le paquetage *Core* et est réutilisé dans l'approche que nous proposons. Par conséquent, il est intégralement décrit dans l'annexe A du chapitre III.

- *ManagedContent* : Il fournit les concepts pour décrire textuellement les éléments d'un processus ou d'une méthode de développement. Ce paquetage permet d'habiller (ajout de propriétés) un modèle de processus ou une méthode de développement. Il « merge » le paquetage *Core*.

- *MethodContent* : Il fournit les concepts pour décrire le contenu réutilisable d'une méthode de développement. Ce contenu peut être réutilisé par un ou plusieurs processus. Le contenu d'une méthode de développement (*MethodContent*) fournit une base de connaissances (savoir-faire) indépendante d'un processus ou d'un cycle de vie concret. Il « merge » le paquetage *ManagedContent*.

- *ProcessBehavior* : Il « merge » les paquetages *ProcessStructure* et *MethodContent,* et definit des concepts permettant d'associer aux modèles de processus SPEM un comportement exprimé dans un formalisme externe (machines à états, diagrammes d'activitéa ou BPMN).

- *ProcessWithMethods* : Il fournit les concepts permettant d'intégrer une méthode de développement définie avec le paquetage *MethodContent* dans un processus défini avec le paquetage *Process Structure*. Un processus représenté par une activité (Activity), prend les éléments d'une méthode de développement et les relie selon l'ordre d'exécution adapté aux besoins spécifiques d'un type de projet. Il « merge » les paquetages *ProcessStructure* et *MethodContent*.

- *MethodPlugin* : Il introduit les concepts pour la gestion, la maintenance et la réutilisation des librairies d'un processus ou d'une méthode de développement. Le mécanisme de plug-in facilite l'extensibilité et la variabilité des processus. Il « merge » les paquetages *ProcessStructure, MethodContent* et *ProcessWithMethods*.

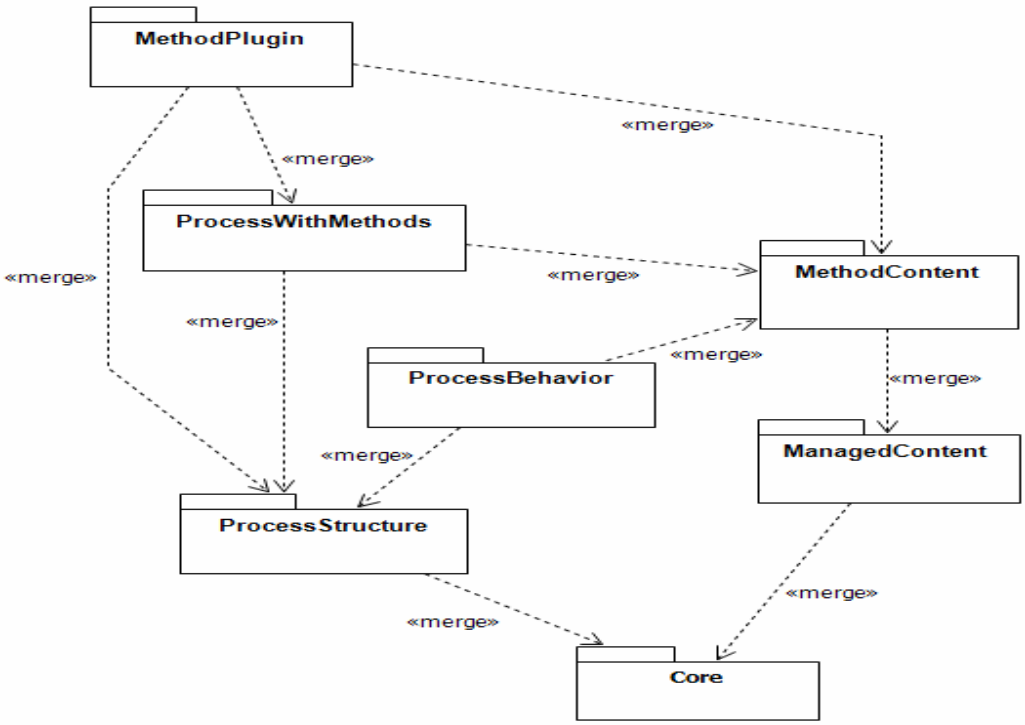


Figure II.4 Structure du métamodèle de SPEM 2.0 [OMG-SPEM 2.0, 2008]

II.3.2.2.4. L'approche MDA (Model-Driven Architecture)

MDA (*Model-Driven Architecture*) [Soley et al., 2000 ; OMG-MDA, 2001] est une initiative de l'OMG rendue publique en 2000. C'est à la fois la proposition d'une architecture et d'une démarche

de développement. L'idée de base de MDA est de séparer les spécifications fonctionnelles d'un système des détails de son implémentation sur une plate-forme donnée. Pour cela, MDA définit une architecture de spécification structurée en plusieurs types de modèles : modèles indépendants de l'informatisation appelés CIM (*Computational Independent Model*), modèles indépendants des plates-formes appelés PIM (*Platform Independent Model*), et modèles spécifiques à une plate-forme appelés PSM (*Platform Specific Model*), générés à partir des PIM en se basant sur les PDM (*Platform Description Model*).

Un CIM modélise les exigences d'un système, son but étant d'aider à la compréhension du problème mais aussi de fixer un vocabulaire commun pour un domaine. Dans UML, le diagramme des cas d'utilisation est un bon candidat pour représenter un CIM.

Le PIM, connu aussi sous le nom de modèle d'analyse ou de conception, est un modèle abstrait indépendant de toute plate-forme d'exécution. Le PIM a pour but de décrire le savoir-faire ou la connaissance métier d'une organisation. Ayant isolé le savoir-faire métier dans des PIM, on a besoin soit de transformer ces modèles PIM en d'autres PIM pour les affiner ou pour des besoins d'interopérabilité, soit de produire des modèles PSM ciblant une plate-forme d'exécution spécifique en se basant sur les PDM pour assurer la portabilité et augmenter la productivité. En effet, la transformation PIM vers PSM permet par la suite de genérer la quasi-totalité du code de l'application, ce qui permet d'avoir un important gain de productivité. Sans cette transformation, les modèles seraient en décalage avec le code de l'application qui serait entièrement écrit à la main.

Le PDM modélise la plate-forme sur laquelle le système va être exécuté (ex. modèles de composants à différents niveaux d'abstraction : PHP, EJB, etc.). Plus précisément, il définit les différentes fonctionnalités de la plate-forme et précise comment les utiliser.

L'initiative MDA de l'OMG a donné lieu à une standardisation des approches pour la modélisation sous la forme d'une structure en quatre niveaux de modélisation appelée encore *Pile de modélisation*. La proposition initiale était d'utiliser le langage UML et ses différentes vues comme unique langage de modélisation. Cependant, il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin d'exprimer de nouveaux concepts relatifs à des domaines d'application spécifiques. Ces extensions devenant de plus en plus importantes, la communauté MDA a élargi son point de vue en considérant les langages de modélisation spécifiques à un domaine (DSML).

La Figure II.5 donne une vue générale d'un processus MDA appelé couramment cycle de développement en Y en faisant apparaître les différents niveaux d'abstraction associés aux modèles.

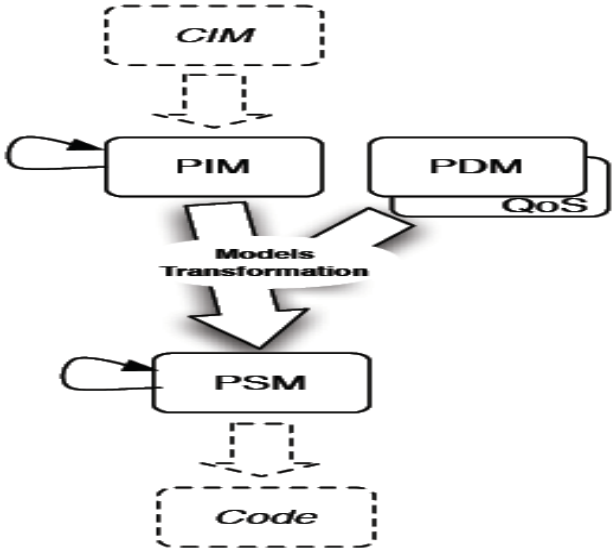


Figure II.5 Principes du processus MDA [Combemale, 2008]

Aujourd'hui beaucoup d'outils s'inscrivent dans la mouvance MDA. Parmi les plus récents nous pouvons citer :

– AndroMDA [AndroMDA, 2008] qui est une plate-forme de génération de code extensible qui transforme les modèles UML en composants déployables sur une plate-forme donnée (J2EE, Spring, .NET, etc.).

– OAW [OAW, 2008], l'acronyme d'Open Architecture Ware. C'est une plate-forme modulaire de génération développée en Java, qui supporte les modèles EMF, UML2, XML ou des modèles exprimés en JavaBeans, etc.

II.3.3. Transformation de modèles

La définition de transformations a pour objectif de rendre les modèles opérationnels dans une approche IDM, ce qui augmente la productivité des applications. La transformation de modèles est une opération très importante dans toute approche orientée modèle. En effet, les transformations assurent les opérations de passage d'un ou plusieurs modèles d'un niveau d'abstraction donné vers un ou plusieurs autres modèles du même niveau (transformation horizontale) ou d'un niveau différent (transformation verticale [Mens et al., 2004]. On peut citer comme exemple de transformation verticale, la transformation PIM vers PSM et comme exemple de transformation horizontale, la transformation PIM vers PIM ou PSM vers PSM. Le modèle transformé est appelé *modèle source* et le modèle résultant de la transformation est appelé *modèle cible*.

II.3.3.1. Principales approches de transformations de modèles

Une classification discutée dans [Czarnecki et al., 2003] distingue deux types de transformation : une transformation modèle vers modèle et une transformation modèle vers code. En général, la transformation modèle vers code peut être vue comme un cas particulier d'une transformation

modèle vers modèle ; il suffit juste de définir le métamodèle du langage de programmation cible. Parmi les approches de transformations modèles nous distinguons l'approche par *programmation*, l'approche par *template* et l'approche par *modélisation ou par règles*. Ces trois approches discutées dans [Blanc, 2005] sont détaillées dans les sections qui suivent.

II.3.3.1.1. L'approche par programmation

L'approche par programmation utilise les langages de programmation en général, et plus particulièrement les langages orientés objet. Cette approche décrit la transformation sous forme d'un programme à l'image de n'importe quelle application informatique. Cette approche reste très utilisée car elle réutilise l'expérience accumulée et l'outillage des langages existants. Cette approche est utilisée dans les outils Softeam MDA Modeler (section II.3.3.5.1) et IBM Rational Software Modeler (section II.3.3.5.2).

II.3.3.1.2. L'approche par template

L'approche par template consiste à définir des modèles cibles en y déclarant des paramètres. Ce modèle paramétré appelé aussi modèle template peut être dans un format graphique (template UML par exemple) ou textuel (template basés sur XMI ou un langage de programmation). L'exécution d'une transformation consiste à prendre un modèle template et à substituer ses paramètres par les informations contenues dans le modèle source. Cette approche par template est implémentée par exemple dans Softeam MDA Modeler. La notion de template est très utilisée dans le développement des sites web basés sur PHP ou JSP. En effet, dans ces langages des pages dynamiques sont créés à partir des canevas de pages (templates) qui contiennent des paramètres qui font référence par exemple aux données d'une base. L'approche par template est aussi supportée par les outils Softeam MDA Modeler (section II.3.3.5.1) et IBM Rational Software Modeler (section II.3.3.5.2).

II.3.3.1.3. L'approche par modélisation

L'approche par modélisation ou par règles consiste à appliquer les principes de l'ingénierie des modèles aux transformations. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs, en exprimant leur indépendance vis-à-vis des plates-formes d'exécution. Le standard MOF 2.0 QVT de l'OMG (voir section II.3.3.3.1) est une illustration parfaite de l'approche par modélisation et a pour but de fournir un moyen standardisé pour établir les transformations de modèles. L'approche par modélisation est aussi choisie par le groupe ATLAS pour le langage ATL (voir II.3.3.3.2). Dans l'approche par modélisation, la transformation se fait par l'intermédiaire de règles de transformations qui décrivent la correspondance entre les entités du modèle source et celles du modèle cible. La transformation se situe en réalité, au niveau des métamodèles source et cible qui décrivent la structure des modèles cible et source.

La Figure II.6 détaille un exemple d'application de l'approche par modélisation. Dans cet exemple, une transformation d'un modèle source est constituée de deux étapes. La première étape consiste à décrire le modèle de transformation appelé aussi règles de la transformation. Ces règles sont spécifiées sur la base des métamodèles source et cible. La seconde étape consiste à générer automatiquement le modèle cible par l'exécution des règles de la transformation.

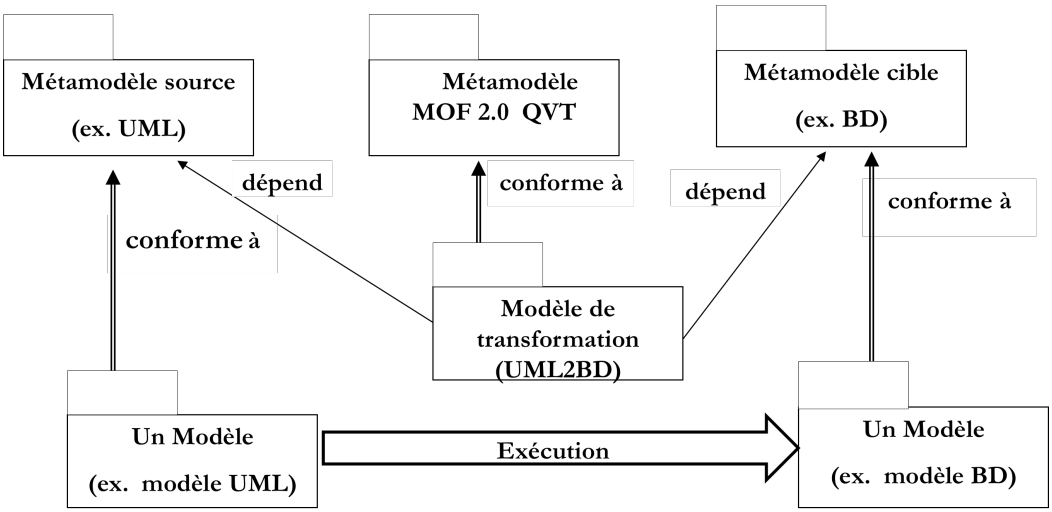


Figure II.6 Exemple d'application de l'approche par modélisation [Blanc, 2005]

II.3.3.1.4. Synthèse des approches de transformations de modèles

En résumé, définir une transformation consiste à spécifier les règles de mapping d'un modèle source vers un modèle cible. Ce qui caractérise les différentes approches, c'est la façon dont les règles de mapping sont implémentées. L'approche par programmation est sans doute la mieux outillée. Elle est aussi la plus facile à utiliser, car elle ne nécessite que peu d'apprentissage pour ceux qui maîtrisent les langages de programmation orientés objet. Mais cette approche a des limites au niveau de la lecture des modèles stockés sous forme de fichiers XMI. L'approche par template n'est pas difficile à mettre en œuvre pourvu qu'on dispose d'un langage approprié de définition de template. L'approche par modélisation, même si elle est la plus complexe à mettre en œuvre, reste la plus prometteuse, car elle offre une solution favorisant la pérennité des transformations et facilite donc leur réutilisation.

II.3.3.2. Typologie et propriétés des transformations

Dans [Mens et al., 2004] nous distinguons les transformations dites *endogènes* et *exogènes* combinées à des transformations dites *verticales* et *horizontales* (voir Figure II.7). Une transformation est dite *endogène* si les métamodèles source et cible sont identiques ; elle *exogène* dans le cas contraire. Une transformation *simple* ou m*u*ltiple peut être *exogène* ou *endogène*. Par contre pour une transformation sur place les métamodèles source et cible sont identiques, donc elle est endogène. Le niveau d'abstraction peut changer dans le cadre d'une transformation. Une transformation est dite *verticale* si elle met en jeu différents niveaux d'abstraction dans la transformation (PIM vers PSM). Une transformation est dite *horizontale* lorsque les modèles source et cible associés à la transformation sont au même niveau d'abstraction (PIM vers PIM ou PSM vers PSM).

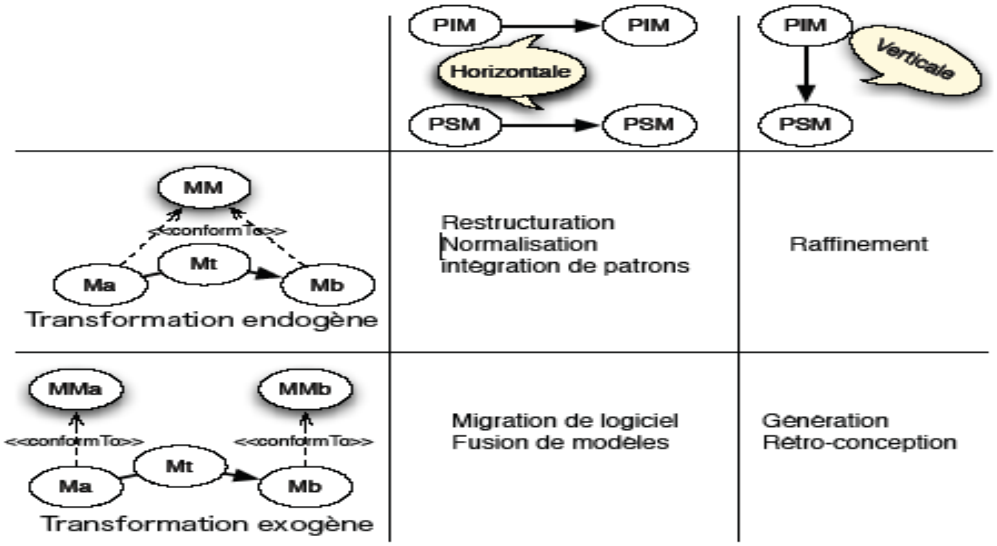


Figure II.7 Typologie des transformations de modèles [Combemale, 2008]

La restructuration, la normalisation et l'intégration de patrons de conception sont des transformations *endogènes* et *horizontales* alors que le raffinement d'un modèle (par exemple le modèle d'analyse UML vers le modèle de conception UML) est une transformation *endogène et verticale*. La migration de logiciel et la fusion de modèles sont des transformations *exogènes* et *horizontales,* tandis que la génération de code et la rétro-conception sont des transformations *exogène*s et *verticales*.

Dans la littérature [Mens et al., 2004 ; Czarnecki et al., 2003], nous distinguons une autre typologie des transformations caractérisée comme suit :

– une *transformation simple* (1 vers 1) transforme un élément d'un modèle source en un élément d'un modèle cible. Un exemple typique est la transformation d'une classe UML en un document XML qui définit sa structure ou en une table de base de donnée relationnelle ;

– une *transformation multiple* (M vers N) transforme un ou plusieurs éléments d'un modèle source en un ou plusieurs éléments d'un modèle cible. Comme exemples de transformations multiples nous pouvons citer les transformations de décomposition de modèles (1 vers N) et de fusion de modèles (N vers 1) ;

– une *transformation de mise à jour* parfois appelée *transformation sur place* permet de modifier un modèle par ajout, modification ou suppression d'un de ses éléments. Dans ce type de transformation, le modèle source est mis à jour sans création explicite du modèle cible. La restructuration de modèles (*Model Refactoring*) qui permet de réorganiser les éléments d'un modèle afin d'améliorer sa structure ou sa lisibilité en est une illustration.

Les propriétés suivantes caractérisent une transformation :

– *Réversibilité* : une transformation est dite réversible si elle est exécutable dans les deux sens. Exemple : Model-to-Code et Code-to-Model.

– *Réutilisabilité* : la réutilisabilité permet de réutiliser les règles d'une transformation dans d'autres transformations de modèles.

– *Ordonnancement* : l'ordonnancement consiste à représenter les niveaux d'imbrication des règles de transformation du fait que l'exécution d'une règle d'une transformation peut déclencher l'exécution d'autres règles.

– *Modularité* : une transformation modulaire permet de mieux modéliser les règles ou le programme de transformation. Un langage de transformation de modèles supportant la modularité facilite la réutilisation des règles de transformation.

II.3.3.3. Les langages/outils dédiés aux transformations de modèles

Dans cette catégorie, on retrouve les outils et langages conçus spécifiquement pour faire de la transformation de modèles et qui peuvent être intégrés dans les environnements de développement standard (par ex. Éclipse).

Dans le monde académique nous retrouvons le standard QVT [OMG-QVT, 2008], le langage ATL du groupe ATLAS de l'INRIA-LINA [Bézivin et al., 2005], *BOTL* (Bidirectional Object oriented Transformation Language) [Marschall et al., 2003], *ModTransf* [ModTransf-website] un langage de transformation basé sur les règles d'XML, *AndroMDA* [AndroMDA, 2008], *Coral* [Alanen et al., 2004] un outil pour créer, éditer, et transformer des modèles à l'exécution, *QVTEclipse* [QVTEclipse-website] une implantation préliminaire du standard QVT dans Eclipse, *UMT-QVT*(UML Model Transformation Tool) [UMT-QVT-website].

Dans le monde industriel, nous retrouvons *Mia-Transformation* [MiaSoftware-website] de Mia-Software qui permet d'exécuter des transformations de modèles prenant en charge différents formats d'entrée et de sortie (XMI, API, etc.), et *PathMate* de PathFinders Solution [PathFinders Solution-website] qui est un outil configurable qui s'intègre dans les AGL implémentant UML.

Dans la suite de cette section, nous présentons essentiellement les langages QVT et ATL qui sont représentatifs de cette catégorie des langages, et font l'objet de nombreuses expérimentations dans la communauté IDM.

II.3.3.3.1. Le standard QVT (Query, Views, Transformation)

Les transformations de modèles étant au cœur de l'IDM, un standard dénommé QVT (*Query, Views, Transformation*) [OMG-QVT, 2008] a été établi pour modéliser ces transformations. L'objectif de QVT est d'établir une manière standardisée pour interroger les modèles MOF, créer des vues et définir les transformations. Le standard QVT vise à atteindre les objectifs suivants :

– normaliser un moyen d'exprimer des correspondances (transformations) entre langages définis avec MOF ;

– exprimer des requêtes (Query) pour filtrer et sélectionner des éléments d'un modèle (notamment sélectionner les éléments source d'une transformation) ;

– proposer un mécanisme pour créer des vues (Views) qui sont des modèles déduits d'un autre pour en révéler des aspects spécifiques ;

– formaliser une manière de décrire des transformations (Transformation).

Les principales recommandations dans la spécification de MOF QVT [OMG-QVT, 2005 ; Jouault *et al.*, 2006] sont les suivantes :

– La syntaxe abstraite du métamodèle QVT sera décrite en MOF et sa syntaxe concrète sera textuelle ou graphique ;

– Le langage de requête devra s'appuyer sur le langage OCL ;

– La gestion des liens de traçabilité devra être automatique ;

– MOF QVT devra permettre plusieurs scenarii d'exécution (transformations uni-directionnelles, multi-directionnelles, incrémentales, etc.).

La dernière version du standard QVT [OMG-QVT, 2008] présente un caractère hybride dans le sens qu'elle est composée de trois langages de transformation différents (voir Figure II.8). La partie déclarative de QVT est définie par les langages *Relations* et *Core* qui ont des niveaux d'abstraction différents. *Relations* est un langage orienté utilisateur permettant de définir des transformations à un niveau d'abstraction élevé. Il a une syntaxe textuelle et graphique. Le langage *Core* forme l'infrastructure de base pour la partie déclarative ; c'est un langage technique de bas niveau défini par une syntaxe textuelle. Il sert à spécifier la sémantique d'exécution du langage Relations, sous la forme d'une transformation Relations2Core. La vision déclarative passe par une association de patterns, côté source et cible pour exprimer la transformation. Manifestement, elle permet une expression plus simple des transformations de type mappings (transformation unidirectionnelle).

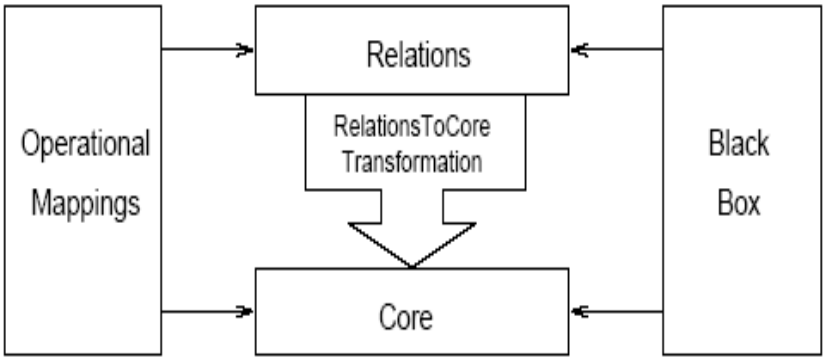


Figure II.8 Architecture du standard QVT [OMG-QVT, 2008]

La composante impérative de QVT est supportée par le langage *Operational Mappings*. La vision impérative impose une navigation explicite et une création explicite des éléments du modèle cible. Le langage *Operational Mappings* propose *un premier mécanisme d'extension* des deux langages déclaratifs de QVT (Relations, Core) en ajoutant des constructions impératives (séquence, sélection, répétition, etc.) ainsi que des constructions OCL à effet de bord. Les langages de style impératif sont mieux adaptés pour des transformations complexes qui comprennent une composante algorithmique importante. Par rapport au style déclaratif, ils ont l'avantage de gérer les cas optionnels dans une transformation. Enfin, QVT propose *un deuxième mécanisme d'extension* pour spécifier des transformations, en permettant d'invoquer des fonctionnalités de transformations implémentées dans un langage externe, appelé « *Black Box* »

Beaucoup d'outils implémentent le standard QVT. Parmi eux, nous pouvons citer : *SmartQVT* [Alizon et al., 2008], *MediniQVT* [Medini QVT-website], etc.

SmartQVT est une implémentation du langage standardisé *QVT-Operational*. Cet outil compile des transformations de modèles exprimées en QVT pour produire du code Java exécutable. Cet outil est fourni sous forme de plug-in Eclipse basé sur le framework de métamodélisation EMF. Comme il implémente un langage impératif, il est bien adapté pour décrire des transformations complexes.

MediniQVT est une implémentation du langage standardisé *QVT-Relations*. Cet outil inclut un débogueur graphique et un éditeur pour spécifier de manière déclarative les transformations.

II.3.3.3.2. Le Langage ATL

ATL [Bézivin et al., 2005] est l'acronyme d'ATLAS Transformation Language ; c'est un langage à vocation déclarative, mais qui est en réalité un langage hybride permettant de faire des transformations de modèles aussi bien endogènes qu'exogènes. Les outils de transformation liés à ATL sont intégrés sous forme de plug-in ADT (ATL Development Tool) dans l'environnement de développement Eclipse. Un modèle de transformation ATL se base sur des définitions de métamodèles au format XMI. Sachant qu'il existe des dialectes d'XMI, ADT est adapté pour interpréter des métamodèles décrits à l'aide d'EMF (Eclipse) ou MDR (NetBeans). Afin d'assurer son indépendance par rapport aux autres outils de modélisation, ADT met à disposition le langage KM3 (*Kernel MetaMetaModel*) qui est une forme textuelle simplifiée d'EMOF permettant de décrire des métamodèles et des modèles. En effet, ADT supporte les modèles décrits dans différents dialectes d'XMI, qui peuvent par exemple être décrits au format KM3 et transformés au format XMI voulu.

ATL est défini par un métamodèle MOF pour sa syntaxe abstraite et possède une syntaxe concrète textuelle. Pour accéder aux éléments d'un modèle, ATL utilise des requêtes sous forme d'expressions OCL. Une requête permet de naviguer entre les éléments d'un modèle et d'appeler des opérations sur ceux-ci. Une règle déclarative d'ATL, appelée M*atched Rule*, est spécifiée par un nom, un ensemble de patrons sources *(InPattern*) mappés avec les éléments sources, et un ensemble de patrons cibles (*OutPattern*) représentant les éléments créés dans le modèle cible. Depuis la version 2006 d'ATL, de nouvelles fonctionnalités ont été ajoutées telles que l'héritage entre les règles et le multiple *pattern matching* (plusieurs modèles en entrée). Le style impératif d'ATL est supporté par deux constructions différentes. En effet, on peut utiliser soit des règles impératives appelées C*alled Rule*, soit un bloc d'instructions impératives (*ActionBlock*) utilisé avec les deux types de règles. A l'image d'une méthode d'une classe, une *Called Rule* est appelée explicitement en utilisant son nom et en initialisant ses paramètres. La Figure II.9 ci-dessous présente un extrait du métamodèle ATL qui décrit la structure d'une règle ATL.

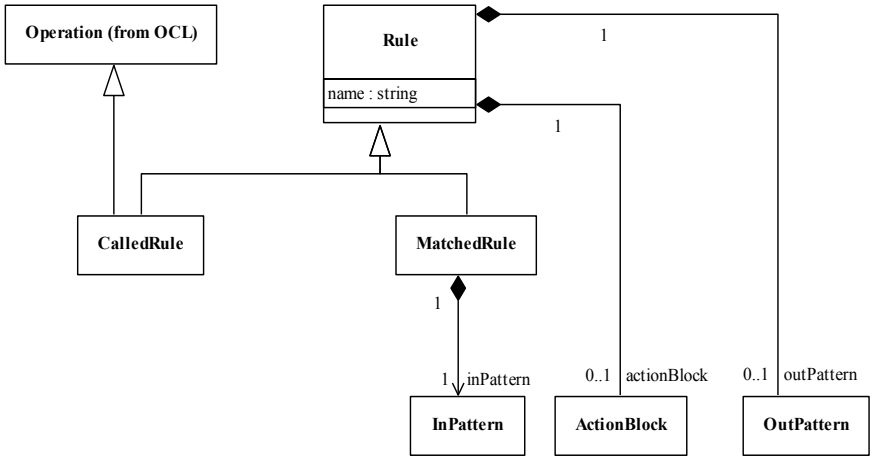


Figure II.9 Extrait du métamodèle ATL [Bézivin, 2003b]

ATL supporte deux modes d'exécution différents : le mode standard et le mode par raffinement. Dans le mode standard, les éléments sont créés seulement quand les patterns sources définis dans les règles déclaratives ont été reconnus dans le modèle ; le système instancie alors les éléments du pattern cible. Une fois l'étape d'instanciation passée, un lien de traçabilité est créé, et associe chaque élément reconnu du modèle source à un élément du modèle cible. Finalement, le système évalue ces liens de traçabilité afin de déterminer les valeurs des propriétés des éléments instanciés. Dans le mode par raffinement, les éléments dont les patterns sources n'ont pas été *matchés* par les règles sont automatiquement copiés dans le modèle cible par le moteur d'exécution. Ceci réduit considérablement l'effort nécessaire au développement de transformations destinées à modifier une petite partie d'un modèle en gardant le reste inchangé.

II.3.3.4. Les outils génériques

Un outil générique permet la transformation de n'importe quel modèle pourvu qu'il soit exprimé dans le format requis. Dans cette catégorie on retrouve les outils de la famille XML (*XLST* [W3C-XLST, 1999], *Xquery* [W3C-Xquery, 2007]) et les outils de transformation de graphes (*AGG* [AGG-website, 2007], *UMLX* [Willink, 2003], *GreAT* [Agrawal et al., 2005]) qui sont principalement utilisés dans le monde académique.

Le langage Xquery est une spécification du W3C qui permet de faire des requêtes sur les documents XML. Il représente pour les documents XML ce que SQL représente pour les bases de données relationnelles. Le langage XLST (eXtensible Stylesheet Language Transformations) est un dialecte du langage XML qui permet de faire une transformation d'un document XML vers un format de type texte. Les outils de la famille XML ont atteint un certain degré de maturité du fait d'une large diffusion dans le monde XML.

II.3.3.5. Les outils intégrés aux AGL

L'objectif de cette catégorie d'outils est d'intégrer un moteur de transformation comme fonctionnalité supplémentaire dans les Ateliers de Génie Logiciel (AGL). Cette intégration permet non seulement d'utiliser les AGL pour décrire les modèles d'un système mais aussi procéder à leurs transformations. Dans cette catégorie d'outils nous retrouvons les outils commerciaux (*Objecteering 6/MDA Modeler* [Softeam-website; Softeam, 2008; Modelio-website] et *IBM Rational Software Modeler* [RSM-website]) et certains outils du monde académique parmi lesquels *OptimalJ* [Compuware-website], *FUJABA* [Burmester et al., 2004]. Dans la suite, nous présentons essentiellement les outils Objecteering MDA Modeler, RSM et FUJABA qui sont représentatifs de cette catégorie d'outils.

II.3.3.5.1. Objecteering MDA Modeler

Objecteering MDA Modeler [Softeam-website; Softeam, 2008] est un outil puissant commercialisé par Objecteering Software. Il permet aux utilisateurs d'élaborer des modèles et d'appliquer des opérations de production sur ceux-ci. Les utilisateurs sont généralement des ingénieurs qualité, des architectes, des ingénieurs méthode ou des chefs de projet (voir Figure II.10). L'objectif est de capitaliser un savoir-faire identifié d'une entreprise à travers des composants MDA afin que ces derniers puissent être exploités directement par d'autres outils tels qu'UML Modeler. Dans [Softeam, 2008], un composant MDA est un ensemble fonctionnel autonome apportant des services et des extensions à un modeleur UML pour en étendre ses capacités selon le domaine couvert par ce composant. Un composant MDA est constitué des éléments suivants : un ensemble de *profils*, des *éléments IHM*, un *modèle Java*, un projet de *test* et un projet *First-Steps*.

MDA Modeler utilise une approche par programmation pour définir des opérations de transformation de modèles. Dans sa dernière version, MDA Modeler est accessible depuis Eclipse ce qui permet de programmer en Java des transformations de modèles par le biais des classes d'implémentation associées à chaque élément d'UML. L'architecture de MDA Modeler ne supporte que des transformations de modèles endogènes (modèle UML vers modèle UML). La création d'une transformation de modèles se fait dans MDA Modeler par le biais de la création de profils UML.

MDA Modeler offre un support graphique de modélisation des profils, représentant les métaclasses référencées, les stéréotypes utilisés et leurs propriétés. La définition d'une transformation de modèles nécessite de référencer toutes les métaclasses UML concernées par la transformation. Ces références contiennent le code de transformation qui sera en langage J ou Java selon le cas. Une fois la transformation implémentée, il est possible de lui associer une commande sous forme de menu utilisateur dans un composant MDA. Lorsque le composant sera déployé dans UML Modeler, les développeurs, grâce à la commande associée à la transformation pourront exécuter la transformation.

En résumé nous pouvons dire que MDA Modeler offre des mécanismes permettant de définir des opérations sur les modèles et ainsi de les rendre productifs. La Figure II.10 ci-dessous résume les principes des outils MDA Modeler et UML Modeler.

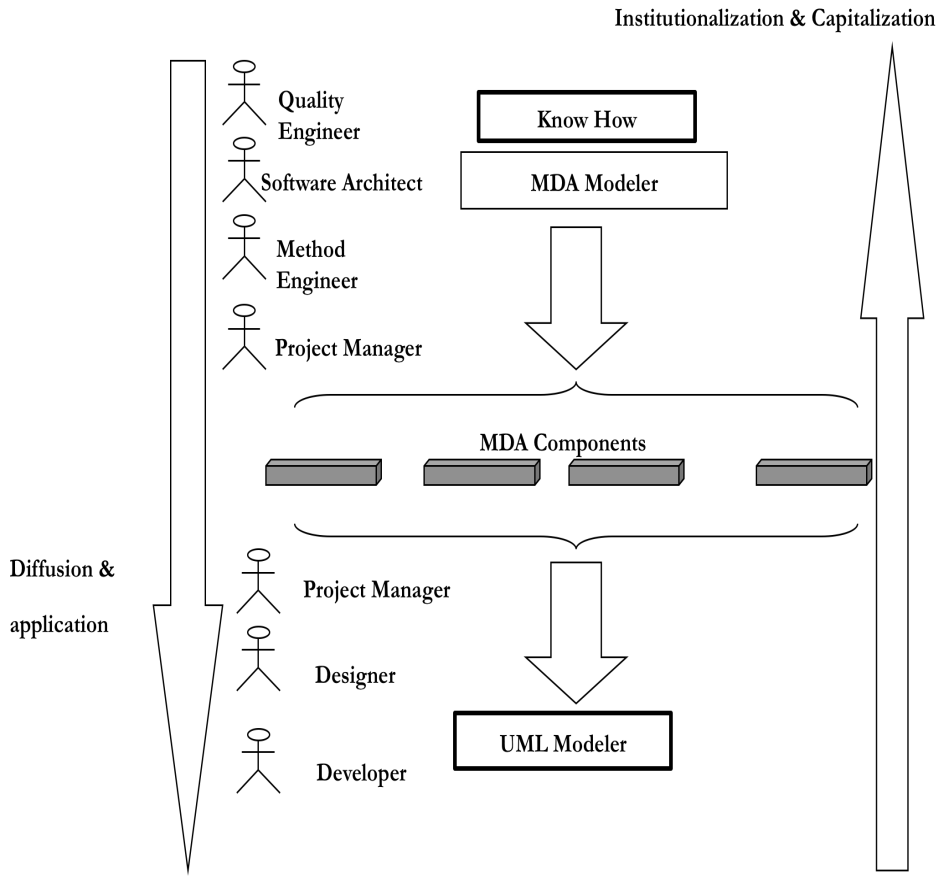


Figure II.10 Principes de MDA Modeler et UML Modeler [Softeam, 2008]

Depuis 2009, Modelio [Modelio-website] est le successeur de la plate-forme Objecteering MDA Modeler. C'est un outil qui suit l'approche MDA en permettant de faire des transformations PIM vers PSM. De plus, Modelio assiste les concepteurs UML en leur offrant un support pour l'analyse des exigences, l'analyse des objectifs, et l'analyse des règles métiers.

II.3.3.5.2. IBM Rational Software Modeler (RSM)

RSM [RSM-website] permet à ses utilisateurs de définir des transformations de modèles exogènes ou endogènes. Après la création d'un projet de transformation de modèles, la définition d'une transformation de modèles passe par la spécification de l'ensemble des règles qui composent la transformation. Chaque règle porte sur un unique élément d'UML – par exemple une classe, un paquetage – et dispose de sa propre classe d'implémentation où seront codées en Java les transformations effectuées sur les modèles. Après avoir défini intégralement la transformation par le biais de règles utilisant des classes d'implémentation, RSM peut packager le projet de transformation sous forme de plug-in afin de le rendre utilisable dans n'importe quel projet de modélisation.

II.3.3.5.3. FUJABA

FUJABA [Burmester et al., 2004], l'acronyme de *From UML to Java and Back Again*, a pour but de fournir un environnement de génération de code Java et de rétro-conception. Il utilise UML

comme langage de modélisation. Durant la dernière décennie, l'environnement de FUJABA est devenu une base pour beaucoup d'activités de recherche notamment dans le domaine des applications distribuées, les systèmes de bases de données ainsi que dans le domaine de la modélisation et de la simulation des systèmes mécaniques et électriques. Ainsi, l'environnement de FUJABA est devenu un projet open-source qui intègre les mécanismes de l'IDM.

II.3.3.5.4. Synthèse

L'intérêt des outils intégrés aux ateliers de génie logiciel est d'une part, leur maturité et, d'autre part, leur excellente intégration dans l'atelier qui les héberge. L'inconvénient en est que la majeure partie d'entre eux intègre des langages de transformation propriétaires qui s'adaptent difficilement à l'évolution de l'IDM. Ces outils montrent aussi leur limite lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir sur une longue période. *MDA Modeler* ne peut faire que des transformations de modèles endogènes (UML vers UML). *RSM* est limité s'il s'agit de faire une transformation multiple de modèles (M vers N). En outre *FUJABA*, même s'il présente des idées novatrices, a besoin d'être renforcé notamment dans le cadre du reverse-engineering qui constitue un thème de recherche prometteur de l'IDM.

II.3.3.6. Les outils de métamodélisation

La dernière catégorie des outils de transformation de modèles est celle des outils de métamodélisation dans lesquels la transformation de modèles revient à l'exécution d'un métaprogramme (un programme qui porte sur les métamodèles). Parmi ces outils nous pouvons citer *Kermeta* [Triskell, 2005 ; Jézéquel et al., 2005 ; Muller et al., 2005] de l'IRISA-INRIA Rennes, *MetaEdit+* [Tolvanen et al., 2003] de MetaCase : un outil qui permet de définir explicitement un métamodèle et au même niveau de programmer tous les outils nécessaires, allant des éditeurs graphiques aux générateurs de code en passant par des transformateurs de modèles), *XMF-Mosaic* [Xactium-website] de la société Xactium, *EMF/Ecore* [Eclipse-EMF-website] de la fondation Eclipse, l'outil *TOPCASED* [Farail et al, 2006 ; TOPCASED-website] : un atelier de développement open source, *OpenEmbedded* : une plate-forme IDM. Dans la suite de cette section, nous présentons essentiellement *Kermeta*, *XMF-Mosaic*, *EMF/Ecore* et *TOPCASED* qui sont représentatifs de cette catégorie d'outils de métamodélisation, et qui font l'objet de nombreuses expérimentations dans la communauté IDM.

II.3.3.6.1. Kermeta

Dans l'approche par programmation classique, on dit qu'un programme est composé de structures de données combinées à des algorithmes. Une description en Kermeta [Triskell, 2005 ; Jézéquel et al., 2005 ; Muller et al., 2005] est assimilable à un programme issu de la fusion d'un ensemble de métadonnées (EMOF) et du métamodèle d'action AS (Action Semantics) qui est maintenant intégré dans UML Superstructure. Le langage Kermeta est donc une sorte de dénominateur commun des langages qui coexistent actuellement dans le paysage de l'IDM. Ces langages sont les langages de métadonnées (EMOF, ECORE, etc.), de transformation de modèles (QVT, ATL, etc.), de contraintes et de requêtes (OCL), et d'actions (Action Semantics, Xion).

Kermeta est un véritable langage de métamodélisation exécutable. En effet, le métamodèle de Kermeta est composé de deux packages, *Core* et *Behavior* correspondant respectivement à EMOF et à un ensemble de métaclasses représentant des expressions impératives. Ces expressions seront utilisées pour programmer la sémantique comportementale des métamodèles. En effet, chaque concept d'un métamodèle Kermeta peut contenir des opérations, le corps de ces opérations étant composé de ces expressions impératives (*package Behavior*). Il est possible ainsi de définir la structure et la sémantique opérationnelle d'un métamodèle, rendant exécutables les modèles qui s'y conforment. Kermeta est intégré à l'environnement d'Eclipse sous la forme d'un plug-in et, grâce aux outils de génération *Ecore2Kermeta*, on peut traduire un métamodèle Ecore en un squelette de code Kermeta. Ce squelette servira de base pour le codage des méta-opérations.

A l'image d'UML 2.2 Infrastructure, Kermeta intervient à deux niveaux dans l'architecture à quatre niveaux de l'OMG. D'une part, il intervient comme un langage de niveau M3 c'est-à-dire que tous les métamodèles lui sont conformes, mais également comme une bibliothèque de base pour construire les métamodèles de niveau M2.

II.3.3.6.2. XMF-Mosaic

XMF-Mosaic [Xactium-website] est un outil destiné à développer des langages de modélisation et à déployer les outils nécessaires incluant éditeurs, analyseurs de code et transformateurs. Dans XMF-Mosaic, les métamodèles représentent des langages spécifiques à un domaine et sont appelés modèles de domaine. XMF-Mosaic est régi par un langage appelé XOCL (*eXtensible Object Constraint Language*) qui est un langage basé sur XML pour représenter les contraintes OCL dans les modèles UML. Parallèlement, l'environnement fournit un ensemble d'outils graphiques qui permettent de créer des métamodèles, de construire des modèles correspondants et de décrire des transformations. Pour la transformation de modèles, XMF-Mosaic propose le langage *Xmap*. A l'aide de ce langage, on peut définir un ensemble de règles de transformation définis chacun entre un ou plusieurs éléments d'un métamodèle source et un élément d'arrivée du métamodèle cible. L'environnement de XMF-Mosaic dispose d'un outil graphique pour définir le squelette d'une règle de transformation qui permet d'indiquer les métaclasses de départ, la métaclasse d'arrivée et de définir des dépendances entre règles.

II.3.3.6.3. EMF

EMF [Eclipse-EMF-website], qui signifie *Eclipse Modeling Framework*, est un environnement de modélisation et de génération de code qui facilite la construction d'outils et d'applications basées sur des modèles de données structurées. Il est à la base de nombreux outils IDM. La rencontre de l'IDM et du phénomène Eclipse a donné lieu à de nombreux projets tels que : *EMP* (Eclipse Modeling Project) qui met l'accent sur l'évolution et la promotion des technologies de développement basées sur les modèles dans la communauté Eclipse, *MoDisco* qui fournit une plate-forme extensible pour développer des outils IDM. *EMP* est un package qui regroupe plusieurs projets qui traitent de la modélisation afin d'unifier les visions et d'améliorer l'interopérabilité. Ces projets sont : *EMF*, *GMF* (Graphical Modeling Framework) qui permet de développer des générateurs d'éditeurs graphiques de modèles, *UML 2*, et *GMT* (Generative Modeling Tools).

Le métamodèle *Ecore* d'EMF sert de pivot et permet donc l'interopérabilité entre les outils IDM. EMF permet aussi le développement rapide et l'intégration de nouveaux plug-ins Eclipse. EMF est composé lui-même d'un ensemble de briques appelées plug-ins. Parmi ces plug-ins nous citons :

– le métamodèle *Ecore* qui est un canevas de classes pour décrire les modèles *EMF* et manipuler les référentiels de modèles ;

– *EMF.Edit* est un canevas de classes pour le développement d'éditeurs arborescents de modèles EMF ;

– le Modèle de génération *GenModel* permet de personnaliser la génération de code Java ;

– *JavaEmitterTemplate* est un moteur de template générique ;

– *JavaMerge* est un outil de fusion de code Java.

II.3.3.6.4. TOPCASED

TOPCASED (Toolkit in Open-source for Critical Application and SystEms Development) [Farail et al, 2006 ; TOPCASED-website] a pour but de fournir un atelier basé sur l'IDM pour le développement des systèmes logiciels et matériels embarqués. Les autorités de certification pour les domaines d'application de TOPCASED (aéronautique, espace, automobile, etc.), imposent des contraintes de qualification fortes pour lesquelles les approches formelles (analyse statique, vérification du modèle, preuve) sont nécessaires. L'outillage TOPCASED a pour principal objectif de simplifier la définition de nouveaux DSLs (Domain Specific Languages) ou de langages de modélisation en fournissant des technologies de niveau méta telles que des générateurs d'éditeurs syntaxiques (textuels et graphiques), des outils de validation statique et d'exécution de modèle, ce qui lui confère les atouts d'un véritable outil de métamodélisation.

II.3.3.6.5. Synthèse

Les outils de métamodélisation proposent des technologies qui permettent de faire des transformations impératives (cas de Kermeta), d'engendrer la syntaxe concrète d'un langage à partir de sa syntaxe abstraite (cas de TOPCASED et de l'outil OpenEmbedded), et qui permettent l'interopérabilité entre les outils IDM (cas de l'environnement de modélisation EMF). Ces outils permettent aujourd'hui de créer un DSML (Domain Specific Modeling Language) exécutable ou non et ses outils associés à moindre coût et dans un délai relativement court.

II.3.3.7. Traçabilité des transformations

La traçabilité a souvent été réduite au suivi des besoins du client dans le cycle de développement des logiciels. Mais avec l'avènement de l'IDM la notion de traçabilité prend une autre dimension ; on parle ainsi traçabilité des transformations. Dans la littérature nous trouvons plusieurs définitions de cette notion de traçabilité [Amar et al., 2011 ; Aizenbud-Resher et al., 2005; Vanhooff et al., 2005; Frédéric Jouault, 2005; Czarnecki et al., 2003]. Cependant, en examinant de prés ces définitions, on s'accorde à dire que la traçabilité d'une transformation est un mécanisme d'enregistrement des liens (traces) entre éléments de modèles sources et éléments de modèles cibles.

Dans certaines approches, un modèle de traçabilité conforme à un métamodèle de traces est décrit ou généré automatiquement afin de pérenniser les traces d'exécution d'une transformation.

Il existe deux approches pour concevoir la traçabilité : approche sans modèle de traces, et approche avec modèle de traces. La première approche implémentée par exemple dans [Aizenbud-Resher et al., 2005; Vanhooff et al., 2005] consiste à exprimer directement les relations entre éléments sources et cibles soit dans le modèle source, soit dans le modèle cible. La deuxième approche implémentée par exemple dans ATL et QVT consiste à générer ou à décrire un modèle de trace qui est conforme à un métamodèle de traces. Cette dernière favorise la pérennité des traces, la séparation des préoccupations, et permet de découpler le code de la transformation du code de la traçabilité. Par exemple il est possible de générer automatiquement un modèle de traces d'une transformation décrite avec le langage *Relation* de QVT. Le langage *Core* de QVT fournit à l'utilisateur un mécanisme pour définir son modèle de trace, mais n'assure pas une gestion automatique de celui-ci. Enfin, dans le langage Operationnal Mappings de QVT, la gestion de la trace est entièrement à la charge de l'utilisateur.

II.3.4. Discussion

Le développement logiciel dirigé par les modèles a pour principal objectif de concevoir des applications en séparant les préoccupations et en plaçant les notions de modèles, métamodèles et transformations de modèles au centre du processus de développement. Dans cette section consacrée à l'IDM, nous avons donc particulièrement mis l'accent sur les concepts, langages et outils associés à la transformation de modèles – paradigme central de l'IDM–. Cette focalisation sur la notion de transformation ne doit pas faire oublier qu'une transformation étant in fine un programme exécutable, toutes les problématiques liées au développement du logiciel (test, vérification, traçabilité, etc.) peuvent s'appliquer à elle. De même, les modèles étant représentables par des graphes (généralement enrichis), les résultats de recherche théoriques sur les graphes sont de plus en plus utilisés pour modéliser et prouver des propriétés sur les modèles.

Cette étude bibliographique sur l'IDM a montré sa rapide évolution et l'engouement des laboratoires académiques et industriels pour cette discipline. L'IDM reste fortement marquée par les propositions de l'OMG (standards, initiative MDA) même si chacun reconnaît aujourd'hui que le développement à base de modèles ne peut pas se cantonner à une vision orientée UML. De fait, nous constatons aujourd'hui que l'IDM à langage unique (UML) perd sa position centrale au profit de l'IDM à langages multiples (approche DSL). Cette situation a pour conséquence le développement de nombreux langages spécifiques (par exemple pour l'expression des contraintes, les actions, les transformations, la sérialisation, etc.). Il en résulte par contre une délicate problématique d'intégration de ces langages qui tendent à disperser les concepts de manipulation de modèles. Il faut donc doter l'IDM de moyens pour contrôler la définition et la fusion de ces langages et créer une synergie entre les différents groupes de recherche dans ce domaine. L'action IDM [Action IDM-website] et plusieurs conférences (telles que MODELS [MODELS-website], EC-MFA, IDM, etc.) ont été ainsi créées pour offrir un cadre d'échange à la communauté IDM.

Enfin, l'un des défis majeurs de l'IDM dans les prochaines années est sans aucun doute le passage à l'échelle et son industrialisation, la notion de méga-modèles et de modèles infinis (modèles de grande taille), et le développement collaboratif de modèles [Sriplakich, 2007]. Il s'agit entre autres de développer des plates-formes supportant des processus de développement IDM collaboratifs et

certifiés pour des logiciels éventuellement critiques, tels que les systèmes embarqués. Parmi les axes dans lesquels notre équipe est d'ores et déjà impliquée, nous pouvons citer : le passage à l'échelle de l'IDM avec des modèles nombreux et de grande taille (prise en compte des points de vue des utilisateurs [Anwar et al., 2010]), la modélisation et la mise en œuvre automatisée de processus IDM collaboratifs [ANR-GALAXY-website], la traçabilité des transformations et la co-évolution de modèles [Amar et al., 2011], la définition et l'application de patrons orientés IDM, la génération par transformations de composants exécutables multi-cibles, etc.

II.4. Exemples de Processus IDM

Cette section présente des exemples de processus IDM [Larrucea et al., 2004 ; Fondement et al., 2004 ; Rios al, 2006 ; Stahl et al., 2006]. Dans cette section, nous avons cherché à focaliser notre attention sur des exemples représentatifs de processus IDM liés à des domaines d'application divers. En se référant à [Kleppe et al., 2003], un processus IDM appelé aussi *processus de développement dirigé par les modèles* est défini comme suit : « a process of developing software using different models on different levels of abstraction with (automated) transformations between these models ». Donc un *processus IDM* peut être vu comme un enchaînement de transformations (automatiques) de modèles, chaque transformation consommant un ou plusieurs modèles sources et produisant un ou plusieurs modèles cibles. L'un des premiers processus IDM explicité est apparu avec l'initiative MDA de l'OMG qui décrit un processus général de transformations de modèles qui peut être appliqué à n'importe quel domaine d'application (voir section II.3.2.2.4 dédiée à l'approche MDA). A partir de l'initiative MDA, d'autres processus IDM furent proposés. On peut citer par exemple un certain nombre de processus qui sont dédiés aux applications web [Koch, 2006], aux services d'architectures [Maciel et al., 2006], aux systèmes d'apprentissage en ligne (E-learning) [Cong et al., 2010], à la composition de modèles [Anwar et al., 2008], le processus MoPCoM [Koudri, 2010], et OpenUP/MDD une version du processus unifié dédié au développement IDM [OpenUP/MDD-website].

II.4.1. Le Processus UWE dédié aux applications web

L'objectif du processus UWE (*UML-based Web Engineering*) [Koch, 2006] est d'offrir aux développeurs un support systématique et semi-automatique pour le développement des applications web basé sur les modèles et leurs transformations. Le processus UWE couvre tout le cycle de développement des systèmes web : des exigences au code exécutable. UWE est un processus IDM qui suit les principes de l'approche MDA. Le processus est composé d'un ensemble de modèles et de transformations de modèles spécifiés par des métamodèles et des langages de transformation de modèles. Les métamodèles sont le métamodèle *WebRE* (Web Requirements Engineering) [Escalona et al, 2006] dédié à la modélisation des exigences web, le métamodèle *WebSA* (Web Software Architecture) [Melià et al., 2005] dédié à la modélisation des architectures web, et le métamodèle *UWE* [Kroiß et al., 2008] qui décrit la structure et le comportement des applications web. Le métamodèle UWE inclut « *Content Metamodel* » qui correspond à la partie du métamodèle d'UML qui décrit les diagrammes de classes, « *Navigation Metamodel* » qui décrit les aspects relatifs à la navigation dans les applications web et « *Presentation Metamodel* » qui décrit les aspects relatifs à la présentation dans les applications web.

Le processus UWE (voir Figure II.11 ci-après) démarre avec la définition du modèle des exigences (*Requirement Models*) de type CIM. Deux ensemble de modèles de type PIM sont dérivés à partir des exigences : les modèles fonctionnels (*Functional Models*) qui représentent les différentes préoccupations des systèmes web (contenu, la navigation, la logique métier, la présentation, et l'adaptation) ; et les modèles architecturaux (*Architecture Models*) qui représentent les caractéristiques architecturales des systèmes web. La transformation entre les modèles fonctionnels est une transformation composite, chaque sous-transformation traitant d'une préoccupation séparée de l'ingénierie web. Les modèles fonctionnels sont par la suite intégrés dans un modèle appelé « *BigPicture Model* » pour une vérification. Une fusion entre le modèle « *BigPicture* » et les modèles architecturaux (*Architecture Models*) produit un modèle intégré (*Integration Model*) qui couvre les aspects fonctionnels et architecturaux. Les modèles spécifiques aux plates-formes ciblées (*modèle J2EE* ou un *modèle .NET*) sont dérivés à partir du modèle intégré. Finalement du code exécutable est généré à partir des modèles spécifiques aux plates-formes J2EE et .NET.

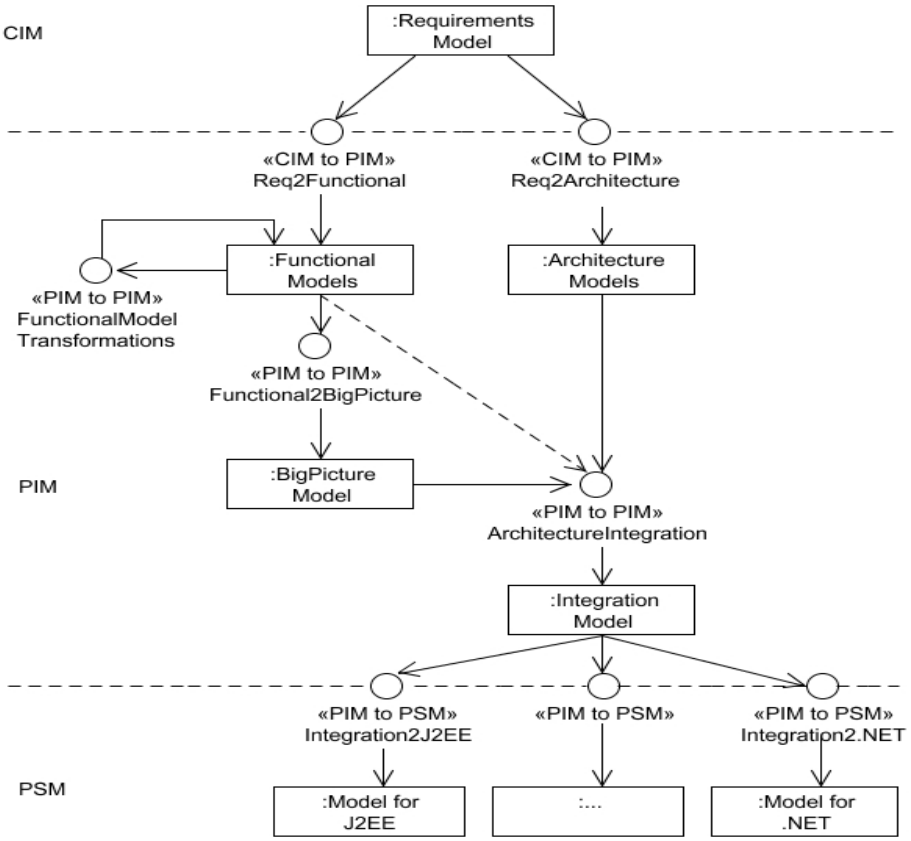


Figure II.11 Vue d'ensemble du processus UWE [Koch, 2006]

II.4.2. Processus IDM dédié aux services d'architectures (Middleware Services)

Le processus de développement dédié aux services d'architectures [Maciel et al., 2006] a pour but de faciliter le développement des applications distribuées. Il est basé sur MDA, utilise les concepts EDOC (Enterprise Distributed Object Computing) [OMG-EDOC, 2002] et est supporté par un outil de transformation appelé *TRANSFORMS* pour automatiser certaines étapes du processus. La construction d'une application distribuée basée sur MDA débute avec la définition d'un

modèle de type PIM indépendant d'une architecture. Par la suite, un modèle de type PSM est défini selon l'architecture sur laquelle l'application va être implémentée. Les PIMs sont décrits grâce au profil EDOC et plusieurs PSM peuvent être générés à partir d'un PIM. Le processus produit trois catégories de modèles : le *modèle du domaine* (Domain Model), le *modèle de conception* (Design Model) et le *modèle opérationnel* (Operational Model).

Le *modèle du domaine* correspond au contexte dans lequel le service doit être appliqué. La portée et les responsabilités de l'application sont définies dans ce modèle via les exigences fonctionnelles de l'application. Le service offert et les informations manipulées par l'application sont définis dans ce modèle. Ce modèle donne une vue informationnelle (*Information View*) et organisationnelle (*Enterprise View*) de l'application.

Le *modèle de conception* décrit une vue relative à l'informatisation (*Computational View*) du modèle du domaine. Ce modèle indépendant de toute plate-forme identifie l'interface des services et les composants qui fournissent les exigences établies dans le *modèle du domaine*. Les exigences non-fonctionnelles sont intégrées dans le *modèle de conception* qui forme avec le *modèle du domaine* le PIM de l'application.

Le *modèle opérationnel* est un modèle d'exécution qui décrit l'environnement d'exécution de l'application sur la plate-forme cible. Les caractéristiques de la plate-forme cible sont intégrées dans le *modèle opérationnel* à travers la vue relative à l'ingénierie de l'application (*Engineering View*). Un mapping du *modèle de conception* et du modèle décrivant la vue relative à l'ingénierie de l'application produit le PSM de l'application qui représente la vue technique (*Technology View*). La Figure II.12 ci-dessous donne la structure de ce processus.

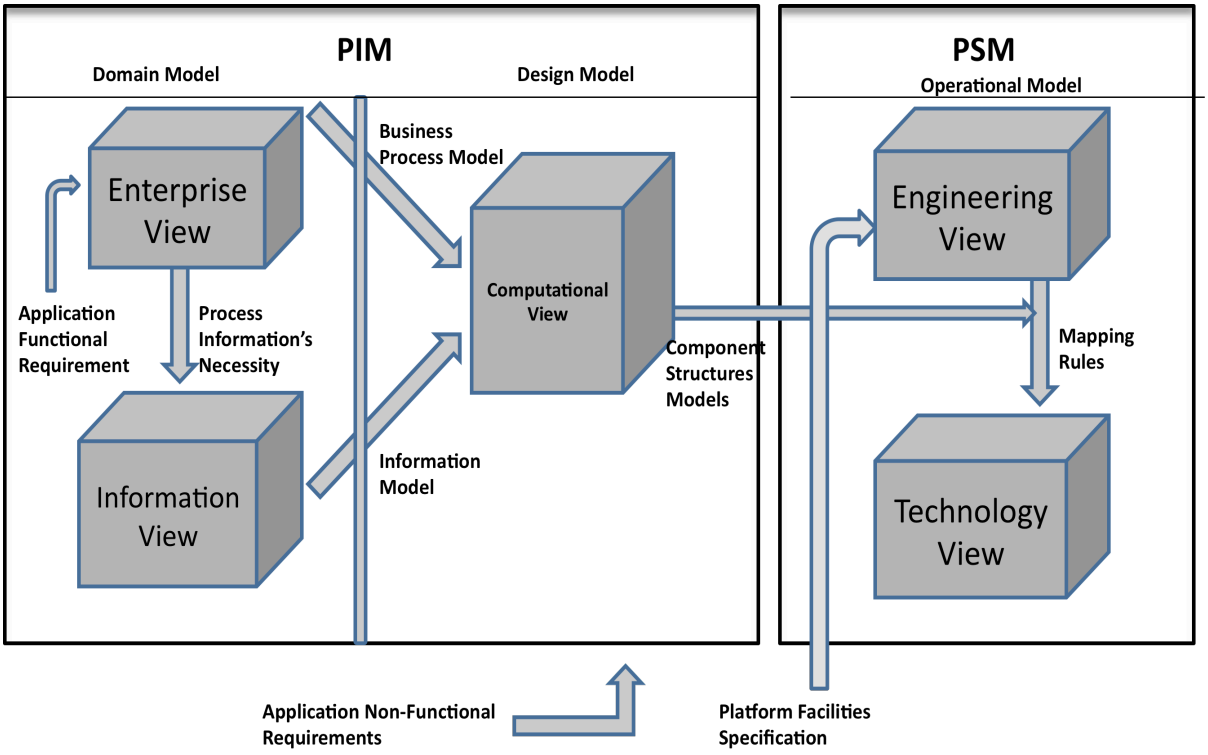


Figure II.12 Processus dédié aux services d'architectures [Maciel et al., 2006].

II.4.3. Processus IDM pour le développement de systèmes d'apprentissage en ligne

Le processus de développement de la plate-forme E-learning décrite dans [Cong et al., 2010] est réalisé par une série de modèles et de transformations de modèles qui permettent de raffiner le modèle des exigences jusqu'à obtenir du code executable pour la plate-forme ciblée. La Figure II.13 ci-dessous donne une vue générale de ce processus. Le processus démarre avec la phase « Définition des exigences » (*Requirements Definition*) qui permet de produire les diagrammes de paquetages et de cas d'utilisation du système. Ces diagrammes décrivent les modèles de type CIM de la plate-forme E-learning. Une analyse de robustesse (*Robustness Analysis*) permet d'affiner le modèle de cas d'utilisation produit dans la phase précédente. Au cours de la conception générale (*Conceptual Design*), les modèles de type PIM sont construits. Ces modèles décrivent la vue statique et dynamique du système cible respectivement à travers les diagrammes de classes et de séquence. A partir des PIMs, un mapping de PIM vers PSM au cours de la phase de conception détaillée (*Detailed Design*) produit le modèle PSM de la plate-forme ciblée. Finalement, la phase d'implémentation (*Implementation*) permet de générer le code de plate-forme ciblée à travers une transformation PSM vers Code.

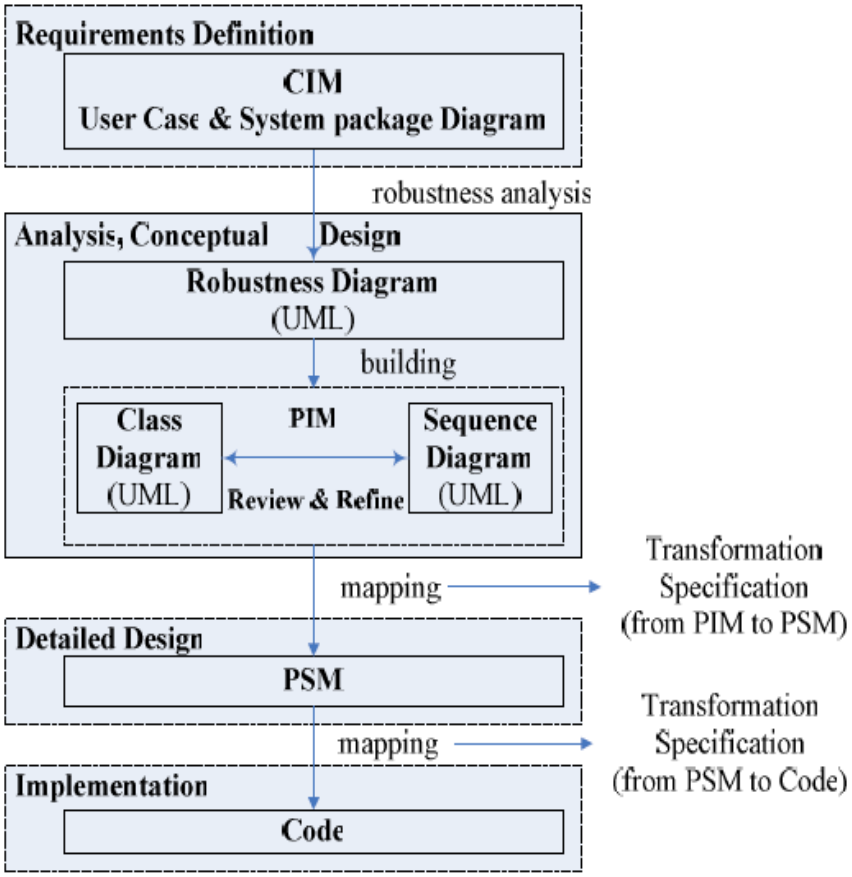


Figure II.13 Le processus IDM dédié à la plate-forme E-learning [Cong et al., 2010]

II.4.4. Processus VUML pour la composition de modèles structurels

VUML [Nassar, 2005 ; Anwar et al., 2010] est une approche de multi-modélisation en UML qui a été développée dans notre équipe. Elle est centrée sur la notion de point de vue et propose un

processus de composition de modèles structurels (voir Figure II.14) pour obtenir le modèle final. La démarche de VUML dans une vision IDM comporte quatre phases :

- *Phase 1 : Analyse Globale* : C'est la phase centralisée de la modélisation des exigences à travers les digrammes de cas d'utilisation UML ;
- *Phase 2 : Analyse et Conception par point de vue*. Dans cette phase, plusieurs concepteurs travaillent séparément pour réaliser des modèles de conception par point de vue appelés modèles partiels ;
- *Phase 3 : Composition*. On compose les modèles partiels de la phase précédente afin d'aboutir à un modèle final appelé *modèle VUML*. Le modèle VUML est un modèle indépendant de toute plate-forme (PIM). La composition structurelle comporte deux étapes. La première étape consiste à résoudre les conflits (homonymies, etc.). La deuxième étape, automatique, est réalisée par le biais de trois catégories de règles de transformation :
    - règles de correspondance qui définissent les correspondances entre les modèles de conception par point de vue **;**
    - règles de fusion qui fusionnent les modèles par le biais des règles de correspondance pour obtenir le modèle VUML ;
    - règles de translation qui copient les éléments des modèles qui n'ont pas été mis en correspondance dans le modèle VUML ;
- *Phase 4 : Application d'un patron de conception* sur le modèle VUML pour produire le modèle d'implémentation.

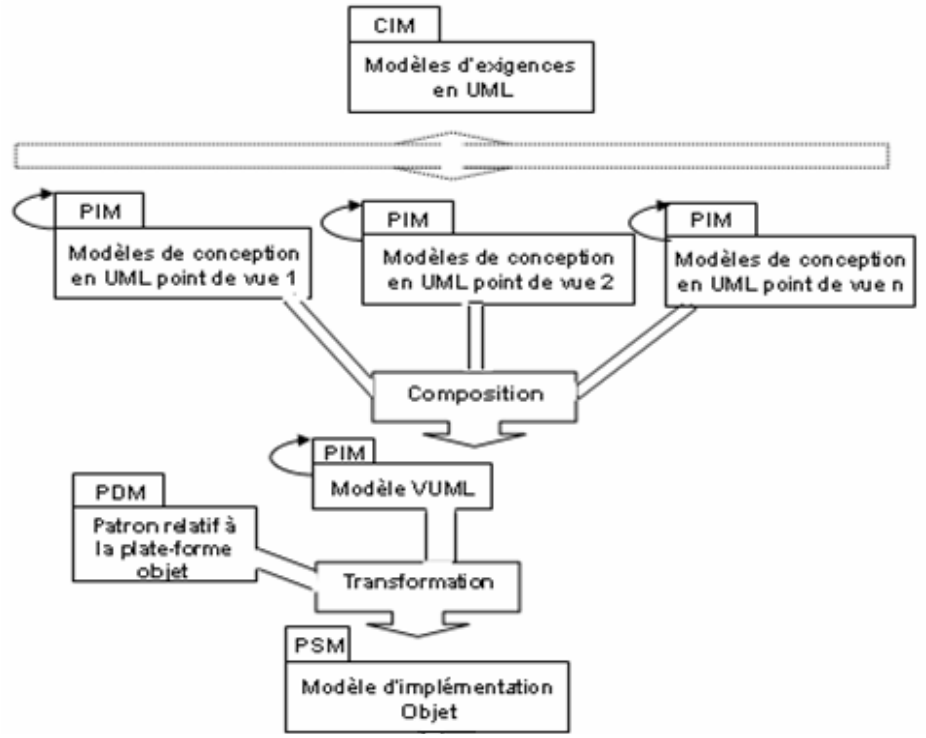


Figure II.14 Processus de composition structurelle dans VUML [Anwar et al., 2010]

II.4.5. Le Processus MoPCoM

Le processus MoPCoM [Koudri, 2010] (voir Figure II.15) est un raffinement du processus dit en « Y » de MDA. Il vise à modéliser une plate-forme à travers trois phases d'analyse qui permettent de construire un système de manière itérative. Le processus MoPCoM démarre avec l'analyse des exigences fonctionnelles et non-fonctionnelles afin de définir les cas d'utilisation du système. Il est composé de trois phases (1) *MoC Analysis* qui correspond à la phase d'analyse du modèle de la plate-forme virtuelle appelé parfois modèle de calcul MoC (Model of Computation), (2) *Topology* & *Schedulability Analysis* qui correspond à la phase d'analyse de la topologie de la plate-forme d'exécution et de l'ordonnancement de ses processus ; et (3) *Cycle Accurate Analysis* qui correspond à la phase d'analyse précise et détaillée de la plate-forme d'exécution. Le processus MoPCoM est décrit aussi par trois niveaux d'abstraction correspondant chacun à une phase du processus :

- Le niveau AML (*Abstract Modeling Level*) décrit l'allocation des blocs fonctionnels sur une plate-forme virtuelle. L'allocation permet aussi de vérifier le modèle fonctionnel par rapport aux exigences non-fonctionnelles (allocation de ressources, communication, synchronisation etc.). Ce niveau définit aussi la communication entre les différents processus de la plate-forme virtuelle à travers un protocole abstrait de haut niveau (granularité large).
- Le niveau EML (*Execution Modeling Level*) définit l'allocation des blocs applicatifs sur les ressources de la plate-forme d'exécution. Ce niveau permet de déterminer l'adéquation entre les algorithmes décrits au niveau AML pour la plate-forme virtuelle et l'architecture physique de la plate-forme d'exécution. Il décrit aussi la topologie de la plate-forme d'exécution. Cette topologie regroupe les nœuds de calcul, de communication ou de mémorisation, interconnectés à travers des bus qui implémentent les protocoles abstraits de haut niveau décrits au niveau AML.
- Le niveau DML (*Detailed Modeling Level*) permet de raffiner la plate-forme d'exécution décrit au niveau EML jusqu'à obtenir une allocation satisfaisante pour la plate-forme détaillée. Ce niveau permet de mener des analyses précises et détaillées. Le code exécutable des différentes parties du matériel est dérivé du modèle d'allocation (Allocated Model) de ce niveau.

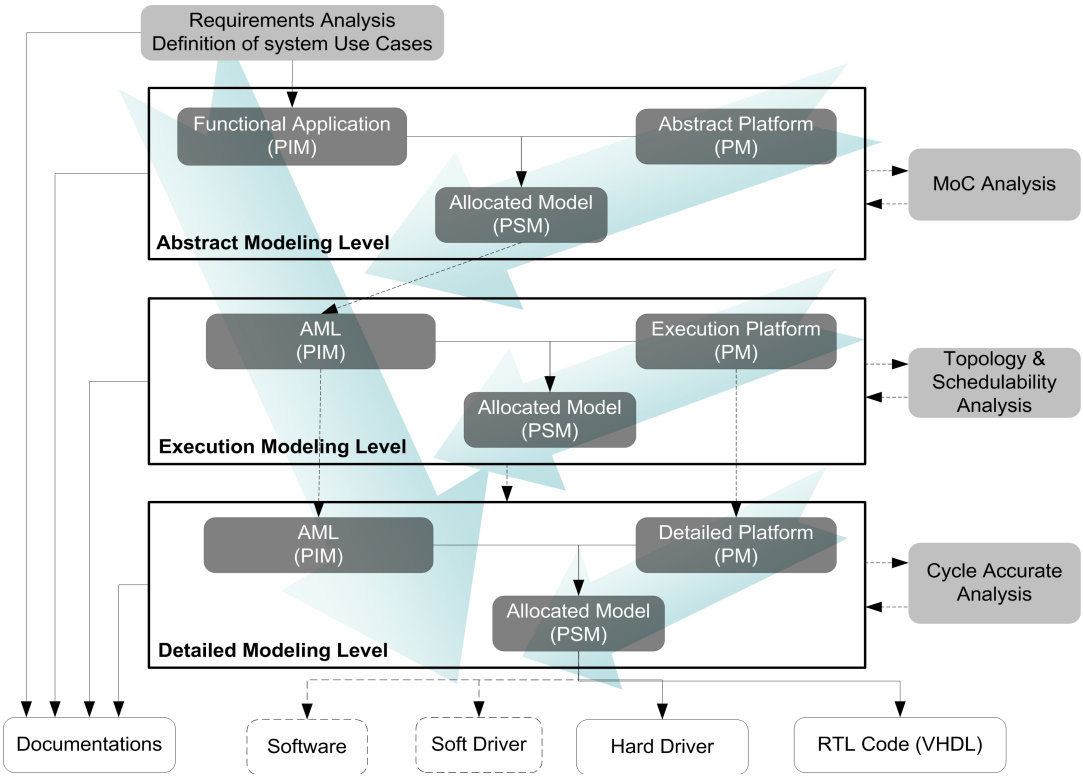


Figure II.15 Vue d'ensemble du processus MoPCoM [Koudri, 2010]

II.5. Les Langages de description de processus IDM

La section précédente a présenté des exemples représentatifs de processus IDM. Cependant, il n y a pas d'homogénéité dans la terminologie utilisée pour décrire ces processus ; chacun adopte son propre langage ou notation, et différents concepts sont utilisés pour décrire les activités et les artéfacts du développement logiciel. Dans [Humphrey et al., 1989] (traduction en français) : la modélisation de processus à travers une terminologie cohérente et unifiée doit permettre la communication, la compréhension, la réutilisation, l'évolution, la gestion, et la standardisation des processus. Le standard SPEM 2.0 de l'OMG dédié à la modélisation des processus logiciels et systèmes propose des concepts qui sont trop génériques pour décrire certains aspects des processus IDM. Plusieurs langages et formalismes ont été proposés pour modéliser les processus logiciels (voir section II.2.3). Cependant, à travers l'étude bibliographique que nous avons menée seules deux approches tentent d'expliciter dans leurs métamodèles les concepts de base de l'IDM (Transformation, modèle, métamodèle, outil IDM). Ces deux approches que nous décrivons dans les sections qui suivent sont celles de Maciel [Maciel et al., 2009] et Poress [Porres et al, 2006].

II.5.1. L'approche de Maciel

L'approche de Maciel [Maciel et al., 2009] propose un langage de modélisation et de mise en œuvre des processus IDM. Ce langage est basé sur le standard SPEM et l'initiative MDA de l'OMG. Certains concepts du métamodèle de SPEM ont été spécialisés afin de fournir un langage capable de

décrire les processus basés sur MDA. Par exemple, le concept *WorkProduct* hérité de SPEM est spécialisé en cinq concepts :

(1) *UMLModel* qui est produit par un rôle ou par une transformation ;

(2) *TransformationRule* qui contient les règles qui permettent de faire d'une part une transformation d'un ou de plusieurs modèles UML sources vers un ou plusieurs modèles UML cibles, d'autre part la génération de code à partir d'un modèle UML ;

(3) *EtraModel* qui permet de représenter les documents du projet ;

(4) *Code* pour représenter le code généré durant le projet ;

(5) *Profile* pour représenter les profils UML.

Un processus est composé d'une séquence de phases, chaque phase comportant une ou plusieurs itérations. Les différentes phases dans cette approche sont : la modélisation des CIM, la modélisation des PIM, la modélisation des PSM, et enfin la génération de code. Une phase de modélisation CIM, PIM, ou PSM peut être basée sur un ou plusieurs profils UML. La Figure II.16 ci-aprés présente le métamodèle de l'approche de Maciel.

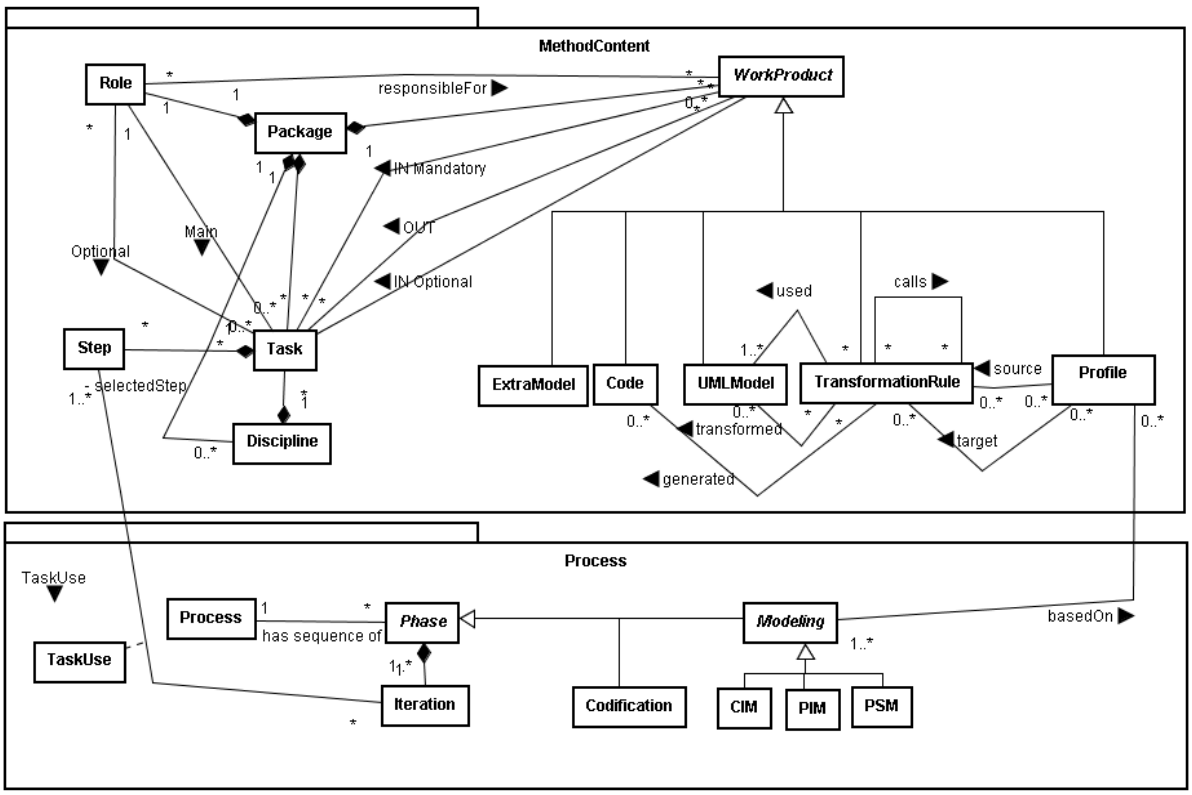


Figure II.16 Le métamodèle de l'approche de Maciel [Maciel et al., 2009]

L'approche de Maciel est supportée par un environnement de modélisation et de mise en œuvre appelé « *Transforms* » (voir Figure II.17) qui peut être utilisé pour instancier un processus MDA.

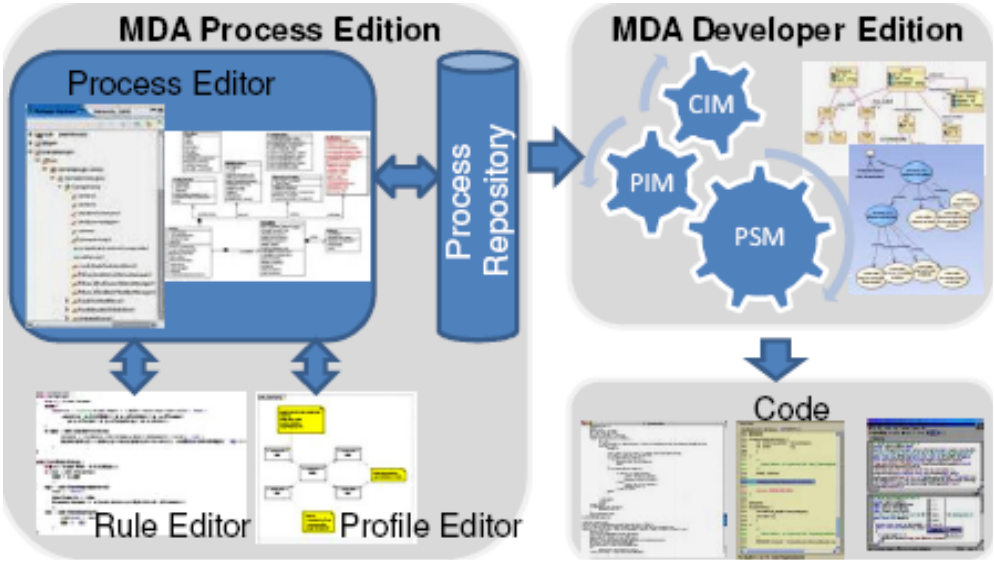


Figure II.17 Vue d'ensemble de l'environnement Transforms [Maciel et al., 2009]

L'environnement « *Transforms* » est divisé en deux modules :

- *MDA Process Edition* : ce module renferme un éditeur de processus MDA (P*rocess Editor*), un éditeur de règles de transformation (*RuleEditor*) et un éditeur de profils UML (*Profile Editor*) et une base pour stocker les processus édités (*Process Repository*).
- *MDA Developer Edition* : ce module à pour objectif d'instancier un processus MDA stocké dans le repository pour un projet particulier et par la suite procéder à son exécution. Dans ce module, une équipe de développement est assignée aux rôles décrits dans le processus et réalise les tâches définies dans chaque phase du développement.

II.5.2. L'approche de Porres

L'approche de Porrres [Porres et al, 2006] cible le développement des logiciels et systèmes utilisant les processus IDM. Le métamodèle proposé dans cette approche (voir Figure II.18) est défini sur la base de l'approche MDA, des diagrammes d'activités d'UML 2.2, du métamodèle SPEM, et de la sémantique des réseaux de Petri.

L'approche de Porres peut être considérée comme un complément du standard SPEM 2.0. Le métamodèle propose des concepts qui permettent de décrire à la fois un processus de développement et un projet basé sur une instance de ce processus.

Une activité hérite de « *WorkDefinition* » et de « Behavior » et est composé d'un ensemble de tâches individuelles (*Step*). L'activité est réalisée par une ressource qui peut être un outil de transformation ou un rôle. Une activité peut avoir une pré ou postcondition. Elle possède un ensemble de paramètres qui sont des modèles consommés, produits, ou modifiés durant le développement. Une phase et un cycle de vie sont des activités spéciales. Le responsable d'une activité est aussi le responsable des tâches qui composent l'activité.

Une tâche (*Step*) étant un « *Behavior* », elle a donc des paramètres et peut être associée à une pré ou postcondition. *InitialStep* et *FinalStep* sont des « *WorkDefinition* » qui représentent les nœuds de

contrôle spécifiques. *InitialStep* est la tâche initiale qui fournit les données qui déclenchent l'exécution d'un processus associé à un cycle de vie, alors que *FinalStep* est la tâche finale qui arrête tous les flots d'exécution d'un cycle de vie.

Le concept *Model* représente un artéfact du développement. L'approche de Porres distingue deux types de modèles : les modèles conformes à MOF (*MOF-compliant Model*) pour lesquels il est possible de définir des contraintes et les modèles non-interprétables (*UnInterpreted Model*). Les contraintes associées aux modèles MOF renseignent sur la validité de ces modèles, ce qui permet de savoir si une activité ou une tâche utilisant ces modèles peut être exécutée ou non. Un flot (*Flow*) est un arc direct qui permet de connecter un « *WorkDefinition* » à un modèle et vice-versa.

Un mapping de l'approche de Porres vers les réseaux de Petri permet de guider l'exécution et de suivre l'évolution du projet représenté par le réseau de Petri. Un « *WorkDefinition* » (Activity, Step, InitialStep, FinalStep) sera décrit par une transition dans le réseau de Petri. La tâche initiale (*InitialStep*) permet de démarrer le réseau de Petri (c'est-à-dire le projet) alors que la tâche finale (*FinalStep*) permet de supprimer tous les jetons du réseau de Petri, ce qui signifie que le projet est terminé. Les modèles manipulés par le processus sont des variables utilisées dans le réseau de Petri. Un modèle est dit actif (*Active*), lorsqu'il représente un paramètre d'entrée valide pour une activité ou une tâche. Un modèle est dit vivant (*Alive*), lorsqu'il est créé ou généré à la suite d'un déclenchement d'une activité ou d'une tâche. Une activité ou une tâche est dite activable ou éligible (*Enabled*) si un modèle actif est connecté à chacun de ses paramètres d'entrée. Une activité ou une tâche activable peut être sélectionnée par un membre de l'équipe de développement pour sa réalisation ou son déclenchement. Une activité ou une tâche non-activable (*Not-enabled*) n'est pas éligible.

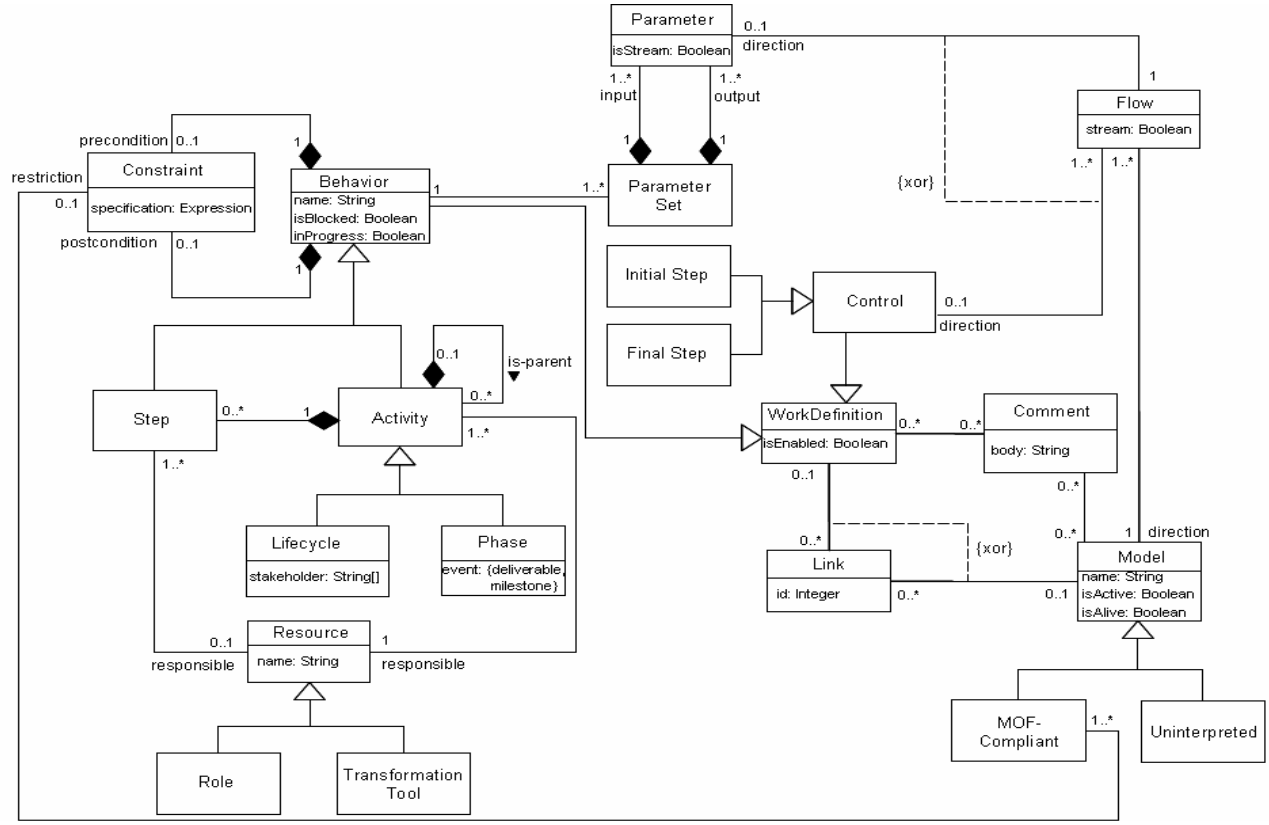


Figure II.18 Le métamodèle de l'approche de Porres [Porres et al, 2006]

II.5.3. Synthèse

Nous avons présenté dans les sections qui précédent deux approches de modélisation et de mise en œuvre des processus IDM.

La première, celle de Maciel, propose un langage de modélisation et de mise en œuvre des processus IDM. Ce langage est basé sur le standard SPEM et l'initiative MDA de l'OMG. Il est supporté par un environnement appelé *Transforms*. Cependant cette approche ne prend en compte que les transformations basées sur les modèles UML. En effet, le développement à base de modèles ne peut pas se cantonner à une vision orientée UML et il faut prendre en compte les autres DSL. De plus l'approche de Maciel ne prend pas en compte les autres approches de transformation basées sur soit sur un programme, soit sur un template. L'approche ne prend pas aussi en compte QVT qui pourtant est le standard dédié à la définition de transformations exécutables. Ces limitations rendent rigides son environnement d'implémentation qui ne prévoit pas l'intégration avec les autres environnements de transformations tels que (ATL, les outils QVT, etc.). L'assistance aux développeurs dans la réalisation des activités autres que les transformations et la traçabilité des transformations ne sont pas aussi prises en compte dans cette approche.

La deuxième approche présentée (*celle de Porres*) propose une approche basée sur MDA, les diagrammes d'activités d'UML 2.2, SPEM, et les réseaux de Petri. Cette approche permet de décrire un modèle de processus conforme au métamodèle, et exécute ce modèle de processus dans le formalisme des réseaux de Petri. Cette approche a l'avantage de séparer la description d'un processus de son exécution dans le formalisme des réseaux de Petri dont le caractère exécutable et sa sémantique bien définie sont reconnus de tous. L'approche offre aussi l'avantage d'utiliser les outils existants dans le domaine des réseaux de Petri. Cependant, cette approche n'explicite ni la notion de transformation et ses modèles et métamodèles associés, ni la relation de conformité entre un modèle et son métamodèle. Les autres aspects du développement IDM tels que la traçabilité des transformations sont absents dans cette approche.

II.6. Bilan des approches étudiées

Plusieurs langages et formalismes ont été proposés pour la modélisation des processus logiciels (voir section II.2.3). Cependant, à travers l'étude bibliographique que nous avons menée seules deux approches explicitent dans leurs métamodèles les concepts de base de l'IDM (transformation, modèle, métamodèle, méta-métamodèle, outil IDM). Ces deux approches qui sont celles de Maciel et Porres présentent des limitations discutées dans la section précédente.

Le standard SPEM de l'OMG dédié à la modélisation et la mise en œuvre des processus ne capture pas la nature exacte des activités et artéfacts d'un développement IDM. En outre, il ne satisfait pas le critère d'exécutabilité dans sa dernière version. En effet, le suivi de l'exécution d'un projet basé sur un processus défini avec SPEM n'est pas supporté par le standard SPEM. Néanmoins, SPEM propose de lier un modèle de processus à un formalisme externe (diagrammes d'activités, machines à états UML, notation BPMN (Business Process Modeling Notation) afin de pouvoir l'exécuter. La manière dont cette liaison doit se faire n'est pas décrite dans SPEM, et reste à la charge des éditeurs d'outils SPEM.

Nous considérons que dans un développement, il est important de prendre en compte deux aspects : un modèle qui définit le processus, et un modèle d'exécution qui décrit ce qui se passe réellement dans le développement.

Le standard QVT fournit un formalisme qui permet d'exécuter les transformations définies dans un processus IDM. IL aborde aussi la traçabilité même si cette dernière mérite d'être renforcée. En effet, seul le langage *Relation* permet de générer automatiquement les traces de l'exécution d'une transformation. Le langage *Core* de QVT fournit à l'utilisateur un mécanisme pour définir son modèle de trace, mais n'assure pas une gestion automatique de celui-ci. Dans le langage *Operationnal Mappings* de QVT, la gestion de la trace est entièrement à la charge de l'utilisateur.

Le tableau suivant décrit la comparaison des approches représentatives. Cette comparaison est faite selon les critères d'études définis dans l'introduction. Nous remarquons que les LMP de première génération et les langages de transformations (ATL, QVT-Core, QVT-Operationnal Mappings) sont moins conviviaux que les autres langages (SPEM, langages basés sur SPEM, langages basés sur UML, approches de Maciel et Porres, QVT-Relation) car ils proposent une notation textuelle alors que les autres langages proposent une notation graphique.

L'exécutabilité est supportée par toutes les approches sauf SPEM, MODAL, et PROMENADE.

En ce qui concerne les concepts IDM seuls les langages QVT, ATL, et les approches de Maciel et Porres, tentent de les expliciter. Nous remarquons aussi que la traçabilité des transformations n'est supportée que par les langages QVT-Relation, QVT-Core et ATL.

Approches / Critères		Convivialité	Exécutabilité	Prise en compte des concepts de l'IDM	Traçabilité
LMP de 1ére génération	APPL-A	+	OUI	NON	NON
	MERLIN	+	OUI	NON	NON
	PBOOL	+	OUI	NON	NON
	Slang	+	OUI	NON	NON
	MSL	+	OUI	NON	NON
	Little JIL	+	OUI	NON	NON
LMP basés sur SPEM	SPEM	++	NON	NON	NON
	eSPEM	++	OUI	NON	NON
	xSPEM	++	OUI	NON	NON
	MODAL	++	NON	NON	NON

LMP basées sur UML	UML4SPM	++	OUI	NON	NON
	PROMENADE	++	NON	NON	NON
	Aapproche de Di Nitto	++	OUI	NON	NON
	Approche de Chou	++	OUI	NON	NON
Langages de transformations	QVT-Relation	++	OUI	OUI	OUI
	QVT-Core	+	OUI	OUI	OUI
	QVT-Operationnal Mappings	+	OUI	OUI	NON
	ATL	+	OUI	OUI	OUI
Langages de description des processus IDM	Approche de Maciel	++	OUI	OUI	NON
	Approche de Porres	++	OUI	OUI	NON

Tableau II-1 Tableau comparatif des approches étudiées

II.7. Conclusion

Nous avons présenté dans ce chapitre un état de l'art dans le domaine de l'Ingénierie des Processus Logiciels (IPL) et de l'Ingénierie Dirigée par les Modèles (IDM). L'IPL est une discipline du génie logiciel qui vise la maîtrise les projets de développement logiciel en fournissant les moyens de modéliser les processus logiciels supports de ces projets. Quant à l'IDM c'est une discipline récente du génie logiciel qui recommande l'utilisation intensive des modèles et des transformations de modèles au cœur du processus de développement.

L'étude bibliographique que nous avons menée s'est focalisée essentiellement sur la notion de transformation avec ses langages et outils associés ainsi que sur les LMP d'avant et après l'IDM. Au regard de cette étude, il est possible de réifier les concepts centraux de l'IDM pour modéliser et mettre en œuvre les processus IDM. Les éditeurs de logiciels ont compris très vite cet enjeu et ont commencé à transformer leurs processus traditionnels en des processus IDM [Larrucea et al., 2004]. La formalisation des processus IDM à travers un langage et un environnement support devra permettre de réduire le coût du développement, de respecter les délais du développement et de séparer nettement les aspects métiers et les aspects techniques par le biais d'une utilisation intensive des modèles et des transformations au cœur des processus de développement.

CHAPITRE III. LE METAMODELE SPEM4MDE

III.1. Introduction

Beaucoup de processus IDM ont émergé avec l'avènement de l'IDM. Cependant, il n'existe pas à l'heure actuelle de langage unifié pour décrire ces processus, chacun adoptant une notation ad hoc et des concepts différents sont utilisés pour décrire les activités et les artéfacts du développement logiciel. Pour répondre à cette problématique, nous proposons le langage de modélisation de processus SPEM4MDE spécifié formellement sous forme d'un métamodèle. Il étend un sous-ensemble du standard SPEM 2.0 [OMG-SPEM 2.0, 2008], réutilise un sous-ensemble d'UML 2.2 [OMG-UML 2.2, 2009] qui décrit les machines à états, et intègre MOF 2.0 QVT [OMG-QVT 1.0, 2008] pour décrire des transformations exécutables. L'extension de SPEM se justifie par le fait que c'est un standard dédié à la modélisation des processus logiciels et systèmes. Cependant il présente certaines lacunes dont l'impossibilité de capturer la nature exacte des artéfacts (modèles) et activités (transformations) d'un développement IDM. En outre, SPEM ne fournit pas de formalisme ni de concepts spécifiques permettant d'exécuter les modèles de processus qui lui sont conformes. En effet, le suivi de l'exécution d'un projet basé sur un processus défini avec SPEM n'est pas supporté par ce standard. Nous considérons que dans un développement, il est important de prendre en compte deux aspects : un modèle qui définit le processus, et un modèle d'exécution de projet qui définit l'instance du modèle de processus et qui décrit ce qui se passe réellement dans le développement. Quant à la norme QVT, c'est le standard dédié à la modélisation des transformations de modèles ; elle permet de définir des transformations exécutables. Le standard UML offre les concepts qui permettent de décrire les machines à états. Nous le réutilisons afin de décrire les modèles comportementaux des processus IDM.

Ce chapitre est organisé comme suit. Dans la section III.2 nous décrivons l'approche globale adoptée dans la construction du métamodèle SPEM4MDE. La section III.3 décrit l'organisation en paquetages du métamodèle SPEM4MDE. La section III.4 présente les concepts de base du métamodèle SPEM4MDE. La section III.5 présente le volet structurel de SPEM4MDE. La section III.6 décrit le volet comportemental de SPEM4MDE. Nous concluons ce chapitre par la section III.7 en rappelant les contributions du métamodèle SPEM4MDE.

III.2. Approche de construction du métamodèle SPEM4MDE

Notre intention première est de fournir à la communauté IDM, un langage de modélisation et de mise en œuvre des processus IDM. Un processus IDM est défini comme un enchaînement de transformations de modèles. Or, les concepts du standard SPEM 2 permettent de décrire certaines

activités basées sur les modèles (l'édition de modèles par exemple) mais ne capturent pas la nature exacte de la majeur partie des activités et des artéfacts d'un développement IDM. Ainsi, pour tirer pleinement profit du standard SPEM 2, nous avons décidé de construire le métamodèle SPEM4MDE en étendant certains de ses concepts par des concepts relatifs au développement IDM (transformation, modèle, métamodèle, outil IDM). Pour réduire la complexité du métamodèle SPEM4MDE, nous ne réutilisons dans SPEM 2 que le paquetage *Process Structure* qui constitue le noyau de SPEM4MDE sur lequel reposent les autres paquetages. Cependant, l'architecture de SPEM4MDE permet à l'avenir une intégration facile des autres paquetages de SPEM 2. SPEM ne fournissant pas de concepts spécifiques pour décrire le comportement d'un processus, son extension permettra donc de décrire la structure des processus IDM mais pas leur comportement. Pour surmonter cet obstacle, SPEM4MDE réutilise le paquetage d'UML 2.2 Superstructure qui décrit les machines à états d'UML. Une machine à états permet de capturer les états d'un élément d'un processus IDM au cours de son cycle de vie. L'exploitation des machines à états dans un environnement de génie logiciel permet d'offrir une assistance aux développeurs. En outre, les transformations décrites dans la partie structurelle d'un modèle de processus SPEM4MDE ne sont pas exécutables. Dans le but d'associer une sémantique exécutable à ces transformations, nous réutilisons le standard QVT. La réutilisation du standard QVT permet de tirer pleinement profit des outils d'implémentation et d'exécution liés à ce standard. SPEM4MDE prend aussi en compte les approches de transformations exécutables basées soit sur un programme, soit sur un template.

III.3. Structuration en paquetages du métamodèle SPEM4MDE

La Figure III.1 décrit l'organisation en paquetages du métamodèle SPEM4MDE. Le paquetage *MDE Process Structure* introduit les concepts qui décrivent la structure d'un processus IDM. La structure d'un processus IDM est décrite en spécifiant les transformations et les autres activités basées sur les modèles (par exemple, éditer un modèle, vérifier un modèle, etc.). *MDE Process Structure* « merge » le paquetage *Process Structure* de SPEM 2.0 qui permet de décrire rigoureusement les activités basées sur les modèles (voir Annexe A). Le paquetage *Model Relationship* importe le paquetage *MDE Process Structure* pour décrire d'une part les différentes relations entre modèles et entre éléments de modèles, d'autre part les traces d'exécution d'une transformation. Le paquetage *MDE Process Behavior* décrit la partie comportementale d'un processus IDM. Il « merge » le paquetage *MDE Process Structure* et les paquetages de MOF 2.0 QVT (voir Annexe C). L'intégration de MOF 2.0 QVT dans SPEM4MDE a pour objectif de décrire la sémantique d'exécution des transformations spécifiées dans un modèle de processus IDM. *MDE Process Behavior* réutilise aussi les concepts du paquetage *BehaviorStateMachines* d'UML 2.2 Superstructure (voir Annexe B) pour décrire le comportement d'un processus IDM.

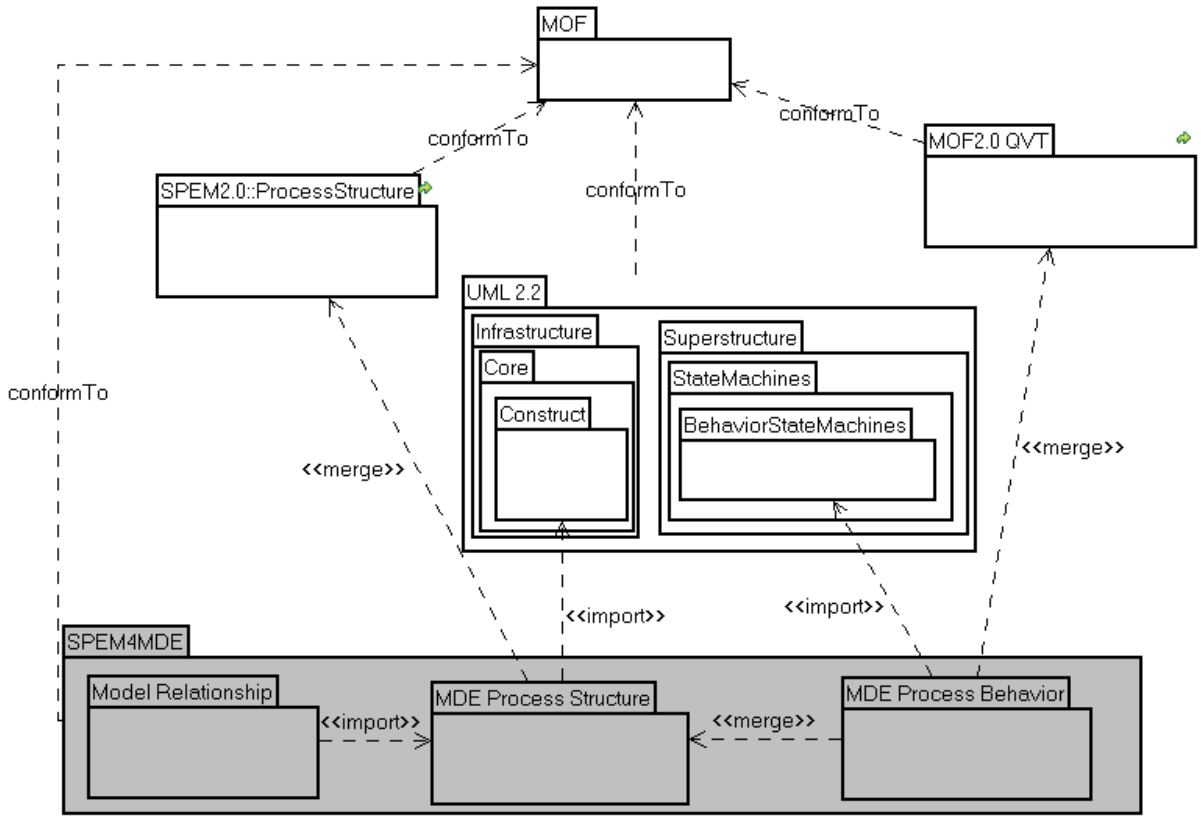


Figure III.1. Organisation en paquetages du métamodèle SPEM4MDE

III.4. Concepts de base de SPEM4MDE

Notre approche a pour but de fournir à la communauté IDM, un langage de modélisation et de mise en œuvre de processus se situant au même niveau que les standards de l'OMG (SPEM 2.0 , UML 2.2 et MOF 2.0 QVT) et permettant aux concepteurs de processus de décrire aussi bien une activité à base de modèles (par exemple l'édition de modèles) qu'une transformation de modèles. L'idée de base est de considérer qu'un processus de développement logiciel basé sur les modèles appelé aussi – processus IDM – est un enchaînement de transformations de modèles et d'activités basées sur les modèles qui aboutit généralement à un artéfact exécutable : le produit logiciel. La définition d'une transformation (*TransformationDefinition*) consiste à spécifier ses *métamodèles source et cible*. Une transformation est décomposable en sous-transformations. Elle peut étendre une autre transformation et peut avoir des liens de précédence. Le concept *TransformationImpl* décrit l'implémentation de la transformation dans un formalisme basé soit sur des règles (par exemple ATL, QVT), soit sur un programme (par exemple Java), soit sur un template (par exemple approche implémentée dans Softeam MDA Modeler). Les modèles sont les paramètres d'entrée et de sortie de *TransformationImpl* et devront être conformes aux métamodèles de *TransformationDefinition*. La séparation de la définition d'une transformation de son implémentation a pour objectif d'avoir plusieurs implémentations pour une définition donnée. Ainsi, un concepteur de transformation pourra choisir l'implémentation qui lui convient le plus en se basant sur une définition. Un modèle peut être le résultat d'une composition de modèles ou d'un raffinement d'un modèle. La Figure III.2 ci-dessous décrit les concepts qui résument l'approche conceptuelle adoptée pour définir le métamodèle SPEM4MDE. Ce diagramme n'est qu'un modèle conceptuel qui résume les concepts de

base de SPEM4MDE et diffère donc du métamodèle SPEM4MDE qui sera présenté dans les sections qui suivent. Notre approche prend aussi en compte les transformations de métamodèles car il suffit de remplacer dans le diagramme ci-dessous les modèles par les métamodèles et les métamodèles par les méta-métamodèles.

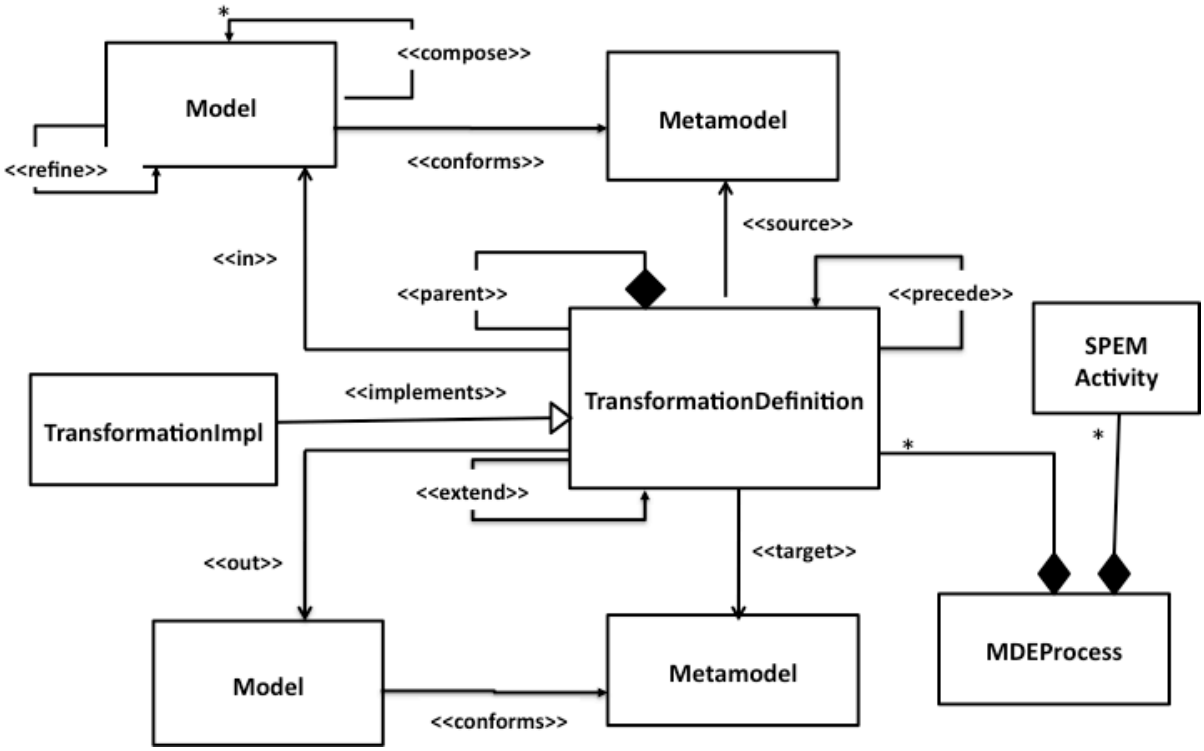


Figure III.2. Modèle conceptuel de l'approche adoptée dans SPEM4MDE

III.5. Volet structurel de SPEM4MDE

Cette section introduit les paquetages du volet structurel de notre métamodèle. Le volet structurel fournit les concepts qui permettent de définir la structure d'un processus IDM. Le volet structurel est décrit par les paquetages *Process Structure* de SPEM 2.0, *MDE Process Structure* et *Model Relationship*. La sémantique statique (structurelle) de ces paquetages sera décrite en OCL.

III.5.1. Le paquetage Process Structure de SPEM 2.0

Le paquetage SPEM 2.0 *Process Structure* (voir Annexe A) contient les concepts de base pour définir la structure d'un processus de développement. Dans SPEM, un processus de développement est une collaboration entre des entités actives et abstraites (les rôles) qui réalisent des opérations (activités) pour produire des entités concrètes et réelles (les produits). SPEM 2.0 *Process Structure* « merge » le paquetage *Core* de SPEM 2.0. Les concepts de *Process Structure* concernés par le « merge » avec *MDE Process Structure* sont *BreakdownElement*, *Activity*, *RoleUse*, *WorkProductUse*.

BreakdownElement est une généralisation abstraite des artéfacts, des rôles, et des activités d'un processus logiciel. Les artéfacts sont spécifiés par *WorkProductUse*, les rôles par *RoleUse* et les activités

par *Activity*. Les artefacts, les rôles, et les activités d'un processus logiciel sont contenus dans une activité via la composition « *nestedBreakdownElement* » entre *Activity* et *BreakdownElement*. Une activité (*Activity*) définit un travail (*WorkDefinition*) assignable à des rôles et liés à des produits (ses entrée et/ou sorties).

WorkProductUse représente soit un artéfact consommé, modifié ou produit par une activité, soit un artéfact contenu dans une activité via la composition « *nestedBreakdownElement* ». Si *WorkProductUse* est un artéfact contenu dans une activité A, alors il sera utilisé par une sous-activité de A comme entrée, sortie, ou entrée-sortie. Par conséquent, il sera lié à un rôle contenu dans A qui en sera le rôle responsable.

RoleUse représente soit le rôle joué par une personne physique qui réalise une activité, soit un rôle contenu dans une activité via la composition « *nestedBreakdownElement* ». Si *RoleUse* est un rôle contenu dans une activité A, alors il sera utilisé par une sous-activité de A comme rôle. Par conséquent, il sera le responsable des produits de cette sous-activité de A.

III.5.2. Le paquetage MDE Process Structure

Le paquetage *MDE Process Structure* (voir Figure III.3) offre les concepts qui permettent de décrire la structure d'un processus IDM. Ce paquetage « merge » le paquetage *Process Structure* de SPEM 2. 0. L'idée sous-jacente est de spécialiser les concepts *Activity* et *WorkProductUse* de SPEM par respectivement les concepts *TransformationDefinition* et *Model*. La spécialisation du concept *Activity* permet de décrire la précondition, la postcondition et l'invariant d'une transformation, de décomposer une transformation en une série de sous-transformations, et enfin de décrire les précédences entre les transformations.

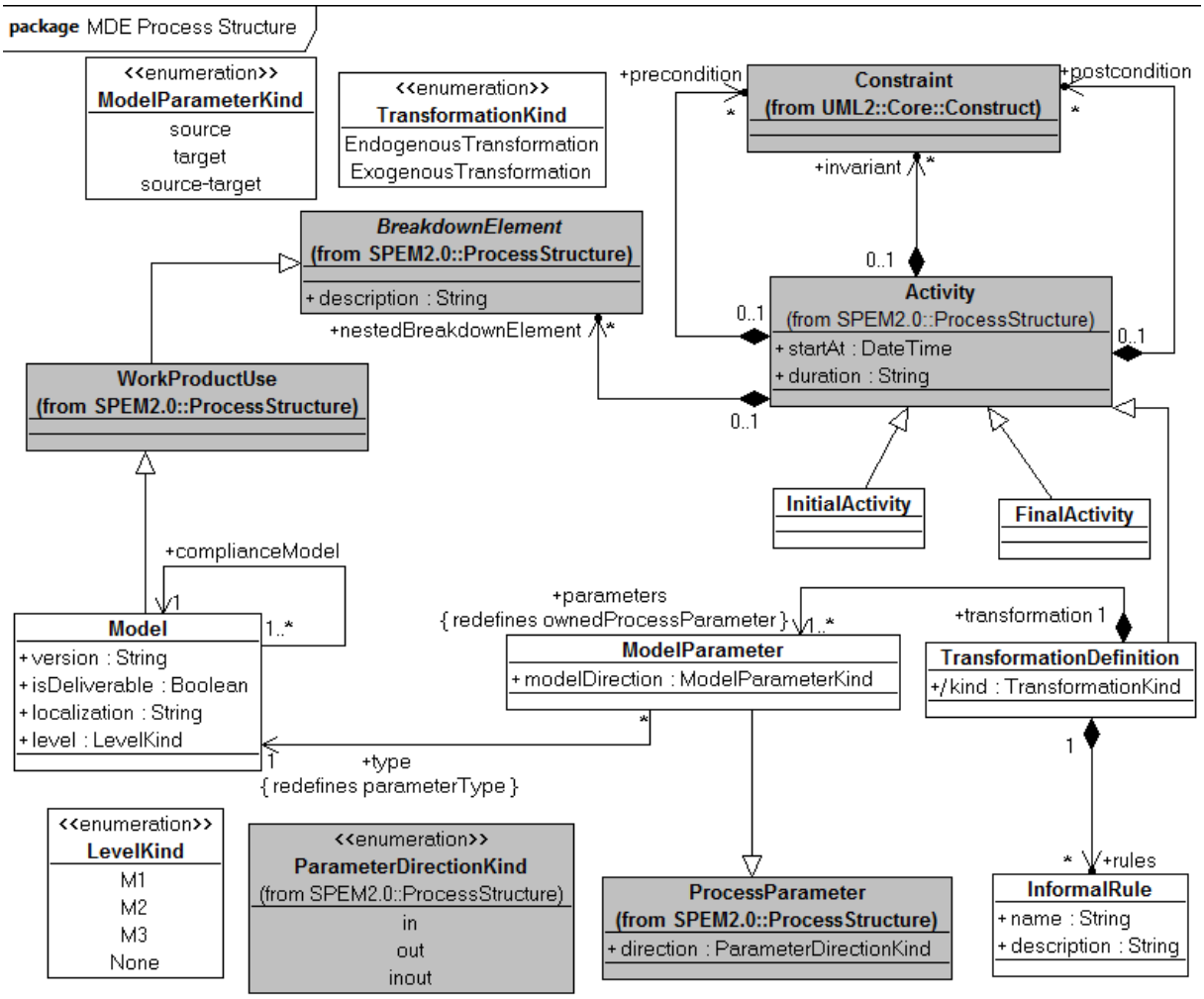


Figure III.3. Paquetage MDE Process Structure

III.5.2.1. BreakdownElement (from SPEM 2.0 Process Structure)

Description (cf. Annexe A, section d)

BreakdownElement est une généralisation abstraite des artefacts spécifiés par *WorkProductUse*, des rôles spécifiés par *RoleUse* et des activités spécifiées par *Activity*.

Attributs (nouveau)

- description: String Cet attribut permet de donner une description textuelle d'un BreakdownElement.

III.5.2.2. Activity (from SPEM 2.0 Process Structure)

Description (cf. Annexe A, section h)

Le concept *Activity* définit un travail assignable à des rôles spécifiés par *RolUse* et lié à des produits (ses entrée et/ou sorties) spécifiés par *WorkProducUse*. Une activité est liée à ses produits via les instances de *ProcessParameter* et à ses rôles via les instances de *ProcessPerformer*. Une activité dans

SPEM spécialise d'une part un « *WorkDefinition* » afin de spécifier sa précondition et sa postcondition et d'autre part un « *WorkBreakdownElement* » afin de spécifier ses successeurs et ses prédécesseurs.

Attributs nouveaux

- startAt : DateTime. Cet attribut spécifie la date de démarrage d'une activité lors de la mise en œuvre.
- duration : String. Cet attribut spécifie la durée prévue pour réaliser une activité.

Associations dans SPEM

- precondition : Constraint [*]. Cette composition redéfinit l'association « *precondition* » associée à « WorkDefinition ». Elle spécifie la ou les préconditions d'une activité c'est-à-dire les conditions à vérifier avant de commencer la mise en oeuvre d'une activité.
- postcondition : Constraint [*]. Cette composition redéfinit l'association « postcondition » associée à « WorkDefinition ». Elle spécifie la ou les postconditions d'une activité c'est-à-dire les conditions à vérifier avant de terminer la mise en œuvre d'une activité.

Association nouvelle

- invariant : Constraint [*]. Cette association spécifie le ou les invariants d'une activité. L'invariant est une condition qui est toujours vraie durant le cycle de vie de l'activité.

Notation graphique :

Activity

III.5.2.3. InitialActivity

Description

Ce concept décrit une pseudo-activité qui permet de démarrer la mise en œuvre d'un processus.

Généralisation

- Activity (from SPEM 2.0 Process Structure)

Notation graphique :

III.5.2.4. FinalActivity

Description

Ce concept décrit une pseudo-activité qui permet de terminer la mise en œuvre d'un processus c'est-à-dire de terminer tous les flots d'exécution d'un processus.

Généralisation

- Activity (from SPEM 2.0 Process Structure)

Notation graphique :

III.5.2.5. Model

Description

Le concept *Model* décrit tout artéfact de type modèle dans un développement IDM. Dans le cadre des transformations, nous nous intéresserons seulement aux modèles conformes à un autre modèle appelé métamodèle. Le terme modèle est pris dans un sens très général, car il représente à la fois un modèle de niveau M1, un modèle de niveau M2 (i.e. un métamodèle), et un modèle de niveau M3 (i.e. méta-métamodèle).

Généralisation

- WorkProductUse (from SPEM 2.0 Process Structure)

Attributs

- version : String. Cet attribut spécifie la version courante d'un modèle. Cet attribut sera renseigné lors de la mise en œuvre.
- isDeliverable : Boolean. Cet attribut spécifie si le modèle est un livrable ou non.
- localization : String. Cet attribut spécifie le chemin d'accès (emplacement) d'un modèle au moment de la mise en œuvre.
- level : LevelKind. Cet attribut spécifie le niveau de modélisation d'un modèle. Les niveaux de modélisation sont précisés par l'énumération *LevelKind* dont les valeurs sont : None, M1, M2 et M3. La valeur *None* spécifie qu'un modèle peut ne pas avoir de niveau fixe (par exemple UML Infrastructure, Kermeta, etc.). UML Infrastructure est vu comme un modèle de niveau M2 s'il est utilisé dans le métamodèle d'UML ou comme un modèle de niveau M3 s'il est utilisé dans MOF.

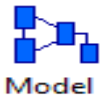
Association

- complianceModel : Model [1]. Cette association spécifie la relation de conformité entre deux modèles. Les règles de bonne modélisation ci-dessous introduisent une restriction entre un modèle et celui auquel il est conforme.

Notation graphique de la relation de conformité :

Règles de bonne modélisation

- WF01 : Un modèle de niveau M1 doit être conforme à un modèle de niveau M2. **Context** Model inv: (self.level=#M1 **and** self.complianceModel→notEmpty()) **implies** (self.complianceModel.level= #M2)
- WF02 : Un modèle de niveau M2 doit être conforme à un modèle de niveau M3. **Context** Model inv : (self.level=#M2 **and** self.complianceModel→notEmpty()) **implies** (self.complianceModel.level= #M3)
- WF03 : Un modèle de niveau M3 doit être conforme à lui-même. **Context** Model inv : (self.level=#M3 **and** self.complianceModel→notEmpty()) **implies** (self.complianceModel=self)

Notation graphique : 

III.5.2.6. ModelParameter

Description

ModelParameter est un concept qui permet de référencer soit des modèles et des métamodèles dans le cas d'une transformation de modèles, soit des métamodèles et des méta-métamodèles dans le cas d'une transformation de métamodèles.

Généralisation

- ProcessParameter (from SPEM 2.0 Process Structure)

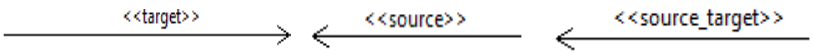
Attribut nouvel

- mdirection : ModelParameterKind. Cet attribut spécifie soit la direction des métamodèles référencés dans le cas d'une transformation de modèles, soit la direction des méta-métamodèles référencés dans le cas d'une transformation de métamodèles. Les valeurs de cet attribut sont spécifiées par l'énumération *ModelParameterKind* dont les valeurs sont source (i.e. source), target (i.e. cible), et source-target. (i.e. source-cible).

Association

- type : Model [1]. Cette association spécifie les modèles (niveau M1, M2, M3) référencés par *ModelParameter*.. Elle redéfinit l'association « *parmaterType* » liée à *WorkProductUse* dans *Process Structure*.

Notation graphique :



III.5.2.7. TransformationDefinition

Description

Le concept *TransformationDefinition* spécifie la définition d'une activité de transformation de modèles. Cette activité est décrite en spécifiant ses paramètres qui sont des modèles (respectivement métamodèles) conformes à des métamodèles (respectivement méta-métamodèles). Cette activité de définition d'une transformation consiste aussi à décrire les règles informelles qui seront implémentées dans *MDE Process Behavior* via le concept *TransformationImpl*. Le concept *TransformationDefinition* spécialise le concept *Activity* afin de décrire sa précondition, sa postconsdition, ses sucesseurs et ses prédécesseurs.

Généralisation

- Activity (from SPEM 2.0 Process Structure)

Attribut

- kind : TransformationKind. C'est un attribut dérivé qui spécifie grâce à l'énumération *TransformationKind* si une transformation est endogène ou exogène. Une transformation est endogène si ses métamodèles source et cible sont identiques ; elle est exogène dans le cas contraire.

Associations

- parameters : ModelParameter [1..*]. Cette composition spécifie les paramètres d'une transformation. Ces paramètres sont des références qui pointent soit vers des modèles et des métamodèles dans le cas d'une transformation de modèles, soit vers des métamodèles et des méta-métamodèles dans le cas d'une transformation de métamodèles. L'association « *parameters* » redéfinit le rôle « *ownedProcessParameter* » qui spécifie les paramètres d'une activité.
- rules : InformalRule [1..*]. Cette association spécifie les règles informelles d'une transformation. Ces règles serviront de guide dans l'implémentation de la transformation.

Règles de bonne modélisation

- WF01 : Les modèles (respectivement métamodèles) d'entrée et de sortie d'une transformation de modèles (respectivement métamodèles) ont le même niveau de modélisation.

  Context TransformationDefinition inv:
  self.parameters →**select** (t|t.direction=#in or t.direction=#out)→ **collect** (t | t.type)→ forall (m1, m2 | m1.level=m2.level)

- WF02 : Les modèles (respectivement métamodèles) d'entée-sortie d'une transformation de modèles (respectivement métamodèles) ont le même niveau de modélisation.

  Context TransformationDefinition inv:
  self.parameters →**select** (t|t.direction=#inout)→ **collect** (t | t.type)→ forall (m1, m2 | m1.level=m2.level)

- WF03 : Le métamodèle (respectivement méta-métamodèles) source et cible d'une transformation de modèles (respectivement métamodèles) ont le même niveau de modélisation.

  Context TransformationDefinition inv:
  (self.parameters→**select** (t|t.modelDirection=#source or t.modelDirection=#target)→ **collect** (t | t.type)→ forall (m1, m2 | m1.level=m2.level)

- WF04 : Une transformation composite ne contient que des transformations (ses sous-transformations), des modèles, et des rôles spécifiés par RoleUse.

  Context TransformationDefinition inv: self.nestedBreakdownElement→ **includesAll** (b: Activity | b.**oclIsTypeOf** (TransformationDefinition) or b.**oclIsTypeOf** (Model) or b.**oclIsTypeOf** (RoleUse))

- WF05: Une transformation simple (non-composite) est associée à un seul rôle.
  Context ProcessPerformer inv : (self.linkedActivity.**oclIsTypeOf** (TransformationDefinition)) **and** (self.linkedActivity.nestedBreakdownElement →isEmpty()) **implies** (self.linkedRoleUse→size()=1)

Notation graphique :

III.5.2.8. InformalRule

Description

Ce concept décrit une règle informelle d'une transformation, cette dernière définit une correspondance entre un concept du métamodèle source et un concept du métamodèle cible. Les règles informelles sont décrites sous forme de notes textuelles associées à la spécification d'une transformation. Les règles informelles seront implémentées dans le volet comportemental dans un langage exécutable interprétable par un outil.

Attributs

- name : String. Cet attribut spécifie le nom d'une règle de transformation.
- description : String. Cet attribut donne une description textuelle d'une règle.

III.5.3. Le paquetage Model Relationship

Le paquetage Model Relationship (voir Figure III.4) décrit les relations entre modèles et éléments de modèles dans un modèle de processus IDM. Il spécifie aussi les traces d'exécution d'une transformation.

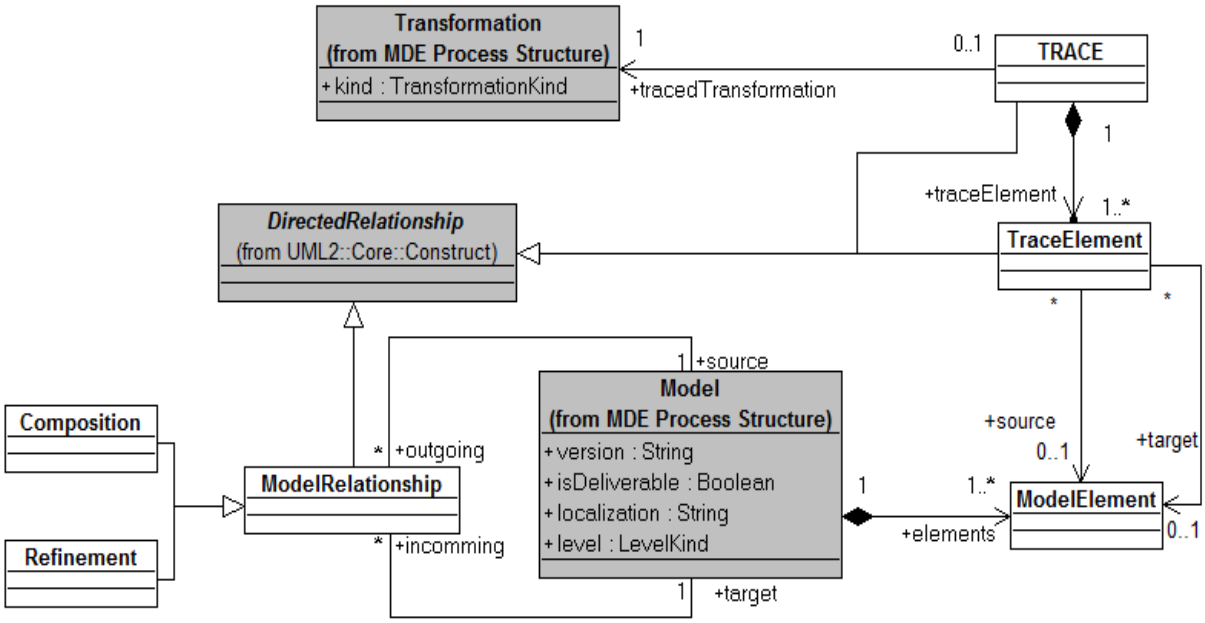


Figure III.4. Paquetage Model Relationship

III.5.3.1. Model

Description cf. Model (section III.5.2.5III.5.2.3)

Attributs Cc. Model (section III.5.2.5)

Associations nouvelles

- outgoing : ModelRelationship [*] : Cette association spécifie les connexions sortantes (i.e. instances de *ModelRelationship*) d'un modèle.
- incoming : ModelRelationShip [*] : Cette association spécifie les connexions entrantes (i.e. instances de *ModelRelationship*) d'un modèle.
- elements : ModelElement [1..*]. Cette association spécifie les éléments qui décrivent un modèle.

III.5.3.2. ModelRelationship

Description

ModelRelationship représente une généralisation des relations entres modèles.

Généralisation

- DirectedRelationship (from UML 2.2 :: Core :: Construct)

Associations

- source : Model. Cette association spécifie le modèle source d'une instance de *ModelRelationship*.
- target : Model. Cette association spécifie le modèle cible d'une instance de *ModelRelationship*.

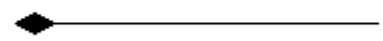
III.5.3.3. Composition

Description

Composition est une spécialisation de *ModelRelationShip* qui définit une composition entre modèles. Par exemple dans VUML [Nassar, 2005 ; Anwar et al., 2010], le modèle VUML final est le résultat de la composition des modèles structurels par point de vue. La notion de composition permet de réduire la complexité d'un modèle de taille importante en le décomposant en plusieurs modèles de petite taille. Elle permet aussi de décrire un modèle composé de modèles hétérogènes (chaque modèle étant conforme à un métamodèle différent).

Généralisation

- ModelRelationship

Notation graphique : 

III.5.3.4. Refinement

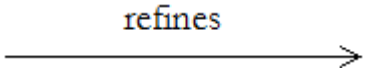
Description

Refinement est une spécialisation de *ModelRelationShip* qui définit une relation de raffinement entre modèles. Par exemple le passage d'un modèle PIM vers un modèle PSM dans l'approche MDA peut être considéré comme un raffinement d'un modèle abstrait vers un modèle concret. Ce concept est

utilisé dans la description structurelle d'un processus lorsqu'un modèle est obtenu par raffinement d'un autre.

Généralisation

- ModelRelationship

Notation graphique : 

III.5.3.5. Trace

Description

Le concept *Trace* spécifie les traces d'exécution d'une transformation. La notion de traçabilité s'apparente à la notion de log dans les applications ou bases de données. La traçabilité est exploitée dans notre approche pour connaître la séquence des modifications subies par un modèle.

Généralisation

- DirectedRelationship (from UML 2.2 :: Core :: Construct)

Associations

- taceElement : TraceElement [1..*]. Cette association décrit l'ensemble des traces d'exécution.
- tracedTransformation [1] : Transformation. Cette association spécifie la transformation tracée.

III.5.3.6. TraceElement

Description

TraceElement est un concept qui permet de décrire la traçabilité entre un élément d'un modèle d'entrée et un élément du modèle de sortie durant l'exécution d'une transformation.

Généralisation

- DirectedRelationship (from UML 2.2 :: Core :: Construct)

Associations

- source : ModelElement [1]. Cette association spécifie l'élément de modèle source d'une trace.
- target : ModelElement [1]. Cette association spécifie l'élément de modèle cible d'une trace.

III.5.3.7. ModelElement

Description

ModelElement spécifie un élément de modélisation. Par exemple la classe *Personne* est un élément de modèle décrit par la méta-classe *Class* dans le métamodèle UML.

III.5.4. Exemple illustratif du volet structurel

Dans cette section nous avons choisi délibérément de décrire un exemple simpliste pour illustrer les concepts *Activity* et *TransformationDefinition*. Un exemple plus détaillé sera présenté dans le chapitre V dédié à l'étude de cas. L'exemple ci-dessous (Figure III.5) consiste à définir une transformation d'un diagramme de classes UML en une base de données relationnelle.

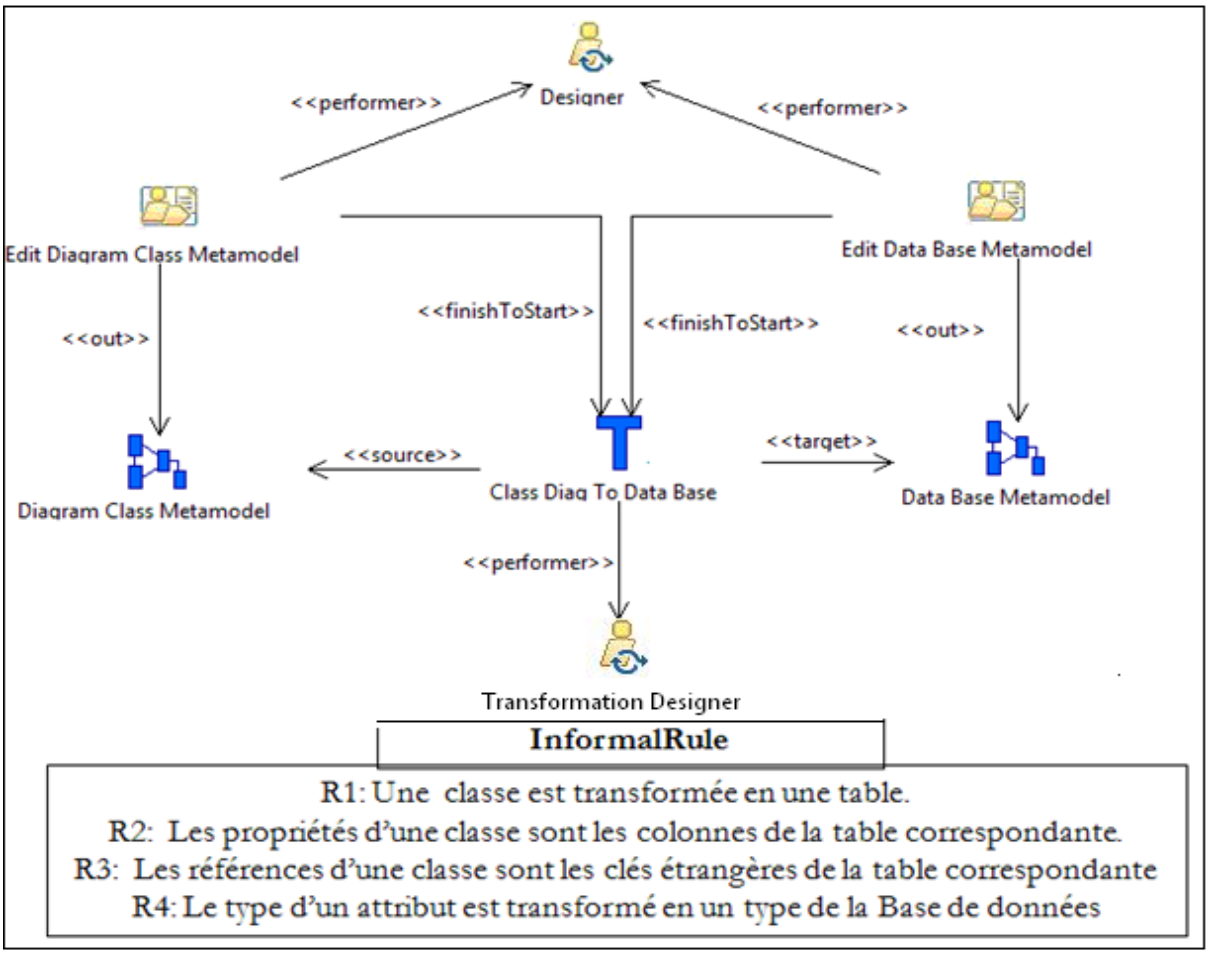


Figure III.5. Exemple d'une définition de transformation en SPEM4MDE.

Nous avons identifié *deux activités* et *une définition de transformation de modèles*. L'activité « *Edit Diagram Class Metamodel* » consiste à décrire le métamodèle « *Diagram Class Metamodel* ». L'activité « *Edit Data Base Metamodel* » consiste à décrire le métamodèle « *Data Base Metamodel* ». Ces deux activités sont réalisées par un concepteur. La description de la transformation « *Class Diag To Data Base* » consiste à indiquer d'abord ses métamodèles source et cible puis à décrire ses règles informelles (R1, R2, R3, et

R4). Cette transformation a pour métamodèle source « *Diagram Class Metamodel* » et pour métamodèle cible « *Data Base Metamodel* » et démarre lorsque les activités « *Edit Diagram Class Metamodel* » et « *Edit Data Base Metamodel* » sont terminées. La définition de la transformation est réalisée par un concepteur de transformation. L'implémentation de la transformation « *Class Diag To Data Base* » dans un formalisme exécutable sera présentée dans le volet comportemental.

III.6. Volet Comportemental de SPEM4MDE

La modélisation comportementale consiste à utiliser un formalisme qui permet de spécifier le comportement d'exécution d'un processus. L'objectif sous-jacent est de permettre un meilleur guidage du développement, une vérification des contraintes des activités/transformations, et la gestion de la cohérence des artéfacts/modèles du développement. Plusieurs formalismes ont été proposés pour décrire le comportement d'un processus (Réseaux de Petri, machines à états d'UML, diagrammes d'activités, notation BPMN, etc.). Nous avons porté notre choix sur les machines à états d'UML. Une machine à états est composée d'états définissant l'ensemble des états d'un élément d'un processus IDM et de transitions définissant les opérateurs de mise en œuvre spécifiques à cet élément. Lorsqu'un développeur exécute un opérateur, les actions associées à cet opérateur sont ainsi exécutées. Par exemple, pour une transformation éligible, quand l'opérateur de mise en œuvre « *run* » est lancé, l'environnement de mise en œuvre va ouvrir automatiquement l'outil requis pour mettre en œuvre la transformation.

Le volet comportemental est décrit par les paquetages *BehaviorStateMachines* d'UML 2.2, les paquetages MOF 2.0 QVT, et le paquetage *MDE Process Behavior* de SPEM4MDE. Le paquetage *MDE Process Behavior* importe le paquetage *BehaviorStateMachines* pour décrire le comportement des éléments d'un processus par le biais des machines à état d'UML 2.2. Il « merge » les paquetages *de* MOF 2.0 QVT pour décrire la sémantique d'exécution des transformations spécifiées dans la structure d'un processus IDM.

III.6.1. Le paquetage BehaviorStateMachines d'UML 2.

Dans le métamodèle SPEM4MDE, les machines à états permettent de décrire le comportement des processus IDM. La description comportementale consiste à décrire les opérateurs de mise en œuvre (opérations associées aux transitions d'une machine à états) et à gérer les états des éléments d'un processus IDM à chaque exécution d'un opérateur.

Le paquetage *UML 2.2 BehaviorSateMachines* (Annexe B) décrit les concepts qui permettent de spécifier une machine à états d'UML. Une machine à états UML est modélisée par un graphe de nœuds d'états interconnectés par un ou plusieurs arcs de transitions. Ces transitions sont déclenchées par l'arrivée d'événements (triggers) lorsqu'une condition (garde) est satisfaite.

III.6.2. Les paquetages de MOF 2.0 QVT

La Figure III.6 ci-dessous décrit l'organisation des paquetages du standard MOF 2.0 QVT. Le standard est composé de huit paquetages dont un issu de EMOF et un provenant de Essential OCL.

Les paquetages *EMOF* et *EssentialOCL* forment le noyau de MOF 2.0 QVT.

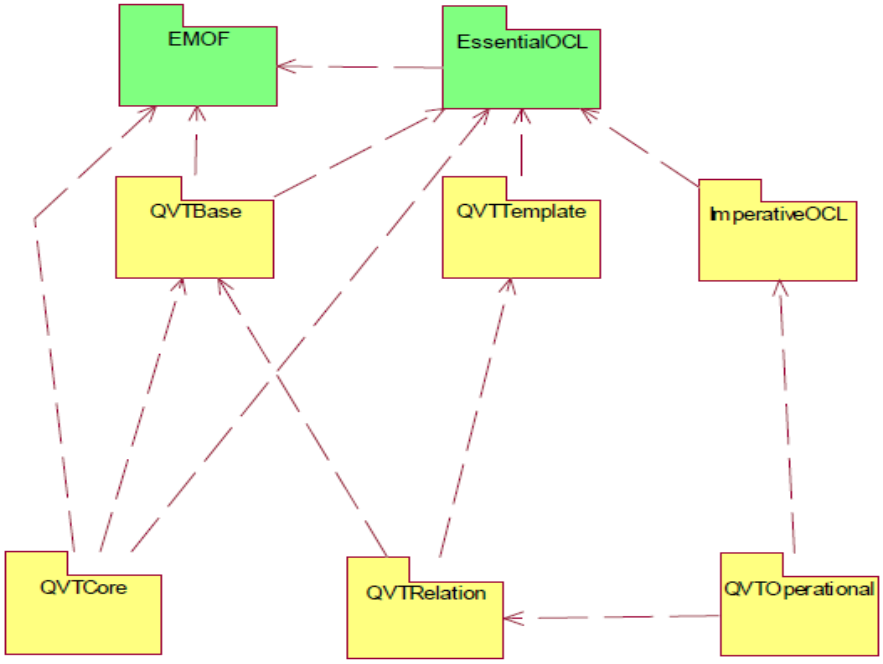


Figure III.6 La structuration en paquetages de MOF 2.0 QVT [OMG-QVT, 2008]

Le paquetage *QVTBase* de MOF 2.0 QVT (Annexe C) contient un ensemble de concepts de base dont une partie provient de la spécification d'EMOF et d'OCL. Ces concepts décrivent la structure de base d'une transformation.

Les paquetages *QVTTemplate* et *ImperativeOCL* dépendent du paquetage *EssentialOCL*. Le paquetage *QVTTemplate* permet de spécifier un pattern permettant de composer un à un les éléments des modèles source et cible d'une transformation. Le paquetage *ImperativeOCL* permet de spécifier des expressions OCL avec effet de bord.

Le paquetage *QVTRelation* dépend des paquetages *QVTBase* et *QVTTemplate*. Il fournit des concepts qui permettent de définir une transformation à un niveau d'abstraction élevé.

Le paquetage *Core* dépend des paquetages *EMOF* et *QVTBase*. Il fournit des concepts qui permettent de spécifier la sémantique d'exécution des transformations définies par le paquetage *QVTRelation*.

Le paquetage *QVTOperational* dépend des paquetages *QVTRelation* et *ImperativeOCL*. Il permet de spécifier des transformations impératives en utilisant des expressions OCL avec effet de bord.

La réutilisation des paquetages de MOF 2.0 QVT dans SPEM4MDE permet d'associer une sémantique exécutable aux transformations décrites dans la partie structurelle d'un modèle de processus SPEM4MDE. Elle favorise aussi la réutilisation des outils qui implémentent le standard QVT.

III.6.3. Le paquetage MDE Process Behavior

Le paquetage MDE Process Behavior (Figure III.7) définit les concepts décrivant le comportement des éléments d'un processus IDM. Ce paquetage « merge » le paquetage *MDE Process Structure* de SPEM4MDE et les paquetages de MOF 2.0 QVT. Il importe aussi le paquetage *BehaviorStateMachines* d'UML 2.2. La réutilisation du paquetage *BehaviorStateMachines* permet de décrire le comportement d'un processus IDM en utilisant les machines à états d'UML. Le « merge » avec les paquetages de MOF 2.0 QVT a pour objectif de décrire le comportement d'exécution des transformations et de réutiliser les outils qui implémentent QVT.

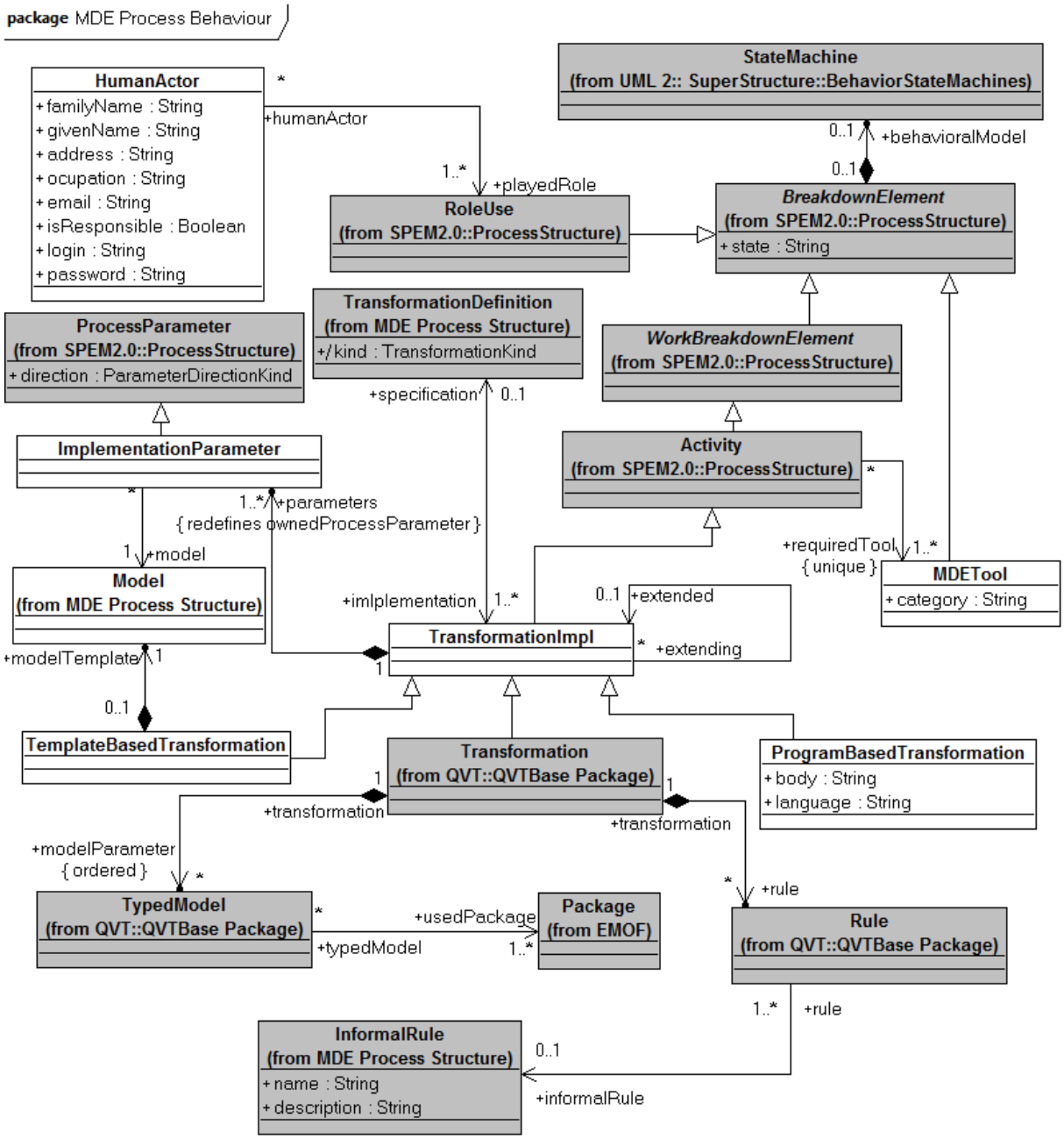


Figure III.7. Paquetage MDE Process Behavior

III.6.3.1. BreakdownElement (from SPEM 2.0 Process Structure)

Description (cf. Annexe A, section d)

BreakdownElement est une généralisation abstraite des artefacts spécifiés par *WorkProductUse*, des rôles spécifiés par *RoleUse* et des activités spécifiées par *Activity*.

Attributs (nouveau)

- state: String. Cet attribut permet de spécifier l'état courant d'un élément de processus exécutable.

Associations

- behavioralModel : StateMachine [0..1]. Cette association spécifie la machine à états éventuellement associée au BreakdownElement.

III.6.3.2. ImplementationParameter

Description

Ce concept décrit la relation entre une transformation implémentée et ses paramètres (modèles source et cible).

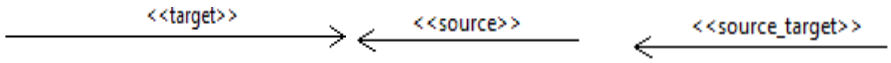
Attribut

- direction : ParameterDirectionKind. Cet attribut hérité de *ProcessParameter* spécifie la direction des modèles qui participent à l'exécution d'une transformation implémentée. L'attribut peut prendre les valeurs (in, out, ou inout).

Association

- model : Model [1]. Cette association spécifie les modèles source et cible de la transformation implémentée.

Notation graphique :



III.6.3.3. TransformationImpl

Description

Ce concept décrit l'activité d'implémentation d'une transformation spécifiée dans la structure d'un processus IDM. Cette activité consiste d'abord à implémenter les règles informelles d'une définition de transformation dans un formalisme interprétable par un outil de transformation, puis à préciser les modèles source et cible. Ce formalisme peut être basé soit sur des règles QVT, soit sur un programme (Java par exemple.), soit sur un template (par exemple approche implémentée dans SOFTEAM MDA Modeler). Le concept *TransformationImpl* spécialise *Activity* afin de décrire l'ordre d'exécution des transformations, la précondition, l'invariant et la postcondition d'une transformation.

Généralisation

- Activity

Associations

- specification : Transformation [1]. Cette association spécifie la définition de la transformation implémentée.
- parameters : ImplementationParameter [1..*]. Cette association spécifie les paramètres de la transformation implémentée (modèles source et cible).
- requiredTool : MDETool [1]. Cette association spécifie l'outil IDM requis pour exécuter la transformation implémentée.
- extended : TransformationImpl [0..1]. Cette association spécifie la transformation étendue.
- extending : TransformationImpl [0..*]. Cette association spécifie le ou les transformations qui étendent.

Règles de bonne modélisation

- WF01 : Les modèles de *TransformationImpl* doivent être des modèles de niveau M1 (cas d'une transformation de modèles) ou des modèles de niveau M2 (cas d'une transformation de métamodèles).

  Context TransformationImpl inv: self.parameters→ collect (i |i.model)→ **forall** (m1, m2 | (m1.level=m2.level) and m1.level<>#M3)

- WF02: Les modèles d'entrée de *TransformationImpl* doivent être conformes au métamodèle source de sa spécification.

  Context TransformationImpl inv: (self.parameters→**select** (i| i.direction=#in) → **collect** (i| i.model.complianceModel))=(self.specification.parameters→**select** (m| m.modelDirection=#source) → **collect** (m| m.type))

- WF03: Les modèles de sortie de *TransformationImpl* doivent être conformes au métamodèle cible de sa spécification.

  Context TransformationImpl inv: self.parameters→**select** (i| i.direction=#out) → **collect** (i| i.model.complianceModel) =self.specification.parameters→**select** (m| m.modelDirection=#target) → **collect** (m| m.type)

- WF04 : Les modèles source-cible de *TransformationImpl* doivent être conformes au métamodèle source-cible de sa spécification.

  Context TransformationImpl inv: self.parameters→**select** (i| i.direction=#inout) → **collect** (i| i.model.complianceModel) =self.specification.parameters→**select** (m| m.modelDirection=#source-target) → **collect** (m| m.type)

- WF05: Une transformation simple (non-composite) est associée à un seul rôle.
  Context ProcessPerformer inv: (self.linkedActivity.**oclIsTypeOf** (TransformationImpl)) **and** (self.linkedActivity.nestedBreakdownElement →isEmpty()) **implies** (self.linkedRoleUse→size()=1)

Notation graphique :

III.6.3.4. MDETool

Description

Ce concept décrit l'outil IDM requis pour l'exécution d'une activité/transformation.

Généralisation

- BreakdownElement

Attribut

- category : String. Cet attribut spécifie la catégorie de l'outil IDM. Parmi ces catégories nous distinguons les outils génériques (Xquery, AGG, etc.), les outils intégrés aux AGL (Softeam MDA Modeler, IBM Rational Software Modeler), les outils dédiés aux transformations (ATL, MediniQVT, etc.), et les outils de métamodélisation (TOPCASED, Kermeta, etc.).

Notation graphique :

III.6.3.5. Transformation (from QVTBase)

Description

Une transformation dans QVT est une transformation exécutable basée sur des règles déclaratives ou impératives qui spécifient son comportement d'exécution. La transformation est exécutée sur un ensemble de modèles dont les types sont définis par des paquetages. Syntaxiquement, le concept *Transformation* est à la fois une spécialisation des concepts *Package* et *Class*. Définie comme un paquetage, la transformation fournit un espace de nommage à ses règles. Définie comme une classe, elle peut définir des propriétés et des opérations. Les propriétés servent à spécifier les valeurs des paramètres de la configuration de l'environnement d'exécution de la transformation. Les opérations servent à implémenter les fonctions utilitaires requises par l'environnement d'exécution.

Généralisation

- TransformationImpl

Règles de bonne modélisation

- WF01: Les métamodèles du concept *TransformationDefinition* sont définis par les paquetages (Package) utilisés dans QVTBase.

  Context Transformation inv: self.modelParameter → **collect** (t| t.usedPackage) =self.specification.parameters→**collect** (m| m.type)

- WF02: Les modèles typés d'une transformation QVT sont identiques aux modèles paramètres de *TransformationImpl.*

  Context Transformation inv: self.modelParameter =self.oclAsType (TransformationImpl).parameters→**collect** (m| m.model)

III.6.3.6. ProgramBasedTransformation

Description

Ce concept décrit une transformation implémentée dans un langage de programmation. L'approche par programmation consiste à utiliser les langages de programmation en général, et plus particulièrement les langages orientés objet. Dans cette approche, la transformation est décrite sous forme d'un programme à l'image de n'importe quelle application informatique.

Généralisation

- TransformationImpl

Attribut

- body : String. Cet attribut permet de décrire le corps du programme de la transformation. Ce programme est l'équivalent des règles formelles dans une approche à base de règles.
- language : String. Cet attribut spécifie le langage utilisé pour décrire le programme de la transformation.

III.6.3.7. TemplateBasedTransformation

Description

Ce concept décrit une transformation implémentée avec l'approche basée sur les template. L'approche par template consiste à définir des canevas pour les modèles cibles souhaités ou modèles template. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les informations contenues dans le modèle source.

Généralisation

- TransformationImpl

Associations

- modelTemplate: Model [1]. Cette association spécifie le modèle template (modèle paramétré) d'une transformation. Ce modèle est l'équivalent du programme de la transformation dans l'approche par programmation. Au moment de l'exécution, les paramètres du modèle template seront substitués par des informations contenues dans le modèle source.

III.6.3.8. HumanActor

Description

Ce concept définit un acteur humain qui joue un ou plusieurs rôles au moment de la mise en œuvre.

Attribut

- familyName : String. Cet attribut spécifie le nom de famille de l'acteur humain.
- givenName : String. Cet attribut spécifie le prénom de l'acteur humain.
- address : String. Cet attribut spécifie l'adresse de l'acteur humain.
- occupation : String. Cet attribut spécifie la fonction de l'acteur humain.
- email : String. Cet attribut spécifie l'adresse électronique de l'acteur humain.
- isResponsible : Boolean. Cet attribut spécifie si l'acteur humain est le responsable de l'activité.
- login : String. Cet attribut spécifie le compte utilisateur de l'acteur pour se connecter à son espace de travail.
- password : String. Cet attribut spécifie le mot de passe de l'acteur pour se connecter à son espace de travail.

Association

- playedRole : RoleUse [1..*]. Cette association spécifie les rôles joués par l'acteur humain dans un projet de développement.

III.6.4. Comportements par défaut fournis par SPEM4MDE

Grâce à la notion d'états et de transitions qu'elles définissent, les machines à états sont appropriées pour observer et contrôler les processus à l'exécution. Une machine à états décrit le cycle de vie d'un élément de processus en répertoriant l'ensemble de ses états au cours de la mise en œuvre. Dans notre approche nous distinguons deux types de transition : automatique et manuelle. Une transition automatique spécifie une condition booléenne pour passer d'un état à un autre. Une transition manuelle appelée *opérateur de mise en œuvre* dans notre approche est une opération dont l'exécution déclenche le passage d'un état à un autre. La connaissance à tout instant de l'état de chaque élément du processus permet d'avoir une vision globale de l'évolution du processus mis en œuvre (projet) afin de mieux le maîtriser. Nous réutilisons les machines à états d'UML afin de décrire les comportements génériques des éléments d'un processus. Ces comportements peuvent donc être adaptés ou réutilisés pour un processus spécifique.

III.6.4.1. Comportement d'une transformation

Le comportement par défaut d'une transformation est illustré par la Figure III.8 ci-dessous. Nous distinguons trois états composites : *NOTRunning*, *Running* et *Finished*. Après l'instanciation du processus, la transformation passe à l'état initial *Executable*.

La transition de l'état *Executable* à l'état *Startable* est déclenchée automatiquement quand la fonction *startable*() retourne la valeur vraie (i.e. la précondition et les précédences sont satisfaites). A ce stade la transformation est éligible c'est-à-dire que toutes les conditions sont réunies pour sa mise en œuvre (exécution). La transformation passe de l'état *Startable* à l'état *Executing* (i.e. la transformation est en cours d'exécution) quand l'opérateur « *run* » est exécuté.

A partir de l'état *Executing*, il y a trois options : soit on termine la transformation (opérateur «

finish »), soit on annule la transformation (opérateur « *cancel* »), soit l'invariant associé à la transformation n'est plus vérifié. Si l'opérateur « *finish* » est exécuté alors la transformation passe à l'état *Terminated*. Si l'opérateur « *cancel* » est exécuté alors la transformation passe à l'état *Aborted* (i.e. l'exécution de la transformation a échoué). Si l'invariant de la transformation n'est plus vrai alors celle-ci passe à l'état *Inconsistent*.

A partir de l'état *Inconsistent* il y a quatre options : soit réconcilier la transformation avec le processus en cours (opérateur « *reconcile* »), soit annuler la transformation (opérateur « *cancel* »), soit terminer la transformation (opérateur « *finish* »), soit retravailler la transformation (opérateur « *rework* »). Si l'opérateur « *reconcile* » est exécuté alors la transformation repasse à l'état *Executing*. Si l'opérateur « *cancel* » est exécuté alors la transformation passe à l'état « *Aborted* ». Si l'opérateur « *finish* » est exécuté alors la transformation passe à l'état *Invalidated*. Si l'opérateur « *rework* » est exécuté alors la transformation repasse à l'état initial *Executable*.

A partir de l'état *Terminated*, la transformation est validée lorsque la fonction *validate()* retourne la valeur vraie (par exemple la postcondition est vérifiée). Elle est invalidée lorsque la postcondition n'est pas vérifiée.

A partir de l'état *Invalidated*, l'application de l'opérateur « *rework* » fait repasser la transformation à l'état initial *Executable*.

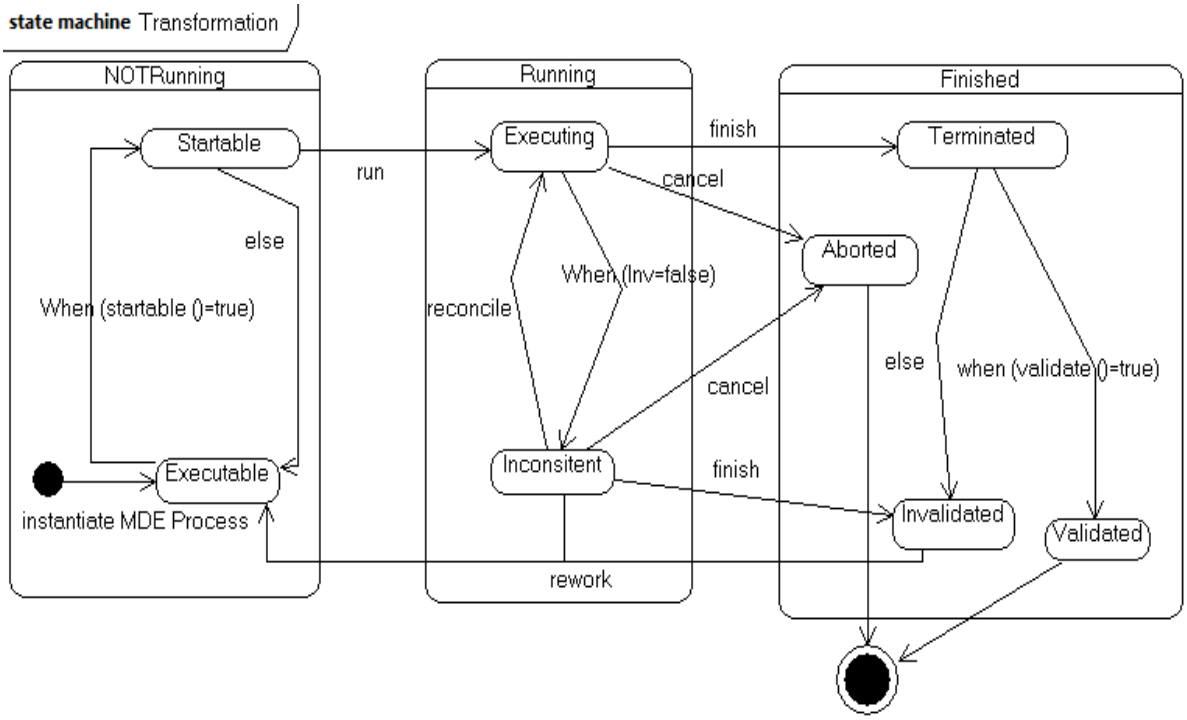


Figure III.8. Comportement par défaut d'une transformation

III.6.4.2. Comportement d'une activité (Activity)

Le comportement par défaut d'une activité est illustré par la Figure III.9 ci-dessous. Nous distinguons trois états composites : *NOTRunning*, *Running* et *Finished*. Après l'instanciation du processus, l'activité passe à l'état initial *Enactable*.

La transition de l'état *Enactable* à l'état *Startable* est déclenchée automatiquement quand la fonction *startable* () retourne la valeur vraie (i.e. la précondition et les précédences sont satisfaites). A ce stade l'activité est éligible c'est-à-dire toutes les conditions sont réunies pour sa mise en œuvre. L'activité passe de l'état *Startable* à l'état *Enacting* (i.e. l'activité est en cours de mise en œuvre) quand l'opérateur « *launch* » est exécuté.

A partir de l'état *Enacting*, il y a cinq options : soit on continue la mise en œuvre de l'activité (opérateur « *perform* »), soit on arrête la mise en œuvre de l'activité (opérateur « *finish* »), soit on annule la mise en œuvre de l'activité (opérateur « *cancel* »), soit on suspend la mise en œuvre de l'activité (opérateur « *suspend* »), soit l'invariant associé à l'activité n'est plus vérifié. Si l'opérateur « *perform* » est exécuté alors l'activité reste à l'état *Enacting*. Si l'opérateur « *finish* » est exécuté alors l'activité passe à l'état *Terminated*. Si l'opérateur « *cancel* » est exécuté alors l'activité passe à l'état *Aborted* (i.e. la mise en œuvre de l'activité est annulée). Si l'opérateur « *suspend* » est exécuté alors l'activité passe à l'état *Suspended* (i.e. la mise en œuvre de l'activité est suspendue pour une période). Si l'invariant de l'activité n'est plus vrai alors celle-ci passe à l'état *Inconsistent*.

A partir de l'état *Suspended*, la mise en œuvre est reprise lorsque l'opérateur « *resume* » est exécuté. L'activité repasse donc à l'état *Enacting*.

A partir de l'état *Inconsistent* il y a quatre options : soit réconcilier l'activité avec le processus en cours (opérateur « *reconcile* »), soit annuler l'activité (opérateur « *cancel* »), soit terminer l'activité (opérateur « *finish* »), soit retravailler l'activité (opérateur « *rework* »). Si l'opérateur « *reconcile* » est exécuté alors l'activité repasse à l'état *Enacting*. Si l'opérateur « *cancel* » est exécuté alors l'activité passe à l'état « *Aborted* ». Si l'opérateur « *finish* » est exécuté alors l'activité passe à l'état *Invalidated*. Si l'opérateur « *rework* » est exécuté alors l'activité repasse à l'état initial *Enactable*.

A partir de l'état *Terminated*, l'activité est validée lorsque la fonction *validate ()* retourne la valeur vraie (par exemple la postcondition est vérifiée). Elle est invalidée lorsque la postcondition n'est pas vérifiée.

A partir de l'état *Invalidated*, l'application de l'opérateur « *rework* » fait repasser l'activité à l'état initial *Enactable*.

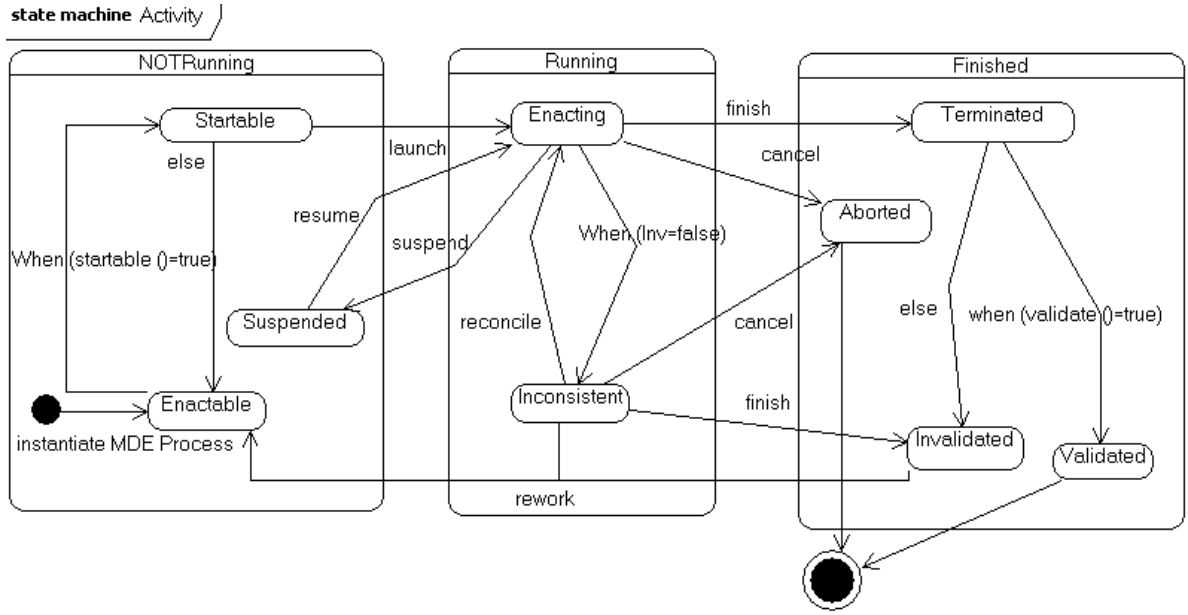


Figure III.9. Comportement par défaut d'une activité

III.6.4.3. Comportement d'un modèle créé par une activité/transformation

Le comportement par défaut d'un modèle créé (voir Figure III.10) est relatif au comportement de l'activité/transformation W qui l'a créé. Lorsque le processus est instancié, le modèle se trouve à l'état initial *Defined* (i.e. le modèle est juste défini dans le processus). L'état *Defined* est l'équivalent de l'état « *Enactable* » pour une activité et de l'état *Executable* pour une transformation. Lorsque l'activité/transformation W échoue le modèle reste à l'état *Defined.* Si l'activité/transformation qui produit le modèle se trouve à l'état *Terminated* alors le modèle produit passe automatiquement à l'état *InitialVersion* (i.e. la version initiale du modèle est créée). A partir de l'état *InitialVersion,* on peut appliquer soit (1) l'opérateur « *finish* » , soit (2) l'opérateur *« startRefactonring ».*

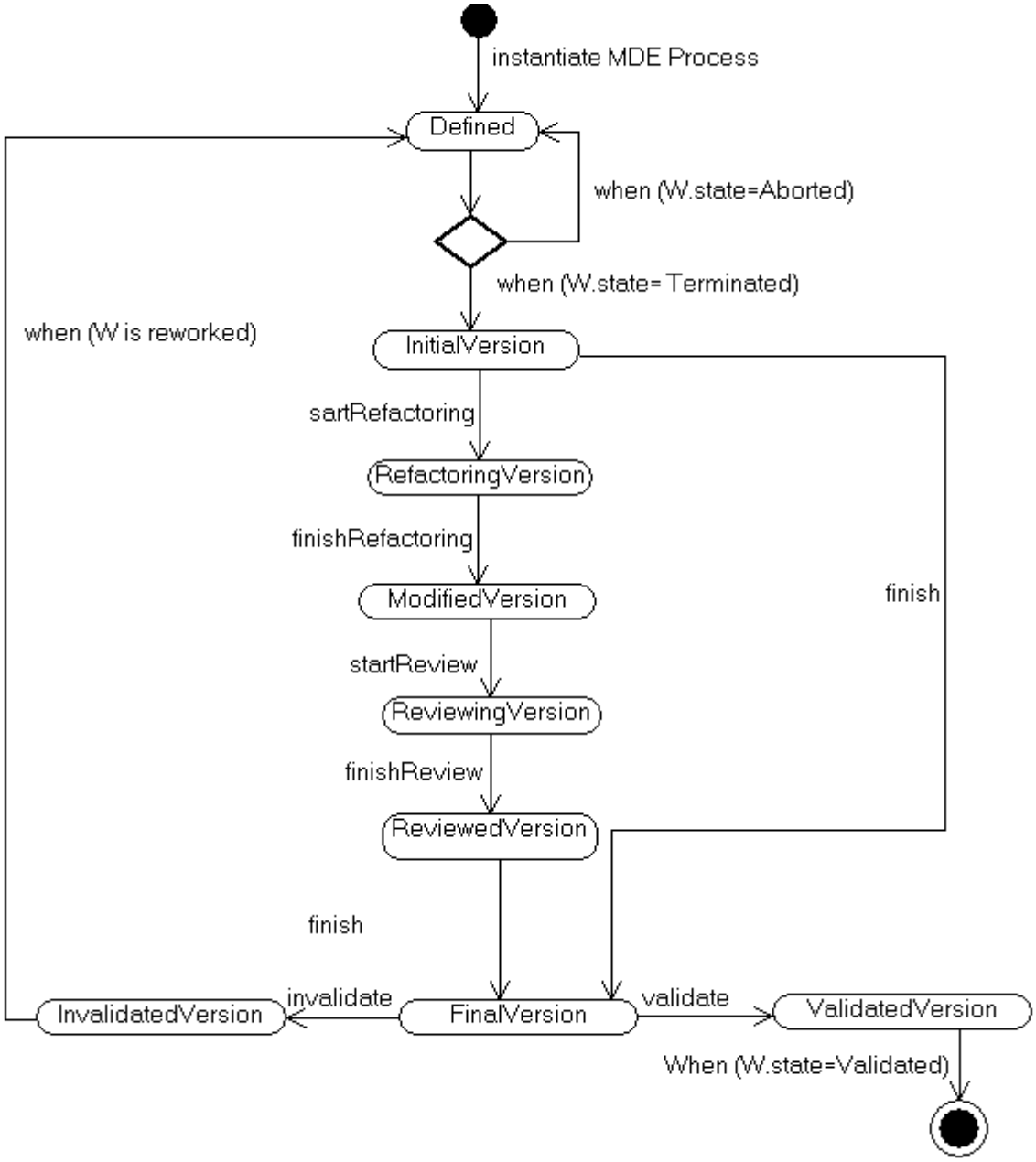


Figure III.10. Comportement par défaut d'un modèle créé par une activité/transformation

(1) L'application de l'opérateur « *finish* » sur le modèle initial fait passer celui-ci à l'état *FinalVersion* (i.e. le modèle se trouve à l'état final). A partir de l'état *FinalVersion*, on peut soit valider le modèle soit l'invalider. L'application de l'opérateur « *validate* » sur le modèle final fait passer celui-ci à l'état *ValidatedVersion* alors que l'opérateur « in*validate* » fait passer le modèle à l'état *InvalidatedVersion*. Si le modèle est invalidé et l'activité/transformation qui l'a produit est retravaillé alors le modèle repasse à l'état *Defined*. Si le modèle est validé et l'activité/transformation qui l'a produit est validée alors le cycle de vie se termine.

(2) L'application de l'opérateur *« startRefactonring* sur le modèle initial fait passer celui-ci à l'état *RefactoringVersion* (i.e. le modèle est en cours de restructuration en vue d'améliorer sa lisibilité). L'opérateur *« finishRefactonring »* termine l'activité de restructuration du modèle, celui-ci passe donc à l'état *ModifiedVersion* (i.e. le modèle a été modifié). A partir de l'état *ModifiedVersion,* on peut appliquer l'opérateur *« startReview* sur le modèle modifié, celui-ci passe donc à l'état *ReviewingVersion* (i.e. le modèle est en cours de revue). L'opérateur *« finishReview »* termine la revue du modèle, celui-ci passe donc à l'état *ReviewedVersion* (i.e. la revue du modèle est terminée). A partir de l'état *ReviewedVersion,* on peut appliquer l'opérateur *« finish »* sur le modèle «reviewé », celui-ci passe donc à l'état *FinalVersion* (i.e. le modèle se trouve à l'état final). A partir de l'état *FinalVersion*, on peut soit valider le modèle soit l'invalider. L'application de l'opérateur « *validate* » sur le modèle final fait passer celui-ci à l'état *ValidatedVersion* alors que l'opérateur « in*validate* » fait passer le modèle à l'état *InvalidatedVersion*. Si le modèle est invalidé et l'activité/transformation qui l'a produit est retravaillé alors le modèle repasse à l'état *Defined*. Si le modèle est validé et l'activité/transformation qui l'a produit est validée alors le cycle de vie se termine.

III.6.4.4. Comportement d'un modèle modifié par une activité/transformation

Le comportement par défaut d'un modèle modifié (voir Figure III.11) est relatif au comportement de l'activité/transformation W qui l'a modifié. Le comportement d'un modèle modifié par une activité/transformation est similaire au comportement d'un modèle créé par une activité/transformation. La principale différence se trouve au niveau des états initiaux. Le cycle de vie d'un modèle modifié démarre avec l'état « *ValidatedVersion* » alors que celui d'un modèle créé démarre avec l'état « *Defined* ». Au fait le modèle qu'on veuille modifier a été produit par une autre activité/transformation, c'est ce qui explique que son cycle de vie démarre avec l'état « *ValidatedVersion* ».

Figure III.11. Comportement par défaut d'un modèle modifié par une activité/transformation

III.6.4.5. Comportement d'un outil MDE requis pour une activité/ transformation

Le comportement par défaut d'un outil IDM (voir Figure III.12) est relatif au comportement de l'activité/transformation W pour laquelle il est requis. Quand le processus est instancié, l'outil se trouve à l'état *Required* (i.e. l'outil est requis pour l'activité/transformation). C'est l'équivalent de l'état *Defined* pour un modèle créé.

L'application de l'opérateur « *open* » sur l'outil fait passer celui-ci à l'état *Opened* (i.e. l'outil est ouvert). L'opérateur « *open* » s'accompagne d'un ensemble d'actions qui permettent d'ouvrir

automatiquement l'outil requis pour la mise en œuvre de l'activité/transformation. Cet opérateur ne peut être exécuté que par le responsable de l'activité/transformation.

A partir de l'état *Opened*, l'application de l'opérateur « *launch* ») fait passer l'outil de l'état *Opened* à l'état *Using* (i.e. l'outil est en train d'être utilisé dans la mise en œuvre de l'activité/transformation W). L'opérateur « *launch* » ne peut être exécuté que si l'activité/transformation W se trouve à l'état « *Startable*. »

Si l'activité/transformation W se trouve à l'état *Terminated* alors l'outil passe automatiquement à l'état *Used* (i.e. l'outil est déjà utilisé dans la mise en œuvre de l'activité/transformation W). A partir de l'état *Used*, l'outil peut être relancé (opérateur« *re-launch* ») en vue de ré-exécuter l'activité/transformation. L'application de l'opérateur « *re-launch* » fait repasser l'outil à l'état *Using*.

Quand l'opérateur « *close* » est exécutée alors l'outil passe à l'état *Closed* (i.e. l'outil est fermé). A partir de l'état *Closed*, l'application de l'opérateur *re-open* fait repasser l'outil à l'état *Opened*. Toujours à partir de l'état *Closed*, si l'activité/transformation W pour laquelle l'outil est requis se trouve à l'état *Validated* alors le cycle de vie se termine.

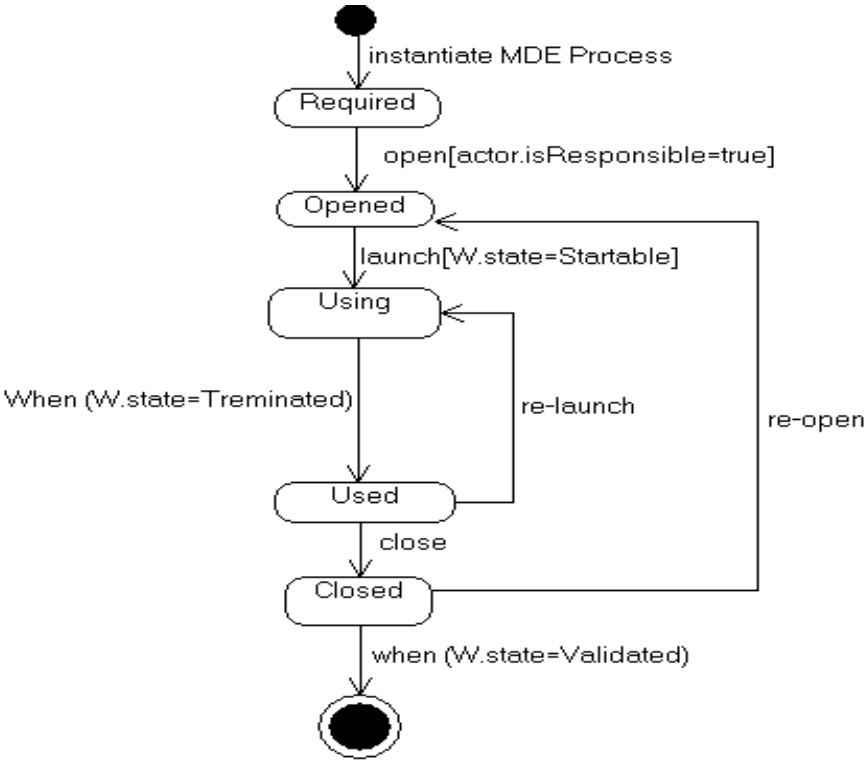


Figure III.12. Comportement par défaut d'un outil MDE

III.6.4.6. Comportement d'un rôle participant à une activité/transformation

Le comportement par défaut d'un rôle (voir Figure III.13) est relatif au comportement de l'activité/ transformation W à laquelle il participe. Lorsque le processus est instancié, le rôle se trouve à l'état *Assigned* (i.e. le rôle est assigné à l'activité/transformation). Si l'activité/transformation à laquelle il participe se trouve dans l'état *Enacting/Executing* alors le rôle passe à l'état *Performing* (i.e. il est en train de participer à la mise en œuvre de l'activité/transformation).

Si l'activité/transformation W se trouve à l'état composite *Finished* (*Terminated*, *Aborted*, *Validated*, ou *Invalidated*) alors le rôle passe à l'état *Performed* (i.e. le rôle a déjà participé à la mise en œuvre de l'activité/transformation). A partir de l'état *Performed*, si l'activité/transformation W se trouve à l'état *Aborted* alors le rôle repasse à l'état *Assigned*. Toujours à partir de l'état *Performed*, si l'activité/transformation est retravaillée (opérateur « rework ») alors le rôle repasse à l'état *Performing*. Finalement, si l'activité/transformation W à laquelle le rôle participe est validée alors le cycle de vie se termine.

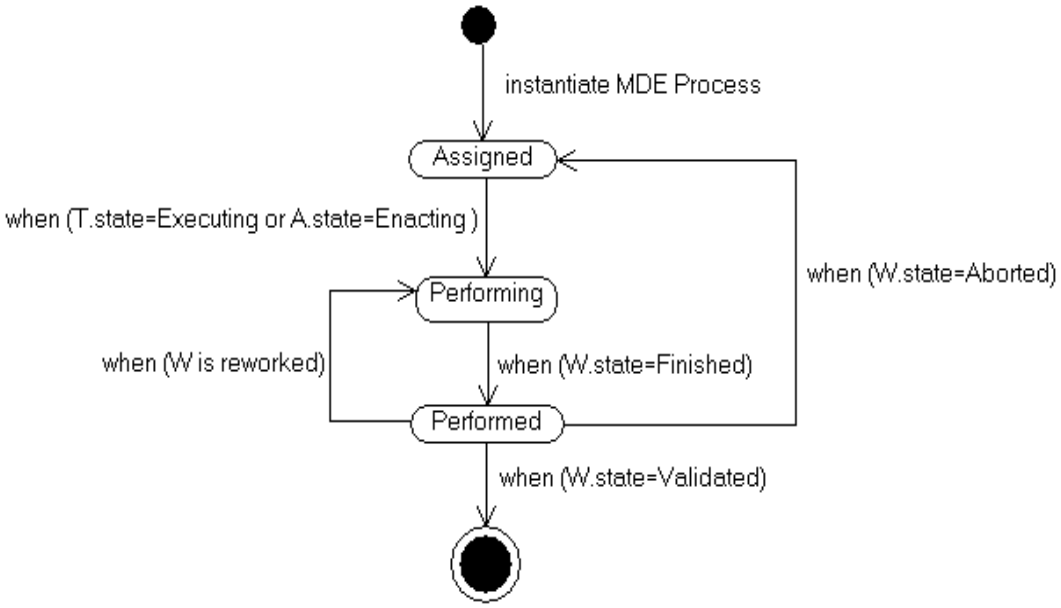


Figure III.13. Comportement par défaut d'un rôle

III.6.5. Exemple illustratif du volet comportemental

Pour illustrer l'aspect comportemental, nous considérons un extrait du processus qui a servi d'illustration dans le volet structurel de SPEM4MDE (voir section III.5.4). Cet extrait de processus (voir Figure III.14) est limité à la définition de la transformation « *Class Diag To Data Base* » avec son métamodèle source «*Class Diag Metamodel* », son métamodèle cible « *Data Base Schema* » et son rôle « *Transformation Designer*» qui va implémenter les règles informelles de la transformation. La description comportementale de cet extrait sera faite en deux parties : une première partie qui décrit l'implémentation de la transformation « *Class Diag To Data Base* » et une seconde partie qui décrit les modèles comportementaux de l'extrait du processus.

III.6.5.1. Description de l'implémentation de la transformation

La première partie décrit l'implémentation de la transformation « *Class Diag To Data Base* » (voir Figure III.14). La transformation «*Class Diag To Data Base Impl* » est liée à sa définition («*Class Diag To Data Base* ») à travers trois associations stéréotypées : *implements*» pour la transformation implémentée, «*conformsTo*» pour les modèles d'entrée et de sortie, et «*plays*» pour l'acteur humain. La transformation « *Class Diag To Data Base Impl* » a pour modèle source « *Class Diagram* » conforme au métamodèle «*Class Diag Metamodel* » et pour modèle cible « *Data Base Schema* » conforme au métamodèle «*Data Base*

Metamodel ». L'acteur Bob qui joue le rôle de développeur utilise l'outil « Medini QVT » pour exécuter la transformation « *Class Diag To Data Base Impl* ».

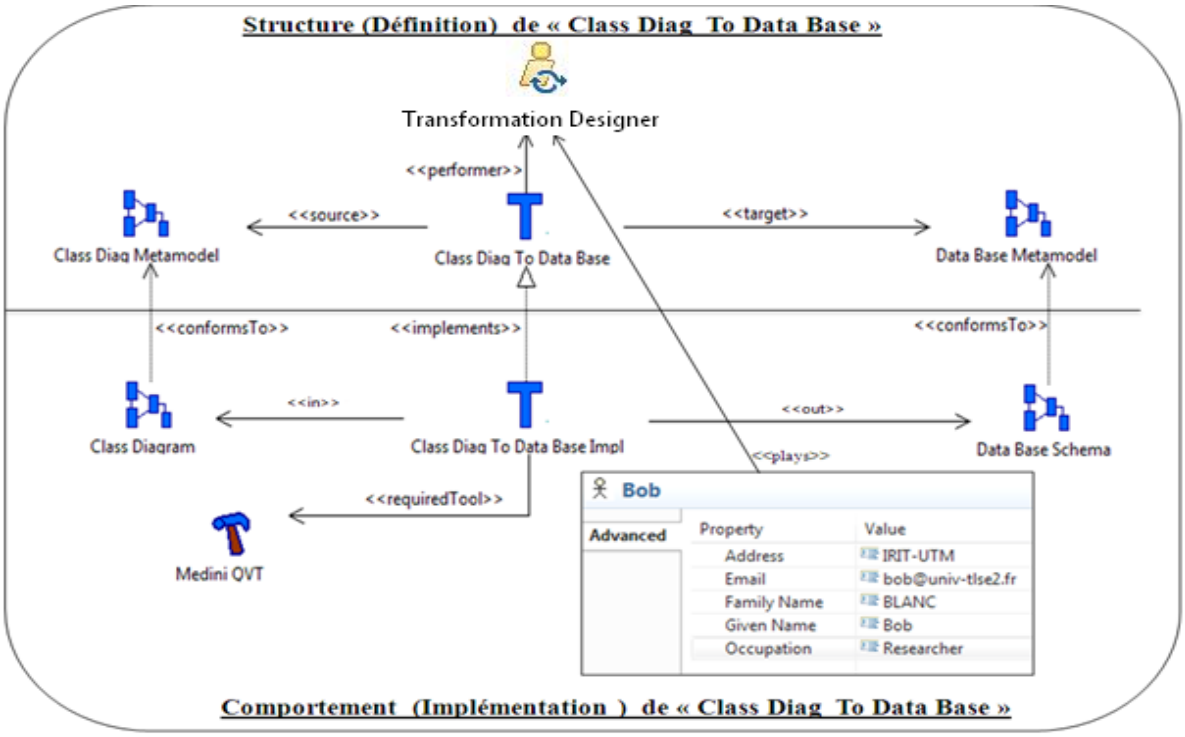


Figure III.14 Exemple d'une implémentation de transformation en SPEM4MDE

Les règles ci-après (voir Listing III.1) associées à la transformation « *Class Diag To Data Base Impl* » sont une implémentation en *QVTRelation* des règles informelles spécifiées dans l'exemple du volet structurel. Ces règles spécifient les associations entres les concepts du métamodèle source «*Class Diag Metamodel* » (voir Figure III.15) et ceux du métamodèle cible « *Data Base Schema* » (voir Figure III.16). La Figure III.15 représente une version simplifiée du métamodèle « *Class Diag Metamodel* ». Ce métamodèle décrit la métaclasse « Class » contenant un ensemble d'attributs ou propriétés spécifiés par la métaclasse « Property », et un ensemble d'opérations spécifiées par la métaclasse « Operation ». Une association lie deux ou plusieurs rôles (« memberEnd »), chacun spécifiant une référence vers une classe. Les types des attributs sont spécifiés par la métaclasse « DataType ».

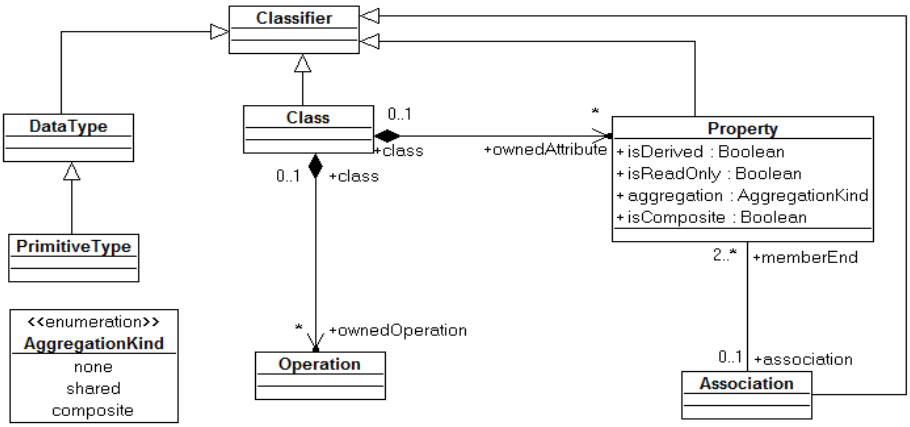


Figure III.15 Version simplifiée du métamodèle « Class Diag Metamodel »

La Figure III.16 représente une version simplifiée du métamodèle « *Data Base Metamodel* ». Ce métamodèle décrit la métaclasse « *Table* » contenant un ensemble de colonnes spécifiées par la métaclasse « *Column* », un ensemble de clés candidates spécifiées par la métaclasse « *Key* », et un ensemble de clés étrangères spécifiées par la métaclasse «*ForeignKey* ». Chaque table a une seule clé primaire choisie parmi une des clés candidates. Un clé candidate est liée à une ou plusieurs colonnes. Une clé étrangère est associée à une ou plusieurs colonnes. Chaque colonne est associée à un type de donnée spécifié par la métaclasse « *BaseDataType* ».

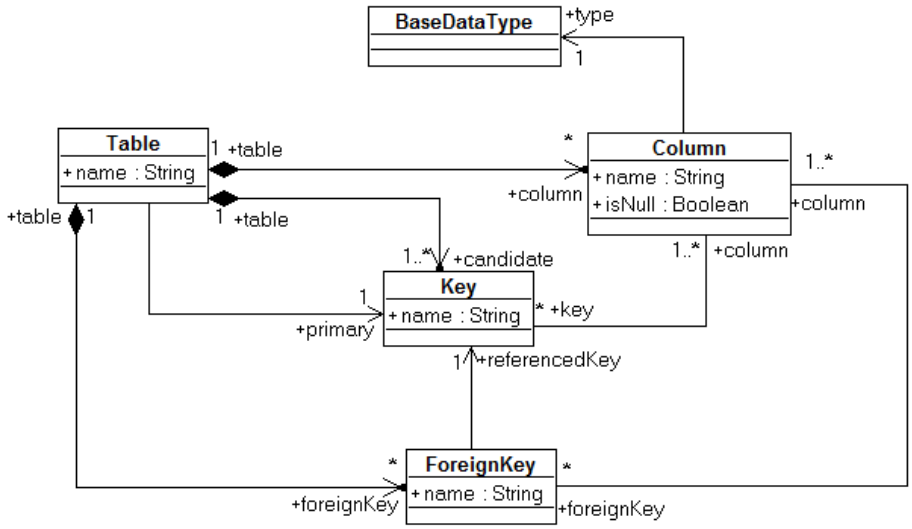


Figure III.16 Version simplifiée du métamodèle « Data Base Metamodel »

```
Transformation ClassDiagToDataBaseImpl (cd:Class Diag Metamodel, db:DataBaseMetamodel)
{
top relation ClassToTable {
checkonly domain cd cl: Class { name = name1 };
enforce domain db tb: Table { name = name1 };
}
top relation PropertyToColumn
{
checkonly domain cd p:Property { name = name2 };
enforce domain db c: Column { name = name2 };
}
top relation memberEndToForeignKey
{
checkonly domain cd m:memberEnd { name = name3 };
enforce domain db f: ForeignKey { name = name3 };
}
top relation DataTypeToBaseDataType
{
checkonly domain cd d: DataType { name = name4 };
enforce domain db b: BaseDataType { name = name4 };
}}
```

Listing III.1 Régles QVT de la transformation « *Class Diag To Data Base Impl* »

III.6.5.2. Description des modèles comportementaux

La seconde partie décrit les modèles comportementaux de la transformation « *Class Diag To Data Base Impl* », du modèle produit « *Data Base Schema* », de l'outil « *Medini QVT* », et de l'acteur Bob.

Pour décrire le modèle comportemental de la transformation « *Class Diag To Data Base Impl* », nous réutilisons le modèle comportemental d'une transformation (voir section III.6.4.1). Les corps des fonctions *startable*() et *validate*() n'étant pas définis dans ce modèle comportemental, nous allons les spécifier en OCL dans le cadre de la description comportementale de la transformation « *Class Diag To Data Base Impl* ». La fonction *startable*() spécifie si la transformation « *Class Diag To Data Base Impl* » est éligible ou non alors que la fonction *validate*() spécifie si la transformation « *Class Diag To Data Base Impl* » est valide ou non.

L'extrait du processus étant limité à la transformation « *Class Diag To Data Base Impl* », cette dernière n'a pas de précédence. La seule condition à vérifier pour que la fonction *startable*() prenne la valeur vraie est la validité du modèle d'entrée (diagramme de classe) de la transformation « *Class Diag To Data Base Impl* ». Le code OCL ci-dessous décrit le corps de la fonction *startable*().

Context « *Class Diag To Data Base Impl* »::startable () : Boolean

post: **result**=self.parameters→**select**(i|i.direction=#in)→**collect**(i | i.model)→ **forall**(m | m.state= ' ValidatedVersion')

La fonction *validate*() prend la valeur vraie si le modèle cible (schéma de la base de données) de la transformation « *Class Diag To Data Base Impl* » se trouve à l'état « *FinalVersion* » . Le code OCL ci-dessous décrit le corps de la fonction *validate*().

Context « *Class Diag To Data Base Impl* »::validate() : Boolean

post: **result**=self.parameters→**select**(i|i.direction=#out)→**collect**(i | i.model)→ **forall**(m | m.state= 'FinalVersion')

Pour décrire le modèle comportemental du modèle de sortie « *Data Base Schema* », nous réutilisons intégralement le modèle comportemental par défaut d'un modèle créé par une activité/transformation (voir section III.6.4.3).

La description du comportement de l'outil « *Medini QVT* » est entièrement basée sur le modèle comportemental par défaut d'un outil MDE requis pour la mise en œuvre d'une activité/transformation (voir section III.6.4.5).

La description du comportement de l'acteur « *Bob* » est basée sur le comportement d'un rôle qui participe à la mise en œuvre d'une activité/transformation (voir section III.6.4.6).

III.7. Conclusion

Dans ce chapitre nous avons présenté le métamodèle SPEM4MDE dédié à la modélisation et la mise en œuvre des processus IDM. SPEM4MDE étend certains concepts de SPEM par des concepts relatifs au développement IDM (transformation, modèle, métamodèle, outil IDM). La réutilisation de

SPEM 2.0 a pour principal intérêt de favoriser l'alignement de SPEM4MDE avec ce standard et de permettre ainsi une meilleure diffusion des concepts de SPEM4MDE. Pour réduire la complexité de SPEM4MDE, nous n'avons réutilisé que le paquetage *Process Structure* de SPEM 2.0 qui fournit les concepts basiques pour décrire la partie structurelle d'un processus.

L'exécution des modèles de processus n'est pas dans le champ d'intérêt de SPEM, même si le standard propose d'exécuter les modèles de processus dans un formalisme externe basé soit sur les diagrammes d'activités UML 2.0, soit sur les machines à états d'UML, soit sur la notation BPMN (Business Process Modeling Notation). Pour décrire le comportement des processus IDM, SPEM4MDE réutilise le paquetage d'UML 2.2 Superstructure qui décrit les machines à états d'UML. Le concept de machines à états est réutilisé afin de définir le comportement des éléments de processus. SPEM4MDE offre aussi un ensemble de comportements par défaut que les concepteurs de processus peuvent réutiliser ou adapter pour un processus spécifique.

SPEM4MDE réutilise aussi le standard QVT afin de spécifier le comportement d'une transformation, l'intérêt étant de tirer pleinement profit des outils d'implémentation et d'exécution liés à ce standard. La séparation de la définition d'une transformation de son implémentation dans SPEM4MDE permet de différer l'implémentation d'une transformation et par conséquent de laisser au développeur le libre choix du formalisme d'implémentation. La définition d'une transformation consiste à préciser ses métamodèles source et cible, et ses règles informelles. Quant à l'implémentation de la définition, elle consiste à décrire les modèles source et cible, et à implanter les règles informelles dans un formalisme interprétable par un outil de transformation.

CHAPITRE IV. SPEM4MDE-PSEE : UNE IMPLEMENTATION DE SPEM4MDE

IV.1. Introduction

Le métamodèle SPEM4MDE présenté dans le chapitre précédent offre des concepts qui permettent de décrire la structure et le comportement des processus IDM. Pour valider la spécification de ce métamodèle et assister les concepteurs/développeurs de processus dans la modélisation/mise en œuvre des processus IDM, nous avons développé un prototype d'un PSEE (Process-centered Software Engineering Environment) appelé SPEM4MDE-PSEE. Ce prototype, développé sous la plate-forme TOPCASED, fournit d'une part un éditeur graphique pour la modélisation structurelle et comportementale des processus IDM (« *SPEM4MDE Process Editor* ») et d'autre part un environnement de mise en œuvre (« *SPEM4MDE Process Enactment Engine* ») s'appuyant sur les modèles comportementaux des processus.

Ce chapitre est présenté comme suit. Dans la section IV.2 nous présentons la plate-forme TOPCASED que nous avons utilisée pour développer le prototype SPEM4MDE-PSEE. La section IV.3 présente l'architecture générale de SPEM4MDE-PSEE. La section IV.4 présente une description fonctionnelle des modules de SPEM4MDE-PSEE à travers les diagrammes de cas d'utilisation UML. La section IV.5 présente l'implémentation du prototype SPEM4MDE-PSEE en faisant ressortir le processus suivi pour générer l'éditeur graphique de modèles de processus SPEM4MDE (« *SPEM4MDE Process Editor* »). Nous concluons par la section IV.6 en faisant un bilan du prototype SPEM4MDE-PSEE.

IV.2. La plate-forme TOPCASED

IV.2.1. Vue d'ensemble de TOPCASED

La plate-forme TOPCASED (Toolkit in OPen-source for Critical Application and SystEms Development) est un environnement open-source pour le développement d'applications critiques et de systèmes embarqués et temps-réel.

L'environnement TOPCASED a pour principal objectif de simplifier la définition de nouveaux DSMLs (Domain Specific Modeling Languages) ou de langages de modélisation spécifiques à un domaine en fournissant des technologies de niveau méta telles que des générateurs d'éditeurs

syntaxiques (textuels et graphiques), des outils de vérification, de transformation et d'exécution de modèles et enfin des outils de génération de documentation. C'est une plate-forme qui suit les principes de l'IDM et qui est basée sur Eclipse. L'architecture de TOPCASED (Figure IV.1) intègre plusieurs outils basés sur Eclipse tels que l'environnement de métamodélisation EMF et le Framework GEF (Graphical Editing Framework) qui permet créer des éditeurs graphiques. TOPCASED supporte l'import, l'export et la gestion des versions des modèles.

TOPCASED définit un point d'accès (*ModelBus*) qui permet aux outils externes d'accéder à ses services selon le principe de l'encapsulation (sans connaître l'implémentation ou la localisation des services fournis). *ModelBus* utilise les points d'extension de l'environnement d'Eclipse pour déclarer les services. Un éditeur est déclaré via un fichier « *plugin.xml* », alors que les diagrammes de ce même éditeur sont déclarés via leur propre fichier « *plugin.xml* ». L'éditeur peut appeler ces diagrammes à travers *ModelBus* sans connaître leur localisation.

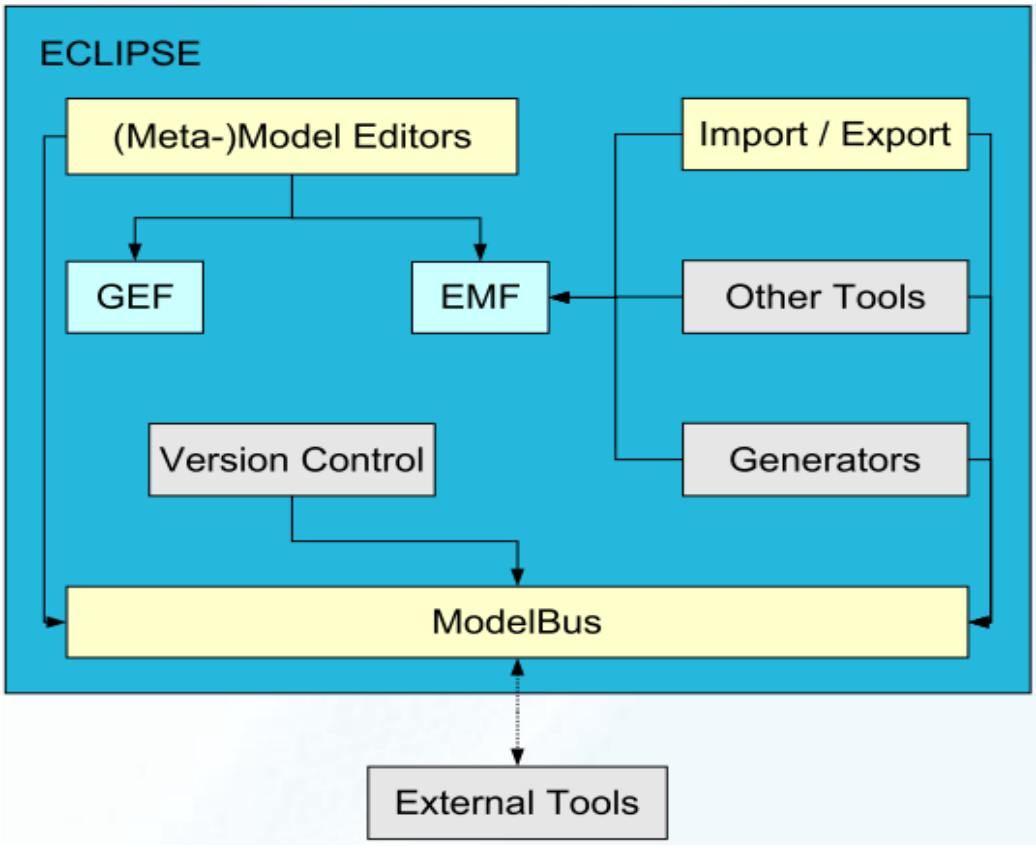


Figure IV.1 Architecture de la plate-forme TOPCASED [Farail et al., 2005]

IV.2.2. Fonctionnalités utilisées dans TOPCASED

Cette section présente essentiellement les fonctionnalités utilisées dans TOPCASED pour développer le prototype SPEM4MDE-PSEE. Ces fonctionnalités sont offertes par l'éditeur Ecore, les outils de configuration et génération d'éditeurs graphiques (*Editorconfigurator, Diagramconfigurator, Outlineconfigurator, Propertiesconfigurator*), et les outils d'édition et de vérification de contraintes OCL (*OCL Editor, OCL Checker*).

TOPCASED propose un ensemble d'éditeurs de modèles (UML, SAM, SysML, Ecore). Dans le cadre de la réalisation de notre prototype, nous avons utilisé l'éditeur Ecore pour décrire le métamodèle SPEM4MDE sous format Ecore.

La génération d'éditeurs graphiques a pour objectif d'outiller un langage de modélisation (UML, SPEM, etc.). Cette génération passe d'abord par la configuration d'un ensemble de modèles: le modèle de configuration de l'éditeur (*Editorconfigurator*), les modèles de configuration des diagrammes de l'éditeur (*Diagramconfigurator*), le modèle de configuration qui produit une vue graphique donnant un aperçu global des modèles réalisés avec l'éditeur (*Outlineconfigurator*) et le modèle de configuration qui produit une vue graphique qui permet de visualiser les propriétés des éléments d'un modèle (*Propertiesconfigurator*). Dans le cadre de la réalisation de notre prototype nous réutilisons ces fonctionnalités afin d'outiller le métamodèle SPEM4MDE.

TOPCASED offre aussi des outils qui supportent l'édition des contraintes d'un modèle exprimées en OCL (*OCL Editor*) et la vérification de ces contraintes (*OCL Checker*). TOPCASED supporte deux modes de vérification : la vérification à la demande (faite par l'utilisateur), et la vérification pendant l'édition de modèles (vérification automatique). Nous réutilisons ces fonctionnalités dans le prototype SPEM4MDE-PSEE pour vérifier les modèles de processus SPEM4MDE vis-à-vis de la sémantique statique du métamodèle SPEM4MDE.

IV.3. Architecture de SPEM4MDE-PSEE

La Figure IV.2 décrit l'architecture générale du prototype SPEM4MDE-PSEE. Dans cette architecture, le prototype est divisé en deux modules : « *SPEM4MDE Process Editor* » et « *SPEM4MDE Process Enactment Engine* ».

Le module « *SPEM4MDE Process Editor* » permet aux concepteurs de processus d'une part de décrire la structure et le comportement des modèles de processus IDM et d'autre part de les vérifier. La description des transformations contenues dans un processus sera entièrement à la charge du concepteur de transformation. Le module « *SPEM4MDE Process Editor* » est connecté aux outils d'implémentation de transformations (ATL, Medini QVT, etc.).

La description d'un modèle de processus IDM inclut la description de sa structure et de son comportement. Pour l'aspect comportemental, le concepteur de processus a le choix de réutiliser un comportement de base fourni par le métamodèle SPEM4MDE ou de l'adapter. Une fois le modèle du processus décrit, il faut procéder à sa vérification. La vérification consiste à exécuter les règles de bonne formation exprimées en OCL et/ou en Java. Il y a deux manières de vérifier les modèles de processus SPEM4MDE, soit à la demande (le concepteur de processus lance lui-même la vérification), soit durant l'édition (la vérification est faite automatiquement par « *SPEM4MDE Process Editor* »). La vérification à la demande est basée sur des règles de bonne formation exprimées en OCL alors que la vérification automatique est basée sur des règles de bonne formation exprimées en OCL et/ou Java. Les résultats de l'édition de processus (modèles de processus, règles de transformation) sont stockés dans une base appelée « *MDE Process Repository* ».

Le module «*SPEM4MDE Process Enactment Engine* » permet aux développeurs de mettre en œuvre un projet basé sur un modèle de processus SPEM4MDE. Le module « *SPEM4MDE Process Enactment Engine* » est connecté aux outils d'exécution de transformations (ATL, Medini QVT, etc.). Dans ce module, le chef de projet (*Project Manager*) sélectionne dans le « *MDE Process Repository* » un

modèle de processus SPEM4MDE et l'adapte selon les spécificités d'un projet. L'adaptation d'un modèle de processus générique consiste à le modifier éventuellement et à assigner toutes les ressources nécessaires à sa mise en œuvre (outils, développeurs, espaces de travail, etc.). Grâce aux modèles comportementaux sur lesquels s'appuie le module « *SPEM4MDE Process Enactment Engine* », les développeurs pourront ensuite mettre en œuvre le modèle de processus adapté. Ils pourront par conséquent connaître à tout instant l'état de chaque élément du processus et les opérateurs de mise en œuvre qui leur sont applicables.

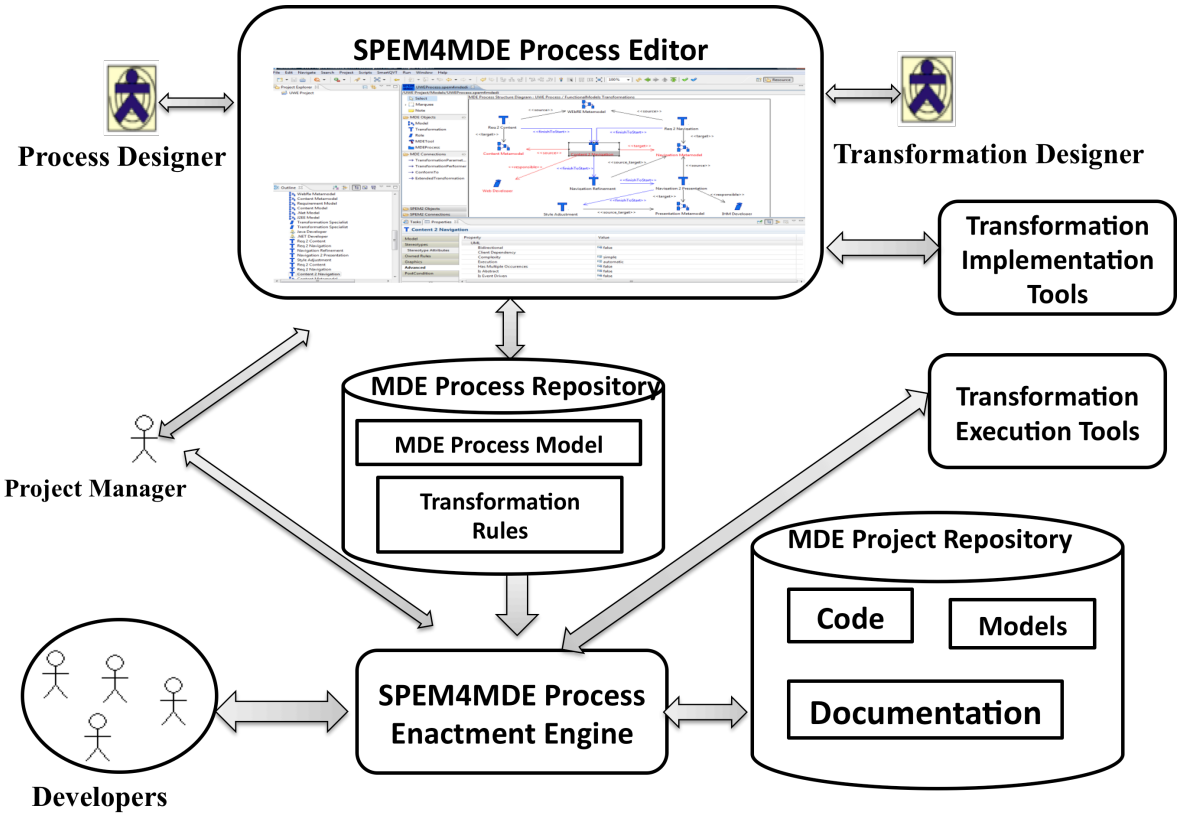


Figure IV.2. Architecture de SPEM4MDE-PSEE

IV.4. Description fonctionnelle de SPEM4MDE-PSEE

Cette section décrit les fonctionnalités des deux modules de SPEM4MDE-PSEE à travers les diagrammes de cas d'utilisation d'UML. Un diagramme de cas d'utilisation est un moyen d'exprimer les fonctionnalités d'un système. Il est composé de cas d'utilisation et d'acteurs liés à ces cas. Un cas d'utilisation est un moyen d'exprimer une fonctionnalité via une interaction entre un acteur et le système.

IV.4.1. Description fonctionnelle de « SPEM4MDE Process Editor »

Cette section décrit les fonctionnalités (voir Figure IV.3) du module « *SPEM4MDE Process Editor* ». Un concepteur de processus peut créer, modifier, vérifier ou rechercher un modèle de processus IDM. La création d'un modèle de processus IDM inclut la description de sa structure et de son comportement. La description de la structure inclut la description des règles de transformations

de modèles et la description des autres activités des processus. La description des règles d'une transformation est réalisée par un spécialiste de transformation. La vérification à la demande (faite par le concepteur) et la vérification durant l'édition de modèles (faite automatiquement) sont des cas particuliers de vérification de modèles.

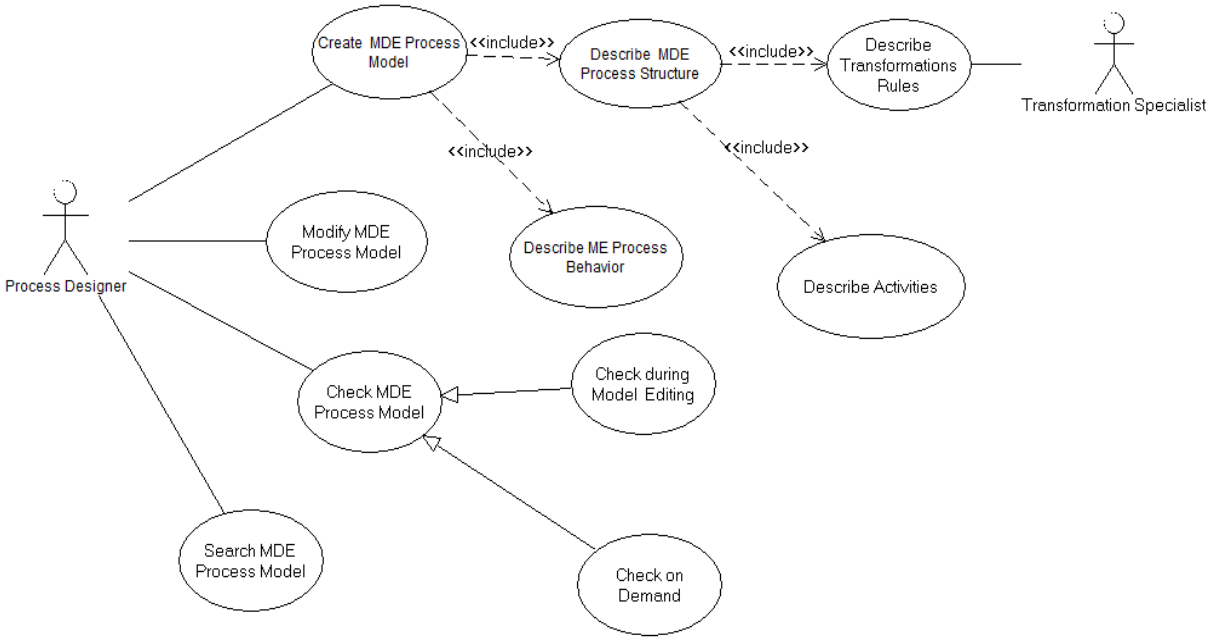


Figure IV.3 Diagramme de cas d'utilisation du module « *SPEM4MDE Process Editor* »

IV.4.2. Description fonctionnelle de « SPEM4MDE Process Enactment Engine »

Cette section décrit les fonctionnalités du module « *SPEM4MDE Process Enactment Engine* ». Ces fonctionnalités sont décrites selon les points de vue du chef de projet et du développeur. La Figure IV.4 décrit les fonctionnalités du module « *SPEM4MDE Process Enactment Engine* » relatives au chef de projet. Le chef de projet peut créer, modifier, ou rechercher un projet IDM. La création d'un projet IDM inclut la recherche d'un modèle de processus SPEM4MDE et son adaptation à un projet spécifique. L'adaptation d'un modèle de processus inclut la modification du modèle de processus et l'assignation des ressources nécessaires à sa mise en œuvre (création des acteurs, création des espaces de travail, assignation d'un espace de travail à un acteur).

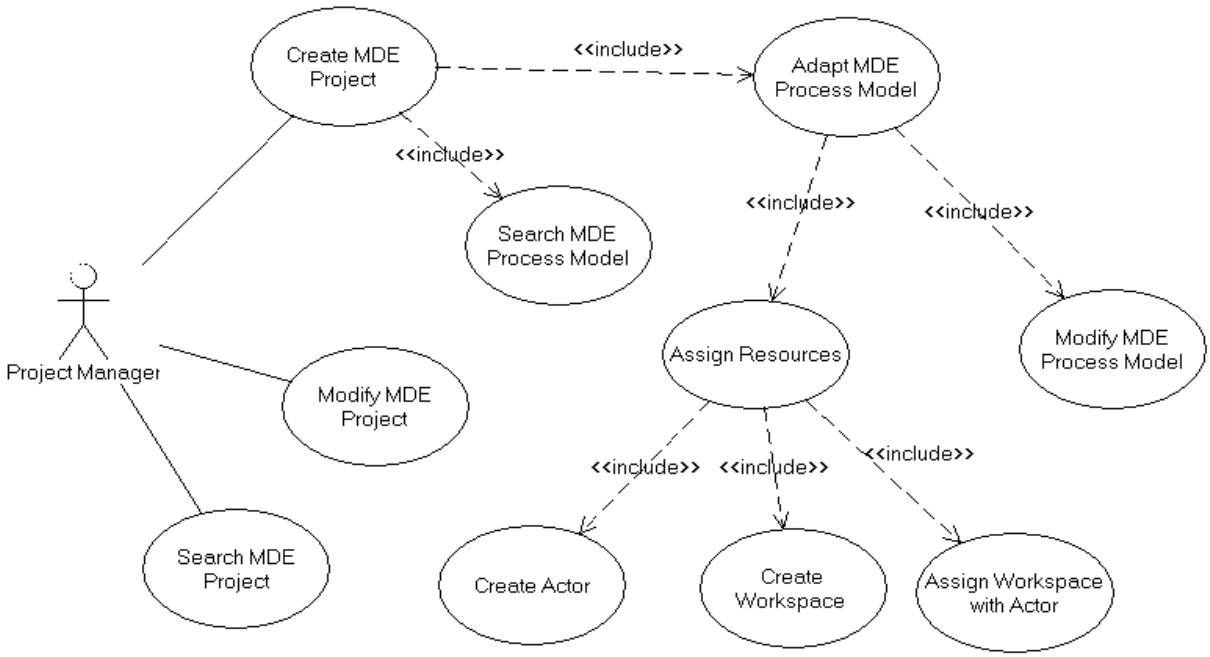


Figure IV.4 Diagramme de cas d'utilisation relatif au Chef de Projet

La Figure IV.5 décrit les fonctionnalités du module « *SPEM4MDE Process Enactment Engine* » relatives au développeur. Un développeur gère son espace de travail, réalise les activités qui se trouvent dans son agenda, et peut consulter à tout moment l'état des activités/transformations dont il a la charge. La mise en oeuvre d'une activité inclut le lancement des opérateurs de mise en œuvre associés à son modèle comportemental. L'exécution d'un opérateur de mise en œuvre consiste à changer l'état de l'élément de processus sur lequel porte cet opérateur, et à exécuter les actions associées à cet opérateur. Par exemple pour une transformation éligible, l'exécution de l'opérateur « *run* » permet de changer l'état de la transformation mais aussi de lancer l'outil requis pour exécuter la transformation.

L'édition de modèles, la vérification de modèles, et l'exécution d'une transformation de modèles sont des cas particuliers de mise en œuvre d'une activité. L'exécution d'une transformation inclut le choix de l'outil d'exécution.

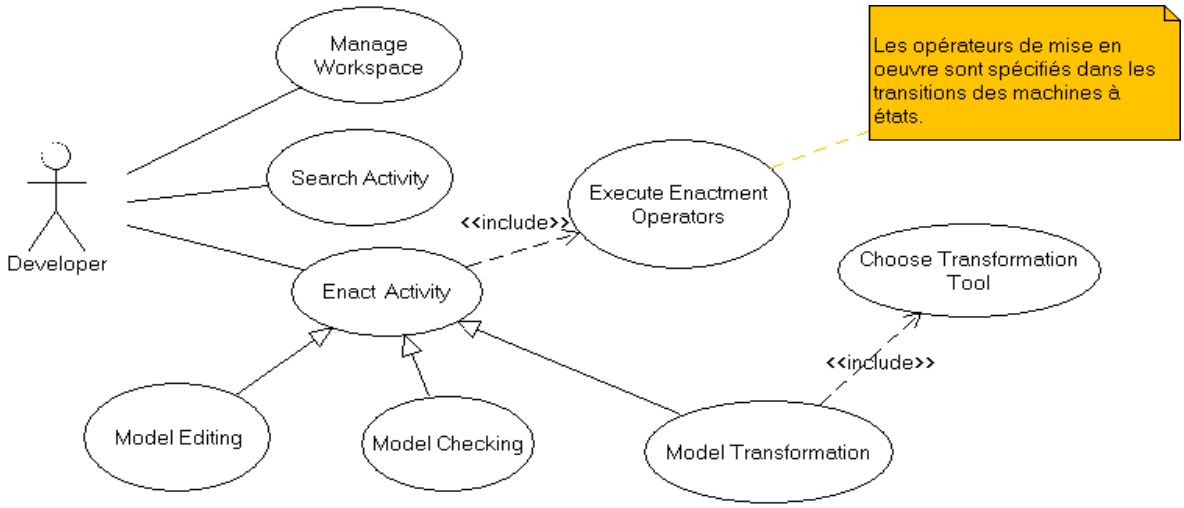


Figure IV.5 Diagramme de cas d'utilisation relatif au développeur

IV.5. Implémentation du prototype SPEM4MDE-PSEE

Cette section décrit l'implémentation des modules « *SPEM4MDE Process Editor* » et « *SPEM4MDE Process Enactment Engine* » du prototype SPE4MDE-PSEE. Le module « *SPEM4MDE Process Editor* » offre les diagrammes qui permettent de décrire graphiquement d'une part la structure et le comportement des processus IDM et d'autre par les relations entre modèles et éléments de modèles. Le module « *SPEM4MDE Process Enactment Engine* » s'appuie sur les modèles comportementaux produits par le module « *SPEM4MDE Process Editor* » pour fournir aux développeurs un environnement de mise en œuvre des modèles de processus SPEM4MDE.

IV.5.1. Implémentation du module « SPEM4MDE Process Editor »

IV.5.1.1. Les diagrammes de l'éditeur graphique de SPEM4MDE

La Figure IV.6 ci-dessous décrit les diagrammes du module « *SPEM4MDE Process Editor* ». Ces diagrammes qui sont au nombre de trois permettent de décrire graphiquement la structure et le comportement des processus IDM ainsi que les relations entre modèles et éléments de modèles. Le diagramme structurel décrit la vue graphique du paquetage « *MDE Process Structure* » de SPEM4MDE. Le diagramme comportemental décrit la vue graphique du paquetage « *MDE Process Behavior* » de SPEM4MDE. Enfin, le diagramme de relations entre modèles décrit la vue graphique du paquetage « *Model Relationship* » de SPEM4MDE.

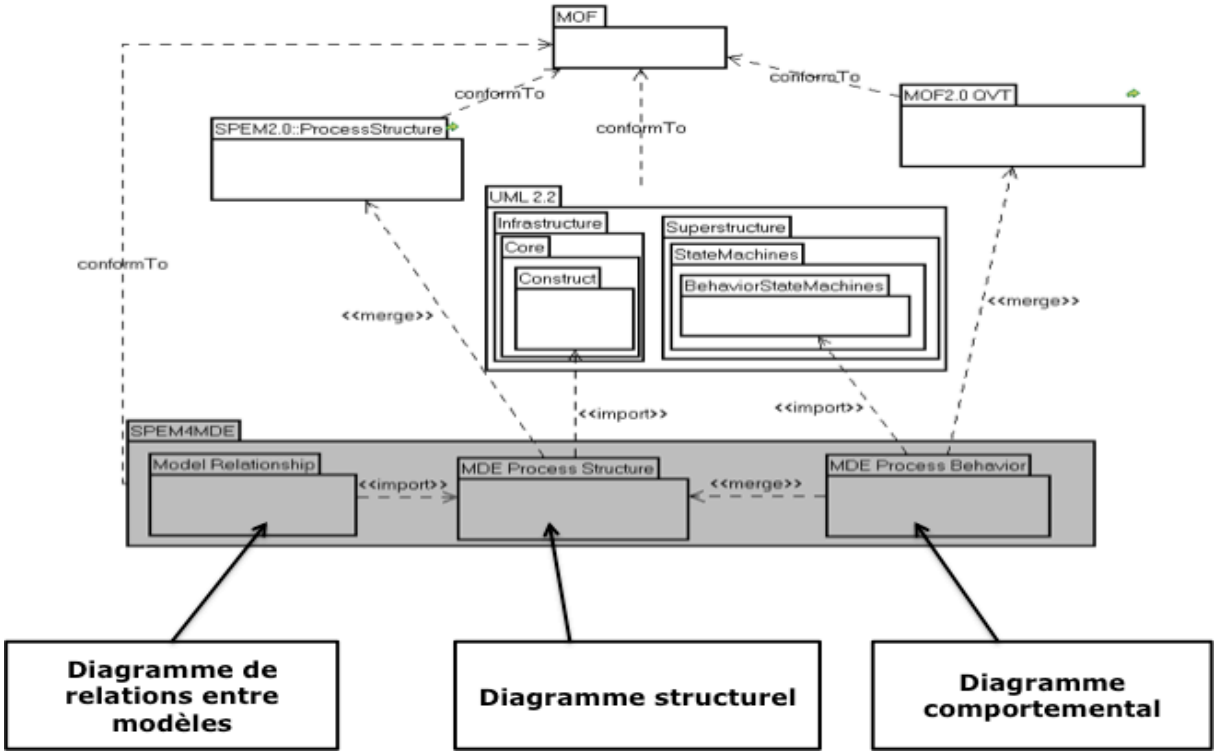


Figure IV.6 Relations entre les diagrammes de l'éditeur et les paquetages de SPEM4MDE

IV.5.1.2. Processus de réalisation de l'éditeur graphique de SPEM4MDE

La Figure IV.7 présente le processus suivi dans TOPCASED pour générer les différents composants de l'éditeur de processus « SPEM4MDE Process Editor ». Ce processus a été décrit avec le diagramme structurel de l'éditeur «SPEM4MDE Process E*ditor* ». Il débute avec la description du métamodèle SPEM4MDE sous format Ecore dont l'objectif est de générer les concepts de SPEM4MDE sous forme d'interfaces Java et de classes d'implémentation.

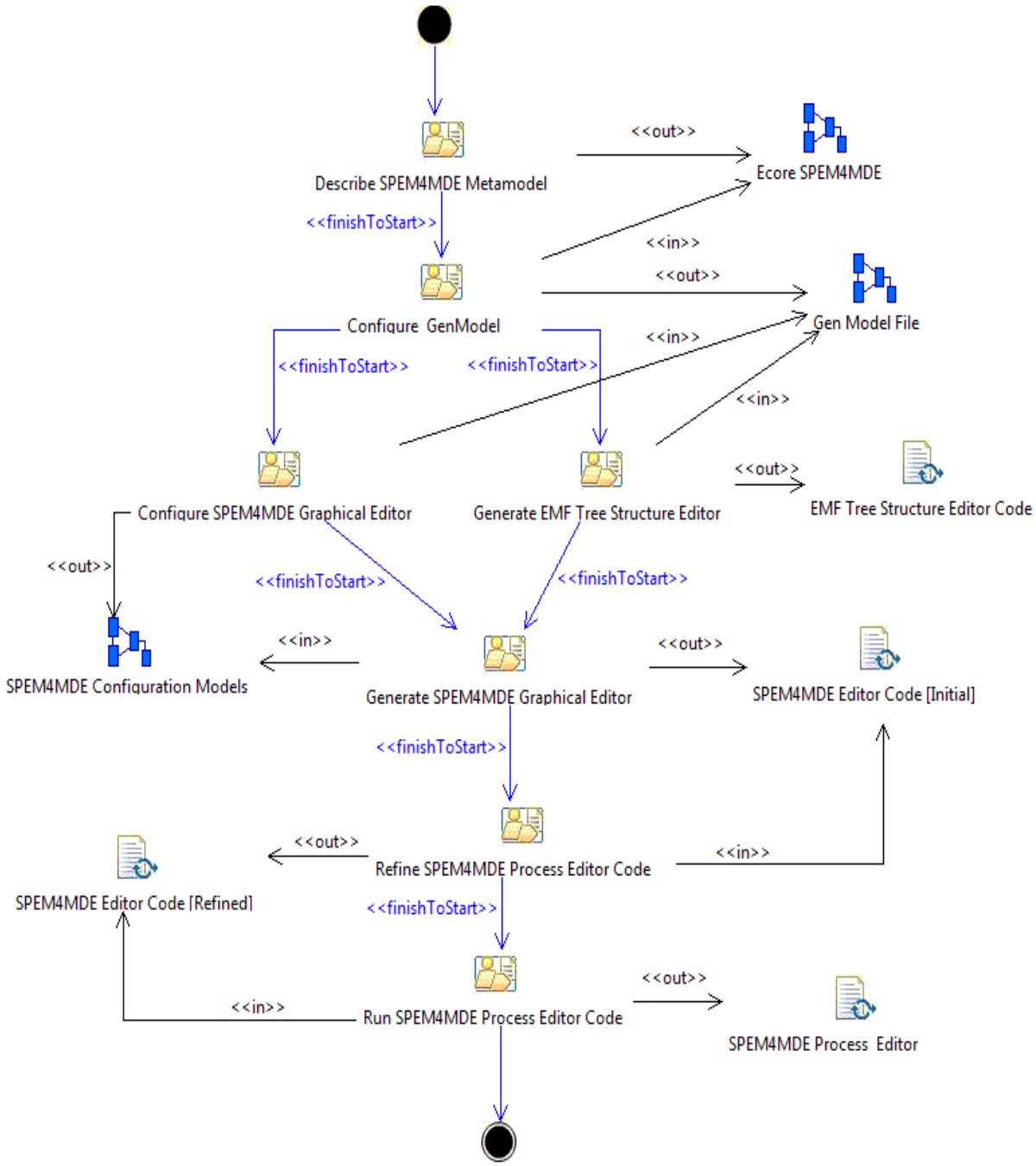


Figure IV.7. Diagramme structurel décrivant le processus de réalisation de l'éditeur graphique de SPEM4MDE

Par la suite, l'activité « *Configure GenModel* » prend en entrée le métamodèle Ecore de SPEM4MDE et produit le modèle « *GenModel* » qui contient les paramètres de génération de code Java. Puis, l'activité « *Generate EMF Tree Structure Editor* » prend en entrée le modèle « *GenModel* » pour produire le code Java des modèles EMF (Model, Edit, Editor) associés à SPEM4MDE.

La relation de précédence « *finishToStart* » spécifie que l'activité « *Configure GenModel* » ne peut démarrer que si l'activité « *Describe SPEM4MDE Metamodel* » est terminée et doit se terminer pour que les activités « *Generate EMF Tree Structure Editor* » et « *Configure Graphical Editor* » puissent démarrer.

L'activité « *Configure Graphical Editor* » prend en entrée le modèle « *GenModel* » pour produire les modèles de configuration de l'éditeur de SPEM4MDE et de ses diagrammes.

L'activité « *Generate Graphical Editor* » prend en entrée les modèles de configuration pour produire le code Java des composants de l'éditeur de SPEM4MDE. Par la suite, ce code est raffiné à travers l'activité «*Refine SPEM4MDE Process Editor Code* » pour introduire de nouvelles fonctionnalités (par exemple la réutilisation des composants de Papyrus, la vérification à la volée, etc.).

Enfin, l'exécution du code Java raffiné produit l'éditeur graphique de SPEM4MDE et ses différents diagrammes.

IV.5.1.2.1. L'activité « Configure GenModel »

Cette activité permet de configurer les paramètres du modèle de génération de code Java (*GenModel*) à partir de la version Ecore du métamodèle SPEM4MDE. Le modèle de génération est sauvegardé dans un fichier avec une extension (*genmodel*). Les paramètres du modèle de génération spécifient les répertoires de génération de code Java des composants d'EMF (*Model*, *Edit*, *Editor*).

IV.5.1.2.2. L'activité « Generate EMF Tree Structure Editor »

Cette activité permet de générer le code Java des modèles EMF (*Model*, *Edit*, *Editor*) à partir du modèle de génération paramétré. Le code généré contient des tags *(@generated*, *@generated not*, etc.) qui permettent de spécifier si certaines parties du code seront modifiées ou non au cours d'une régénération. *Model* contient les classes et interfaces Java qui décrivent les concepts de SPEM4MDE. *Edit* contient les classes Java qui permettent de manipuler les éléments d'un modèle de processus SPEM4MDE dans l'éditeur arborescent. *Editor* contient les classes Java de l'éditeur arborescent.

IV.5.1.2.3. L'activité « Configure SPEM4MDE Graphical Editor »

Cette activité permet de configurer les composants de l'éditeur graphique de SPEM4MDE en fournissant leurs modèles de génération. Cette activité est une activité composite (voir Figure IV.8) composée des activités de configuration de l'éditeur graphique et de ses différents diagrammes. Elle produit le modèle de configuration de l'éditeur graphique de SPEM4MDE (« SPEM4MDE.editorconfigurator ») et les modèles de configuration de ses diagrammes (« *MDEProcessStructure.diagramconfigurator* », « *ModelRelationship.diagramconfigurator* », et « *MDE ProcessBehavior.diagramconfigurator* »).

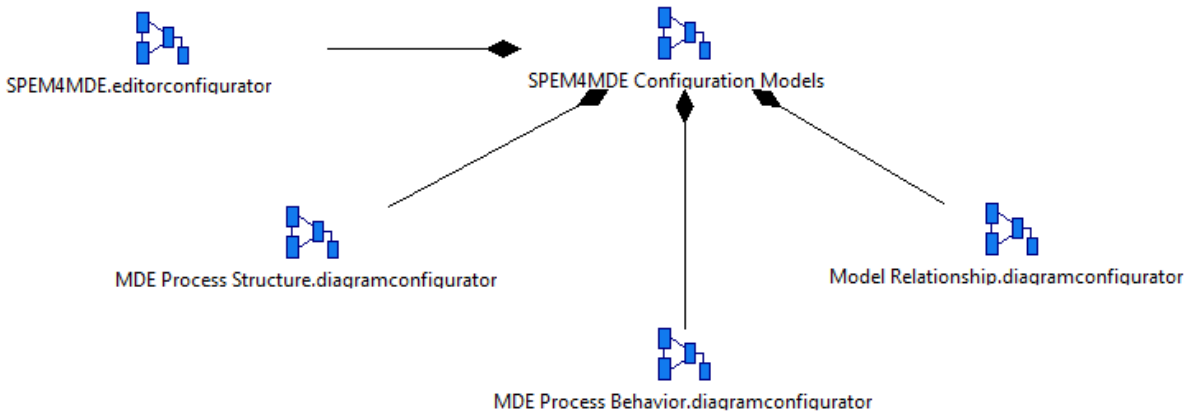


Figure IV.8 Les modèles de configuration de l'éditeur « *SPEM4MDE Process Editor* »

« *SPEM4MDE.editorconfigurator* » est le modèle de configuration de l'éditeur graphique de SPEM4MDE. Il contient cinq paramètres (voir Figure IV.9) (le nom de l'éditeur, le nom du projet ou plug-in, le modèle de génération utilisé, la version du plugin, et le provider).

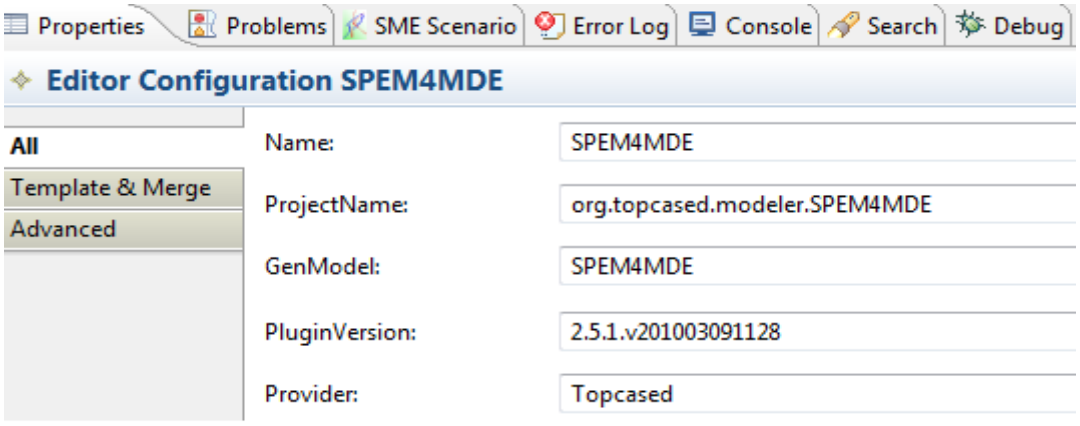


Figure IV.9 Les paramètres de « *SPEM4MDE*.editorconfigurator»

Les modèles de configuration des diagrammes (« *MDEProcessStructure.diagramconfigurator* », « *ModelRelationship.diagramconfigurator* » et « *MDEProcessBehavior.diagramconfigurator* ») définissent les diagrammes supportés par l'éditeur graphique de SPEM4MDE.

Un diagramme de configuration (*Diagram Configuration*) est composé de nœuds (*Node Part*) et de connexions (*Edge Part*). Chaque nœud référence un objet (*Model Object*) qui représente un concept de SPEM4MDE ou un objet simple (*Simple Object*) créé pour les besoins de l'éditeur. Une connexion (*Edge Part*) lie deux nœuds (la source et la cible) et peut comporter un nœud d'objet (*Edge Object*) qui peut représenter une propriété d'un concept de SPEM4MDE. A l'image d'un nœud, une connexion (*Edge Part*) référence un «*Model Object* » ou un «*Simple Object* ».

Le diagramme de configuration contient aussi le modèle de configuration de la palette (*Palette Configuration*). Ce modèle contient les différentes catégories (*Palette Category*). Chaque catégorie contient un ensemble d'items (*Palette Item*), chacun référençant soit un « *Node Part* », soit un « *Edge Part* ».

La Figure IV.10 ci-après décrit le modèle de configuration du diagramme associé au paquetage *ModelRelationship* de SPEM4MDE. Les autres modèles de configuration sont décrits de façon similaire.

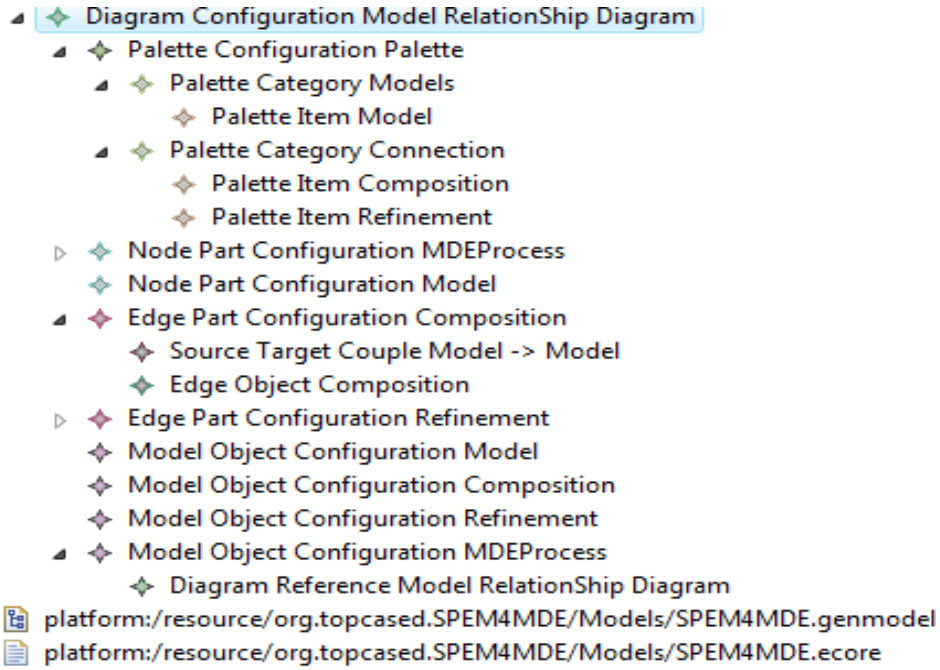


Figure IV.10 Modèle de configuration du diagramme « *Model Relationship* »

IV.5.1.2.4. L'activité « Generate SPEM4MDE Graphical Editor »

Cette activité permet de générer le code Java de l'éditeur graphique et de ses diagrammes à partir des modèles de configuration produit par l'activité « *Configure SPEM4MDE Graphical Editor* ». L'activité organise aussi le code généré sous forme de composants (voir Figure IV.11). Les composants en fond gris sont ceux dont le code généré est modifié. Le code de certains composants est modifié afin de prendre en compte certaines fonctionnalités (la vérification à la volée, la réutilisation des composants de papyrus pour éditer en OCL les pré et postconditions d'une activité/transformation, etc.).

Le composant *« SPEM4MDE Process Editor »* est composé de cinq sous-composants : « *SPEM4MDE Graphical Editor* », « *MDE Process Structure* », « *MDE Process Behavior* », « *Model Relationship », « Properties View »*. Le composant « *SPEM4MDE Graphical Editor* » est utilisé par les autres composants de « *SPEM4MDE Process Editor* ». Il contient les classes Java de l'éditeur graphique. Les composants de *« SPEM4MDE Process Editor »* utilisent les composants *Model* et *Edit* d'EMF, le composant *Utils,* et le composant *« OCL Editor & Checker »*.

Le composant *Edit* d'EMF utilise le composant *Model* d'EMF, tandis que le composant *Editor* d'EMF utilise le composant *Edit* d'EMF.

Le composant « *Properties View* » utilise le composant *Core* de Papyrus afin de proposer au concepteur de processus un éditeur convivial pour décrire les contraintes (précondition, postcondition, invariant) d'une activité/transformation.

Le composant « *MDE Process Structure* » contient les classes Java qui permettent de décrire graphiquement les modèles structurels d'un processus IDM.

Le composant « *Model Relationship* » contient les classes Java qui permettent de décrire graphiquement les relations entre modèles et éléments de modèles dans un processus IDM.

Le composant « *MDE Process Behavior* » contient les classes Java qui permettent de décrire graphiquement les modèles comportementaux d'un processus IDM. Ces modèles comportementaux sont des machines à états UML.

Le composant « *Properties View* » contient les classes Java qui permettent d'une part de visualiser les propriétés des éléments d'un modèle de processus et d'autre part d'éditer les contraintes d'une activité/transformation.

Le composant *Utils* contient les classes Java qui décrivent les icônes graphiques associées aux concepts de SPEM4MDE et aux éléments de la palette de l'éditeur de SPEM4MDE.

Le composant *Model* d'EMF contient les classes et interfaces Java qui décrivent les concepts du métamodèle SPEM4MDE. Le composant *Edit* d'EMF contient les classes Java qui permettent de manipuler les éléments d'un modèle de processus SPEM4MDE dans l'éditeur arborescent. Le composant *Editor* d'EMF contient les classes de l'éditeur arborescent de SPEM4MDE.

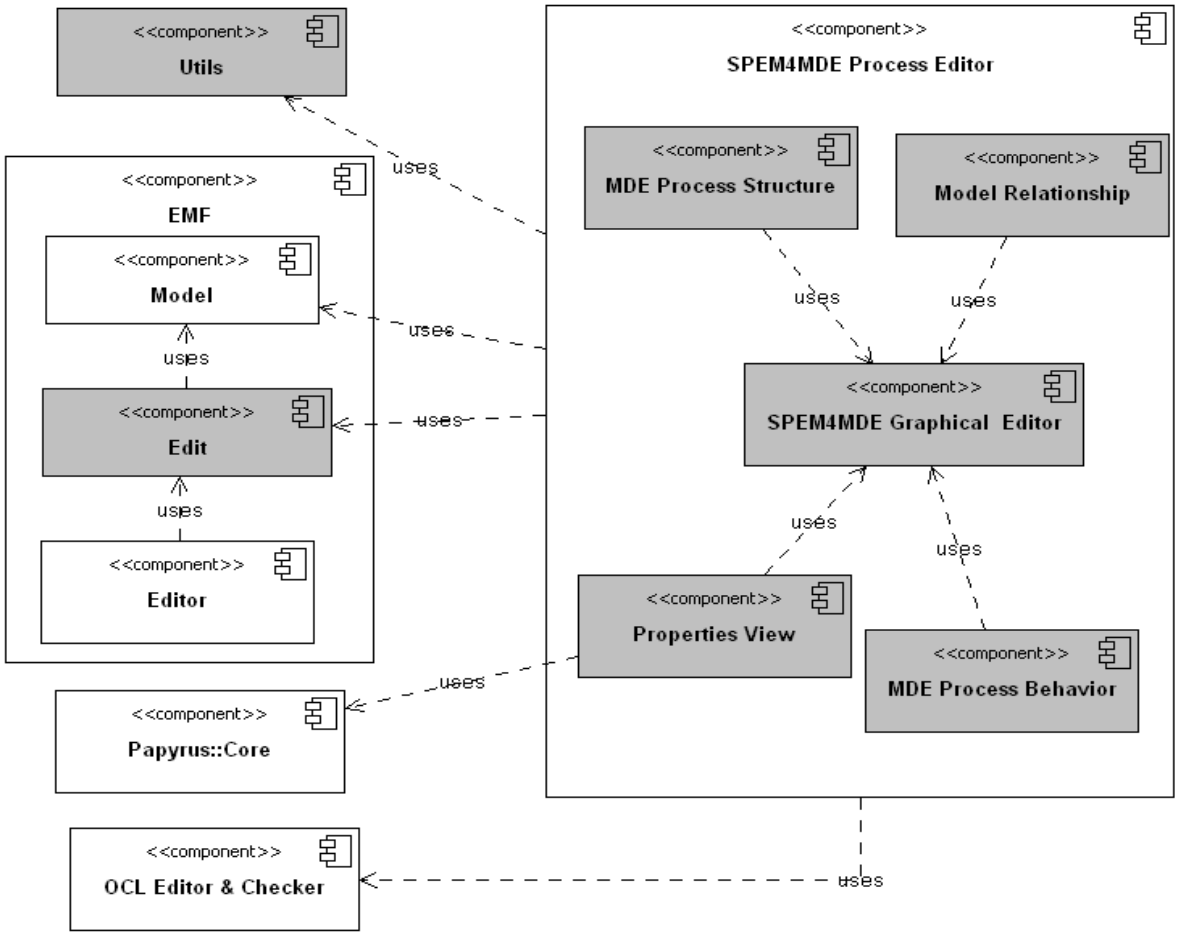


Figure IV.11 Architecture logicielle de « *SPEM4MDE Process Editor* »

En guise d'illustration, la Figure IV.12 ci-dessous présente l'association entre les éléments du modèle de configuration du composant « *Model Relationship* » et les fichiers contenant leur code Java.

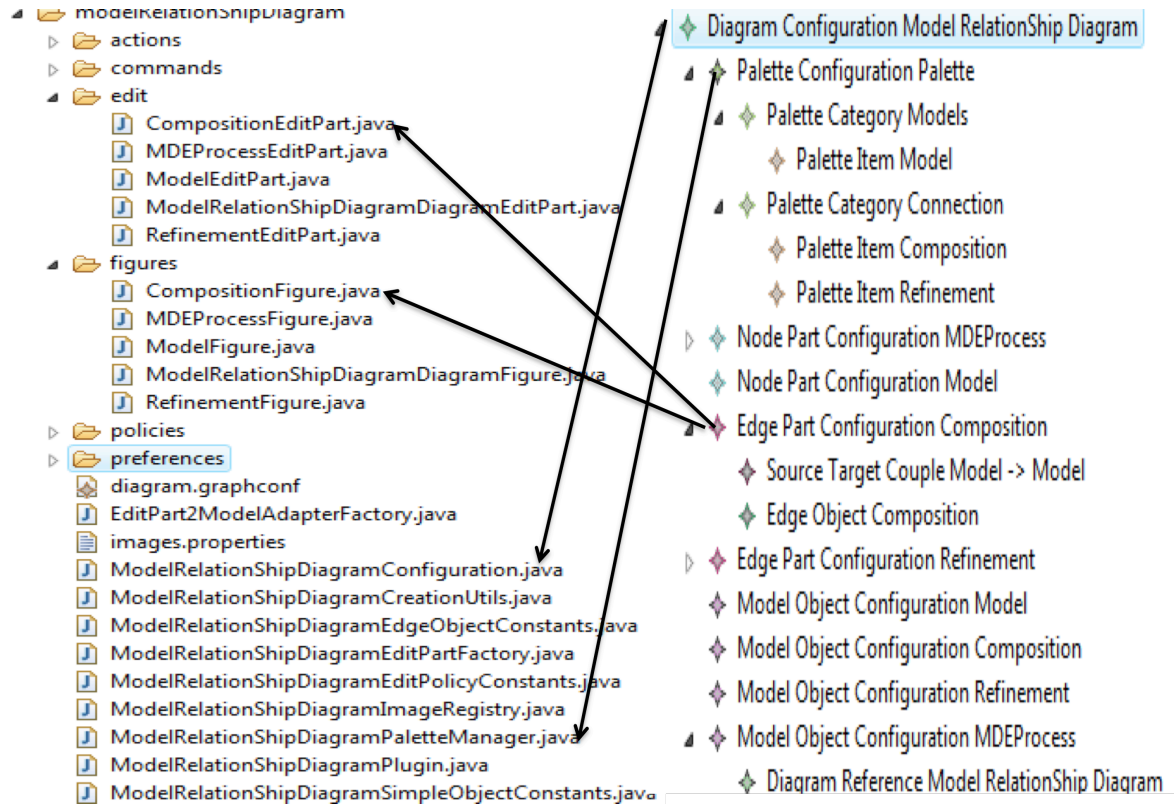


Figure IV.12 Fichiers contenant les classes Java du composant « *Model Relationship* »

La Figure IV.13 ci-dessous présente l'association entre les éléments du modèle de configuration du composant « *Properties View* » et les fichiers contenant leur code Java.

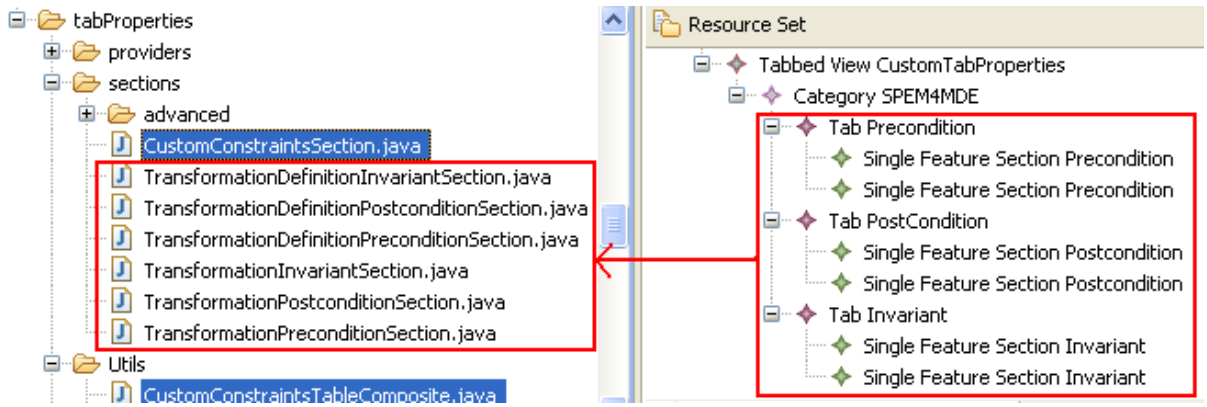


Figure IV.13 Fichiers contenant les classes Java du composant « *Properties view* »

Pour illustrer la vérification de modèles pendant l'édition, nous présentons ci-dessous le code Java de la méthode « *checkTargetForSource* » de la classe « *ConformToEdgeCreationEditPolicy* ». Cette méthode vérifie pendant l'édition la relation de conformité entre deux modèles. Un modèle de niveau M1 est conforme à un modèle de niveau M2 qui est conforme à un modèle de niveau M3. Un modèle de niveau M3 est conforme à lui même.

```
protected boolean checkTargetForSource(GraphElement source,
        GraphElement target) {
     EObject sourceObject = Utils.getElement(source);
     EObject targetObject = Utils.getElement(target);

     if (sourceObject instanceof org.topcased.spem4mde.Model
          && targetObject instanceof org.topcased.spem4mde.Model)
{
        Model sourceModel = (Model) sourceObject;
        Model targetModel = (Model) targetObject;
if (sourceModel.getLevel().equals(LevelKind.M1)
            && (targetModel.getLevel().equals(LevelKind.M1) ||
targetModel.getLevel().equals(LevelKind.M3)))
          return false;
        else
if (sourceModel.getLevel().equals(LevelKind.M2)
            && (targetModel.getLevel().equals(LevelKind.M2) ||
targetModel.getLevel().equals(LevelKind.M1)))
          return false;
        else
if (sourceModel.getLevel().equals(LevelKind.M3) &&
(!targetModel.equals(sourceModel)))
          return false;

return true;
}
        }
```

IV.5.1.2.5. L'activité « Refine SPEM4MDE Process Editor Code »

Cette activité permet de raffiner le code Java des composants de l'éditeur graphique afin d'introduire de nouvelles fonctionnalités (par exemple la réutilisation des composants de Papyrus, la vérification de modèles à la volée, etc.).

IV.5.1.2.6. L'activité « Run SPEM4MDE Process Editor Code »

Cette activité permet d'exécuter le code Java des composants qui participent à la réalisation du module « *SPEM4MDE Process Editor* » pour produire l'éditeur graphique de SPEM4MDE et ses diagrammes associés (digramme structurel, diagramme comportemental, diagramme de relations entre modèles). La Figure IV.14 ci-dessous donne une représentation graphique de l'éditeur de SPEM4MDE décrivant un diagramme structurel. Cette représentation comprend l'explorateur de projet qui montre les fichiers du projet, l'éditeur qui montre la représentation graphique d'un modèle de processus, *Outline* qui donne une vue arborescente des éléments d'un modèle de processus, et « *Properties View* » qui permet de visualiser les propriétés des éléments d'un modèle de processus.

La Figure IV.15 montre l'interface d'édition des contraintes d'une activité/transformation.

La Figure IV.16 ci-dessous montre une représentation graphique de l'éditeur de SPEM4MDE décrivant un diagramme de relations entre modèles.

La Figure IV.17 montre une représentation graphique de SPEM4MDE décrivant un diagramme comportemental.

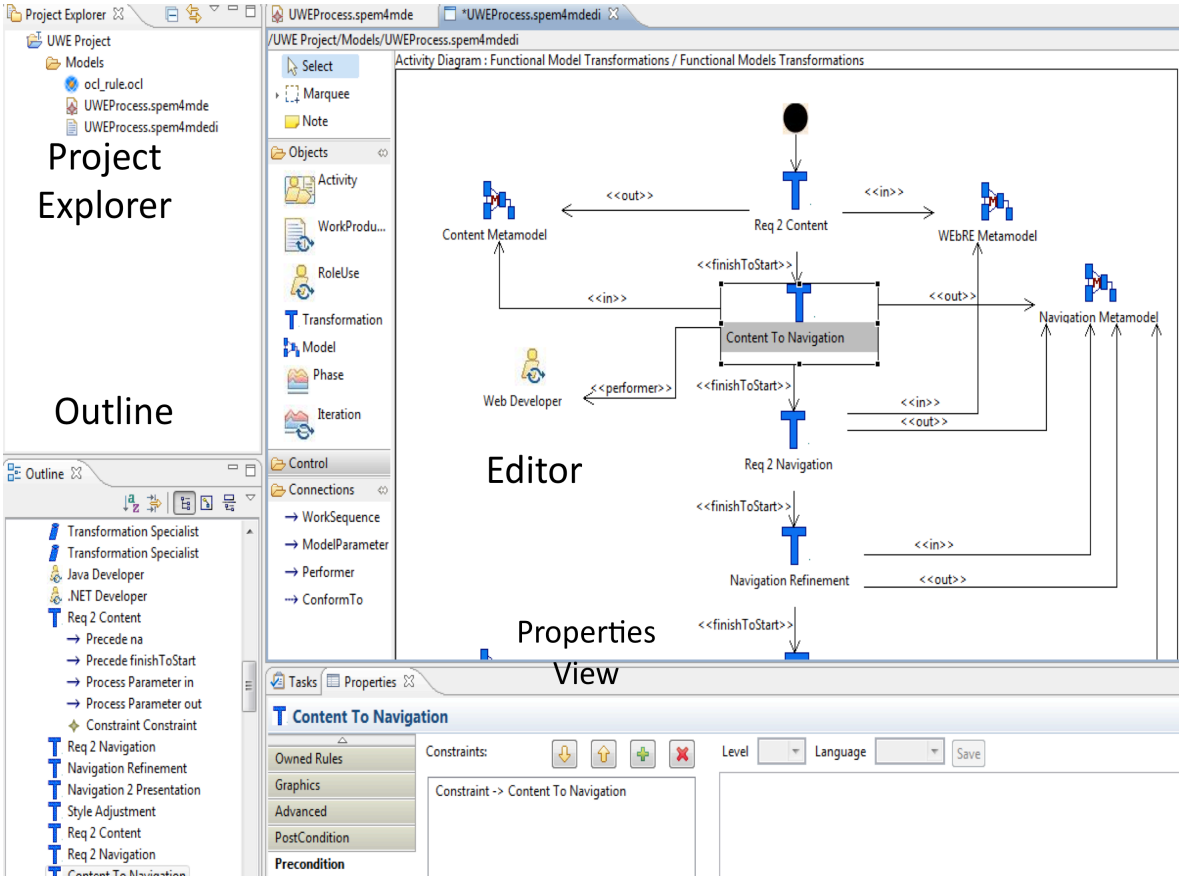


Figure IV.14 Vue graphique du composant « MDE Process Structure »

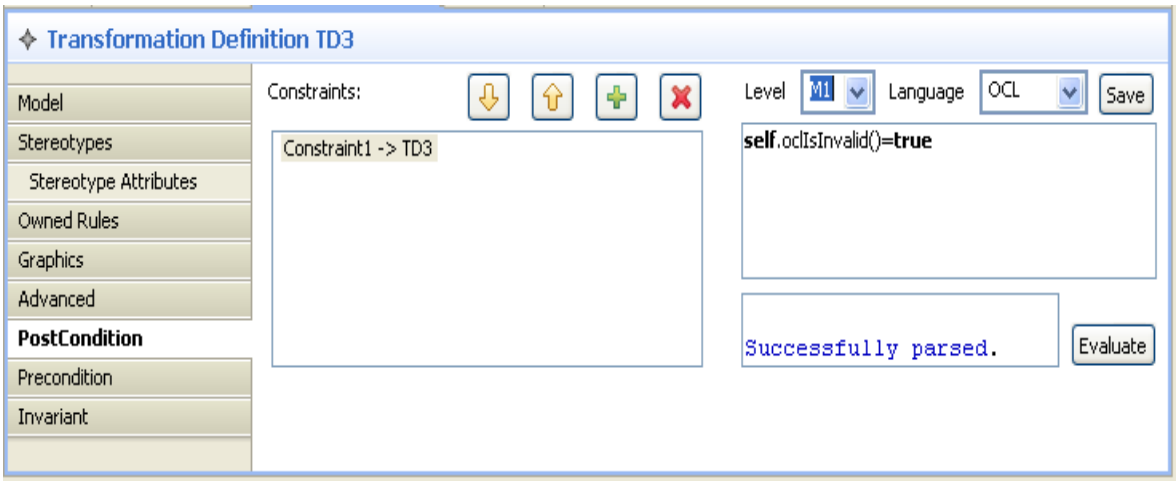


Figure IV.15 Interface graphique de spécification des contraintes d'une activité/transformation

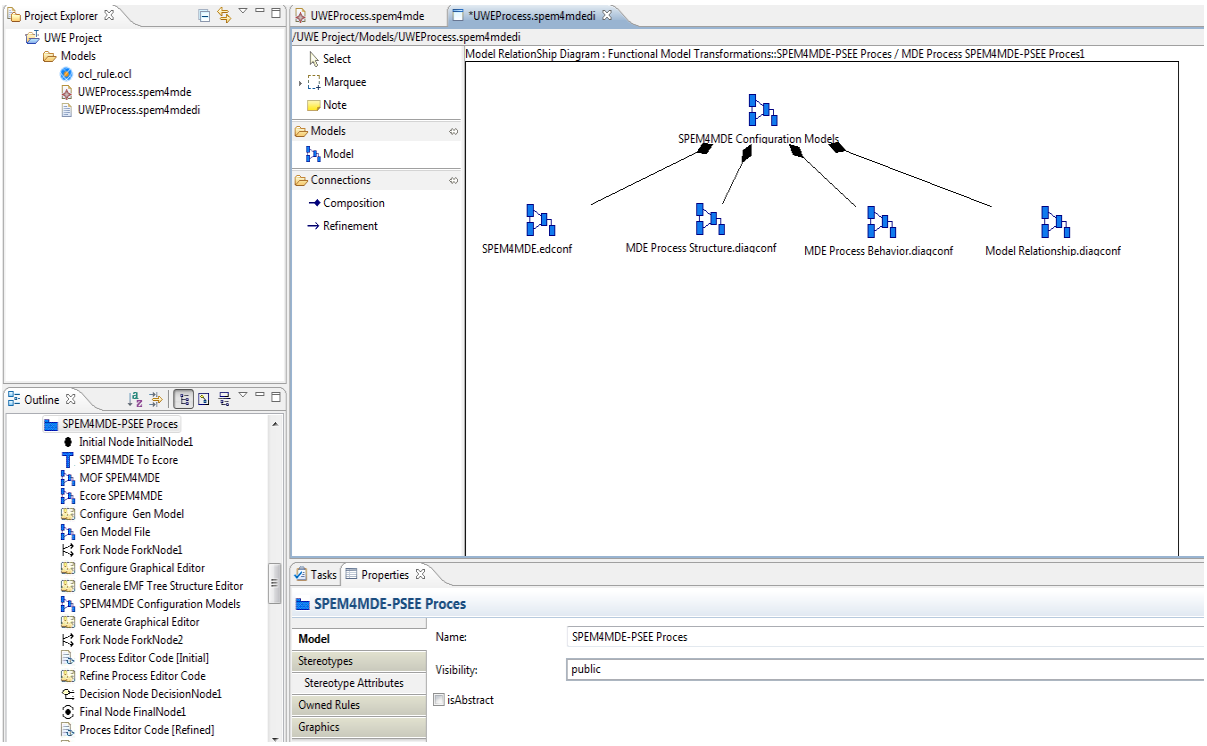


Figure IV.16 Vue graphique du composant « Model Relationship »

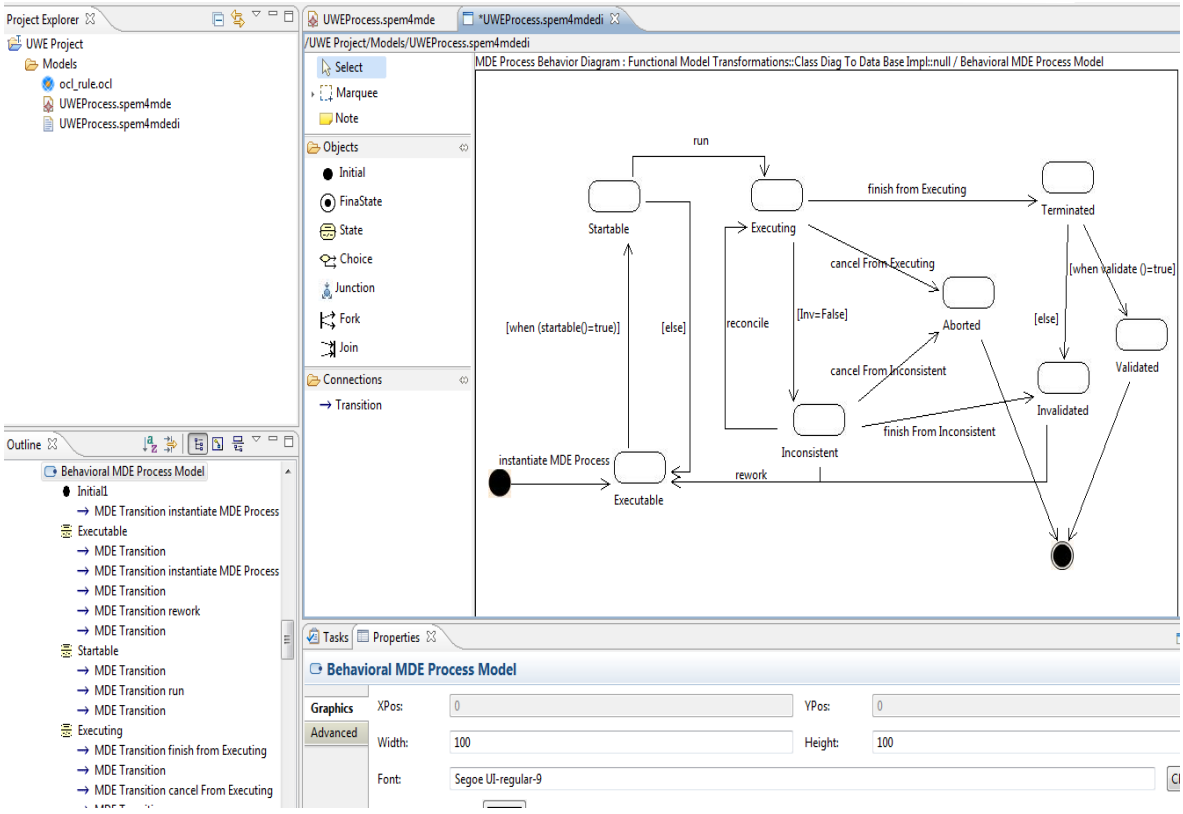


Figure IV.17 Vue graphique du composant « MDE Process Behavior »

IV.5.2. Implémentation du module « SPEM4MDE Process Enactment Engine »

Ce module permet de mettre en œuvre un projet basé sur un modèle de processus SPEM4MDE. La mise en œuvre d'un modèle de processus SPEM4MDE s'appuie sur ses modèles comportementaux. Elle consiste d'une part à exécuter les activités automatiques d'un modèle de processus SPEM4MDE et d'autre part à fournir une assistance aux développeurs en leur indiquant à tout instant l'état de chaque élément du processus ainsi que les opérateurs de mise en œuvre spécifiques. Dans ce module, les opérateurs de mise en œuvre sont implantés sous forme d'entrées de menu graphique en vue de permettre une interaction entre les développeurs et l'environnement de mise en œuvre.

IV.5.2.1. Illustration du module « SPEM4MDE Process Enactment Engine »

Pour illustrer le module « *SPEM4MDE Process Enactment Engine* », la Figure IV.18 présente sous forme de menu contextuel les opérateurs de mise en œuvre du modèle comportemental par défaut d'une transformation (voir Chapitre III, Figure III.8).

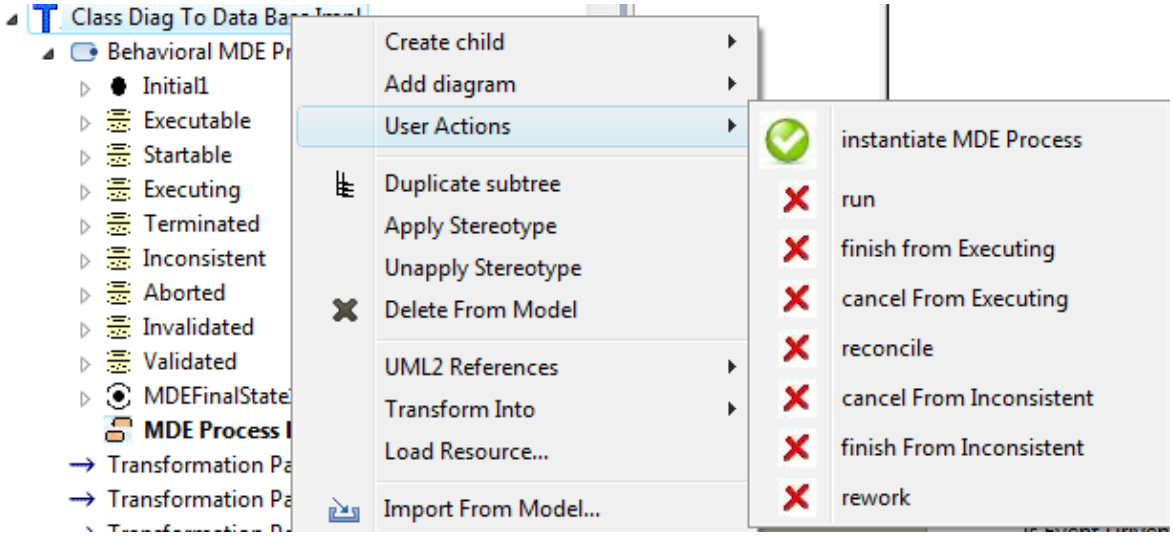


Figure IV.18 Les opérateurs de mise en œuvre d'une transformation

Selon l'état courant de la transformation, les opérateurs éligibles (activables) sont représentés par leurs noms précédés d'une marque verte. Par contre les opérateurs inéligibles (non activables) sont représentés par leurs noms précédés d'une croix rouge. Si un opérateur éligible est lancé par le développeur, alors les actions qui lui associées sont exécutées automatiquement et la transformation change d'état.

Par exemple si le chef de projet instancie un modèle de processus IDM (opérateur « *instantiate MDE Process* »), l'état courant de la transformation est initialisé à « *Executable.* Ce changement d'état est notifié à l'environnement de mise en œuvre qui par la suite positionne tous les autres éléments du modèle de processus à leurs états initiaux conformément à leurs modèles comportementaux.

Si la précondition (par exemple le modèle source est l'état « *ValidatedVersion* ») et les contraintes de précédences de la transformation sont satisfaites, la transformation passe à l'état « *Startable* ». Par conséquent l'opérateur « *instantiate MDE Process* » devient inéligible tandis que l'opérateur « *run* » devient éligible (voir Figure IV.19).

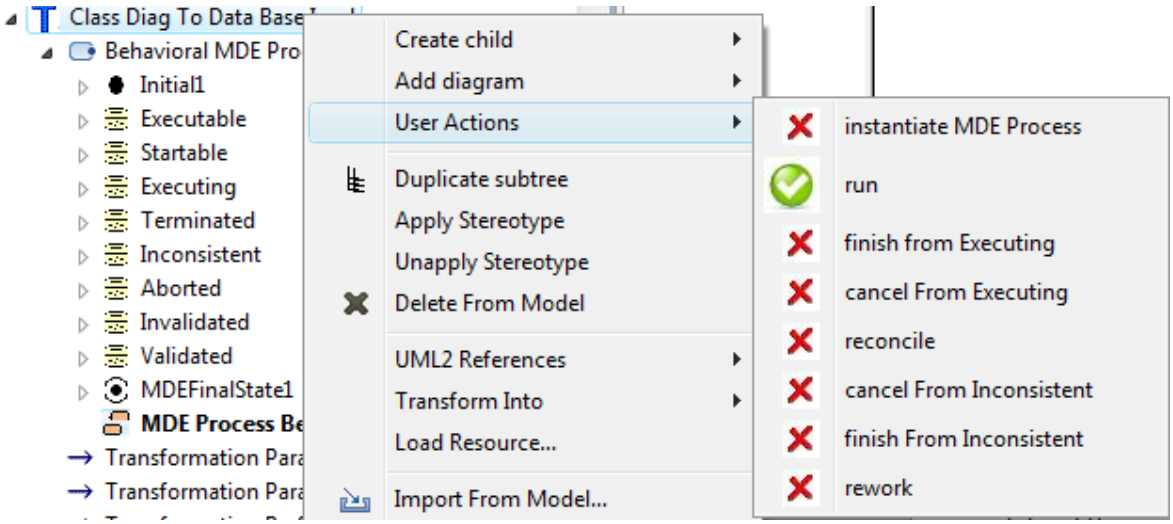


Figure IV.19 Etats des opérateurs de mise en œuvre après l'exécution de l'opérateur « *instantiate MDE Process* »

L'exécution de l'opérateur « *run* » s'accompagne d'un ensemble d'actions telles que l'invocation de l'outil d'exécution de la transformation. L'exécution de cet opérateur fait passer la transformation de l'état « *Startable* » à l'état « *Executing* ». Par la suite, l'opérateur « *run* » devient inéligible tandis que les opérateurs « *finish*» et « *cancel* » deviennent éligibles (voir Figure IV.20).

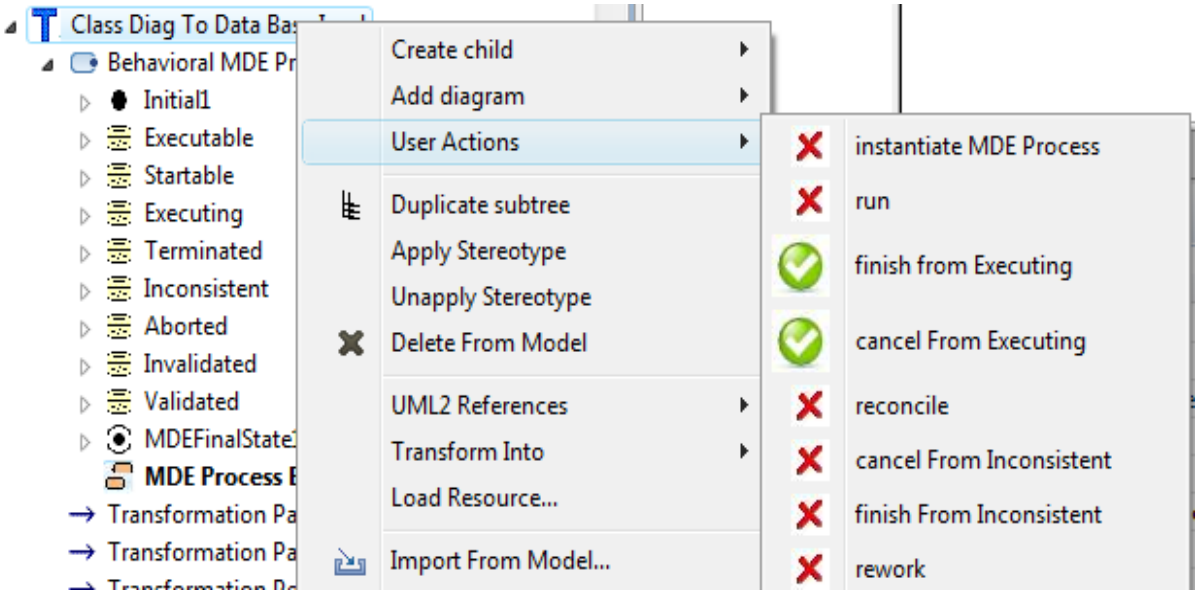


Figure IV.20 Etats des opérateurs de mise en œuvre après l'exécution de l'opérateur « *run* »

Le développeur a maintenant le choix entre terminer la mise en œuvre de la transformation (opérateur « *finish from Executing* ») ou annuler la mise en œuvre (opérateur « *cancel From Executing* »). Supposons que le développeur exécute l'opérateur « *finish from Executing* », alors la transformation passe à l'état « *Terminated* ». Par la suite, tous les opérateurs deviennent inéligibles. Il ne reste plus qu'à valider ou invalider la transformation. Si la fonction « *validate()* » retourne vraie alors la transformation passe automatiquement à l'état « *Validated* », sinon elle passe à l'état « *Invalidated* ».

IV.5.2.2. Mécanisme d'écoute et de traitement des changements d'états

Dans le cadre de la mise en œuvre d'un processus, un changement d'état d'un élément du processus peut provoquer des changements d'états des autres éléments du processus. Pour gérer ce mécanisme de changement d'état, nous avons intégré dans l'environnement de mise en œuvre un module qui écoute les changements d'états des éléments d'un modèle de processus et qui les traitent. La Figure IV.21 ci-après montre ce mécanisme d'écoute et de traitement de ces changements d'états. Tout changement d'état (tout objet qui invoque la méthode *setState ()*) est notifié à l'écouteur *SPEM4mdeEngine* grâce à la méthode *firePropertyChange* de la classe *PropertyChangeSupport*. La méthode *firePropertyChange* contient le nom de la propriété qui a changé de valeur, l'ancienne valeur de la propriété, et la nouvelle valeur de la propriété. Une fois ce changement d'état notifié, l'écouteur *SPEM4mdeEngine* capture grâce au paramètre *event* de la méthode *propertyChange,* le nom de l'objet (élément de modèle de processus) dont la propriété « *state* » a changé de valeur. Une fois le nom de l'objet capturé, l'écouteur procède au traitement de l'événement en faisant appel à la méthode *changeStateMDEProcessElement*. Cette méthode modifie l'état de l'objet capturé et procède éventuellement au changement d'état des autres objets qui dépendent de ce changement.

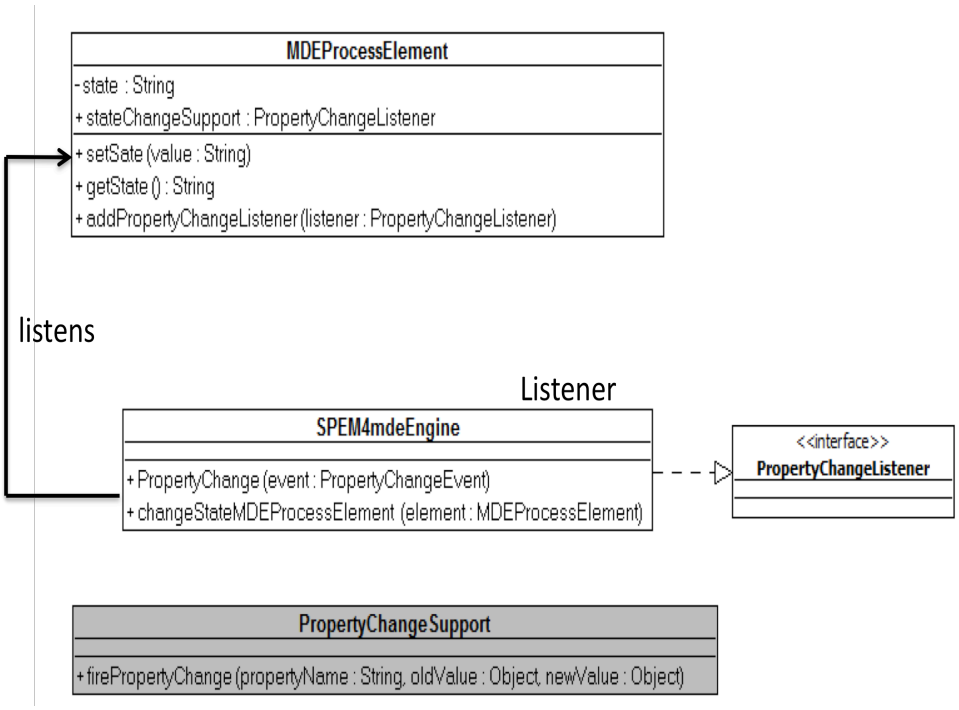


Figure IV.21 Mécanisme d'écoute et de traitement d'un événement de changement d'état

La méthode ci-dessous présente le code Java qui permet de changer l'état d'un élément du modèle de processus et de notifier ce changement (*firePropertyChange*) au module de gestion et de traitement des changements d'état.

```
public void setState(String newState) {
     String oldState = state;
     state = newState;

   stateChangeSupport.firePropertyChange("state",oldState,
newState);
     if (eNotificationRequired())
       eNotify(new ENotificationImpl
            (this, Notification.SET,
Spem4mdePackage.PROCESS_ELEMENT__STATE, oldState,
state));}
```

La méthode ci-dessous présente le code Java qui permet de capturer l'élément du modèle de processus qui a changé d'état. Cette méthode affiche sous forme de message, l'ancien et le nouvel état à chaque fois qu'un opérateur de mise en œuvre est exécuté par un développeur.

```
public void propertyChange(PropertyChangeEvent event) {
     // TODO Auto-generated method stub
     try {
     EObject currentObject=null;
     currentObject =(EObject)event.getSource();
     String oldState=(String) event.getOldValue();
     String newState=(String) event.getNewValue();

     BreakdownElement
element=(BreakdownElement)(currentObject);
       JOptionPane.showMessageDialog(null,
element.getName()+"passe de "+oldState+ à "+newState,
"Message", JOptionPane.INFORMATION_MESSAGE);
        changeStateMDEProcessElement (currentObject,
(MDEProcess)element.getActivity() );
     }
      catch (Exception e)
      {e.printStackTrace();}
}
```

IV.6. Conclusion

Nous avons présenté dans ce chapitre le prototype SPEM4MDE-PSEE qui constitue une implémentation et une validation du métamodèle SPEM4MDE. Nous avons d'abord présenté la plate-forme TOPCASED utilisée pour développer le prototype SPEM4MDE-PSEE, puis l'architecture de ce prototype. Cette architecture comporte deux modules « *SPEM4MDE Process Editor* » qui offre les composants qui permettent de modéliser graphiquement la structure et le comportement d'un processus IDM, et « *SPEM4MDE Process Enactment Engine* » qui permet de mettre en œuvre le modèle de processus adapté. La mise en œuvre consiste d'une part à exécuter les activités

automatiques d'un processus, et d'autre part à offrir une assistance aux développeurs en leur fournissant à tout instant l'état courant de chaque élément du processus ainsi que les opérateurs de mise en œuvre qui leur sont spécifiques. La description fonctionnelle de ces deux modules nous a permis de comprendre les fonctionnalités attendues de SPEM4MDE-PSEE. Certaines de ces fonctionnalités (édition et exécution des règles de transformation, édition de modèles, vérification de modèle de processus, etc.) seront réalisées par des outils externes (ATL, UML Editor, Medini QVT, SmartQVT, OCL Editor & Checker, etc.) qui peuvent être intégrés en tant que plug-in dans *SPEM4MDE*-PSEE.

L'utilisation de la plate-forme TOPCASED nous a permis de générer automatiquement l'éditeur graphique de SPEM4MDE (« *SPEM4MDE Process Editor* »). L'implémentation du module « *SPEM4MDE Process Enactment Engine* » consiste d'une part à traduire sous forme d'entrées de menu graphique les opérateurs de mise en œuvre de chaque modèle comportemental, et d'autre part à gérer puis traiter les changements d'état à chaque exécution d'un opérateur de mise en œuvre.

Dans le prototype que nous avons développé, nous n'avons pas en réalité découplé les deux modules de SPEM4MDE-PSEE. Nous décrivons le modèle structurel et les modèles comportementaux d'un processus IDM et procédons ensuite à la traduction sous forme d'entrées de menu graphique des opérateurs de mise en œuvre de chaque modèle comportemental.

Cependant, comme expliqué dans l'architecture de SPEM4MDE-PSEE, l'idéal serait de décrire la structure et le comportement d'un processus IDM avec le module « *SPEM4MDE Process Editor* » (ce qui est le cas actuellement) et de développer à part un environnement de mise en œuvre complet à travers le module « *SPEM4MDE Process Enactment Engine* ». L'environnement de mise en œuvre devrait permettre de sélectionner un modèle de processus IDM dans le « *repository* » de « *SPEM4MDE Process Editor* », de l'adapter à un projet spécifique et de le mettre en œuvre. L'adaptation consiste à éventuellement modifier le modèle de processus générique et à assigner les ressources nécessaires à sa mise en œuvre (espace de travail pour chaque acteur, assignation des rôles du processus aux acteurs du projet, assignation des outils nécessaires à la mise en œuvre des activités/transformations). Nous comptons réaliser à court terme cet environnement de mise en œuvre qui serait complètement dissocié de l'éditeur de modèle de processus SPEM4MDE.

Le prototype que nous avons développé est mono-dévéloppeur et ne supporte pas la traçabilité des transformations, cette préoccupation étant gérée par les outils d'exécution de transformations qui sont intégrés dans SPEM4MDE-PSEE sous forme de plug-in. A moyen terme, nous comptons implanter dans SPEM4MDE-PSEE un environnement de mise en œuvre multi-développeur qui intégrera un environnement de traçabilité indépendant des outils d'exécution de transformations. Le prototype présenté dans ce chapitre sera utilisé pour illustrer l'étude de cas portant sur le processus UWE (UML-Based Web Engineering) du chapitre V.

CHAPITRE V. ÉTUDE DE CAS : MODELISATION ET MISE EN ŒUVRE DU PROCESSUS UWE

V.1. Introduction

Dans les chapitres III et IV précédents, nous avons présenté respectivement le métamodèle SPEM4MDE et le prototype d'implémentation de ce métamodèle. Pour valider la spécification et l'implémentation du métamodèle SPEM4MDE, nous présentons dans ce chapitre une étude de cas portant sur le processus UWE (UML-Based Web Engineering) qui est tout à fait représentatif des processus IDM. L'objectif du processus UWE [Koch, 2006] est d'offrir aux développeurs un support systématique et semi-automatique pour le développement des applications web basé sur les modèles et leurs transformations. Pour cela, UWE couvre tout le cycle de développement des systèmes web, des exigences jusqu'au code exécutable.

Ce chapitre est organisé comme suit. Nous présentons d'abord une démarche générale proposée dans SPEM4MDE pour modéliser et mettre en œuvre les processus IDM (section V.2). La section 0 présente une description informelle du processus UWE tandis que la section V.4 présente l'application de la démarche de SPEM4MDE au processus UWE. Nous concluons par la section V.5 en résumant les grandes idées développées dans ce chapitre.

V.2. Démarche de modélisation et de mise en œuvre de processus avec SPEM4MDE

SPEM4MDE propose une démarche générale de modélisation et de mise en œuvre des processus IDM. L'intérêt de cette démarche est de guider la modélisation et la mise en œuvre des processus IDM en fournissant l'ensemble des étapes nécessaires à la production des livrables d'un projet de développement IDM (modèles, code, documentation).

V.2.1. Description informelle de la démarche

La Figure V.1 ci-dessous décrit une démarche générale pour modéliser et mettre en œuvre un processus IDM avec SPEM4MDE-PSEE. Cette démarche est déclinée en quatre activités:

La *première* est l'activité de description du processus IDM. Cette activité est réalisée par un concepteur de processus IDM et/ou un spécialiste de transformation via l'éditeur « *SPEM4MDE Process Editor* » de SPEM4MDE-PSEE. Cette activité de description d'un processus IDM produit deux modèles : la modèle structurel (*Structural MDE Process Model*) et le modèle comportemental

(*Behavioral MDE Process Model*). Le modèle comportemental est décrit par le biais des machines à états d'UML. C'est un modèle composé d'un ensemble d'états et d'opérateurs de mise en œuvre qui permettent de déclencher les changements d'états. Le modèle structurel et le modèle comportemental sont conformes au métamodèle SPEM4MDE. Une fois les deux modèles décrits, le concepteur de processus peut faire appel au vérificateur de modèles (*Design Checker*) pour vérifier les contraintes OCL spécifiées dans le métamodèle SPEM4MDE.

La *deuxième* activité de cette démarche est l'adaptation du modèle structurel du processus pour un projet spécifique. Cette adaptation est faite par le chef de projet en utilisant l'éditeur *SPEM4MDE Process Editor*. Elle consiste à assigner les ressources nécessaires (développeurs, outils, espaces de travail) à un modèle de processus IDM pour qu'il devienne exécutable. Le modèle de processus IDM exécutable (*Enactable MDE Process Model*) produit par cette activité représente la version initiale du projet de développement.

La *dernière* activité de cette démarche est la mise en œuvre du modèle de processus adapté. Cette mise en œuvre est réalisée par les développeurs du projet en utilisant les outils IDM mis à leur disposition. Ces développeurs sont assistés dans leurs tâches par l'environnement d'exécution de processus appelé « *SPEM4MDE Process Enactment Engine* ». L'assistance consiste à leur fournir à tout instant les activités/transformation éligibles et l'état courant de chaque activité/transformation. L'environnement de mise en œuvre s'appuie sur les modèles comportementaux du processus afin d'offrir aux développeurs les opérateurs de mise en œuvre sous forme d'entrées de menus graphiques. Cette activité produit les livrables du projet (code, modèles, documentation, etc.).

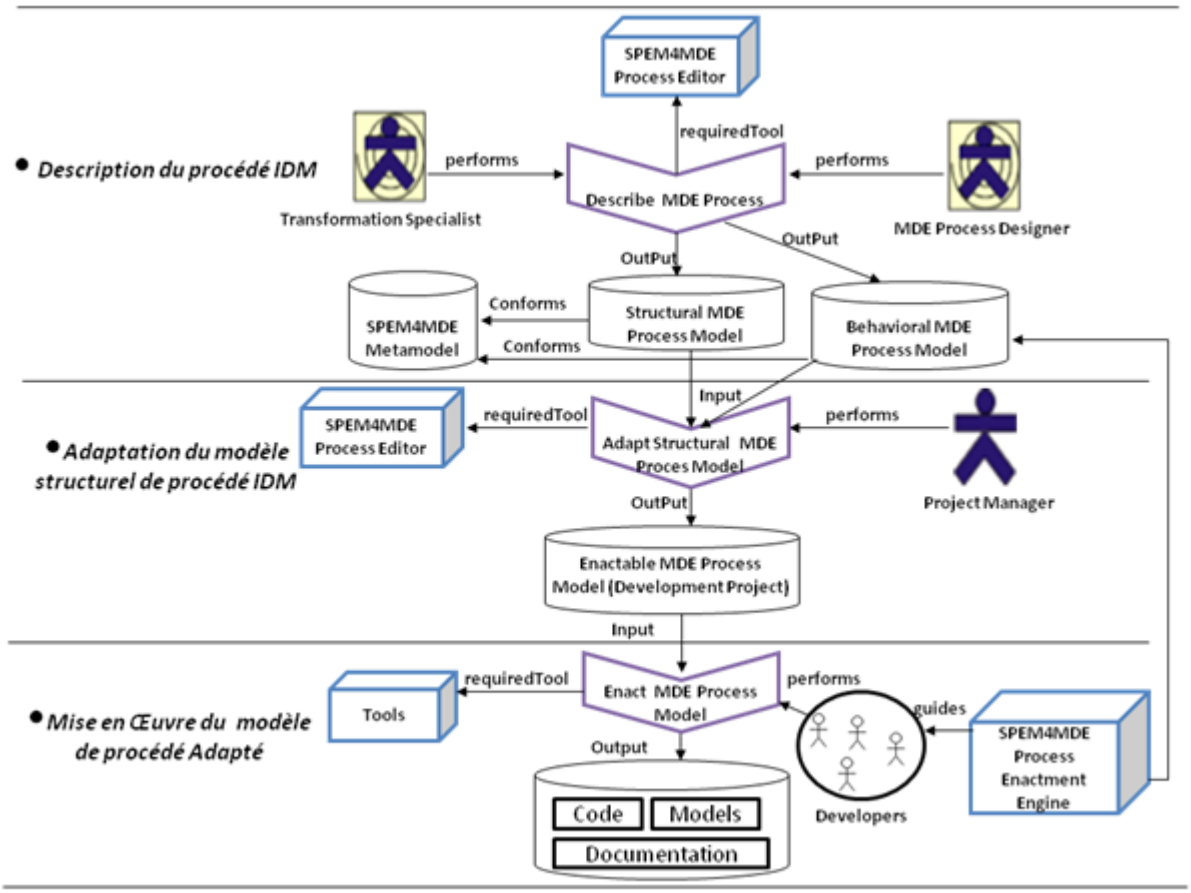


Figure V.1 Démarche générale de modélisation et de mise en œuvre dans SPEM4MDE

V.2.2. Formalisation de cette démarche avec SPEM4MDE

Dans cette section nous formalisons la démarche générale de modélisation et de mise en œuvre de processus IDM en utilisant les concepts de SPEM4DE (voir Figure V.2). Dans cette figure, les activités de cette démarche sont décrites en précisant pour chacune le modèle d'entrée et/ou le modèle de sortie. Nous n'avons pas décrit les outils utilisés dans chaque activité pour des raisons de lisibilité de la figure. Dans cette démarche, le concepteur de processus décrit d'abord les modèles structurel et comportemental du processus. Par la suite ces deux modèles sont vérifiés par rapport aux contraintes OCL spécifiées dans le métamodèle SPEM4MDE. Le chef de projet adapte le modèle structurel à un projet spécifique. Le modèle adapté est par la suite mis en œuvre par les acteurs du projet qui jouent le rôle de développeur afin de produire les livrables du projet (code, modèles, documentation). La mise en œuvre du modèle adpaté s'appuie sur les modèles comportementaux de processus.

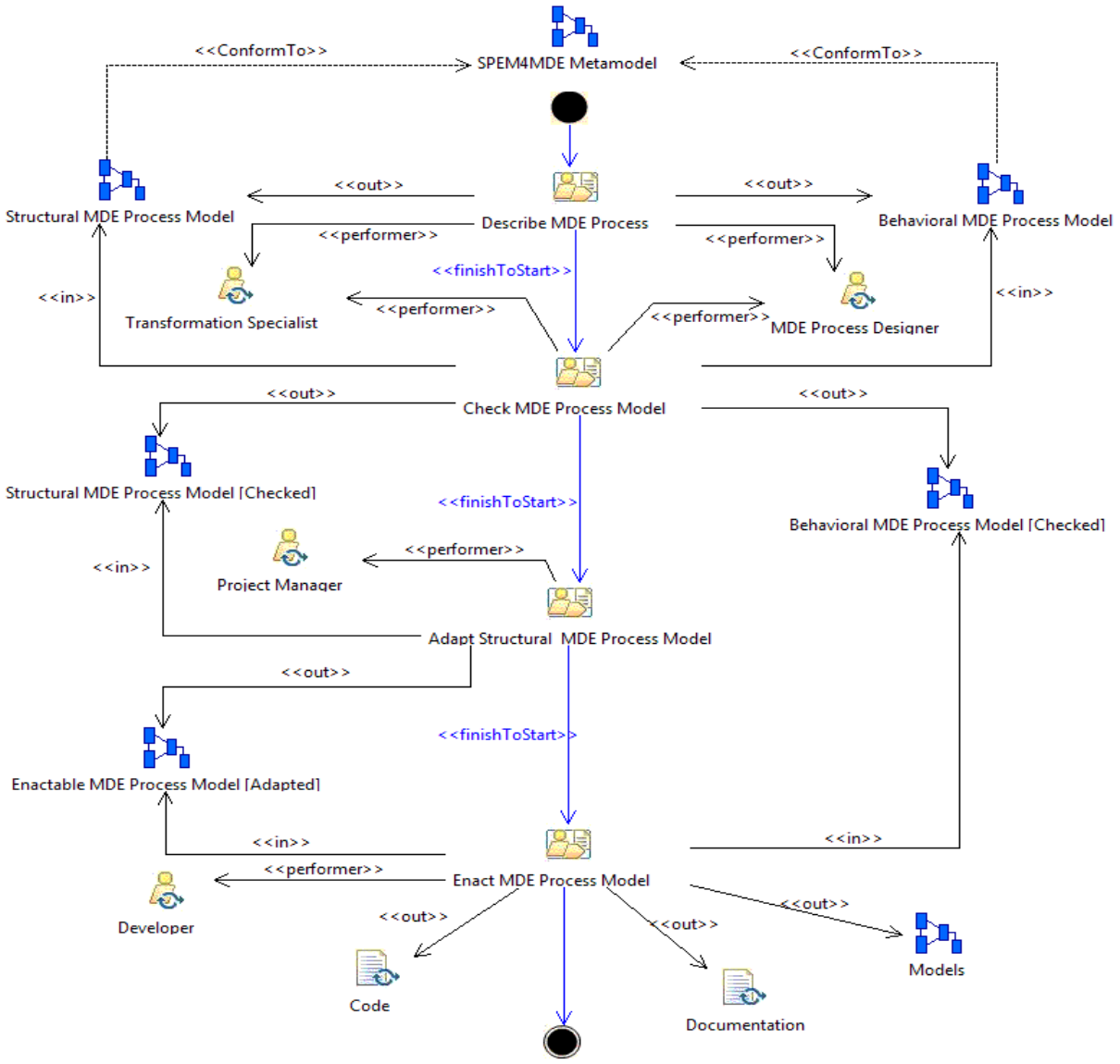


Figure V.2 Démarche de modélisation et de mise en œuvre avec SPEM4MDE

V.3. Description informelle du processus UWE

Le processus UWE (*UML-based Web Engineering*) [Koch, 2006] a été décrit dans la section II.4.1 du Chapitre II. Son objectif est d'offrir aux développeurs un support systématique et semi-automatique pour le développement des systèmes web basés sur les modèles et leurs transformations. Le processus UWE couvre tout le cycle de développement des systèmes web : des exigences au code exécutable. Le processus est composé d'un ensemble de modèles et de transformations spécifiés par des métamodèles et des langages de transformation de modèles. Les métamodèles sont le métamodèle *WebRE* (Web Requirements Engineering) (annexe D, section c)) dédié à la modélisation des exigences web et le métamodèle *UWE* (annexe D, section a) qui spécifie les autres préoccupations des systèmes web (contenu, navigation, présentation).

V.3.1. Les modèles du processus UWE

Dans cette section, nous décrivons essentiellement les modèles du processus UWE. Ces modèles sont : le modèle des exigences, le modèle du contenu, le modèle de navigation, le modèle de présentation, le modèle « BigPicture », le modèle architectural, et le modèle d'intégration.

Le *modèle des exigences* (*Requirements Model*) est un modèle de type CIM qui a pour objectif de définir les fonctionnalités d'un système web via les cas d'utilisation UML. UWE distingue deux types de cas d'utilisation : les cas d'utilisation relatifs à la navigation (unités d'information qui décrivent un contenu du système web) et les cas d'utilisation relatifs au processus métier (unité d'information qui décrit une activité réalisée dans un système web par un utilisateur). Les digrammes d'activités sont utilisés pour détailler un cas d'utilisation relatif au processus métier. Les diagrammes de cas d'utilisation et d'activités sont décrits en utilisant le profil WebRE (annexe D, section d).

Le *modèle de contenu* (*Content Model*) dans UWE est un modèle conceptuel de type PIM (Platform Independent Model). Il est décrit par un diagramme des classes pour sa partie structurelle, et des machines à états et diagrammes de séquences pour sa partie comportementale.

Le *modèle de navigation* (*Navigation Model*) est un modèle de type PIM qui décrit les nœuds d'un système web et leurs relations. Un lien permet de connecter un nœud (la source) à plusieurs nœuds (les cibles). A chaque fois qu'un nœud est atteint durant la navigation, certaines informations sont fournies à l'utilisateur qui exécute certaines actions. Un nœud peut représenter par exemple une page web mais pas son contenu. Le contenu d'une page web est représenté par le modèle de présentation. Le paquetage *Navigation* du métamodèle UWE (voir annexe D, FIG. 6) distingue quatre types de nœuds: *NavigationClass* (les nœuds qui décrivent une unité d'information d'un système web), *ProcessClass* (les nœuds qui correspondent aux activités d'un processus métier), *Menu* (un nœud à partir duquel plusieurs nœuds sont atteignables), et *AccessPrimitive* (les primitives d'accès). Les primitives d'accès sont : *Index* (liste d'éléments), *Query* (requête qui permet d'extraire des informations dans une base), et *GuidedTour* (un nœud spécial qui permet de reconstruire l'historique des nœuds visités). Nous rappelons que le nœud *Menu* dans le modèle de navigation ne sera pas forcément transformé en un menu d'interface utilisateur dans le modèle de présentation même si cela peut être parfois le cas.

Le *modèle de présentation* (*Presentation Model*) (voir annexe D, FIG. 8 et FIG. 9) est un modèle de type PIM qui fournit les concepts permettant de décrire les interfaces utilisateur (zone de texte, image, bouton, formulaire, lien, etc.) d'un système web. Il ne décrit que la structure basique des interfaces utilisateur c'est-à-dire quelle interface utilisateur sera utilisée pour présenter un nœud du modèle de

navigation. Certains aspects spécifiques aux interfaces utilisateur telles que la couleur, la police, et l'emplacement des interfaces utilisateurs sur une page web ne sont pas spécifiés dans ce modèle. Les interfaces utilisateurs d'un modèle de présentation sont indépendantes d'une technologie donnée.

Le *modèle « BigPicture »* est la fusion des modèles fonctionnels (*Content*, *Navigation*, *Presentation*) dans un modèle unique dans le but de les vérifier et de les valider.

Le *modèle architectural* (*Architecture Model*) est un modèle de type PIM qui décrit l'organisation logicielle et les aspects non-fonctionnels d'un système web tels que la performance, la « *maintenabilité* » et la modularité.

Le *modèle d'intégration* (*Integration Model*) est un modèle de type PIM qui constitue le résultat de la fusion des modèles fonctionnels et des modèles architecturaux. Une fois le modèle d'intégration créé, on pourra générer le modèle PSM selon la plate-forme ciblée (J2EE ou .NET).

V.3.2. Les transformations du processus UWE

Le tableau ci-dessous décrit les transformations du processus UWE et les caractéristiques associées (voir Tableau V.I). La caractéristique « *Type* » spécifie le passage d'un modèle à un autre dans l'approche MDA. Nous distinguons les types de transformation suivants « *CIM 2 PIM* », « *PIM 2 PIM* », « *PIM 2 PSM* », et « *PSM 2 PSM* ».

La complexité permet de mesurer le degré de complexité d'une transformation. Nous distinguons dans le processus UWE, les degrés de complexité suivants : « *simple* », « *merge* ». Une *transformation simple* (1 vers 1) transforme un élément d'un modèle source en un élément d'un modèle cible. Une *transformation « merge »* (N vers 1) transforme un ou plusieurs modèles sources en un modèle cible. La caractéristique « *Execution* » spécifie le type d'exécution d'une transformation. Une transformation du processus UWE est automatique si elle ne nécessite aucune intervention de l'utilisateur pendant son exécution. Elle est semi-automatique quand l'utilisateur intervient dans le processus de transformation en choisissant certains éléments du modèle source qui seront transformés. Elle est manuelle quand l'utilisateur doit produire le modèle. La caractéristique « *Technique* » spécifie l'approche utilisée pour implémenter une transformation du processus UWE. Les approches utilisées sont QVT, ATL, Java, OCL, les graphes.

Caractéristiques / Transformation	Type	Complexité	Exécution	Technique
Requirements 2 Achirecture	CIM 2 PIM	simple	manuelle	□
Requirements 2 Content	CIM 2 PIM	simple	automatique	QVT
Content 2 Navigation	PIM 2 PIM	simple	semi-automatique	Java (OCL), ATL
Requirements 2 Navigation	CIM 2 PIM	merge	automatique	QVT
Navigation Refinement	PIM 2 PIM	simple	automatique	Java (OCL)
Navigation 2 Presentataion	PIM 2 PIM	simple	automatique	Java (OCL), ATL

Style Adjustment	PIM 2 PIM	merge	automatique	Java
Functional 2 BigPicture	PIM 2 PIM	merge	automatique	Graphes
Architecture Integration	PIM 2 PIM	merge	automatique	QVT
Integration 2 J2EE	PIM 2 PSM	merge	automatique	QVT, ATL
Integration 2 .NET	PIM 2 PSM	merge	automatique	QVT, ATL

Tableau V.I Caractéristiques des transformations du processus UWE [Koch, 2006]

V.4. Application de la démarche au processus UWE

Cette section décrit la modélisation et la mise en œuvre du processus UWE basées sur une démarche proposée dans SPEM4MDE. Comme décrit dans la démarche, nous présentons par ordre la description structurelle du processus UWE, sa description comportementale, l'adaptation du modèle de processus UWE à un projet de développement d'une application web de messagerie, et enfin la mise en œuvre du modèle de processus adapté.

V.4.1. Description structurelle du processus UWE

La Figure V.3 décrit l'enchainement des transformations dans le processus UWE. Seul le rôle « *Web Designer* » est représenté pour l'activité « *Describe Requirements Model* ». Nous considérons que les transformations du processus UWE seront réalisées par le rôle « *Transformation Specialist* ». Cependant, pour des rasions de lisibilité de la Figure V.3, les rôles des transformations ne sont pas décrits dans le processus UWE.

Le processus UWE démarre avec la description du modèle des exigences (*Requirement Models*). Par l'intermédiaire des transformations « *Requirements 2 Functional* » et « *Requirements 2 Architecture* », deux ensembles de modèles sont dérivés à partir des exigences : les modèles fonctionnels (*Functional Models*) qui représentent les différentes préoccupations d'un système web (contenu, navigation, présentation) et les modèles architecturaux (*Architecture Models*). La transformation des modèles fonctionnels (*Functional Model Transformations*) (voir Figure V.4) est une transformation composite dans laquelle chacune des sous-transformations traite d'une préoccupation séparée de l'ingénierie web. Ces sous-transformations sont au nombre de six (voir Figure V.4): « *Requirements 2 Content* », « *Content 2 Navigation* », « *Requirements 2 Navigation* », « *Navigation Refinement* », « *Navigation 2 Presentation* », et « *Style Adjustment* ». Les modèles fonctionnels sont par la suite intégrés dans un modèle appelé « *BigPicture Model* » pour une vérification. Une fusion entre le modèle « *BigPicture* » et les modèles architecturaux (*Architecture Models*) produit un modèle intégré (*Integration Model*) qui couvre les aspects fonctionnels et architecturaux. Les modèles spécifiques aux plates-formes ciblés (*modèle J2EE* ou un *modèle .NET*) sont dérivés à partir du modèle d'intégration. Finalement du code exécutable est généré à partir des modèles spécifiques aux plates-formes J2EE et .NET ou toute autre plate-forme ciblée.

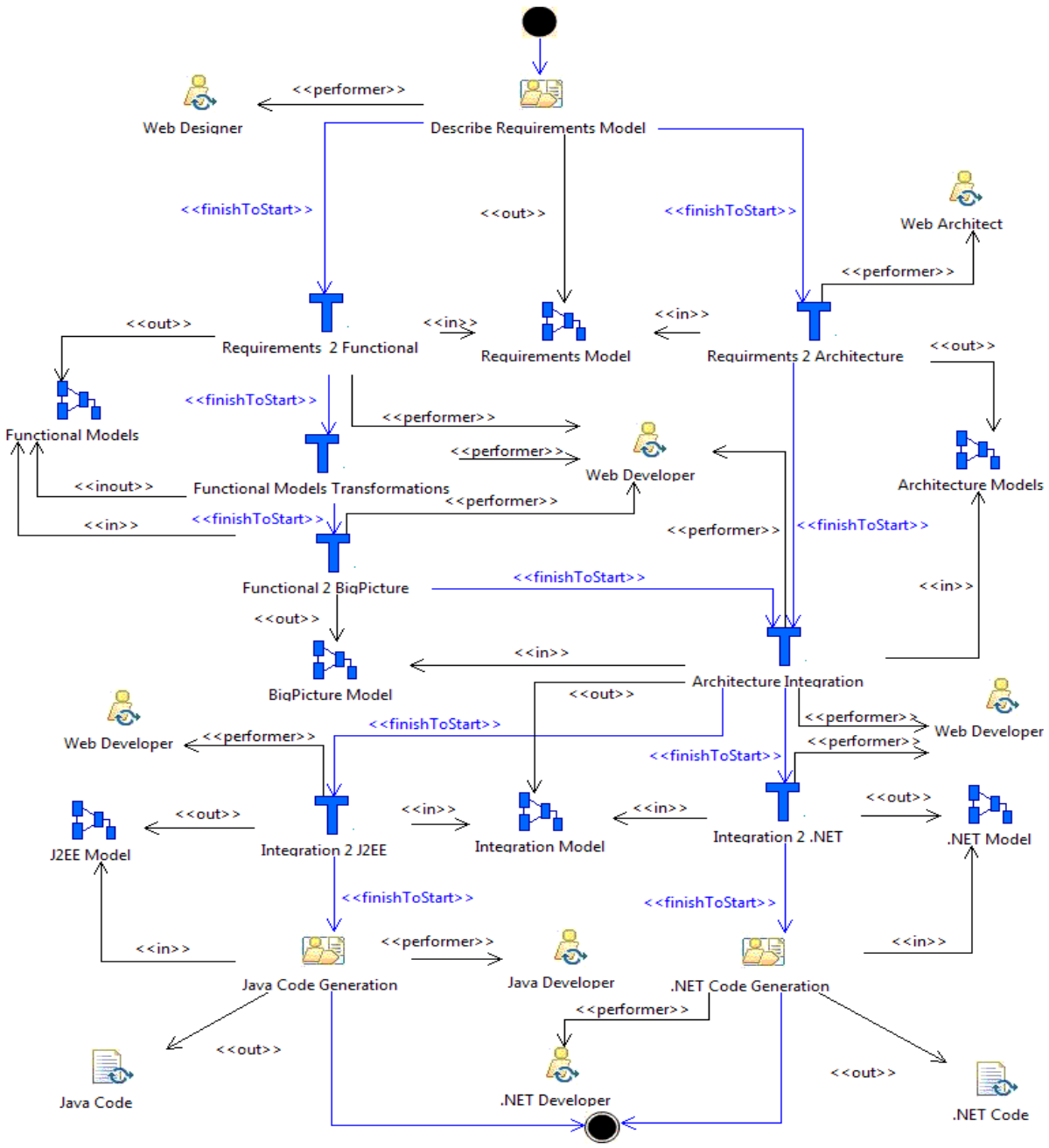


Figure V.3 Description structurelle du Processus UWE

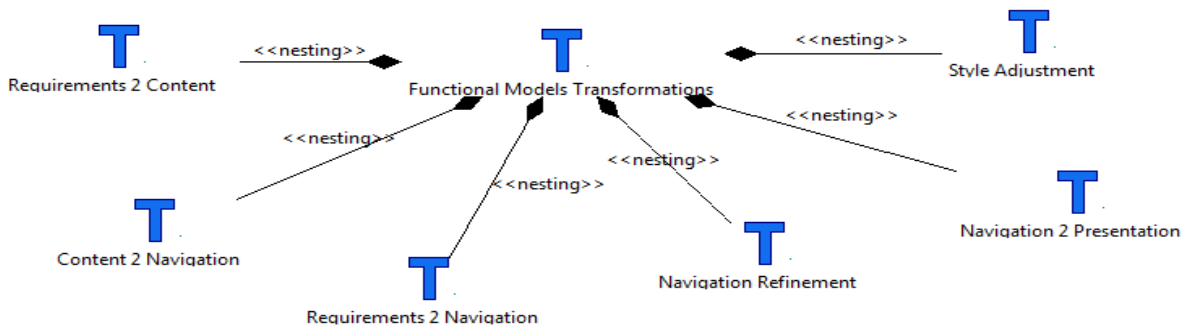


Figure V.4 Les transformations des modèles fonctionnels du processus UWE

V.4.1.1. La transformation « Requirements 2 Architecture »

Cette transformation permet de générer les modèles architecturaux à partir des modèles des exigences non-fonctionnelles. Cette transformation n'a pas été automatisée pour le moment dans le processus UWE. La Figure V.5 ci-dessous présente les relations entre la transformation « *Requirements 2 Architecture* » et sa définition « *Requirements 2 Architecture Definition* ». La définition d'une transformation spécifie les métamodèles source et cible de la transformation et les règles informelles de la transformation qui seront par la suite implémentées dans un formalisme basé sur des règles, un « template », ou un programme. La transformation « *Requirements 2 Architecture* », instance de *TransformationImpl* du métamodèle SPEM4MDE implémente sa définition qui est une instance de *TransformationDefinition* du métamodèle SPEM4MDE. De cette relation d'implémentation, découle une relation de conformité entre le modèle source (respectivement le modèle cible) de « *Requirements 2 Architecture* » et le métamodèle source (respectivement le métamodèle cible) de « *Requirements 2 Architecture Definition* ».

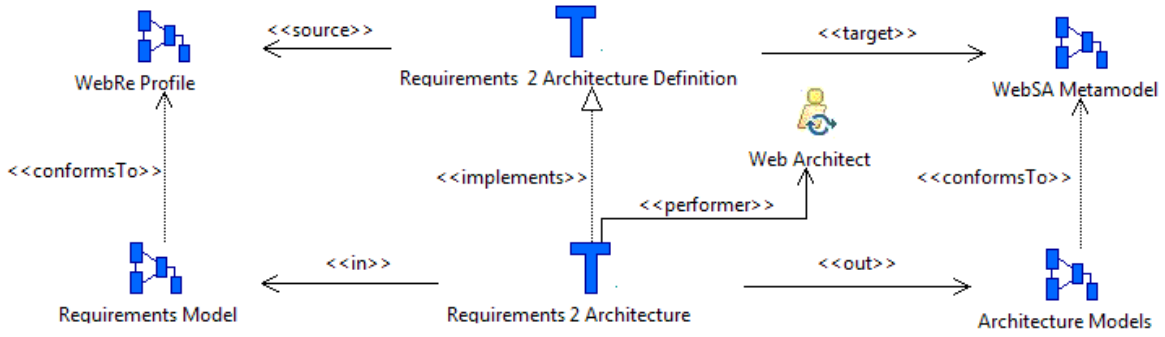


Figure V.5 Relations entre la transformation « *Requirements 2 Architecture* » et sa définition

V.4.1.2. Les transformations entre les modèles fonctionnels

La Figure V.6 ci-dessous décrit l'enchaînement des transformations des modèles fonctionnels dans le processus UWE. Ces transformations sont : « *Requirements 2 Content* », « *Content 2 Navigation* », « *Requirements 2 Navigation* », « *Navigation Refinement* », « *Navigation 2 Presentation* », et « *Style Adjustment* ». Le processus de transformations entre les modèles fonctionnels la transformation « *Requirements 2 Content* » qui produit le modèle du contenu (*Content Model*) à partir du modèle des exigences. La transformation « *Content 2 Navigation* » produit une première version du modèle de navigation (*Navigation Model*) à partir de certains éléments du modèle du contenu. Comme illustrée par la figure ci-dessus, la transformation « *Requirements 2 Content* » ne peut démarrer que si l'activité « *Describe Requirements Model* » est terminée et doit être terminée avant que la transformation « *Content 2 Navigation* » ne démarre. La transformation « *Requirements 2 Navigation* » permet d'enrichir la première version du modèle de navigation à partir du modèle des exigences. Un raffinement du modèle de navigation enrichi permet de produire la version finale du modèle de navigation. Par le biais de la transformation « *Navigation 2 Presentation* », une première version du modèle de présentation est générée. Enfin, le modèle de présentation est fusionné avec les styles pour produire la version finale du modèle de présentation.

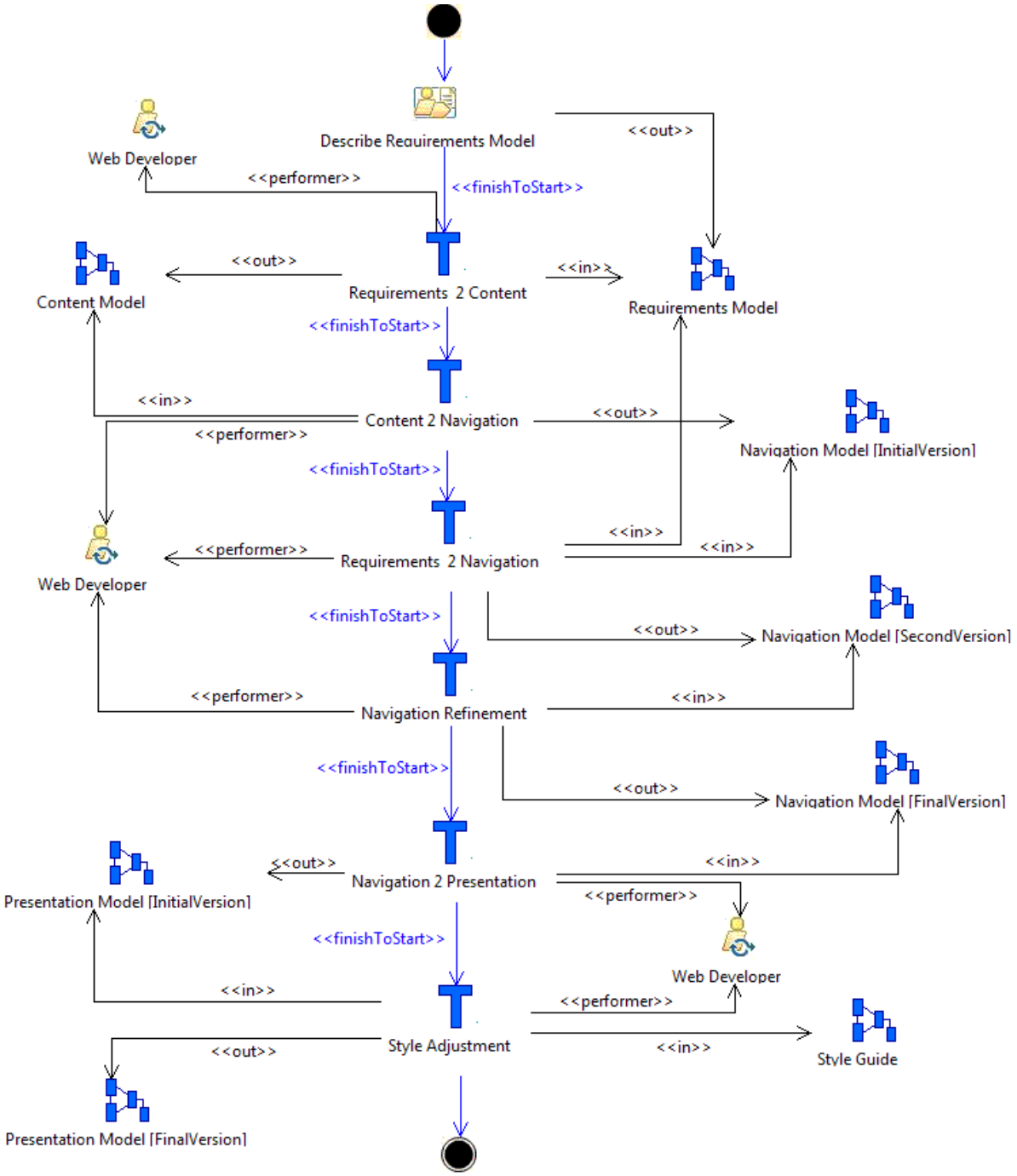


Figure V.6 Enchaînement des transformations des modèles fonctionnels

V.4.1.2.1. La transformation « Requirements 2 Content »

La transformation « *Requirements 2 Content* » est une transformation CIM (*Computational Independent Model*), vers PIM (*Platform Independent Model*), basée sur des règles QVT. Elle permet de générer le modèle de contenu d'une application à partir du modèle des exigences. La transformation « *Requirements 2 Content* » consiste à mapper les instances de la méta-classe *Content* du métamodèle *WebRE* (voir annexe D, section c) vers les classes du modèle de contenu. La Figure V.7 ci-dessous présente les relations entre la transformation « *Requirements 2 Content* » et sa définition.

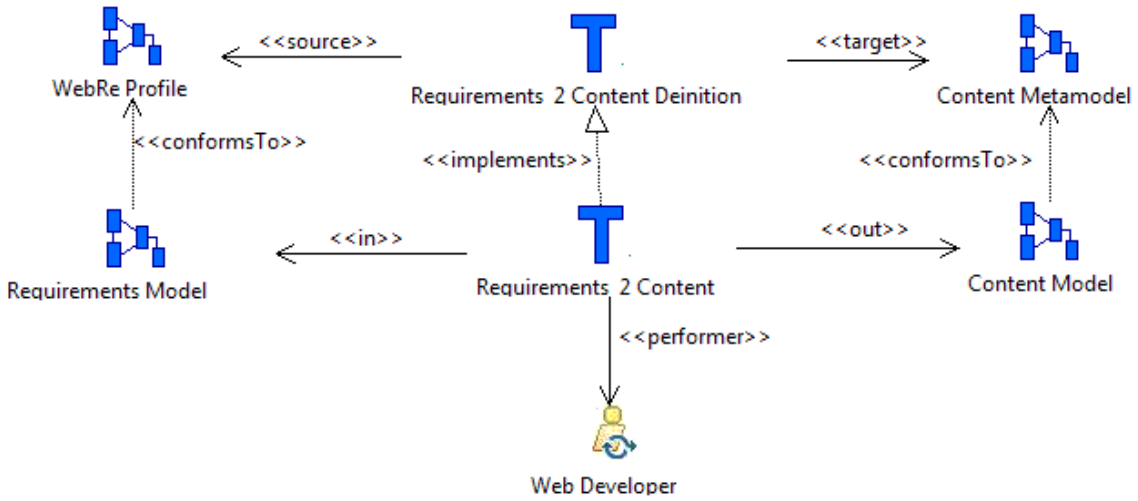


Figure V.7 Relations entre la transformation « *Requirements 2 Content* » et sa définition

L'extrait de code ci-dessous (voir Listing V.1) décrit les règles d'implémentation de la transformation « *Requirements 2 Content* » en QVT-Relations. Ce code sera exploité au moment de la mise en œuvre par un outil qui implémente le langage *QVT-Relation*. La règle ou relation R1 spécifie que le nom d'une instance de « Content » du modèle « webRe » sera le nom d'une instance de « Class » du modèle « uwe ». La relation R2 spécifie que les propriétés d'une instance de « Content » du modèle « webRe » seront les propriétés d'une instance de « Class » du modèle « uwe ».

```
Transformation Requirements2Content (webre:WebRE,uwe:UWE)
{
  top relation R1   {
  checkonly domain webre c:Content { name = n };
  enforce domain uwe cc: Class { name = n };
                    }
  top relation R2 {
  cn: String;
  checkonly domain webre p: Property { namespace=c:
  Content {}, name = cn};
  enforce domain uwe p1:Property { namespace = cc: Class{};
  name = cn}
  when {R1 (c,cc); }}}
```

Listing V.1 Description dans le formalisme QVT-Relation de la transformation « *Requirements 2 Content* »

V.4.1.2.2. La transformation « Content 2 Navigation »

La transformation « *Content 2 Navigation* » permet de générer le modèle de navigation à partir du modèle de contenu. C'est une transformation PIM vers PIM basée sur Java et ATL. Elle est semi-automatique car elle requiert une intervention humaine pour slectionner certains éléments du modèle du contenu (*Content Model*) qui sont essentiels pour générer le modèle de navigation. La Figure V.8

présente les relations entre la transformation « *Content 2 Navigation* » et sa définition.

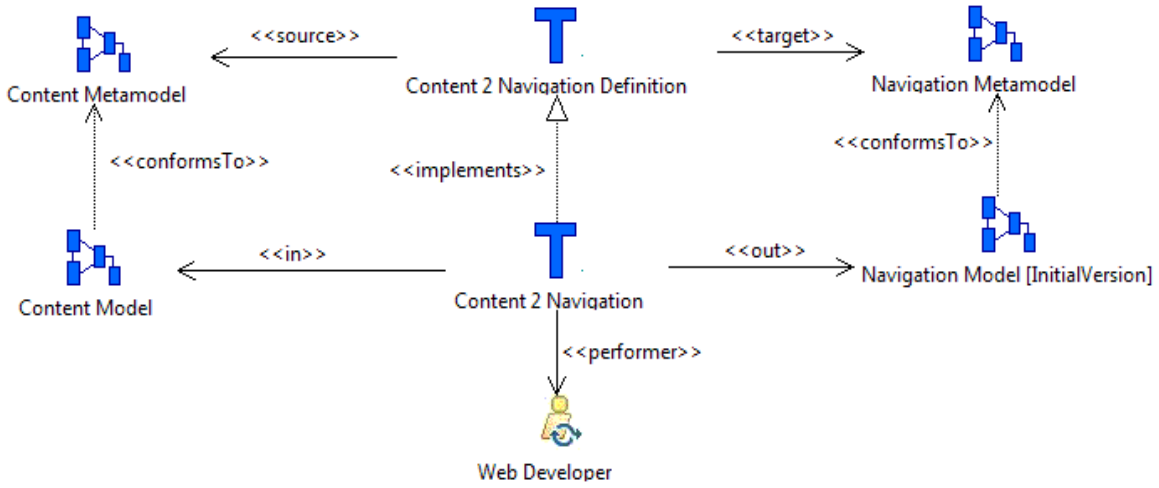


Figure V.8 La transformation « *Content 2 Navigation* » et sa définition

V.4.1.2.3. La transformation « Requirements 2 Navigation »

Le modèle des exigences contient des informations qui peuvent enrichir le modèle de navigation produit par la transformation précédente. La transformation « *Requirements 2 Navigation* » permet d'enrichir le modèle de navigation à partir du modèle des exigences. La Figure V.9 présente les relations entre la transformation « *Requirements 2 Navigation*» et sa définition. La transformation « *Requirements 2 Navigation* » a pour modèles source « *Requirements Model* » et « *Navigation Model* » conformes respectivement au profil *WebRE* et au métamodèle « *Navigation Metamodel* » et pour modèle cible « *Navigation Model* » conforme à « *Navigation Metamodel* ».

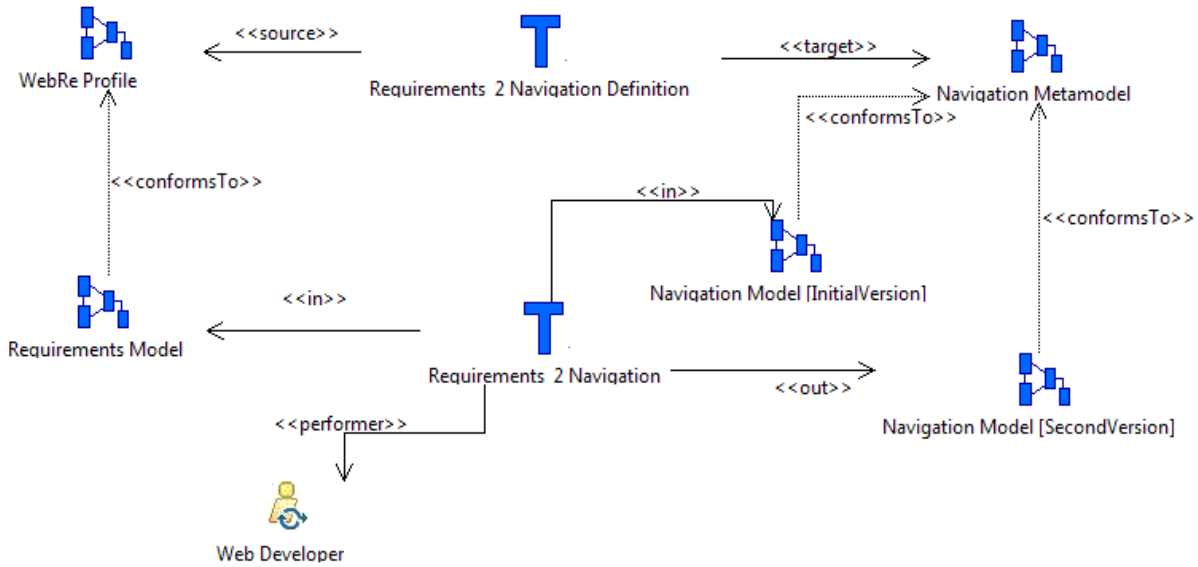


Figure V.9 La transformation « *Requirements 2 Navigation* » et sa définition

V.4.1.2.4. Raffiner le modèle de navigation (« Navigation Refinement »)

Le modèle de navigation produit par la transformation précédente n'est qu'une version initiale. Un raffinement de ce modèle est nécessaire pour prendre en compte certaines informations. C'est une transformation endogène car les métamodèles source et cible sont identiques. La Figure V.10 présente les relations entre la transformation « *Navigation Refinement*» et sa définition.

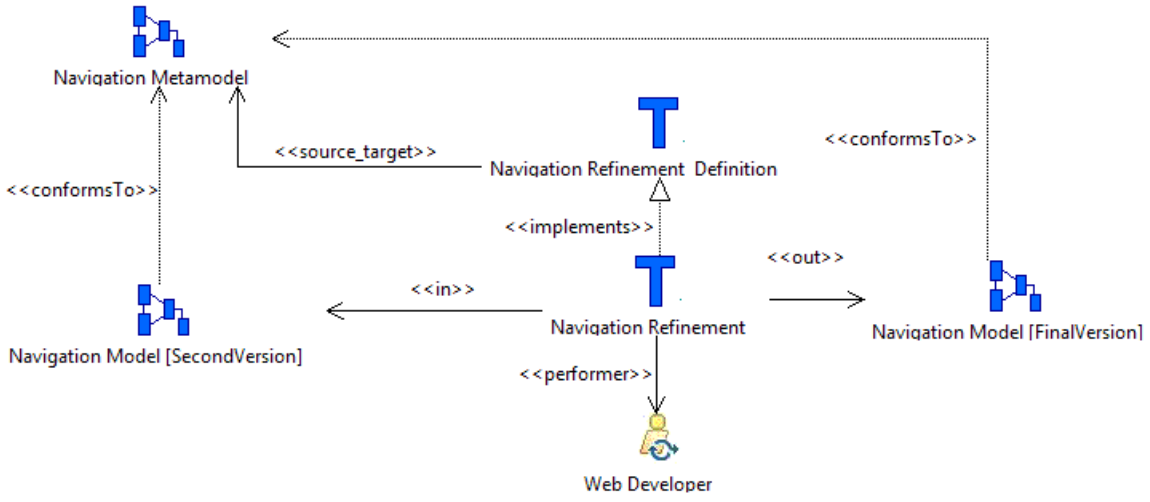


Figure V.10 La transformation « *Navigation Refinement* » et sa définition

V.4.1.2.5. La transformation « Navigation 2 Presentation »

Cette transformation permet de créer la première version du modèle de présentation à partir du modèle de navigation produit par la transformation précédente. Le modèle de présentation décrit les interfaces utilisateurs associées à chaque élément du modèle de navigation. La Figure V.11 présente les relations entre la transformation « *Navigation 2 Presentation* » et sa définition.

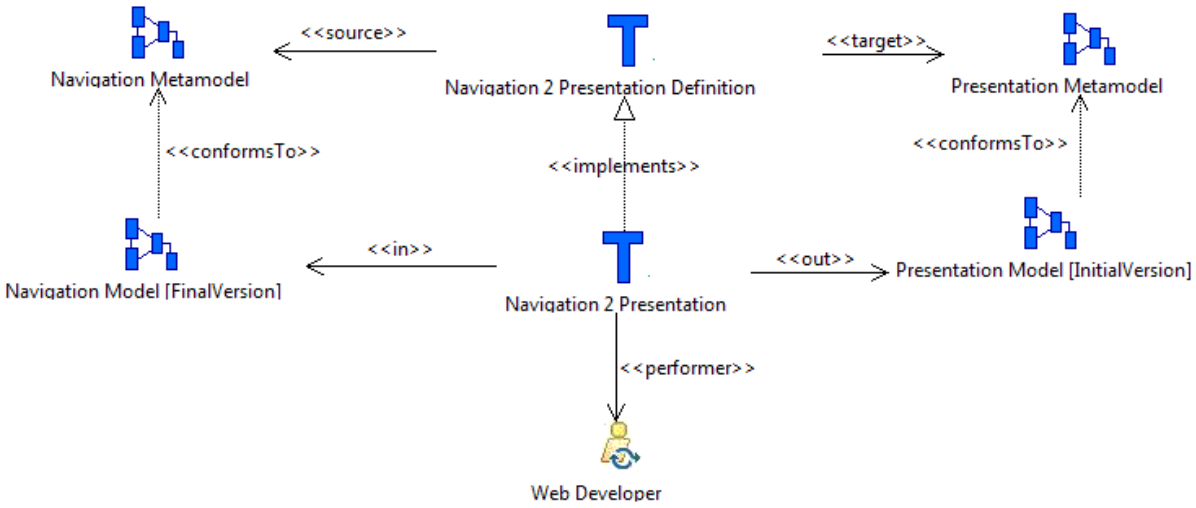


Figure V.11 La transformation « *Navigation* 2 *Presentation* » et sa définition

V.4.1.2.6. Application des styles sur le modèle de présentation (« Style Adjustment »)

Cette transformation permet de produire la version finale du modèle de présentation en y intégrant les styles. Les styles décrivent l'ergonomie des pages web d'une application. Ils décrivent les couleurs des interfaces utilisateur, la police, la taille, etc. La Figure V.12 présente les relations entre la transformation « *Style Adjustement*» et sa définition.

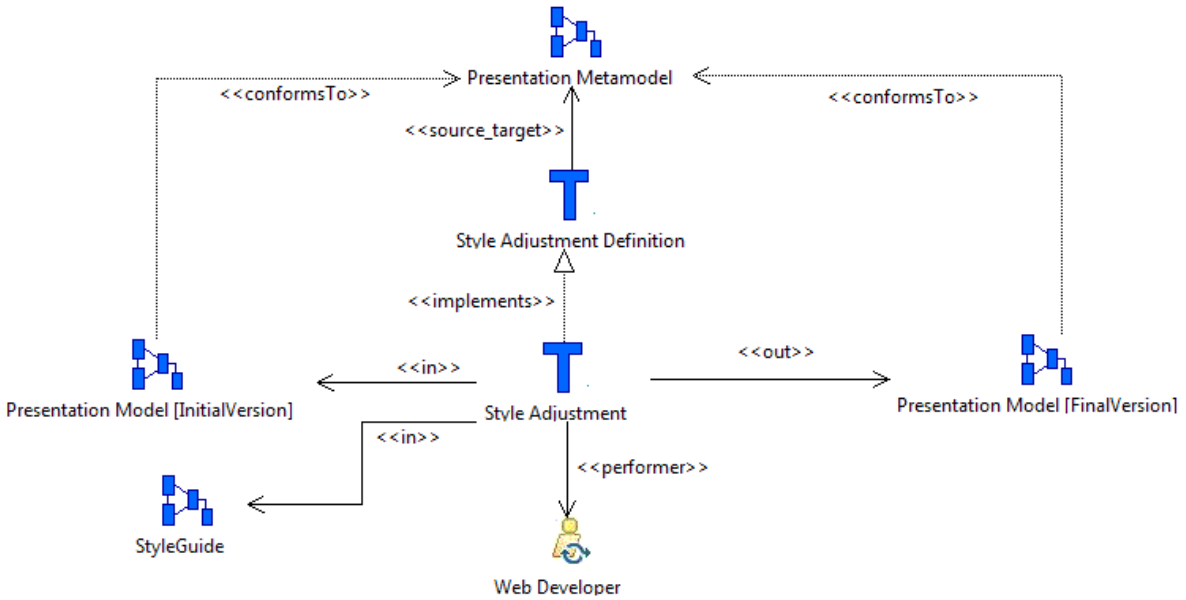


Figure V.12 La transformation « *Style Adjustment* » et sa définition

V.4.1.3. La transformation « Functional 2 BigPicture »

V.4.1.3. La transformation « *Functional 2 BigPicture* » est une transformation automatique qui permet de fusionner les modèles fonctionnels du processus UWE dans un modèle unique afin de les vérifier et de les valider. La Figure V.13 présente les relations entre la transformation « *Functional 2 BigPicture*» et sa définition.

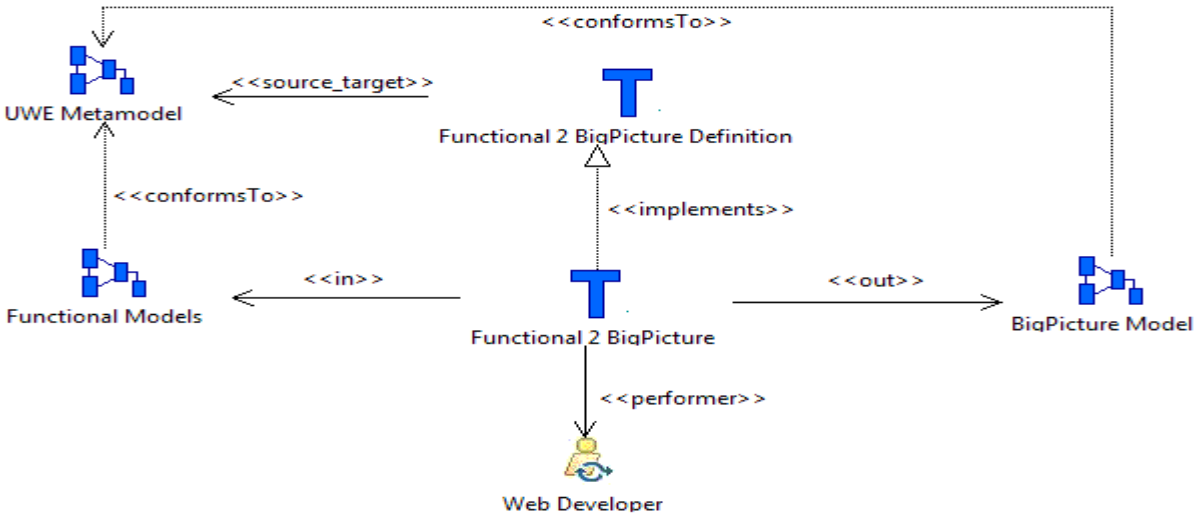


Figure V.13 La transformation « *Functional2BigPicture* » et sa définition

V.4.1.4. La transformation « Architecture Integration »

La transformation « *Architecture Integration* » permet d'intégrer le modèle *BigPicture* produit par la transformation précédente et les modèles architecturaux produits par la transformation « *Requirements 2 Architecture* » dans un modèle unique appelé modèle d'intégration. La Figure V.14 présente les relations entre la transformation « *Architecture Integration* » et sa définition.

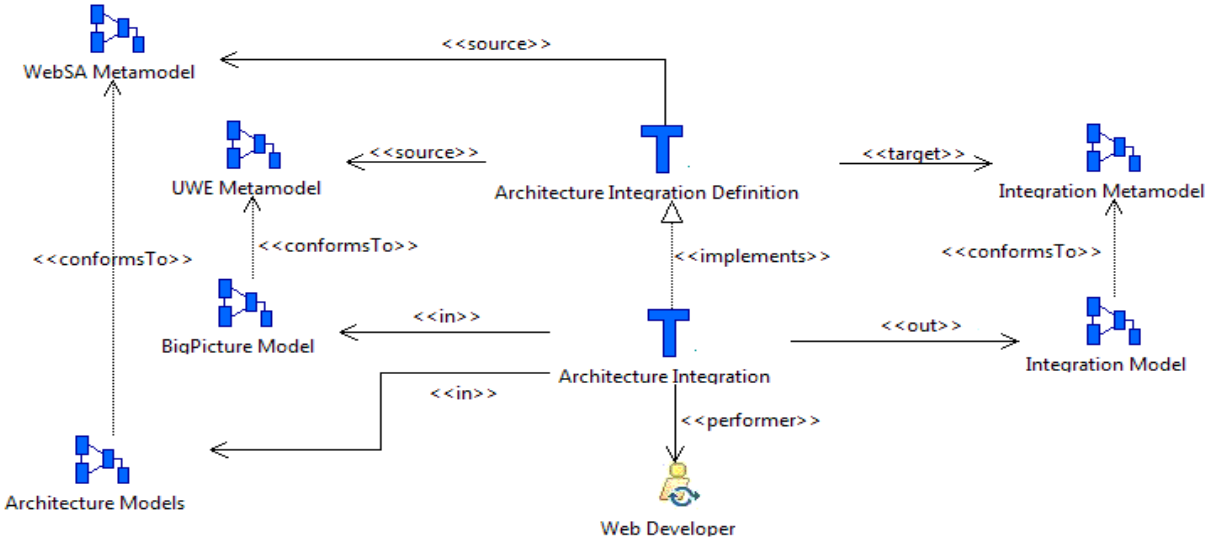


Figure V.14 La transformation « *Architecture Integration* » et sa définition

V.4.1.5. La génération des PSM (« Integration 2 J2EE » et « Integration 2 .NET »)

Les transformations « *Integration 2 J2EE* » et « *Integration 2 .NET* » produisent respectivement le modèle de la plate-forme J2EE et le modèle de la plate-forme .NET à partir du modèle d'intégration produit par la transformation précédente. La Figure V.15 présente les relations entre la transformation « *Integration 2 J2EE* » et sa définition, alors que la Figure V.16 présente les relations entre la transformation « *Integration 2 .NET* » et sa définition

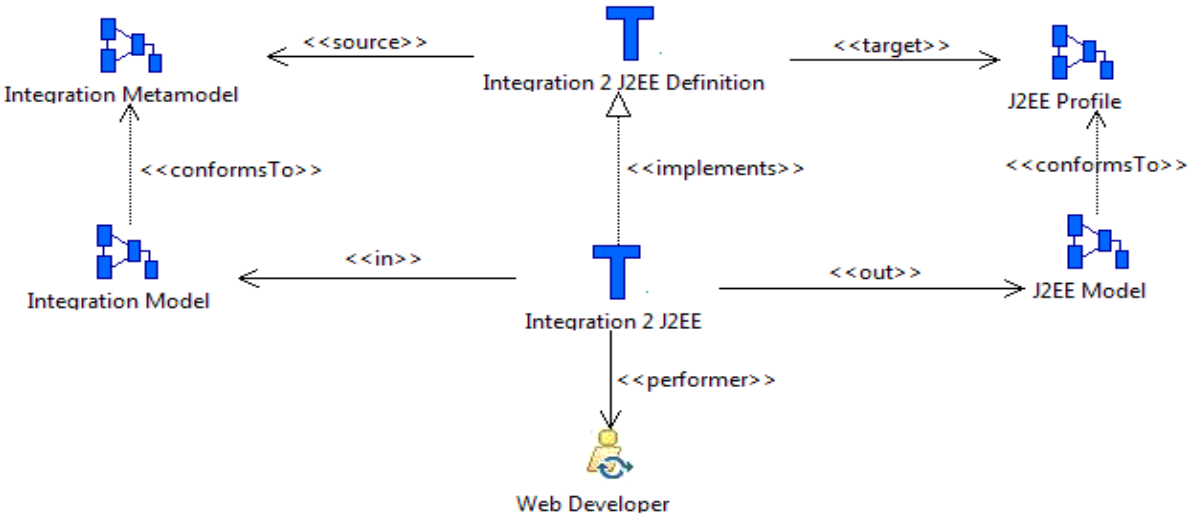


Figure V.15 La transformation « *Integration 2 J2EE* » et sa définition

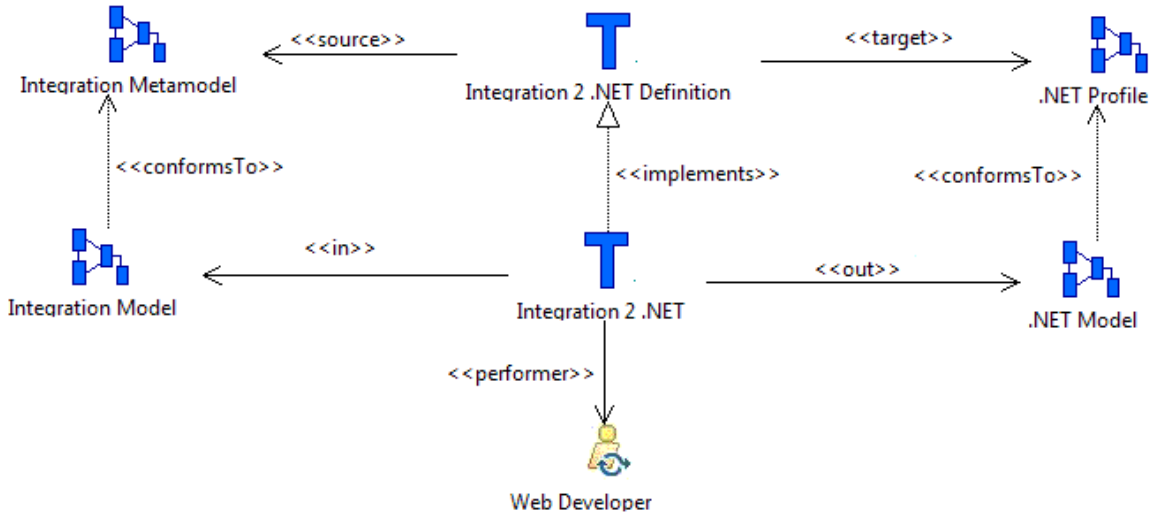


Figure V.16 La transformation « *Integration 2 .NET* » et sa définition

V.4.2. Description comportementale du processus UWE

Dans cette section, nous décrivons les modèles comportementaux du processus UWE. Ces modèles décrivent les comportements des activités/transformations, des modèles, des outils et des rôles du processus UWE. Nous réutilisons dans la suite de ce chapitre les modèles comportementaux génériques décrits dans le chapitre III.

V.4.2.1. Comportement de l'activité « Describe Requirements Model »

Pour décrire le comportement de l'activité *« Describe Requirements Model »*, nous réutilisons le comportement générique d'une activité (voir Figure III.9). La fonction « *startable()* » qui définit la précondition et les précédences de l'activité *« Describe Requirements Model »* retourne vraie car cette activité n'a ni précondition ni précédence. La fonction « *validate()* » ci-dessous décrit la postcondition de l'activité *« Describe Requirements Model »*. Cette fonction retourne vraie si le modèle produit par l'activité *« Describe Requirements Model »* se trouve dans l'état *« FinalVersion »*.

Postcondition de l'activité ***« Describe Requirements Model »***

Context ***«Describe Requirements Model»*** ::validate() : Boolean

post: **result**=self.ownedProcessParameter→**select**(i|i.direction=#out)→**collect**(i | i.parameterType)→ **forall**(m | m.state= ' FinalVersion')

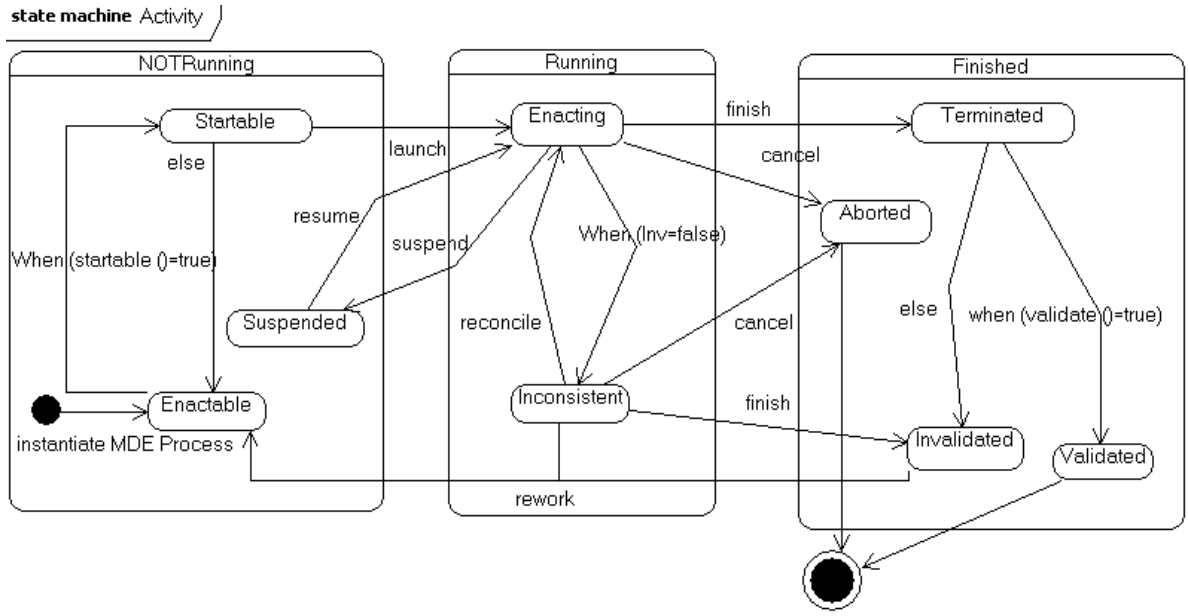


Figure V.17. Rappel du comportement de l'activité « *Describe Requirements Model* »

V.4.2.2. Comportement d'une transformation du processus UWE

Les comportements des transformations du processus UWE (« *Requirements 2 Architecture* », « *Requirements 2 Content* », « *Content 2 Navigation* », « *Requirements 2 Navigation* », « *Navigation Refinement* », « *Navigation 2 Presentation* », « *Style Adjustment* », « *Functional 2 BigPicture*», « *Architecture Integration* », « *Integration 2 J2EE* », et « *Integration 2 .NET* ») sont décrits en réutilisant le comportement générique d'une transformation (voir Figure III.8).

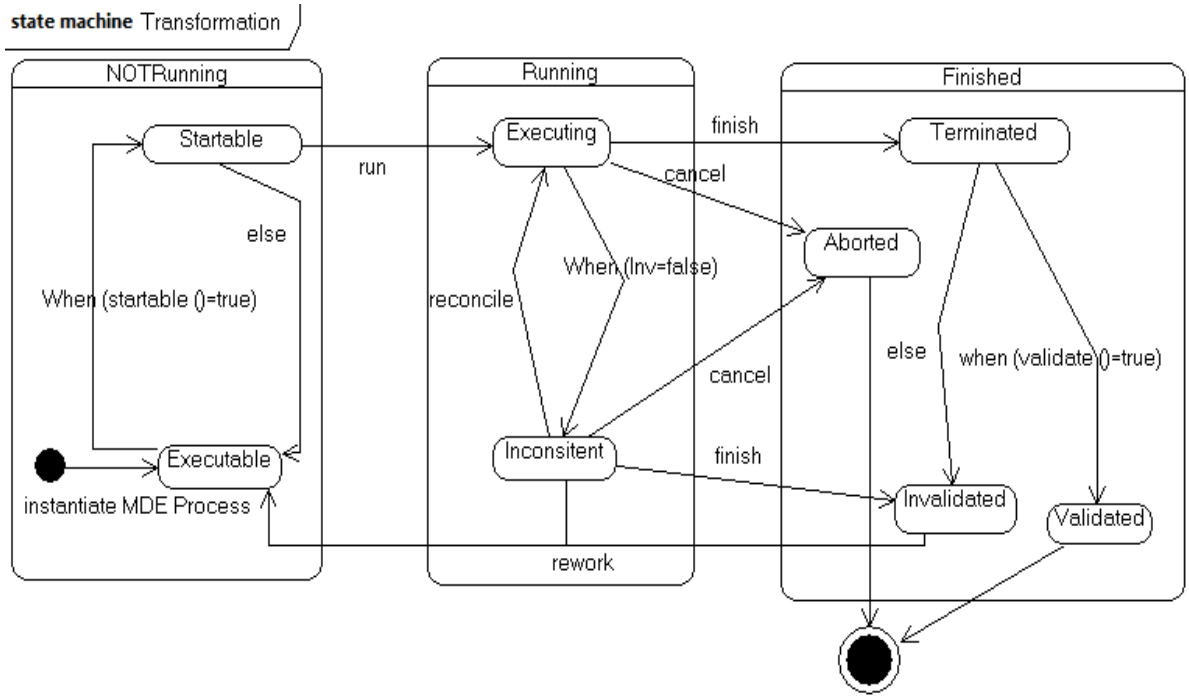


Figure V.18. Rappel du comportement d'une transformation du processus UWE

La fonction « *startable*() » ci-dessous spécifie les conditions (la précondition et les contraintes de précédences) à vérifier avant de commencer la mise en œuvre d'une transformation du processus UWE alors que la fonction « *validate*() » décrit les conditions à vérifier (postcondition) pour qu'une transformation du processus UWE soit déclarée valide.

Précondition d'une transformation du Processus UWE

Context TransformationImpl::startable() : Boolean

post: **result**=self.parameters→**select**(i|i.direction=#in or i.direction=#inout)→**collect**(i | i.model)→**forall**(m|m.state='ValidatedVersion') and self.linkToPredecessor)→**collect**(p|p.predecessor)→ **forall**(p|p.state='Finished')

Postcondition d'une transformation du Processus UWE

Context TransformationImpl::validate() : Boolean

post: **result**=self.parameters→**select** (i|i.direction=#out or.direction=#inout)→**collect**(i | i.model)→**forall**(m|m.state='FinalVersion')

V.4.2.3. Comportement d'un modèle créé ou modifié par une activité/transformation du processus UWE

Pour décrire le comportement du modèle « *Requirements Model* » créé par l'activité *« Describe Requirements Model »* et des modèles *(« Architecture Models », « Content Model », « Navigation Model », « Presentation Model », « BigPicture Model », « Integration Model », « J2EE Model », et «. NET Model »*) créés par les transformations du processus UWE, nous réutilisons le comportement générique d'un modèle créé par une activité ou une transformation (voir chapitre III, Figure III.10).

Pour décrire le comportement du modèle *« Navigation Model »* modifié par les transformations « *Requirements 2 Navigation* » et *« Navigation Refinement* » et du modèle *« Presentation Model »* modifié par la transformation « *Style Adjustment* » nous réutilisons le comportement générique d'un modèle modifié par une activité ou une transformation (voir chapitre III, Figure III.11).

V.4.2.4. Comportements d'un outil et d'un rôle du processus UWE

La description des comportements des outils utilisés dans le processus UWE (*ArgoUWE, MediniQVT, ATL*) est basée sur le comportement générique d'un outil IDM (voir chapitre III, Figure III.12). Pour décrire le comportement des rôles du processus UWE *(« Web Designer, « Web Architect », « Web Developer »*), nous réutilisons le comportement générique d'un rôle (voir chapitre III, Figure III.13).

V.4.3. Adaptation du modèle de processus UWE

Avant de mettre en œuvre un modèle de processus, il est impératif de l'adapter à un projet spécifique. Cette adaptation réalisée par le biais du module « *SPEM4MDE Process Editor* » de notre prototype consiste à modifier si nécessaire le modèle de processus générique et à assigner toutes les ressources nécessaires à sa mise en œuvre (acteurs humains devant jouer les rôles, espace de travail pour chaque acteur du projet, outils d'implémentation et d'exécution des transformations, outils requis par les activités). L'adaptation porte uniquement sur les modèles structurels ; l'adaptation des modèles comportementaux ayant été décrite dans la section précédente.

Pour illustrer l'adaptation du modèle structurel du processus UWE, nous considérons un exemple de projet de développement d'un portail web de messagerie et un extrait du processus UWE (voir Figure V.3) limité à l'activité « *Describe Requirements Model* » et aux trois transformations « *Requirements 2 Content* », « *Content 2 Navigation* » et « *Navigation 2 Presentation* ».

Le portail web devra permettre aux utilisateurs de créer des comptes de messagerie. Pour créer un compte, un utilisateur fournit un nom utilisateur, un mot de passe, son nom, son prénom, sa zone géographique, et son e-mail de récupération. L'utilisateur se connecte avec son nom d'utilisateur et son mot de passe. Une fois connecté, il peut visualiser, rechercher, composer, envoyer, supprimer, ou enregistrer un message. Il pourra aussi mettre une image et une signature pour son profil.

V.4.3.1. Adapter l'activité « Describe Requirements Model »

La Figure V.19 illustre l'adaptation de l'activité « *Describe Requirements Model* » à notre exemple d'application. L'acteur Bob joue le rôle d'un concepteur web (Web Designer) et réalise l'activité « *Describe Requirements Model* » à l'aide de l'outil ArgoUWE. A ce stade et conformément aux états initiaux des modèles comportementaux, l'activité se trouve à l'état « *Enactable* », le modèle qu'elle devra produire (« *Mail Portal Requirements Model* ») se trouve à l'état « *Defined* », l'acteur Bob se trouve à l'état « *Assigned* », et l'outil ArgoUWE se trouve à l'état « *Required* ».

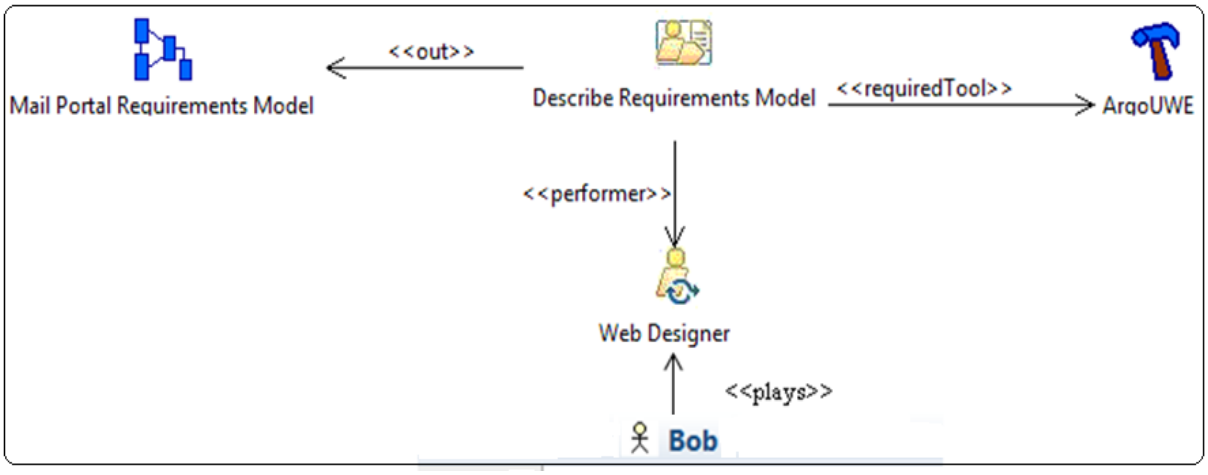


Figure V.19 Adaptation de l'activité « *Describe Requirements Model* »

V.4.3.2. Adapter la transformation « *Requirements 2 Content* »

L'adaptation de la transformation de la transformation « *Requirements 2 Content* » à notre exemple d'application est illustrée par la Figure V.20 ci-après. Elle consiste à assigner les ressources nécessaires à la mise en œuvre de la transformation. Ces ressources sont l'acteur humain Jean qui joue le rôle d'un développeur web, le modèle source de la transformation adaptée (*Mail Portal Requirements Model*), le modèle cible de la transformation adaptée (*Mail Portal Content Model*), et l'outil MediniQVT pour exécuter la transformation adaptée. A ce stade, la transformation « *Requirements 2 Content* » se trouve à l'état « *Executable* », ses modèles source et cible se trouvent à l'état « *Defined* », l'acteur Jean se trouve à l'état « *Assigned* » et l'outil *MediniQVT* se trouve à l'état « *Required* ».

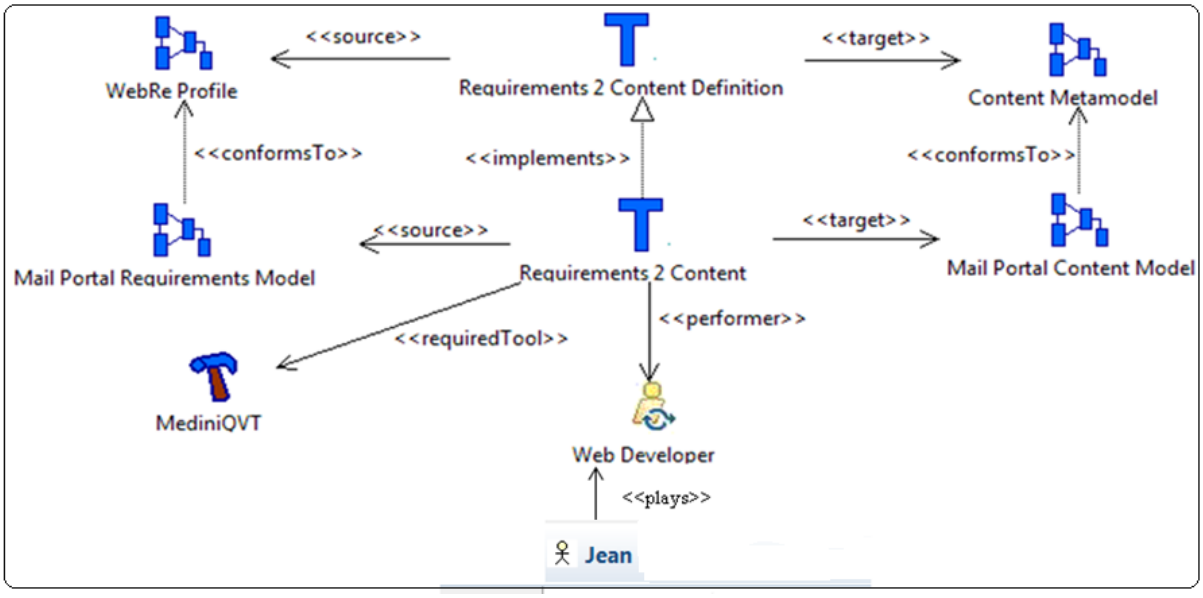


Figure V.20 Adaptation de la transformation « *Requirements 2 Content* »

V.4.3.3. Adapter la transformation « *Content 2 Navigation* »

La Figure V.21 illustre l'adaptation de la transformation « *Content 2 Navigation* » à notre exemple d'application. L'acteur Amara joue le rôle d'un développeur web et réalise la transformation « *Content 2 Navigation* » à l'aide de l'outil ATL. A ce stade, la transformation « *Content 2 Navigation* » se trouve à l'état « *Executable* », ses modèles source et cible se trouvent à l'état « *Defined* », l'acteur Amara se trouve à l'état « *Assigned* » et l'outil ATL se trouve à l'état « *Required* ».

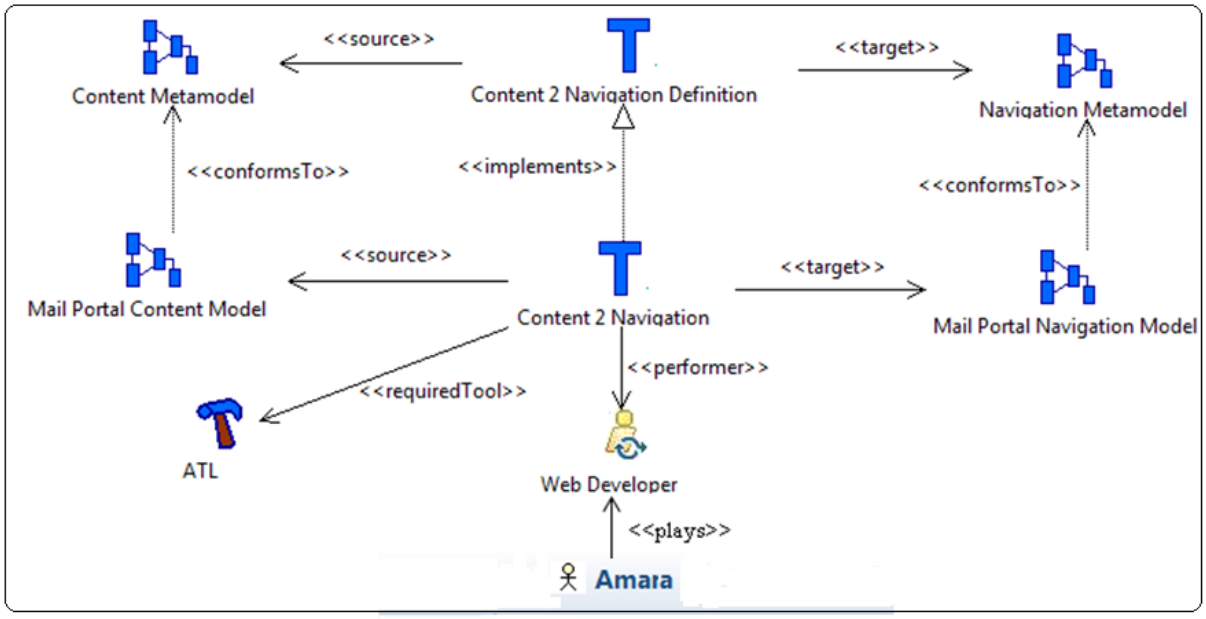


Figure V.21 Adaptation de la transformation « *Content 2 Navigation* »

V.4.3.4. Adapter la transformation « *Navigation 2 Presentation* »

La Figure V.22 illustre l'adaptation de la transformation « *Navigation 2 Presentation* » à notre exemple d'application. L'acteur Dame joue le rôle d'un développeur web et réalise la transformation « *Navigation 2 Presentation* » à l'aide de l'outil ATL. A ce stade, la transformation « *Navigation 2 Presentation* » se trouve à l'état « *Executable* », ses modèles source et cible se trouvent à l'état « *Defined* » l'acteur Dame se trouve à l'état « *Assigned* » et l'outil ATL se trouve à l'état « *Required* ».

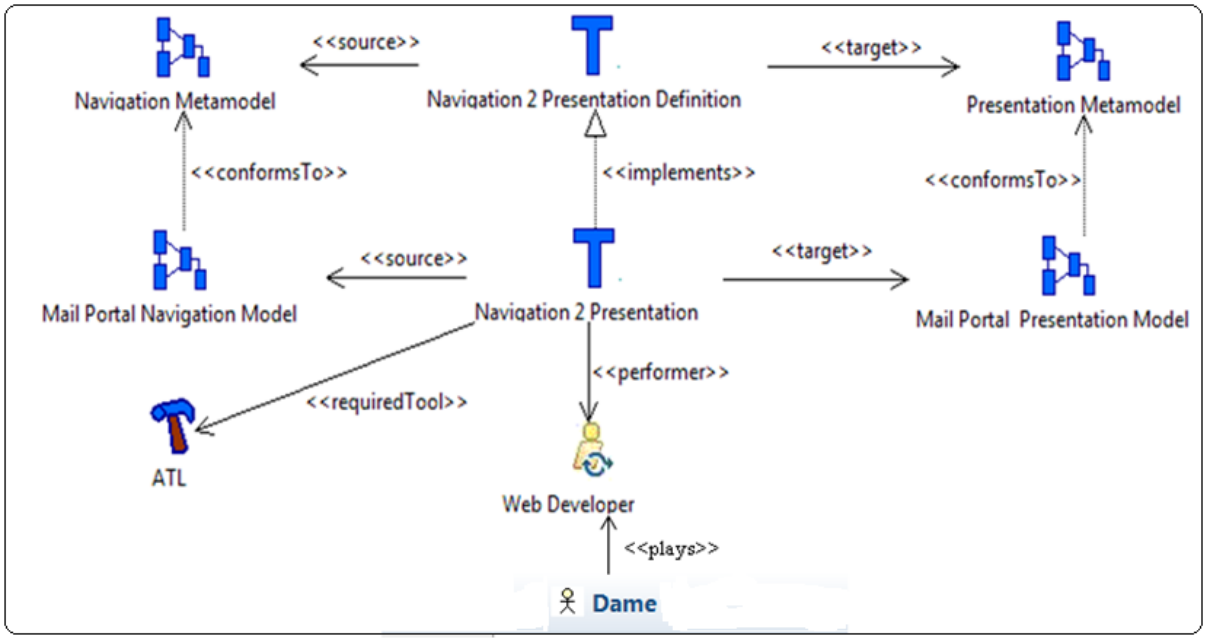


Figure V.22 Adaptation de la transformation « *Navigation 2 Presentation* ».

V.4.4. Mise en œuvre du modèle de processus UWE

La mise en œuvre du processus UWE est basée sur les modèles comportementaux décrits dans la section V.4.2. Grâce à ces modèles comportementaux, le module « *SPEM4MDE Process Enactment Engine* » offre aux développeurs des entrées de menu graphique qui montrent pour chaque élément du processus, les opérateurs de mise en œuvre spécifiques. Les opérateurs de mise en œuvre sont spécifiés par les transitions de la machine à états de chaque élément du processus qui a un comportement. Selon l'état courant de l'élément du processus UWE, ces entrées distinguent à tout instant les opérateurs éligibles (*nom de l'opérateur précédé d'une marque verte*) des opérateurs inéligibles (*nom de l'opérateur précédé d'une croix rouge*). Si un développeur exécute un opérateur éligible alors l'élément de processus lié change d'état après qu'une suite d'actions soit réalisée. Par contre si un développeur exécute un opérateur inéligible alors le module « *SPEM4MDE Process Enactment Engine* » indique sous forme de message les actions à faire pour que cet opérateur puisse être éligible.

Pour illustrer la mise en œuvre, nous considérons l'extrait du processus UWE adapté qui est décrit dans la section précédente. D'après les relations de précédence (voir Figure V.3), la mise en œuvre de la transformation « *Requirements 2 Content* » ne peut démarrer que si la mise en œuvre de l'activité « *Describe Requirements Model* » est terminée, et doit être terminée avant que la mise en œuvre de la transformation « *Content 2 Navigation* » ne démarre.

Toujours d'après les relations de précédence (voir Figure V.6) la mise en œuvre de la transformation « *Content 2 Navigation* » ne peut démarrer que si la mise en œuvre de la transformation « *Requirements 2 Content* » est terminée, et doit être terminée avant que la mise en œuvre de la transformation « *Navigation 2 Presentation* » ne démarre.

Nous rappelons au lecteur que la présentation de cette section est exhaustive mais assez similaire surtout au niveau de la mise en œuvre des transformations.

V.4.4.1. Mise en œuvre de l'activité « *Describe Requirements Model* »

La mise en œuvre de l'activité « *Describe Requirements Model* » est basée sur son modèle comportemental (voir section V.4.2.1). Le résultat attendu pour cette activité est un modèle d'exigences sous forme de diagramme de cas d'utilisation. L'activité étant déjà adaptée à notre exemple d'application, elle se trouve à l'état « *Enactable* », son réalisateur Bob se trouve à l'état « *Assigned* », et l'outil *ArgoUWE* utilisé pour sa réalisation se trouve à l'état «*Required* ». Cette activité n'a pas de précédence ni de précondition donc elle passe automatiquement de l'état « *Enactable* » à l'état « *Startable* ». La Figure V.23 ci-après montre les opérateurs de mise en œuvre de l'activité « *Describe Requirements Model* » quand elle atteint l'état « *Startable* ».

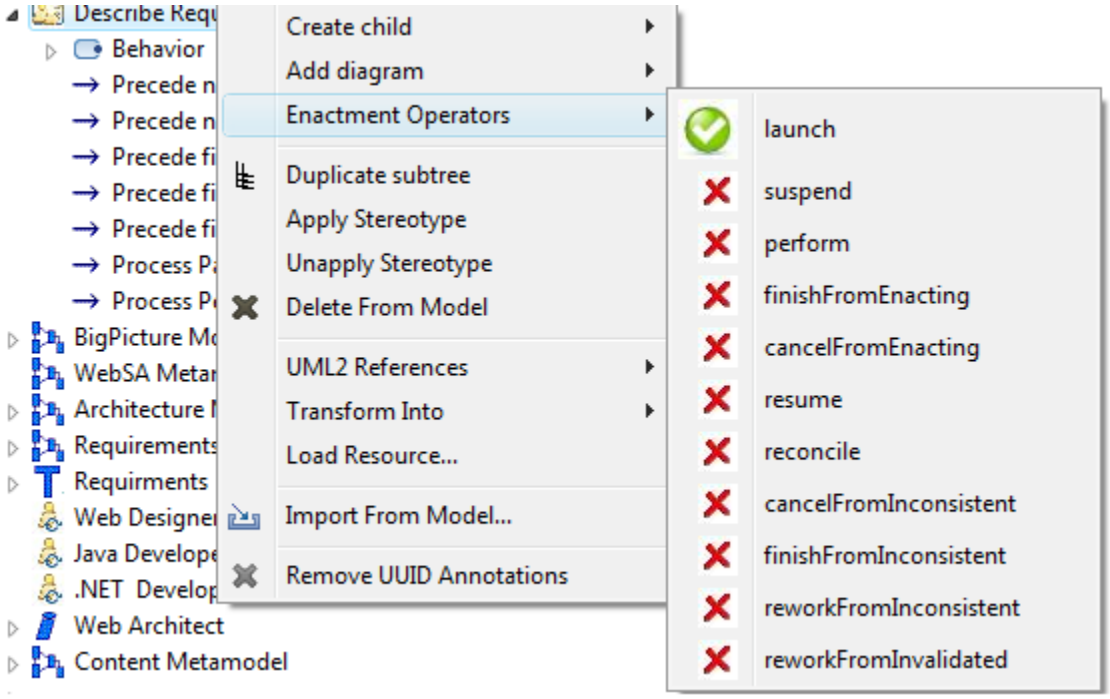
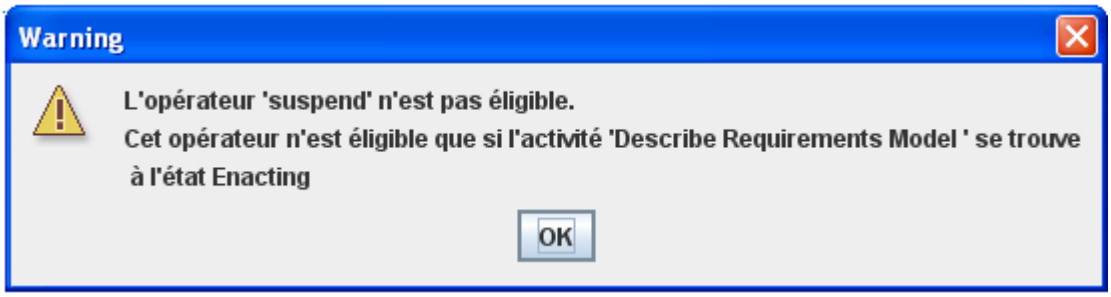
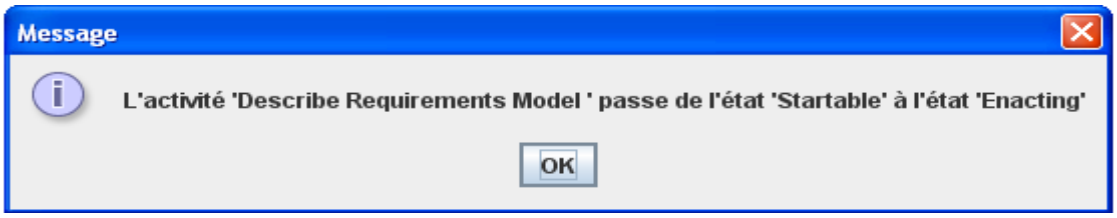


Figure V.23 Opérateurs de mise en œuvre de l'activité « *Describe Requirements Model* »

Quand l'activité se trouve à l'état « *Startable*, seul l'opérateur « *launch* » est éligible. Si Bob exécute par exemple l'opérateur « *suspend* » qui n'est pas éligible, le message suivant est affiché.



Si l'opérateur « *launch*» est exécuté sur l'activité « *Describe Requirements Model* », celle-ci passe de l'état «*Startable* » à l'état « *Enacting* » (voir message ci-après).



On suppose que Bob a déjà ouvert l'outil « *ArgoUWE* » en vue de mettre en œuvre l'activité « *Describe Requirements Model* », l'opérateur « *launch* » devient donc éligible (voir Figure V.24 ci-dessous). L'exécution de l'opérateur « *launch* » entraine simultanément le passage de l'outil à l'état « *Running* » et le passage de Bob à l'état « *Performing* ».

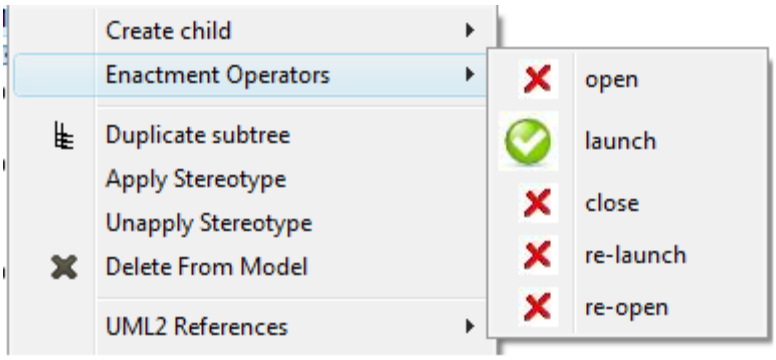


Figure V.24 État des opérateurs de mise en œuvre si l'outil *ArgoUWE* est ouvert

Conformément au modèle comportemental d'une activité, le passage de l'activité« *Describe Requirements Model* » de l'état « *Startable* » à l'état « *Enacting* » rend éligibles les opérateurs « *suspend* », «*perform* », « *finishFromEnacting* », et « *cancelFromEnacting* » (voir Figure V.25).

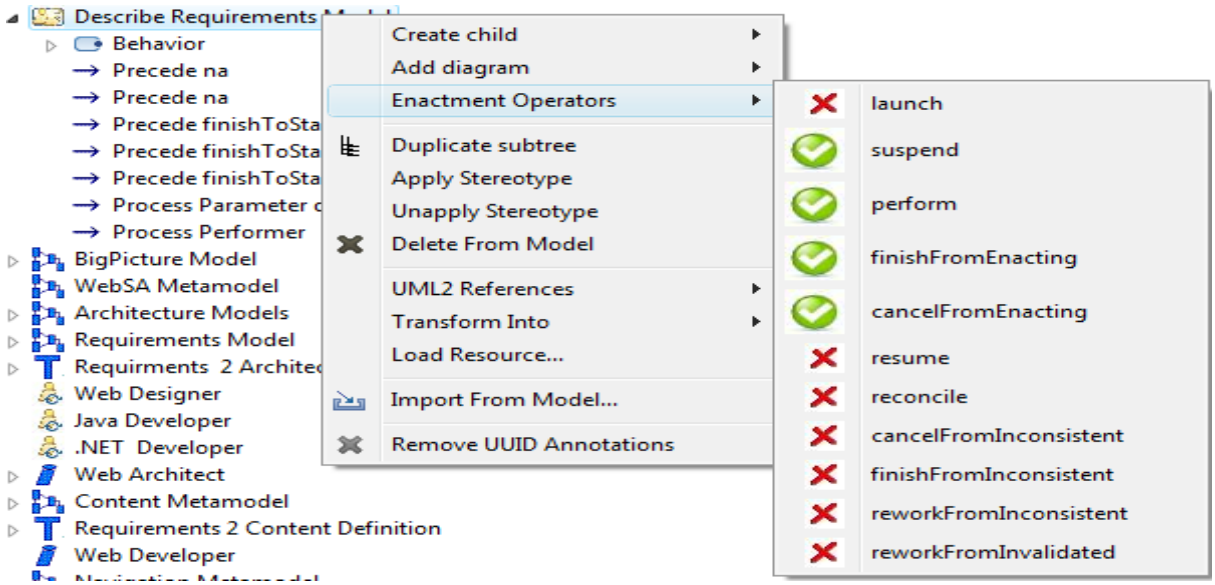


Figure V.25 État des opérateurs de mise en œuvre de l'activité « *Describe Requirements Model* » après son lancement

Nous supposons par la suite que Bob choisit de lancer l'opérateur « *perform* » sur l'activité « *Describe Requirements Model* ». L'activité « *Describe Requirements Model* » et l'outil *ArgoUWE* restent respectivement à l'état « *Enacting* » et à l'état « *Running* ». L'activité se trouvant à l'état « *Enacting* », les opérateurs « *suspend* », «*perform* », « *finishFromEnacting* », et « *cancelFromEnacting* » sont toujours éligibles (voir Figure V.25).

Si Bob décide de terminer la mise en œuvre de l'activité « *Describe Requirements Model* », il exécute l'opérateur « *finishFromEnacting* ». L'activité « *Describe Requirements Model* » passe donc de l'état « *Enacting* » à l'état « *Terminated* » produisant ainsi le modèle des exigences (voir Figure V.26). Le passage de l'activité « *Describe Requirements Model* » à l'état « *Terminated* » entraine automatiquement le passage du modèle des exigences à l'état « *InitialVersion* », de l'outil *ArgoUWE* à l'état « *Used* », et de l'acteur Bob à l'état « *Performed* ».

La Figure V.26 Figure V.26 ci-après illustre sous forme de diagramme de cas d'utilisation le

modèle des exigences produit par l'activité « *Describe Requirements Model* ». Un utilisateur web peut créer un compte de messagerie. Il utilise un compte utilisateur et un mot de passe qui ont été fournis au moment de l'enregistrement pour se connecter sur le serveur de messagerie. Une fois connecté, l'utilisateur peut visualiser, rechercher, composer, envoyer, supprimer, et enregistrer un message électronique. Les cas d'utilisation relatifs au processus métier sont : « *Se connecter* », « *Se Déconnecter* », « *Créer Compte* », « *Supprimer Message* », « *Composer Message* », « *Enregistrer Message* », et « *Envoyer Message* ». Les cas d'utilisation relatifs à la navigation sont : « *Rechercher Message* » et « *Visualiser Message* ».

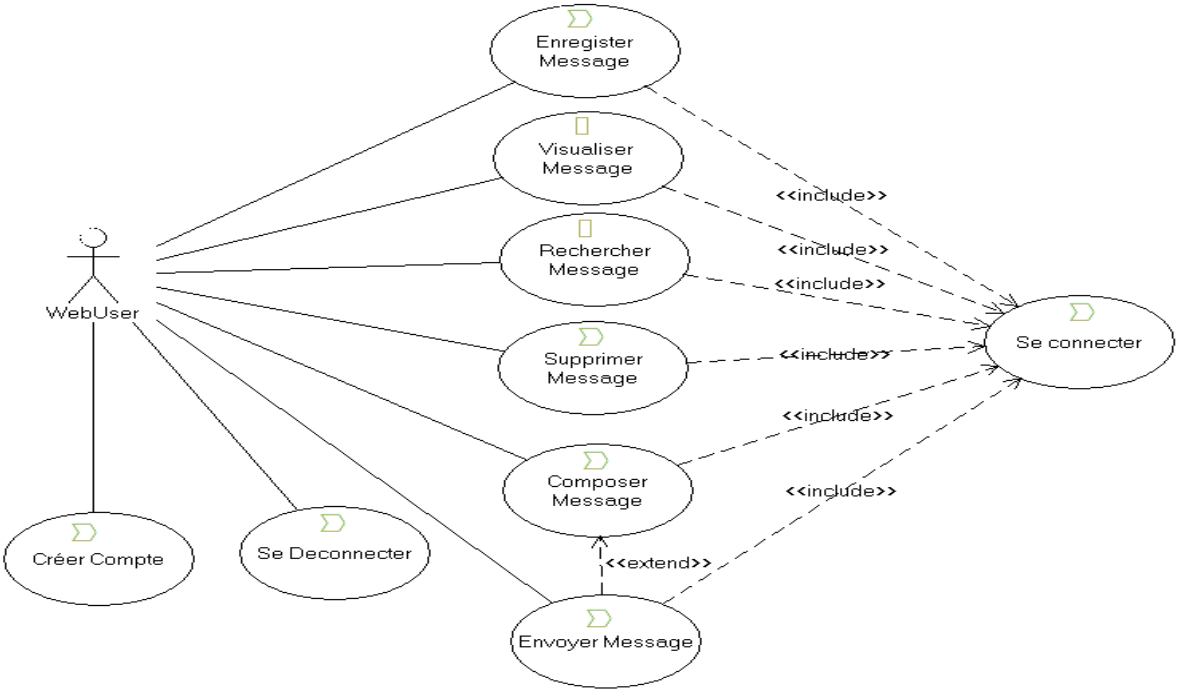


Figure V.26 Modèle des exigences produit par l'activité « *Describe Requirements Model* »

Le modèle produit par l'activité « *Describe Requirements Model* » se trouve à l'état « *InitialVersion* » donc les opérateurs « *finishFromInitialVersion* » et « *startRefactoring* » deviennent éligibles sur celui-ci (voir Figure V.27).

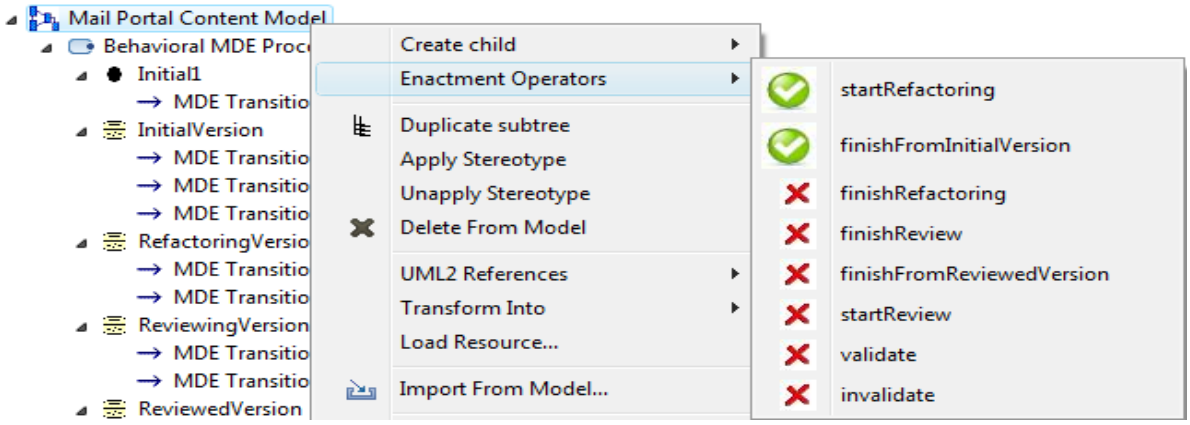


Figure V.27 État des opérateurs de mise en œuvre après le passage du modèle « *Mail Portal Requirements Model* » à l'état « *InitialVersion* »

L'outil se trouvant à l'état «Used », les opérateurs « close « et « re-launch » deviennent éligibles (voir Figure V.28).

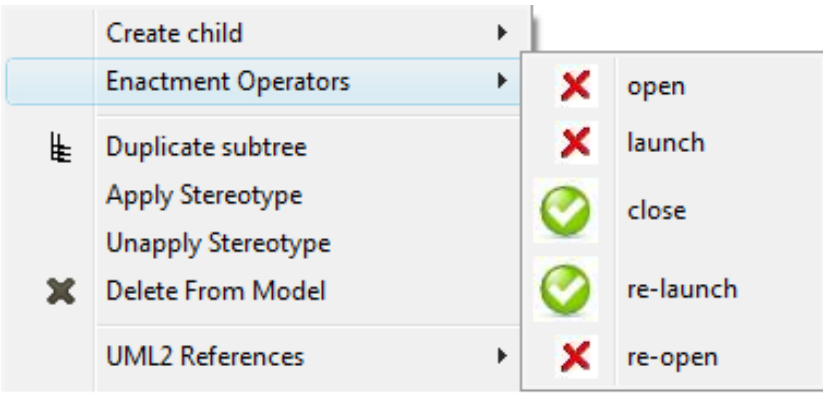


Figure V.28 État des opérateurs de mise en œuvre après le passage de l'outil *ArgoUWE* à l'état « *Used* ».

Nous supposons que Bob exécute l'opérateur « *finishFromInitialVersion* », le modèle produit passe alors à l'état « *FinalVersion* ». A partir de cet état, les opérateurs « *validate*» et « *invalidate* » deviennent éligibles (voir Figure V.29).

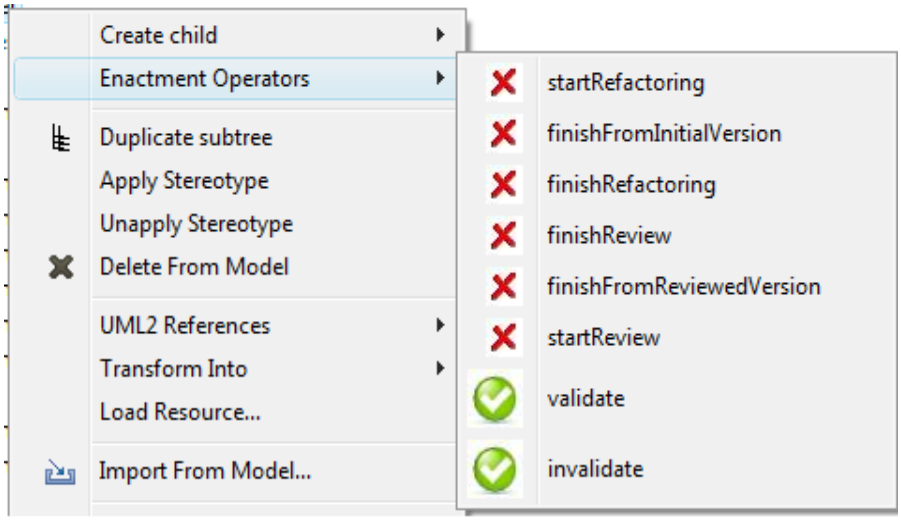


Figure V.29 État des opérateurs après le passage du modèle produit à l'état « *FinalVersion* »

Si Bob exécute l'opérateur «*validate*», le modèle produit (« *Mail Portal Requirements Model* ») passe à l'état « *ValidatedVersion* » (la postcondition de l'activité est donc vérifiée). La postcondition étant vérifiée, l'activité « *Describe Requirements Model* » passe automatiquement à l'état « *Validated* ». Le passage de l'activité à l'état « *Validated* » entraine la fin du cycle de vie du modèle des exigences.

Par la suite, l'outil *ArgoUWE* peut être fermé (application de l'opérateur « *close* », voir Figure V.28) entrainant son passage à l'état « *Closed* ». Le passage de l'activité à l'état « *Validated* » entraine la fin du cycle de vie de l'outil ArgoUWE.

V.4.4.2. Mise en œuvre de la transformation « *Requirements 2 Content* »

La mise en œuvre de la transformation « *Requirements 2 Content* » est basée sur le modèle comportemental d'une transformation (voir section V.4.2.2). La transformation étant déjà adaptée, elle se trouve à l'état « *Executable* », son réalisateur Jean se trouve à l'état « *Assigned* », son modèle source se trouve à l'état « *ValidatedVersion* », son modèle cible à l'état « *Defined* », et l'outil *MediniQVT* utilisé pour sa réalisation à l'état « *Required* ». A ce stade, aucun opérateur de mise en œuvre n'est activable sur la transformation (voir Figure V.30 ci-dessous). Par contre l'opérateur «*open* » est activable sur l'outil *MediniQVT* (voir Figure V.31).

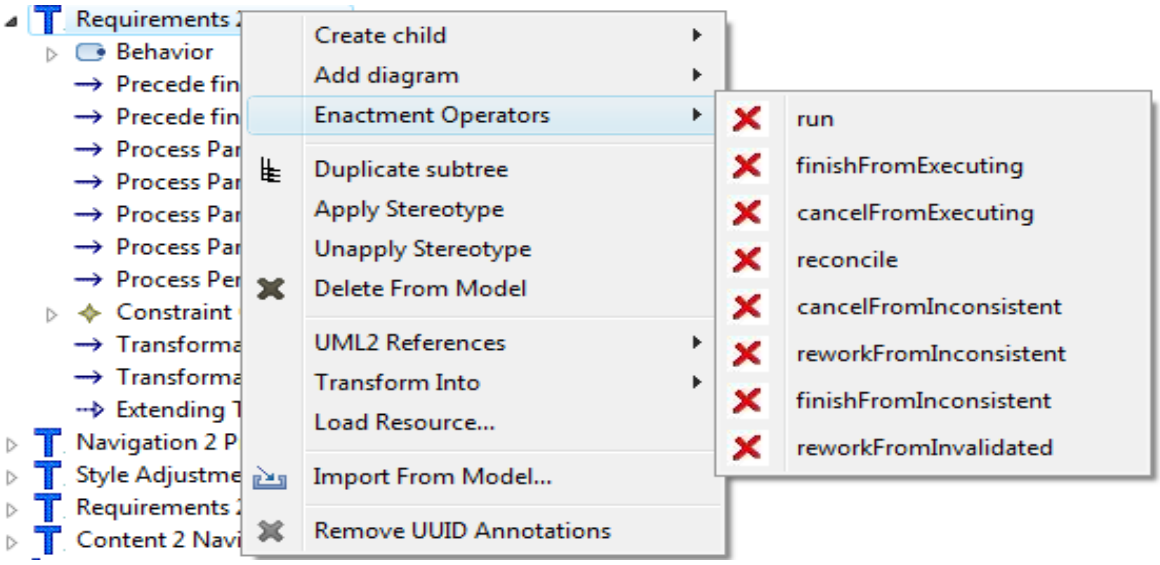


Figure V.30 Opérateurs de mise en œuvre de la transformation « *Requirements 2 Content* »

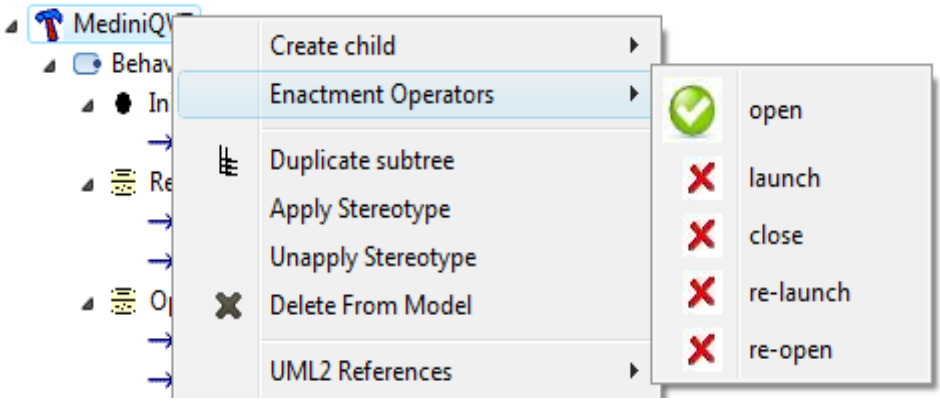


Figure V.31 Opérateurs de mise en œuvre de l'outil *MediniQVT*

La précédence de la transformation « *Requirements 2 Content* » est vérifiée car l'activité « *Describe Requirements Model* » est terminée produisant ainsi le modèle « *Mail Portal Requirements Model* ». Sa précondition est aussi vérifiée car son modèle source (« *Mail Portal Requirements Model* ») se trouve à l'état « *ValidatedVersion*». La précondition et la précédence étant vérifiées, la transformation « *Requirements 2 Content* » passe automatiquement de l'état « *Executable* » à l'état « *Startable* » rendant éligible l'opérateur « *run* » (voir Figure V.32). A ce stade, Jean ouvre l'outil « *MediniQVT* » (opérateur

« *open* ») qui passe à l'état « *Opened* » rendant ainsi éligible l'opérateur « *launch* » (voir Figure V.33).

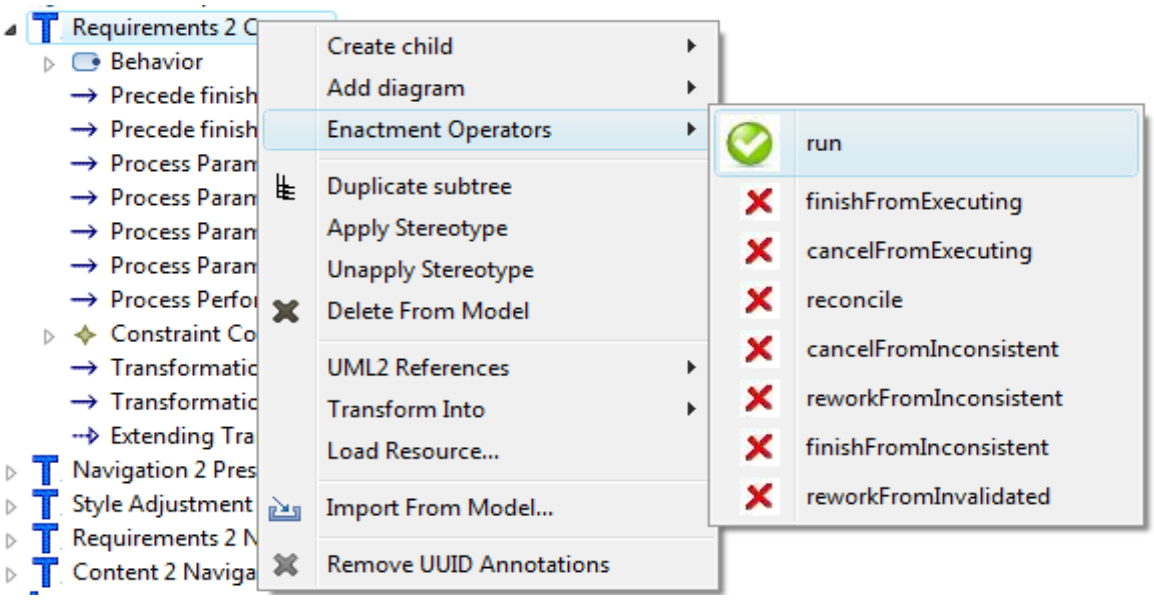


Figure V.32 État des opérateurs de mise en œuvre de « *Requirements 2 Content* » après son passage à l'état « *Startable* »

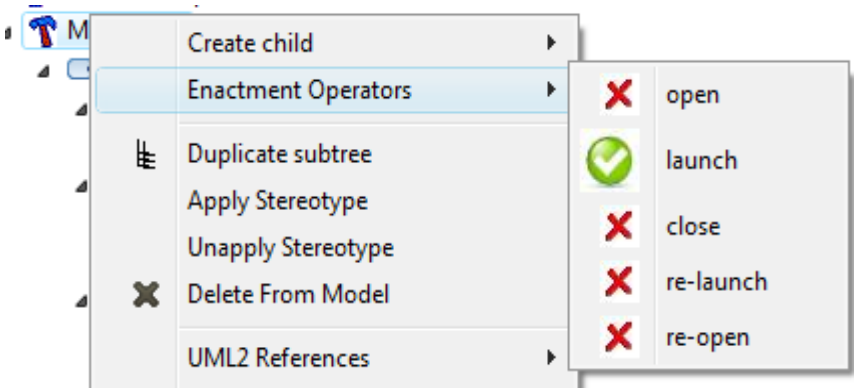


Figure V.33 État des opérateurs de l'outil « *MediniQVT* » après son passage à l'état « *Opened* »

Si l'acteur Jean exécute l'opérateur « *run* » sur la transformation « *Requirements 2 Content* », elle passe à l'état « *Executing* » rendant ainsi éligibles les opérateurs « *finishFromExecuting* » et « *cancelFromExecuting* » (voir Figure V.34). L'exécution de l'opérateur « *run* » sur la transformation déclenche automatiquement l'exécution de l'opérateur « *launch* » sur l'outil *MediniQVT* qui de fait passe de l'état « *Opened* » à l'état « *Running* ». A ce stade l'acteur Jean passe à l'état « *Performing* » conformément à son modèle comportemental (voir chapitre III, Figure III.13).

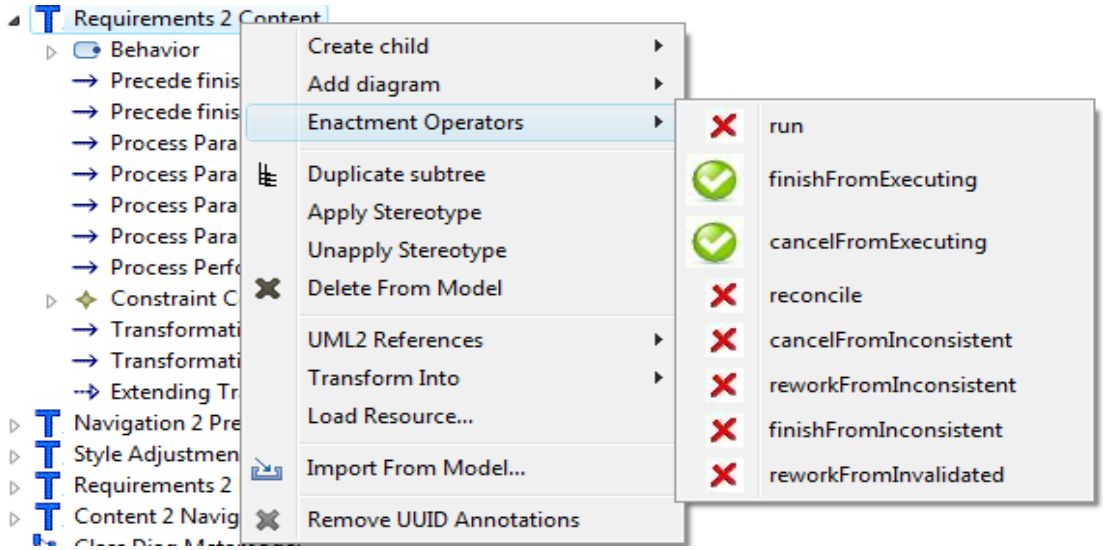


Figure V.34 État des opérateurs de mise en œuvre après l'exécution de l'opérateur « *run* ».

Si Jean exécute l'opérateur « *finishFromEnacting* », la transformation « *Requirements 2 Content* » passe à l'état « *Terminated* » produisant ainsi la version initiale du modèle du contenu de l'application (voir Figure V.35). La transformation se trouvant à l'état à l'état « *Terminated* », l'outil « *MediniQVT* » passe à l'état « *Used* » et l'acteur Jean passe à l'état « *Performed* » conformément à leurs modèles comportementaux.

La Figure V.35 décrit le modèle du contenu produit par la transformation « *Requirements 2 Content* ». Un utilisateur est identifié par son nom, son prénom, sa zone géographique et son e-mail de récupération. Un utilisateur dispose d'un ou de plusieurs comptes. Un compte est identifié par un nom d'utilisateur, un mot de passe, une image du profil du compte, et la signature visible en bas de chaque message envoyé. Un compte peut avoir un ou plusieurs messages électroniques (mail) enregistrés, envoyés, ou reçus. Un message électronique est identifié par son objet, son contenu et le destinataire du message.

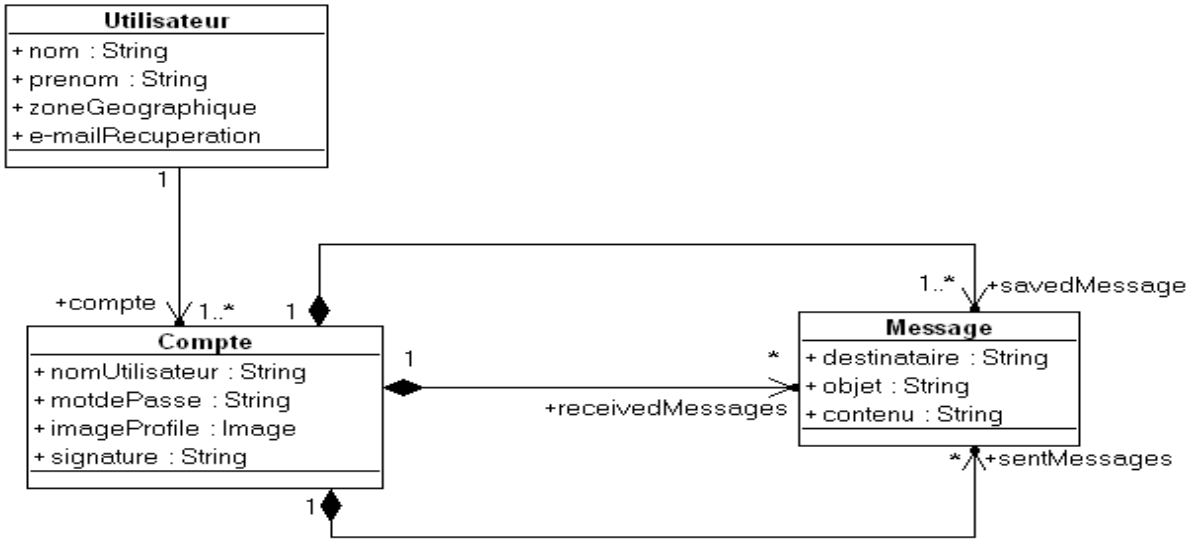


Figure V.35 Modèle de contenu produit par la transformation « *Requirements 2 Content* »

Le modèle de contenu se trouvant à l'état « *InitialVersion* », les opérateurs « *startRefactoring* » et « *finishFromInitialVersion* » deviennent éligibles (voir Figure V.36).

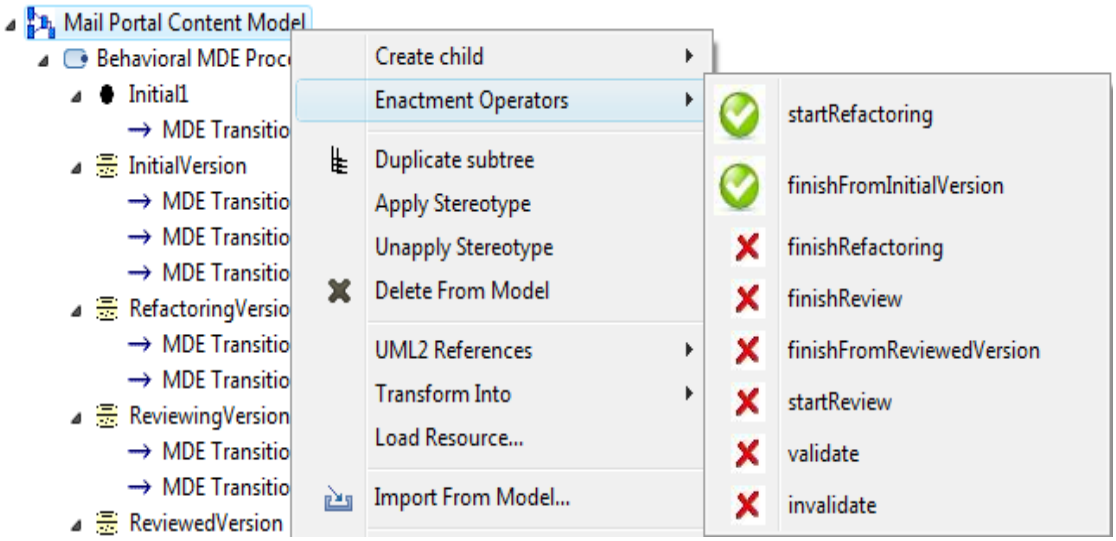


Figure V.36 État des opérateurs du modèle du contenu après son passage à l'état « *InitialVersion* »

L'outil se trouvant dans l'état « *Used* », les opérateurs « *close* » et « *re-launch* » deviennent éligibles (voir Figure V.37).

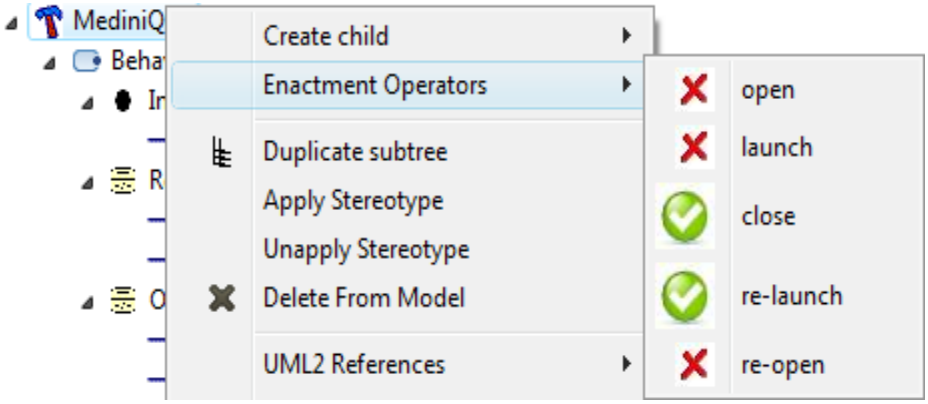


Figure V.37 État des opérateurs après le passage de l'outil à l'état « *Used* ».

Si Jean exécute l'opérateur « *finishFromInitialVersion* » sur le modèle de contenu, celui-ci passe à l'état « *FinalVersion* » rendant ainsi éligibles les opérateurs « *validate* » et « *invalidate* » (voir Figure V.38).

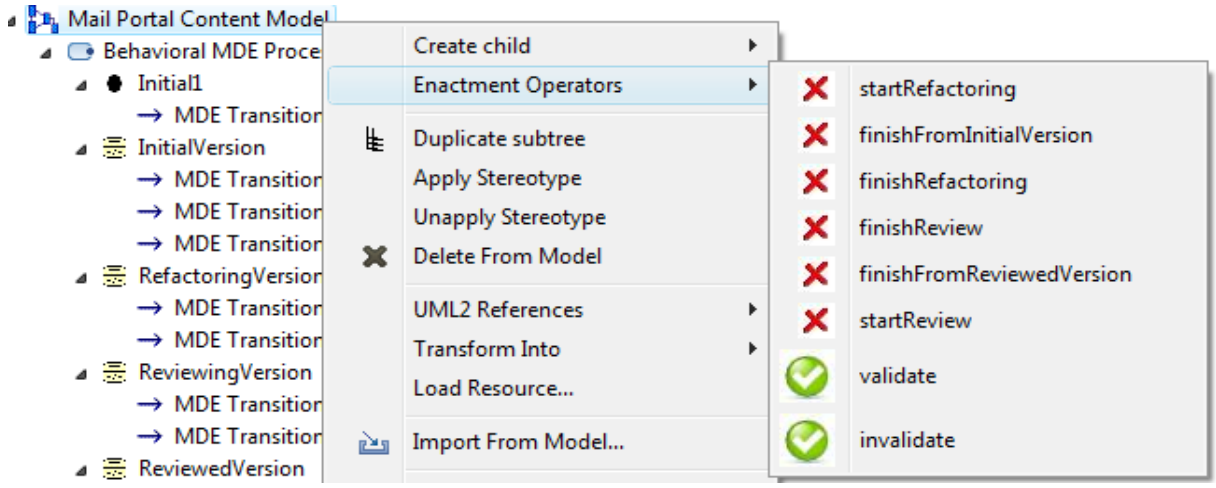


Figure V.38 État des opérateurs du modèle du contenu après son passage à l'état « *FinalVersion* »

Si Jean exécute l'opérateur « *validate*», le modèle de contenu passe à l'état « *ValidatedVersion* » (la postcondition de la transformation est donc vérifiée) et sa transformation passe automatiquement à l'état « *Validated* ».

La transformation se trouvant à l'état « *Validated* », ce qui entraine la fin des cycles de vie de l'outil « *MediniQVT* », de l'acteur « Jean », et du modèle de contenu.

V.4.4.3. Mise en œuvre de la transformation « Content 2 Navigation »

La mise en œuvre de la transformation « *Content 2 Navigation* » est similaire à celle de la transformation « *Requirements 2 Content* ». La Figure V.39 ci-après décrit le modèle de navigation produit par la transformation « *Content 2 Navigation* ». A partir du nœud racine « *Accueil* », deux nœuds sont atteignables (« *Connexion* » et « *Création Compte* »). Après la création d'un compte de messagerie, l'utilisateur peut se connecter. A partir du nœud « *Connexion* », les nœuds « *Rechercher Message* », « *Liste des Messages* », « *Composer Message* », et « *Deconnexion* » sont accessibles. A partir du nœud « *Composer Message* », l'utilisateur peut enregistrer, envoyer ou supprimer un message. La recherche de messages retourne les messages trouvés sous forme d'indexes. L'utilisateur peut supprimer un message dans la boite d'envoi et de réception ou parmi les messages trouvés.

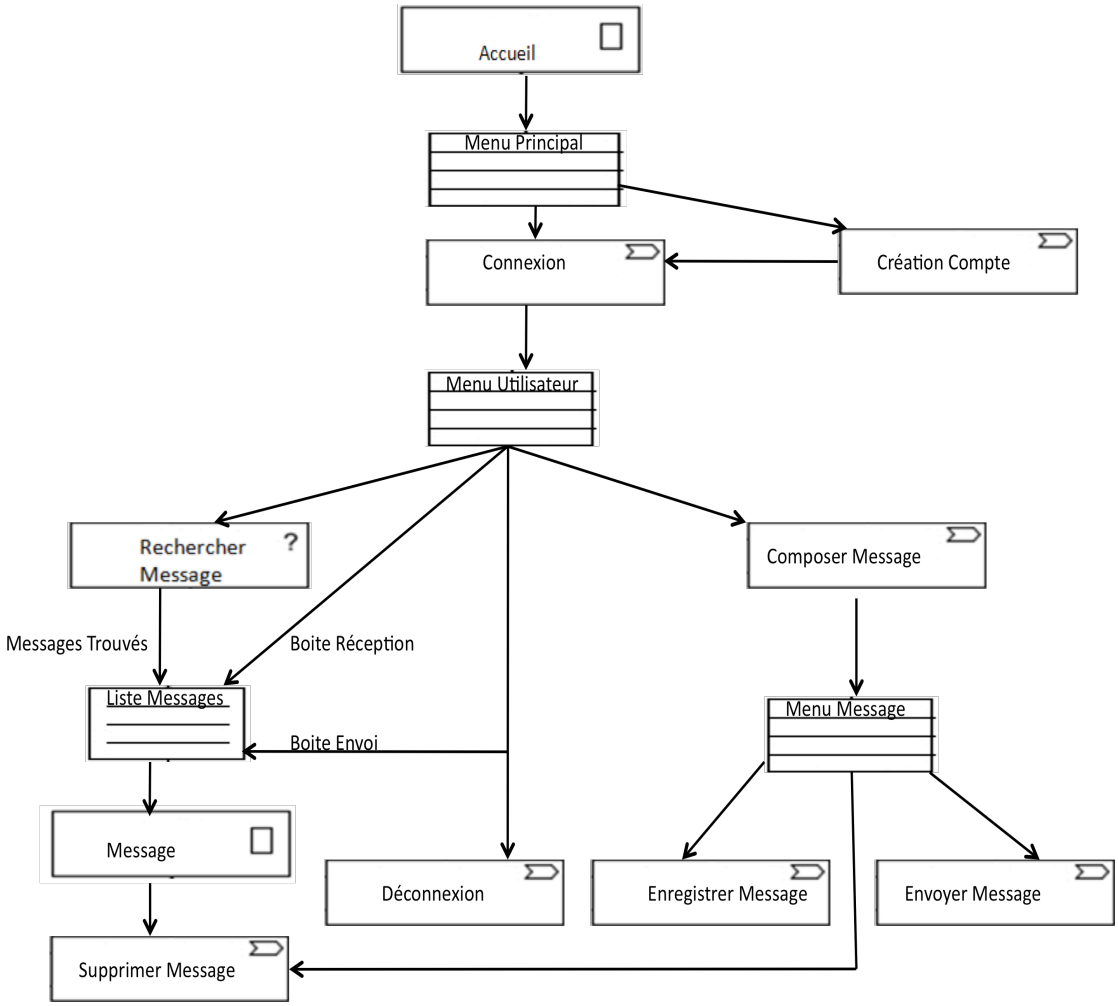


Figure V.39 Modèle de navigation produit par la transformation « *Content 2 Navigation* »

V.4.4.4. Mise en œuvre de la transformation « *Navigation 2 Presentation* »

La mise en œuvre de la transformation « *Navigation 2 Presentation* » est similaire à celle de la transformation « *Requirements 2 Content* ». La Figure V.40 ci-après décrit le modèle de présentation produit par la transformation « *Navigation 2 Presentation* ». La page de présentation « *Gestion Messagerie* » comporte une classe de présentation « *Accueil* », et une classe de présentation « *Utilisateur* » (voir Figure V.41) visible seulement si l'utilisateur est connecté. La page d'accueil comporte deux zones de saisie du nom d'utilisateur et du mot de passe, un bouton de connexion, un bouton de création d'un nouveau compte, un label pour afficher un message d'erreur si le processus de connexion échoue et une classe de présentation « *NouveauCompte* » visible seulement si l'utilisateur crée un nouveau compte. La classe de présentation « *NouveauCompte* » comporte une zone de saisie du nom de famille, du prénom, du nom d'utilisateur, du mot de passe, de la zone géographique, un bouton pour créer le compte, et un label pour afficher un message d'erreur si le processus de création d'un nouveau compte échoue.

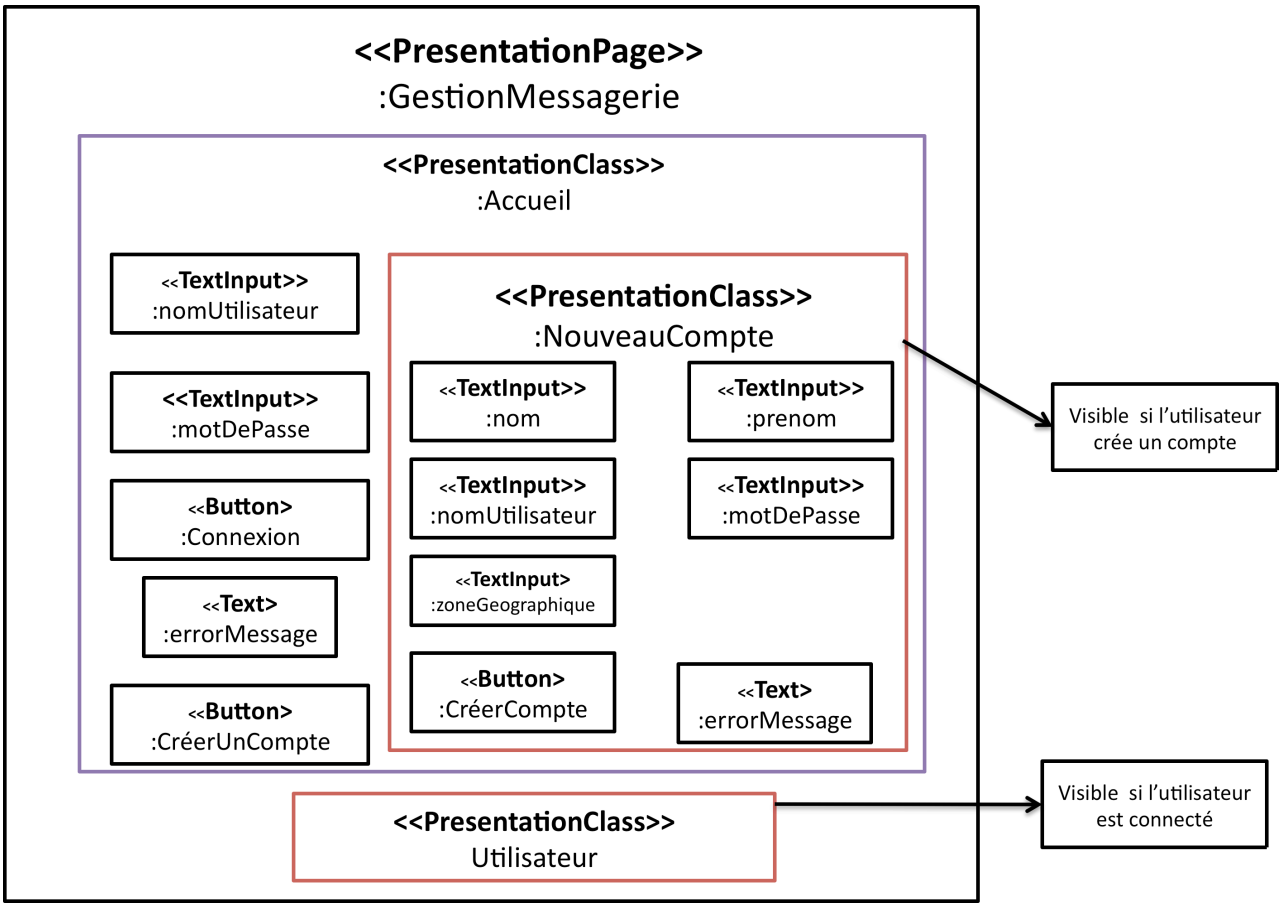


Figure V.40 Modèle de présentation produit par la transformation « *Navigation 2 Presentation* »

La classe de présentation « *Utilisateur* » (voir Figure V.41) comporte un label pour afficher le nom de l'utilisateur qui s'est connecté, un lien pour se déconnecter, un bouton de création d'un nouveau message, trois classes de présentation « *BoiteReception* », « *BoiteEnvoi* », et « *NouveauMessage* ». La classe « *BoiteReception* » est une collection de messages reçus. La classe « *BoiteEnvoi* » est une collection de messages envoyés.

La classe de présentation « *NouveauMessage* » comporte une zone de saisie de l'objet du message, du contenu du message, et du destinataire du message. Elle comporte aussi un bouton d'envoi de message, d'enregistrement de message, et de suppression d'un message, ainsi qu'un lien pour joindre un fichier au message.

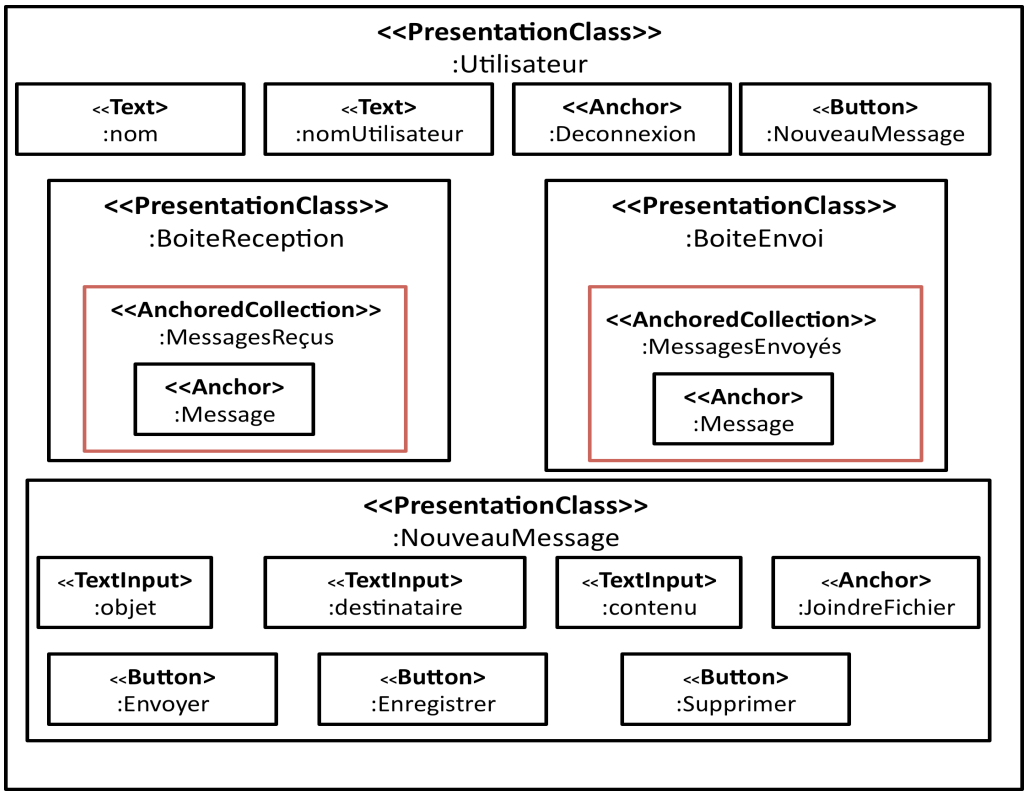


Figure V.41 Modèle de présentation de la classe « Utilisateur »

V.5. Conclusion

L'objectif de ce chapitre était de valider le métamodèle SPEM4MDE à travers une étude de cas. Pour ce faire, nous avons présenté une démarche générale de modélisation et de mise en œuvre de processus IDM et son application au processus UWE. La partie structurelle et comportementale du processus UWE a été décrite grâce au module « *SPEM4MDE Process Editor.* » Pour spécifier la partie comportementale du processus UWE, nous avons réutilisé les comportements génériques définis dans le métamodèle SPEM4MDE. Ces comportements génériques sont des machines à états qui indiquent les états possibles d'un élément du processus UWE (modèle créé ou modifié, transformation, activité, outil, rôle) et les opérateurs de mise en œuvre qui permettent de changer l'état courant d'un élément du processus UWE. Une fois le processus UWE décrit, nous l'avons adapté à projet de développement d'un portail web de messagerie. Grâce au module « *SPEM4MDE Process Enactment Engine* », les opérateurs de mise en œuvre sont traduits sous forme d'entrées de menu graphique. Ces entrées, selon l'état courant de l'élément du processus UWE, distinguent à tout instant les opérateurs éligibles (*nom de l'opérateur précédé d'une marque verte*) et les opérateurs inéligibles (*nom de l'opérateur précédé d'une croix rouge*). Si un développeur exécute un opérateur éligible alors l'élément de processus concerné change d'état. Par contre si un développeur exécute un opérateur inéligible alors le système lui indique les actions à faire pour que cet opérateur devienne éligible. Grâce à l'aide directe consistant à pouvoir visualiser les actions exécutables, les développeurs pourront par la suite utiliser les outils de transformations intégrés sous forme de plug-in dans SPEM4MDE-PSEE pour exécuter les transformations définies avec le module « *SPEM4MDE Process Editor* ».

CHAPITRE VI. CONCLUSION GENERALE ET PERSPECTIVES

VI.1. Rappel de la problématique et des objectifs de la thèse

L'avènement des processus IDM a suscité beaucoup d'intérêt de la part des organisations qui de fait commencent à transformer leur processus de développement traditionnel en un processus IDM [Larrucea et al., 2004]. Plusieurs processus IDM ont ainsi été proposés [Koch, 2006 ; Cong et al., 2010 ; Anwar et al., 2010 ; Koudri, 2010 ; Maciel et al., 2006 ; OpenUP/MDD-website]. Même si ces processus commencent à émerger, nous notons l'absence d'un langage et d'un environnement support reconnus par la communauté IDM pour les décrire puis les mettre en œuvre. Le standard SPEM de l'OMG dédié à la modélisation des processus logiciels et systèmes propose des concepts qui n'arrivent pas à capturer la nature exacte des activités et des artéfacts d'un développement IDM. En effet, la majeure partie des activités dans un développement IDM se résument essentiellement à l'édition et la transformation de modèles. En outre, SPEM ne satisfait pas le critère d'exécutabilité, critère pourtant essentiel pour décrire le comportement des processus IDM. Le standard QVT permet de décrire des transformations exécutables basées sur des règles.

En outre, l'étude des autres LMP (Langage de Modélisation de Processus) proposés dans la littérature montre que seules deux approches tentent d'expliciter les notions de base de l'IDM. Il s'agit des approches de Maciel [Maciel et al., 2009] et de Poress [Porres et al, 2006]. L'approche de Maciel est uniquement centrée sur le langage UML et ses extensions (profil UML). L'approche de Poress n'explicite ni le concept de transformation ni ses métamodèles et modèles associés mais elle met plutôt l'accent sur l'outil de transformation.

L'objectif global des travaux présentés dans cette thèse était de pallier l'insuffisance des LMP proposés dans la littérature en proposant un langage et un environnement support pour décrire et mettre en œuvre les processus IDM.

VI.2. Contribution du travail de thèse

L'étude menée dans le cadre de cette thèse a été conduite selon trois objectif : (1) proposer une extension de SPEM dans laquelle les concepts centraux des processus IDM sont réifiés ; (2) proposer un langage dédié à la modélisation comportementale des processus IDM ; (3) proposer une architecture conceptuelle d'un environnement logiciel d'aide à la modélisation et à la mise en œuvre des processus IDM.

Notre contribution dans le cadre des deux premiéres stratégies a été présentée dans le chapitre III. Elle consistait à définir un langage de modélisation et de mise en œuvre des processus IDM spécifié formellement sous forme d'un métamodèle appelé SPEM4MDE. Le métamodèle SPEM4MDE a été spécifié à travers la réification des concepts centraux de l'IDM (transformation, modèle, métamodèle, outil IDM) et la réutilisation de trois standards de l'OMG : SPEM pour le domaine de la modélisation des processus logiciels et systèmes, QVT pour la description des transformations exécutables à base de règles, et UML pour la description du comportement des processus IDM.

Notre contribution dans la cadre de la troisiéme stratégie a été présentée dans les chapitres IV et V. Elle consistait à élaborer une architecture conceptuelle d'un environnement logiciel fondé sur une démarche en trois étapes.

La première étape a pour but de décrire le modèle structurel et le modèle comportemental d'un processus IDM, conformément au métamodèle SPEM4MDE. Par la suite, ces deux modèles sont validés sur la base des contraintes OCL spécifiées dans le métamodèle SPEM4MDE.

La deuxième étape a pour but de réutiliser et/ou d'adapter les modèles définis dans l'étape précédentes aux spécificités d'un projet de développement déterminé et d'assigner les ressources nécessaires à la mise en œuvre (développeurs, outils, espaces de travail, …).

La troisième et dernière étape de cette démarche est la mise en œuvre du modèle de processus adapté. Elle est réalisée par les développeurs du projet en utilisant les outils IDM mis à leur disposition. Les développeurs sont assistés dans leurs tâches par un environnement de mise en œuvre qui s'appuie sur les modèles comportementaux du processus. Cette étape produit les livrables du projet (code, modèles, documentation, etc.).

Pour valider notre approche, un prototype a été développé sous l'environnement TOPCASED. Ce prototype fournit d'une part un éditeur graphique pour la modélisation structurelle et comportementale des processus IDM et d'autre part un environnement de mise en œuvre s'appuyant sur les modèles comportementaux des processus. Nous avons également appliqué notre approche à une étude de cas significatif: le processus UWE (UML-based Web Engineering), qui est un processus IDM dédié au développement d'applications web.

VI.3. Limites de notre approche et perspectives

L'approche proposée dans cette thèse comporte actuellement certaines limitations. La première limitation est dûe au fait que le métamodèle SPEM4MDE n'étend que la partie *Process Structure* de SPEM 2.0, en laissant de côté les paquetages de SPEM relatifs à la réutilisation d'élements de processus (Method Content, Managed Content, Process With Methods, Method Plug-in). L'extension du seul paquetage *Process Structure* de SPEM 2.0 était motivée par le fait que nous avons cherché à focaliser notre attention d'abord sur les concepts permettant de décrire les aspects structurels et comportementaux d'un processus MDE sans se soucier de la réutilisation. Cependant, l'architecture que nous avons proposée permet d'intégrer facilement les autres paquetages de SPEM dans le métamodèle SPEM4MDE. En effet, il suffit de créer dans SPEM4MDE : un paquetage *MDEMethodContent* qui étendra le paquetage *SPEM::MethodContent* en y intégrant les définitions de transformations ; un paquetage *MDEProcessWithMDEMethods* qui étend les paquetages *SPEM:ProcessWithMethods* et *MDEMethodContent* afin de réutiliser les transformations définies dans

MDEMethodContent et les lier selon un ordre d'exécution ; et enfin un paquetage *MDEMehtodPlugin* qui étend *SPEM::MethodPlugin* et *MDEProcessWithMDEMethods* afin de réutiliser les concepts des plug-in introduits de SPEM.

Une deuxième limitation de notre approche réside dans le fait qu'elle ne prend pas en compte les aspects collaboratifs des processus de développement logiciel. Ces aspects soulèvent de nombreuses questions, loin d'être triviales, par exemple : comment assurer la coordination des développeurs qui travaillent sur un même artéfact ? Comment modéliser les scénarii de collaboration ? Quel scénario de collaboration faut-il appliquer dans telle ou telle situation ? D'autre part, si la gestion de la cohérence des versions est relativement maîtrisé dans le cadre artéfacts textuels (documents, codes sources), il n'en est pas de même pour les modèles du fait de leur caractères non arborescent qui peut conduire à de multiples dépendances dans le cas de modèles de grande taille notamment. Comment pourrait-on alors adpater les techniques classiques de gestion des versions dans le cadre d'un dévéloppement collaboratif IDM ? De fait, cette problématique dépasse le cadre de cette thèse et fait actuellement l'objet d'une thèse au sein de notre équipe, dans le contexte du projet ANR-Galaxy [ANR-GALAXY-website].

Deux autres limitations du travail réalisé dans cette thèse relèvent de l'implémentation. Tout d'abord, dans sa version actuelle, prototype SPEM4MDE-PSEE que nous avons développé contient à la fois le module d'édition de modèles de processus *SPEM4MDE Process Editor* et le module de mise en œuvre *SPEM4MDE Process Enactment Engine*. Par conséquent, c'est dans le même environnement que le concepteur de processus (Process Designer) décrit un modèle processus IDM, le chef de projet adapte ce modèle et le développeur procède à sa mise en œuvre. Cependant, comme présenté dans l'architecture du prototype SPEM4MDE-PSEE, l'idéal serait de décrire un processus IDM avec le module « *SPEM4MDE Process Editor »,* puis de le packager et de le sauvegarder dans le «*MDE Process Repository* ». Par la suite un chef de projet pourra sélectionner un modèle de processus dans ce « *repository* », procéder à son adaptation pour un projet spécifique puis lancer sa mise en œuvre. L'autre limitation réside dans le fait que le prototype n'offre actuellement qu'une version mono-développeur. Il nécessite d'être étendu à une version multi-développeur basée sur une architecture client-serveur.

Enfin, une autre perspective d'extension serait d'offrir un environnement de mise œuvre intègrant la traçabilité, aussi bien des transformations de modèles que de la mise en œuvre des processus. La traçabilité des transformations permettrait de garder les traces d'exécution de chaque règle d'une transformation, de vérifier si une règle a été déjà exécutée ou non, de déboguer une transformation et ses modèles associés et enfin de synchroniser les modèles source et cible d'une transformation. La traçabilité de la mise en œuvre des processus permettrait, quant à elle, d'intégrer différentes techniques de l'ingénierie des processus telles que l'évaluation (Process Assesment), l'amélioration (Process Improvement), la gestion des risques (Risk management), le pilotage et l'aide à la décision au cours du développement (Process Descision Support), etc.

Signalons pour conclure ce mémoire que la validation de notre approche par une étude de cas simplifié (process UWE présenté dans le chapitre Chapitre V) reste insuffisante. Une validation sur un cas d'étude industriel réel d'AIRBUS est prévue dans la dernière année du projet ANR-Galaxy [ANR-GALAXY-website]. C'est à travers cette validation que nous pourrons mesurer réellement les points forts et les points faibles de notre approche.

LISTE DES PUBLICATIONS

Cette section présente par catégorie, les publications que nous avons réalisées pendant cette thèse.

- **Revues nationales**
  - Samba Diaw, Redouane Lbath, Bernard Coulette. État de l'art sur le développement logiciel basé sur les transformations de modèles. Dans : Technique et Science Informatiques, Hermès Science Publications, Numéro spécial Ingénierie Dirigée par les Modèles, Vol. 29, N. 4-5/2010, p. 505-536, juin 2010.

- **Conférences et workshop internationaux avec comité de lecture et actes publiés**
  - Samba Diaw, Redouane Lbath, Bernard Coulette: Specification and Implementation of SPEM4MDE, a metamodel for MDE software processes (regular paper). In: 3rd International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. -. Knowledge Systems Institute, Miami, Floride, USA (2011).
  - Samba Diaw, Redouane Lbath, Vinh Le Thai, Bernard Coulette: SPEM4MDE: a Metamodel for MDE Software Processes Modeling and Enactment. In: 3rd Workshop on Model-Driven Tool & Process Integration - Associated to EC-MFA, pp. 109-121. Fraunhofer, Paris (Juin 2010.)

- **Conférences et workshops nationaux avec comité de lecture et actes publiés**
  - Samba Diaw, Redouane Lbath, Bernard Coulette: Spécification dans une vision IDM des processus de développement logiciel. Dans : Colloque National sur la Recherche en Informatique et ses Applications (CNRIA 2010), Saint-Louis, Sénégal, , Laboratoire d'Analyse Numérique et d'Informatique (LANI), (support électronique), avril 2010.
  - Samba Diaw, Redouane Lbath, Bernard Coulette: SPEM4MDE : un métamodèle basé sur SPEM2 pour la spécification des processus MDE (regular paper). Dans : MAnifestation des Jeunes Chercheurs STIC (MajecStic 2009), Avignon, 16/11/2009-18/11/2009, Laboratoire Informatique d'Avignon, (support électronique), novembre 2009.

- **Conférences avec comité de lecture mais sans actes publiés**
  - Samba Diaw, Redouane Lbath, Bernard Coulette: Feedback on Using TOPCASED Components for Developing a MDE Process-centered Environment: In TOPCASED Days, Toulouse, France, February 2nd–4th, 2011.

BIBLIOGRAPHIE

Action IDM, http://www.actionidm.org.

Acuña S.T., Ferré X.: *Software Process Modelling*. In: Proc. of the World Multiconference on Systemics, Cybernetics and Informatics SCI'01, pp. 237-242, Orlando, USA (2001).

AGG, The Attributed Graph Grammar system (AGG), http://tfs.cs.tu-berlin.de/agg/ (2007).

Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A.: *The Design of a Language for Model Transformations*. In: Software and Systems Modeling, vol. 5, n° 3, pp. 261-288 (2006).

Aizenbud-Resher N., Paige R. F., Rubin, J., Shalam-Gafni Y., Kolovos D. S.: *Operational semantics for traceability*. In ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings.

Alanen M., Porres I.: *Coral: A Metamodel Kernel for Transformation Engines*. In: Proceedings of the Second European Workshop on Model-Driven Architecture (MDA), pp. 165-170. University of kent, Canterbury (2004).

Alizon F., Belaunde M., Dupre G., Nicolas B., Poivre S., Simonin J. : *Les modèles dans l'action à France Télécom avec SmartQVT*. Génie Logiciel, source : Journées Neptune n° 5, p. 35-42 (2008).

Amar B., Leblanc H., Coulette B. : *Traçabilité des transformations et co-évolution de modèles*. Dans : Information - Interaction - Intelligence, Cépaduès Editions, Numéro spécial Réutilisation et Traçabilité des systèmes d'Information, Vol. 10, N. 2, (en ligne), janvier 2011. Accès : http://www.revue-i3.org/

AndroMDA, http://www.andromda.org/(2008).

ANR-GALAXY, http://www.irit.fr/GALAXY

Anwar A, Ebersold S, Nassar M, Coulette B, Kriouile A.: *A Rule-Driven Approach for composition of Viewpoint-oriented Models*. In : JOT (Journal of Object Technology), Mars 2010

Anwar A..: *Formalisation par une approche IDM de la composition de modèles dans le profil VUML*. Thèse de doctorat, Université de Toulouse, 09 Décembre 2009.

Anwar A., Ebersold S., Coulette B., Nassar M., Kriouile A.: *A QVT-based Approach for Model Composition - Application to the VUML Profile*. In: 10th International Conference on Enterprise Information Systems (ICEIS), pp. 360-367. INSTICC Press, Barcelona (2008)

Armenise P., Bandinelli S., Ghezzi C., Morzenti A.: *Survey and assessment the process representing formalisms*. Technical report No. 015, GOODSTEP (1993).

ATLAS group, *KM3: Kernel MetaMetaModel*, Technical Report version 0.3, August 2005, LINA&INRIA.

Bandinelli, S., Fuggetta, A., Ghezzi C., Lavazza L.: *SPADE: An Environment for Software Process Analysis, Design, and Enactment*. In: Finkelstein, A., Kramer, G., Nuseibeh, B. (eds.) Software Process Modelling and technology. Research Studies Press Advanced Software Development Series, pp. 131—151. John Wiley & Sons Inc. (1994).

Bendraou R., Combemale B., Crégut X. Gervais., M.P.: *Definition of an eXecutable SPEM2*.0. In: 14th Asia-Pacific Software Engineering Conference (APSEC), pp. 390-397. IEEE Computer Society, Nagoya, Japan (2007).

Bendraou R., Gervais M.P., Blanc X.: *UML4SPM: A UML 2.0-Based Metamodel for Software Process Mod*eling. In: Briand

L., Williams C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 17-38. Springer, Montego Bay, Jamaica (2005).

Bézivin J., Jouault F.: *Using ATL for Checking Models*. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), Session 1, pp. 1-12.Tallinn, Estonia (2005).

Bézivin J., Breton E.: *Applying the Basic Principles of Model Engineering to the Field of Process Engineering*. European Journal for the Informatics Professional. Vol. 5, pp. 27-33 (2004).

Bézivin J. : *Sur les principes de base de l'ingénierie des mod*èles. *RTSI-L'objet*, vol. 10, n° 4, pp. 145-157 (2004).

Bézivin J. : *La transformation de modèles*. INRIA-ATLAS & Université de Nantes, 2003. Dans: Ecole d'Eté d'Informatique cours #6, CEA EDF INRIA (2003a).

Bezivin J., Dupé G., Jouault F., Pitette G., Rougui E. J.: *First experiments with the atl model transformation language: Transforming xslt into xquery*. In OOPSLA 2003 Workshop, Anaheim, California (2003b). [Online]. Available: http://www.softmetaware.com/oopsla2003/bezivin.pdf

Bézivin J., Gerbé O.: *Towards a precise definition of the OMG/MDA Framework*. In: Proceedings of the 16th IEEE international conference on Automated Software Engineering (ASE), pp. 273. IEEE Press, San Diego, (USA) (2001).

Blanc X. : *MDA en action Ingénierie logicielle guidée par les modèles*. Eyrolles, Paris (2005).

Boehm B.: *A spiral model for software development and enhancement*. ACM SIGSOFT Software Engineering Notes. 11, 14--24 (1988).

Bolcer G.A., Taylor R.N.: *Endeavors: A Process System Integration Infrastructure*. In: 4th. International Conference on Software Proces (ICSP), December 2-6, 1(996).

Burmester S., Giese H., Niere J., Tichy M., Wadsack J., Wagner R., Wendehals L., Zundorf A.: *Tool Integration at the Meta-Model Level within the FUJABA Tool Suite*. In: International Journal on Software Tools for Technology Transfer (STTT), vol. 6, n° 3, pp. 203-218 (2004).

Chou S.C.: *A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language*. Journal of Object Technology.vol. 4, 137--163 (2002).

Combemale B.: *Approche de métamodélisation pour la simulation et la vérification de modèle Application à l'ingénierie des procédés*. Thèse de doctorat, Institut National Polytechnique de Toulouse, 11 Juillet 2008.

Combemale B., Cregut X., Pantel M. : *Transformations de modèles : Principes, Standards et Exemples*. Rapport de recherche, IRIT&CNRS (2007).

Compuware-website, http://www.compuware.com, http://www.optimalj.com/ (OptimalJ).

Cong X., Zhang H., Zhou D., Lu P., Qin L.: *Model-Driven Architecture Approach for Developing E-Learning Platform*. In: Zhang X., Zhong S., Pan Z., Wong K., Yun R. (eds), International Conference on E-learning and Games. LNCS, vol. 6249, pp. 111-122. Springer, Changchun, China (2010).

Coulette B., Crégut X., Dong T. B. T., Tran D. T.: *RHODES, a Process Component Centered Software Engineering Environment*. In: 2nd International Conference on Enterprise Information Systems (ICEIS), pp. 253—260. INSTICC Press, Stafford (2000).

Crégut X. : *Un environnement d'assistance rigoureuse pour la description et l'exécution de procédé de conception - Application à l'approche objet*, Thèse doctorale, Institut National Polytechnique de Toulouse, France, 1998.

Crégut X., Coulette B.: *PBOOL: an Object-Oriented Language for Definition and Reuse of Enactable Processes*. Int. Rev. Software Concepts and Tools, vol 18, n° 2(1997).

Curtis W., Kellner M. I., Over J. *Process Modelling*. Communication of ACM, 35 (9), pp. 75-90 (1992).

Czarnecki K., Helsen S.: Classification des approches de transformation de modèles. Dans : OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)

Derniame J. C., Kaba B.A., Wastell D. (Editors), *Software Process: Principles, Methodology and Technology*. Lecture Notes in Computer Science 1500, Springer (1999).

Diaw S., Lbath R., Coulette B.: *Specification and Implementation of SPEM4MDE, a metamodel for MDE software processes*. In: 3rd International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. -. Knowledge Systems Institute, Miami, Floride, USA (2011).

Diaw S., Thai Le V., Lbath R., Tran H. N., Coulette B.: *Feedback on Using TOPCASED Components for Developing a*

MDE Process-centered Environment. In: TOPCASED Days, (support electronique). Toulouse, France (2011)

Diaw S., Lbath R., Coulette B.: *Etat de l'art sur le développement logiciel basé sur les transformations de modèles*. Technique et Science Informatiques, Vol. 29, N° 4-5, pp. 505-536 (2010a).

Diaw S., Lbath R., Thai Le V., Coulette B.: *SPEM4MDE: a Metamodel for MDE Software Processes Modeling and Enactment*. In: 3rd International Workshop on Model-Driven Tool & Process Integration - Associated to EC-MFA, pp. 109-121. Fraunhofer, Paris, France (2010b).

Diaw S., Lbath R., Coulette B.: *Spécification dans une vision IDM des processus de développement logiciel*. Dans : Colloque National sur la Recherche en Informatique et ses Applications (CNRIA), (support electronique). Saint-Louis, Sénégal (2010c)

Diaw S.., Lbath R., Coulette B. : *SPEM4MDE: un métamodèle basé sur SPEM 2 pour la spécification des processus MDE*. Dans : MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (MajecSTIC), (support electronique). Avignon, France (2009).

Di Nitto E., Lavazza L., Schiavoni, M., Tracanella E., Trombetta M.: *Deriving executable process descriptions from UML*. In: Proceedings of the 24th International Conference on Software Engineering (ICSE), pp. 155—165. ACM, Orlando, Florida (2002).

Dowson M., Fernstrom C.: Towards Requirements for Enactment Mechanisms. In: Third European Workshop on Software Process Techonology (1994).

Eclipse-EMF, http://www.eclipsetotale.com; http://www.eclipse.org/modeling/emf/

Elner R., Al-Hilank S., Bediaga A., Drexler J., Jung M., Kips D., Philippssen M.: *eSPEM – A SPEM Extension for Enactable Behavior Modeling*. In: Kuhne T., Seloc B., Gervais M.P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 116-131. Springer, Paris (2010).

Escalona M. J, Koch N.: *Metamodeling the Requirements of Web System*. In: 2nd International Conference on Web Information System and Technologies (WEBIST), pp. 310-317. INSTICC Press, Setúbal, Potugal (2006).

Farail P., Gaufillet P., Canals A., Camus C. L., Sciamma D., Michel P., Crégut X., Pantel M.: *The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design*. In: Embedded Real Time Software (ERTS), Toulouse (2006).

Farail P., Gaufillet P.: *Topcased – Un environnement de développement OpenSource pour les systèmes embarqués*. Dans Journée de travail NEPTUNE N°2, pp. 16-20. Paris (2005).

Favre J., Estublier J., Blay-Fornarino M. : *L'ingénierie Dirigée par les Modèles. Au-delà du MDA*. Hermes-Lavoisier, Cachan (2006).

Feiler P.H., Humphrey W.S.: *Software Process Development and Enactment: Concepts and Definitions*. In: International Conference on Software Process (ICSP), pp. 28-40. Berlin, Germany (1993).

Finkelstein A., Kramer J., Nuseibeh B. (editors): *Software Process Modelling and Technology*. Research Studies Press (Wiley) (1994).

Fondement F., Silaghi R: *Defining Model Driven Engineering Processes*. In: 3rd UML Workshop in Software Model Engineering (WiSME), Springer, Lisbonne (2004).

France R., Rumpe B.: *Model-driven Development of Complex Software: A Research Roadmap*. In: Proc. of the International Conference on Software Engineering (ICSE), pp. 37-54. IEEE Press, Minneapolis, Minnesota, USA (2007).

Greenfield J., Short K., Cook S., Kent S.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. Wiley, ISBN 0-471-20284-3 (2004).

Humphrey W. S., Kelner M: *Software Modeling: Principles of Entity Process Models*. SEI - Carnegie Mellon University. Pittsburgh, Pennsylvania, (1989).

Humphrey W.S.: *Characterizing the software process: a maturity framework*. IEEE Software. 5, 73 —79 (1988a).

Humphrey W.S.: *The Software Engineering Process: Definition and Scope*. In: Proceedings of the 4th international software process workshop on Representing and enacting of the software process, pp. 82 - 83. ACM Press, (1988b)

Jézéquel J., Fleurey F., Drey Z., Muller P., Pantel M., Maurel C. : *Kermeta : Un langage de méta-modélisation exécutable compatible avec EMOF/ECORE, et son environnement de développement sous Eclipse*. Dans : Actes des

Premières Journées sur l'Ingénierie Dirigée par les Modèles IDM'05, session Démos&Posters, p. 1-4. Université Pierre & Marie Curie, Paris (2005)

Jouault F.: *Loosely coupled traceability for ATL*. In ECMDA Traceability Workshop (ECMDA-TW) 2005 Proceedings.

Jouault J., Kurtev I.: *On the Architectural Alignment of ATL and QVT*. In: Proceedings of the 2006 ACM symposium on applied computing, session Model transformation, pp. 1188-1195. ACM Press, Dijon (2006)

Junkermann G., Peuschel B., Schafer W., Wolf S.: *MERLIN: Supporting cooperation in software development through a knowledge-based environment*. In: A. Finkelstein, J. Kramer, and B. Nuseibeh (editors), Software Process Modelling and Technology, pp. 103 - 129. John Wiley & Sons (1994).

Kaiser G.E., Barghouti N.S., Sokolsky M.H: *Preliminary experience with process modeling in the Marvel software development environment kernel*. In: In Proc. of the 23th Annual Hawaii Int'l Conf. on System Sciences, pp. 131-140. IEEE Press, Hawaii,USA (1990).

Kent S.: *Model Driven Engineering*. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286-298. Springer, Turku, Finland (2002).

Kleppe A., Warmer J., Bast, W.: *MDA EXPLAINED the Model Driven Architecture: Practice and Promise*. Addison-Wesley (2003).

Koch, N.: *Transformations Techniques in the Model-Driven Development Process of UWE*. In: 6th International Conference on Web Engineering (ICWE), Volume 155 Article N° 3. ACM, California (2006).

Koudri A., Champeau J.: *MODAL: A SPEM extension to Improve Co-design Process Models*. In: Münch, J., Yang, Y., Schäfer, W. (eds.) ICSP 2010. LNCS, vol. 6195, pp. 248-259. Springer, Parderborn, Germany (2010).

Koudri A.: *Processus de Conception Conjointe Logiciel Matériel Dirigés par les Modèles*. Thèse de doctorat,Université des Sciences et Technologies de Lille, 13 Juillet 2010.

Kroiß C., Koch N.: *UWE Metamodel and Profile*: User Guide and Reference. LMU, Technical Report (2008).

Larrucea, X., Garcia Diez, A.B., Mansell, J.X.: *Practical Model Driven Development Process*. In: Second European Workshop on Model Driven Architecture (MDA) with emphasis on Methodologies and Transformations, pp. 99-108. Computing Laboratory, University of Kent, Canterbury, UK (2004).

Maciel R.S.P., Silva B.C., Magalhães A.P.F., Rosa N.S.: *An approach to model-driven development process specification*. In: 11th International Conference on Enterprise Information Systems (ICEIS), pp. 27-32. INSTICC Press, Milan (2009).

Maciel R. S. P., Silva B. C., Mascarenhas L. A.: *An Edoc-based Approach for Specific Middleware Services Development*. In: 4th Workshop on MBD of Computer Based System, pp.135-143. IEEE Press, Postdam (2006).

Marschall F., Braun P.: *Model Transformations for the MDA with BOTL*. In: Proceedings of the Workshop on Model-Driven Architecture: Foundations and Applications, pp 25-36. University of Twente, Enschede, the Netherlands (2003).

Melià S., Kraus A., Koch N.: *MDA Transformations Applied to Web Application Development*. In Proc. 5th Int. Conf. on Web Engineering (ICWE), LNCS 3579, Springer, Sydney, Australia (2005).

Mens T., Czarnecki K., Van Gorp P.: *A Taxonomy of Model Transformations*. In: Language Engineering for Model-Driven Software Development (2004).

MiaSoftware-website, Mia – Transformation: e-Source: http://www.mia-software.com/.

Microsoft, Visual Studio.Net, www.microsoft.com/visualstudio, DSL Tools, http://msdn. microsoft.com/vstudio/dsltools/default.aspx (2005).

Microsoft-website, http://www.microsoft.com/ (Microsoft DSL Tools).

MIC-website, MIC: http://www.isis.vanderbilt.edu/research/MIC.

Medini QVT, http://projects.ikv.de/qvt/.

MODELS, Electronic source: http://www.modelsconference.org/

Modelio, http://www.modeliosoft.com/.

ModTransf, Electronic Source: http://www.lifl.fr/west/modtransf/.

Montangero C., Derniame J.C., Kaba B.A., Warboys B.: *The software process: Modelling and technology*. In: Derniame J.C., Kaba B. A., Wastell D. G. (eds.) Software Process: Principles, Methodology and Technology. LNCS, vol. 1500, pp. 1--14. Springer, Berlin (1999).

Muller P. A., Fleurey F., Jézéquel J. M.: *Weaving Executability into Object-Oriented Meta-Languages*. In: Briand L., Williams C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264-278. Springer, Montego Bay, Jamaica (2005).

Muller P. A. : *De la modélisation objet des logiciels à la métamodélisation des langages informatiques*. HDR, Université de Rennes 1, Hawaii 20 Novembre 2006.

Nassar M. : *Analyse/conception par objets et points de vue : le profil VUML*. Thèse de doctorat, Institut National Polytechnique de Toulouse, septembre 2005.

OAW, http://www.openarchitectureware.org (2008).

OPEN UP Component-MDD at: http://www.eclipse.org/epf/openup_component/mdd.php.

Osterweil L.: *Software Processes are Software Too*. In: Proceedings of the 9th International Conference on Software Engineering, Monterey, USA (1987).

OMG-DI 2.0, http://www.omg.org/cgi-bin/doc?ptc/2003-09-01 (2003).

OMG-EDOC UML Profile for Enterprise Distributed Object Computing Specification. (2002).

OMG-MDA, http://www.omg.org/mda/executive_overview.htm.

OMG-MOF 2.0, http://www.omg.org/cgi-bin/doc?formal/06-01-01 (2006).

OMG-OCL 2.0, http://www.omg.org/spec/OCL/2.0/ (2005).

OMG-QVT, RFP, http://www.omg.org/docs/ad/02-04-10 (2002).

OMG-QVT, Final Adopted specification, http://www.omg.org/docs/ptc/05-11-01 (2005).

OMG-QVT, version 1.0, http://www.omg.org/spec/QVT/1.0/PDF/ (2008).

OMG-SPEM 2.0, http://www.omg.org/spec/SPEM/2.0 (2008).

OMG-UML 2.2, http://www.omg.org/spec/UML/2.2/ (2009).

OMG-XMI 2.1, http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm, Décembre 2007.

PathfinderSolutions, source: http://www.pathfindermda.com.

Paulk M. C., Weber C. V., Chrissis M. B.: *The Capability Maturity for Software*. Software Engineering Institute, USA (1994.)

Porres I., Valiente M. C.: *Process Definition and Project Tracking in Model Driven Engineering*. In: Münch, J., Vierimaa, M. (eds.) PROFES 2006. LNCS, vol. 4034, pp. 127-141. Springer, Amsterdam (2006)

QVTEclipse, Electronic Source: http://qvtp.org/downloads/qvtp-eclipse/.

Ribó J.M., Franch X.: *PROMENADE, a PML intended to enhance standardization, expressiveness and modularity in Software Process Modelling*. In: Research report LSI-00-34-R, Dept. LSI of Politechnical University of Catalonia (2000).

Rios E., Bozheva T., Bediaga A., Guilloreau N.: *MDD Maturity Model: A Roadmap for Introducing Model-Driven Development*. In Rensink A., Warmer J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 78-89. Springer, Bilbao (2006)

Royce W. W.: *Managing the development of large software systems: concepts and techniques*. In: 9th international conference on Software Engineering, pp. 328—338. IEEE Press, (1987).

RSM-website, http://www.rpi.edu/dept/cis/software/rationalrose/Modeler6/disk1/readme/fr_FR/readme.html

Softeam-website, http://www.softeam.fr/ (Softeam MDA Modeler).

Softeam, support de formation Objecteering 6/MDA Modeler version 2.0, Paris (2008).

Software Engineering Institute (SEI): *CMMI for Development*, Version 1.3 CMMI-DEV, November 2010, available at: http://www.sei.cmu.edu/library/abstracts/reports/10tr033.cfm

Soley R., OMG Staff Strategy Group: *Model Driven Architecture*. White paper, Object Management Group (2000).

Sriplakich P. : *ModelBus : Un environnement réparti et ouvert pour l'ingénierie des modèles*. Thèse de doctorat, Université de

Paris VI, 05 Septembre 2007.

Stahl T., Volter M.: *The Model-Driven Software Development*. Translation copyright by John Wiley & Sons, Ltd (2006).

Sutton S.M, Lerner B. S., Osterweil L.: *Experience Using the JIL Process Programming Language to Specify Design Processes*, Technical Report 97-68, Department of Computer Science, University of Massachusetts at Amherst, September 6, 1997.

Sutton Jr S.M., Heimbigner D., Osterweil L.: *APPL/A: A Language for Software Process Programming*. ACM Transaction on Software Engineering Methodology (TOSEM). Vol. 4, pp. 221--286 (1995).

SysML-website, http://www.sysml.org/

SYSTEM@TIC PARIS REGION, http://www.usine-logicielle.org/

Sztipanovits J., Karsai G., Biegl C., Bapty T., Ldeczi K., Misra A.: *MULTIGRAPH: architecture for model-integrated computing*. In: Proceedings of the 1st International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 361-368. IEEE Press, Washington DC (1995).

Thai Le Vinh: *Un méta-modèle pour les processus IDM : implémentation sous l'environnement TOPCASED*. Master 2 Recherche « Informatique et Télécommunications », Parcours « Systèmes Informatiques et Génie Logiciel ». Université Toulouse II-Le Mirail, 24 Septembre 2010.

Tolvanen J. P., Rossi M.: *MetaEdit+: defining and using domain-specific modeling languages and code generators*. In: Proceedings of the conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 92-93. ACM Press, Anaheim, California(2003).

TOPCASED-WP5 : *Guide méthodologique pour les transformations de modèles*. Rapport de recherche, version 0.1, IRIT/MACAO 18, Novembre 2008.

TOPCASED-website, http://www.topcased.org

Triskell, Kermeta, IRISA, Rennes, http://www.irisa.fr/triskell; http://www.kermeta.org (2005).

UMT-QVT, UMT, Electronic Source: http://umt-qvt.sourceforge.net/.

Vanhooff B., Berbers Y.: *Supporting modular transformation units with precise transformation traceability metadata*. In ECMDA Traceability Workshop (ECMDA- TW) 2005 Proceedings

W3C-SVG, http://www.w3.org/TR/SVG/ (2003).

W3C-XLST, http://xmlfr.org/w3c/TR/xslt/ (1999).

W3C-Xquery, http://www.w3.org/TR/xquery/ (2007).

Xactium-website, XMF-Mosaic, http://www.xactium.com.

ANNEXE A : LE PAQUETAGE PROCESS STRUCTURE DE SPEM 2.0

Le paquetage SPEM 2.0 *Process Structure* (voir FIG. 1) contient les concepts de base pour définir un processus de développement. Un processus de développement décrit comment les projets basés sur celui-ci doivent être exécutés. Dans SPEM, un processus de développement est une collaboration entre des entités actives et abstraites (les rôles) qui réalisent des opérations (activités) pour produire des entités concrètes et réelles (les produits). SPEM 2.0 *Process Structure* « merge » le paquetage *Core* de SPEM 2.0. Nous donnons ci-dessous la description complète des concepts du paquetage *Process Structure* de SPEM 2.0. Ces descriptions sont tirées de la spécification de SPEM 2. 0 [OMG-SPEM 2.0, 2008].

a) WorkDefinition

Description

Le concept *WorkDefinition* est une méta-classe abstraite qui généralise toute définition d'un travail dans SPEM. *WorkDefinition* est lié aux concepts *WorkDefinitionParameter* et *Constraint*. Les contraintes associées à un *WorkDefinition* définissent sa précondition et/ou sa postcondition. La précondition est une condition à vérifier avant de commencer la mise en œuvre du *WorkDefinition*, tandis que la postcondition spécifie une condition à vérifier pour terminer sa mise en oeuvre.

Généralisation

- Classifier (from UML 2.2 Infrastructure::Core ::Construct)

Associations

- precondition : Constraint [*]. Cette composition spécifie la ou les préconditions d'un *WorkDefinition*.
- postcondition : Constraint [*]. Cette composition spécifie la ou les postconditions d'un *WorkDefinition*.
- ownedParameter : WorkDefinitionParameter [*]. Cette composition définit un ensemble ordonné de paramètres qui spécifient les entrées et les sorties d'un *WorkDefinition*.

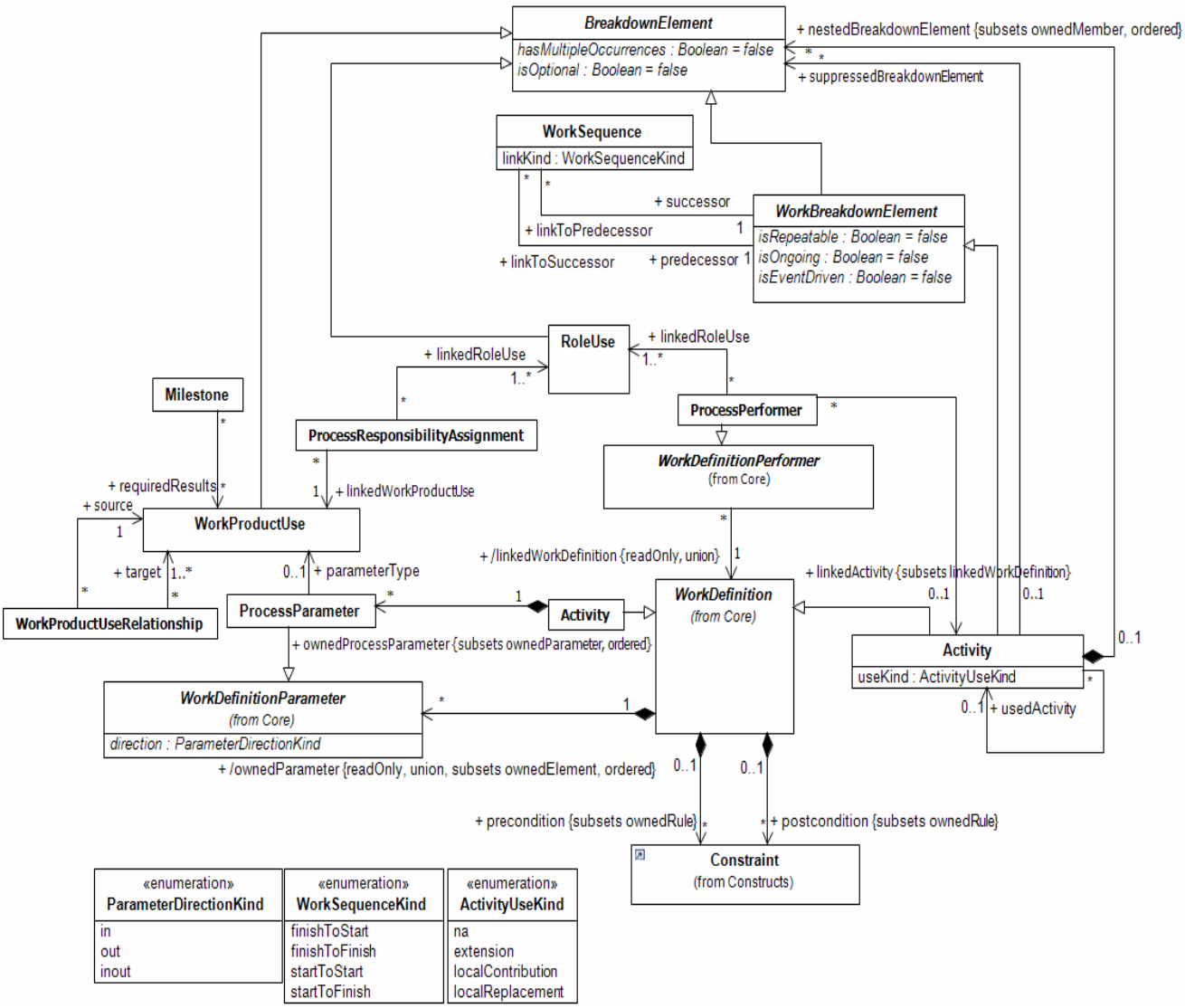


FIG. 1 Paquetage SPEM 2.0 Process Structure [OMG-SPEM 2.0, 2008]

b) WorkDefinitionParameter

Description

Ce concept définit les paramètres d'un *WorkDefinition*. Ces paramètres sont des références qui permettent de spécifier les entrées et les sorties d'un *WorkDefinition*.

Généralisation

- Classifier (from UML 2.2 Infrastructure::Core ::Construct)

Attribut

- direction : ParemeterDirectionKind. Cet attribut spécifie la direction d'un paramètre qui peut être de type in (entrée), out (sortie) ou inout (entrée-sortie).

c) WorkDefinitionPerformer

Description

WorkDefinitionPerformer est un concept qui spécifie le réalisateur (i.e. performer) d'un *WorkDefinition*. Les spécialisations de *WorkDefinitionPerformer* introduiront les différents types de réalisateurs.

Généralisation

- Classifier (from UML 2.2 Infrastructure::Core ::Construct)

Association

- linkedWorkDefinition : WorkDefinition [1]. Cette association spécifie le *WorkDefinition* du réalisateur.

d) BreakdownElement

Description

BreakdownElement est une généralisation abstraite des artefacts spécfiés par *WorkProductUse*, des rôles spécifiées par *RoleUse* et des activités spécifiées par *Activity*. Il définit un ensemble de propriétés disponibles pour ses sous-classes. Les sous-classes de *BreakdownElement* (*WorkBreakdownElement*, *RoleUse*, *WorkProductUse*) sont contenues dans une activité via la composition « nestedBreakdownElement » entre *Activity* et *BreakdownElement*.

Généralisation

- ProcessElement

Attributs

- hasMultipleOccurrences: Boolean=false. Si la propriété est vraie alors *BreakdownElement* aura plusieurs occurrences (i.e. instances). Les occurrences d'un *BreakdownElement* dont l'attribut *hasMultipleOccurrence* peuvent être exécutées en parallèle pendant la mise en œuvre.
- isOptional: Boolean = false. Cet attribut est vrai, si *BreakdownElement* est obligatoire au moment de la mise en œuvre, faux dans le cas contraire.

e) WorkBreakdownElement

Description

WorkBreakdownElement est une spécialisation *BreakdownElement* qui fournit des propriétés spécifiques pour les instances de « «*BreakdownElement »* qui représentent un travail.

Généralisation

- BreakdownElement

Attributs

- isRepeatable: Boolean = false. Cette propriété spécifie si un travail est répétitif ou non (par exemple une itération). Les occurrences d'un *WorkBreakdownElement* dont

l'attribut *isRepeatable* est vrai seront exécutées séquentiellement, ce qui induit une dépendance contrairement à l'attribut *hasMultipleOccurrences* défini dans *BreakdownElement*.

- isOngoing: Boolean = false. L'attribut est vrai pour les instances de *WorkBreakdownElement* qui décrivent un travail qui n'a pas une durée fixe ou un état fini. Le travail d'un administrateur dans un projet de développement afin de maintenir un système dans une certaine cohérence en est une illustration. La durée de ce travail peut varier suivant le degré de l'incident du système.
- isEventDriven: Boolean = false. Cette propriété spécifie le fait qu'un travail démarre non pas parce qu'il a été prévu dans le processus, mais plutôt parce qu'un événement déclenche son démarrage. De tels événements sont les exceptions ou les situations dans un projet qui requièrent qu'un travail soit exécuté à la suite de l'apparition de l'événement. Ces événements ne sont pas modélisés dans SPEM.

Associations

- linkToPredecessor: WorkSequence [*]. Cette association connecte une instance de WorkBreakdownElement à ses prédécesseurs.
- linkToSuccessor: WorkSequence [*]. Cette association connecte une instance de WorkBreakdownElement à ses successeurs.

f) WorkSequence

Description

WorkSequence est un *BreakdownElement* qui représente une relation de précédence entre deux instances de *WorkBreakdownElement*, dans laquelle l'une des instances dépend du démarrage ou de la fin de l'autre pour démarrer ou pour finir.

Généralisation

- BreakdownElement

Attribut

- linkKind: WorkSequenceKind. Cette propriété exprime le type de séquencement par une assignation d'une valeur à partir de l'énumération *WorkSequenceKind*.

Associations

- successor: WorkBreakdownElement [1]. Cette association connecte une instance de *WorkSequence* à une instance de *WorkBreakdownElement* (le successeur).
- predecessor: WorkBreakdownElement [1]. Cette association connecte une instance de *WorkSequence* à une instance de *WorkBreakdownElement* (le prédécesseur).

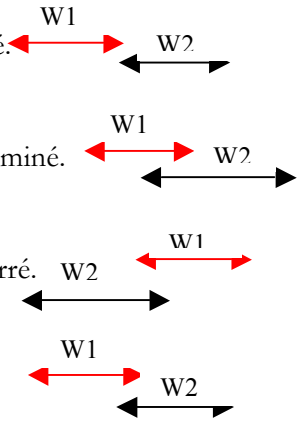
g) WorkSequenceKind

Description

WorkSequenceKind est une énumération qui définit les différents types de précédences entre deux instances de *WorkBreakdownElement* : W1 le prédécesseur (flèche rouge) et W2 le successeur (flèche noire).

Valeurs de l'énumération (Enumeration Literals)

- finishToStart. W2 ne peut démarrer que si W1 est terminé.
- finishToFinish. W2 ne peut être terminé que si W1 est terminé.
- startToFinish. W2 ne peut être terminé que si W1 a démarré.
- startToStart. W2 ne peut démarrer que si W1 a démarré.



h) Activity

Description

Le concept *Activity* définit un travail assignable à des rôles spécifiés par *RolUse* et lié à des produits (ses entrée et/ou sorties) spécifiés par *WorkProducUse*. Une activité est liée à ses produits via les instances de *ProcessParameter* et à ses rôles via les instances de *ProcessPerformer*. La relation de composition entre *Activity* et *BreakdownElement* définit les éléments de processus qui contenus dans une activité.

Activity spécialise les concepts *WorkDefinition* et *WorkBreakdownElement*. La spécialisation de *WorkDefinition* permet de définir la précondition et la postcondition d'une activité. La spécialisation de *WorkBreakdownElement* permet de définir les relations de précédence entre activités.

Généralisation

- WorkBreakdownElement, WorkDefinition

Attribut

- useKind: ActivityUseKind = na . Cette propriété exprime les types de réutilisation entre deux activités, spécifiés par l'énumération ActivityUseKind.

Associations

- nestedBreakdownElement: BreakdownElement [*]. Cette composition exprime les éléments contenus dans une activité.
- suppressedBreakdownElement: BreakdownElement [*]. Cette association permet de cacher certains *BreakdownElement* lors de l'interprétation de la structure d'un processus. Elle est utilisée avec l'association *usedActivity* décrite ci-dessous. L'activité qui réutilise peut définir localement ses propres éléments qui référencent certains éléments de l'activité de base (activité utilisée) afin d'exprimer la suppression.
- ownedProcessParameter : ProcessParameter [*]. Cette composition spécifie un ensemble ordonné de paramètres d'une activité. Cet ensemble est un sous-ensemble

de « ownedParameter » qui spécifie les paramètres d'un WorkDefinition.

- usedActivity: Activity [0..1]. Cette association exprime une relation de réutilisation entre deux activités selon la sémantique définit par l'énumération *ActivityUseKind*.

i) ActivityUseKind

Description

Cette énumération définit la sémantique de la réutilisation entre deux activités. La réutilisation entre deux activités dans SPEM, définit la capacité de réutiliser la structure définit pour une activité A1 via ses « nestedBreakdownElement » dans une deuxième activité A2 sans pour autant avoir besoin de copier physiquement cette structure.

Valeurs de l'énumération (Enumeration Literals)

- na. C'est la valeur par défaut pour les activités qui n'instancient pas l'association *usedActivity*.
- extension. L'extension fournit un mécanisme de réutilisation dynamique de toute la structure d'une activité dans d'autres activités. Un exemple typique de l'extension est l'application des patrons de processus (activité de base) dans un autre processus (activité qui réutilise). L'application de ces patrons de processus consiste à modéliser la structure d'un processus en réutilisant ces patrons via l'association *usedActivity*.
- localContribution. Elle permet à un concepteur de processus de modifier la structure initiale d'une activité A1 en y apportant sa contribution (par exemple en y ajoutant une sous-activité A1.3 en supposant que les activités A1.1 et A1.2 composent l'activité A1).
- localReplacement. Elle permet à un concepteur de processus de remplacer la structure initiale d'une activité A1 par une autre structure définit par une activité A2 tout en gardant l'emplacement et les relations de A1 dans son processus d'origine.

La FIG. 2 illustre les types de réutilisation entre deux activités. La relation d'extension est définie entre les activités *Process* 1 et *Process* 2, spécifiant que la structure de *Process* 2 sera copiée dynamiquement dans *Process 1*.

La relation « *suppressed* » définit dans *Process 2* une activité *Activity 2.1* qui supprime l'activité *Activity 1.1* dans *Process 1*.

La relation « *local contribution* » définit dans *Process 2* une activité *Activity 2.2* qui contribue localement à l'activité *Activity 1.2* en y ajoutant une sous-activité *Activity 2.2.1*.

Finalement la relation « *local replacement* » définit dans *Process 2* une activité *Activity 2.3* qui se substitue à l'activité *Activity 1.3* de *Process 1*. Cette substitution consiste à remplacer la structure de l'activité *Activity 1.3* par celle de l'activité *Activity 2.3* tout en gardant la précédence qu'avait l'activité *Activity 1.3* dans *Process 1*.

Le résultat de l'interprétation de ces quatre relations est donné à droite de la FIG. 2.

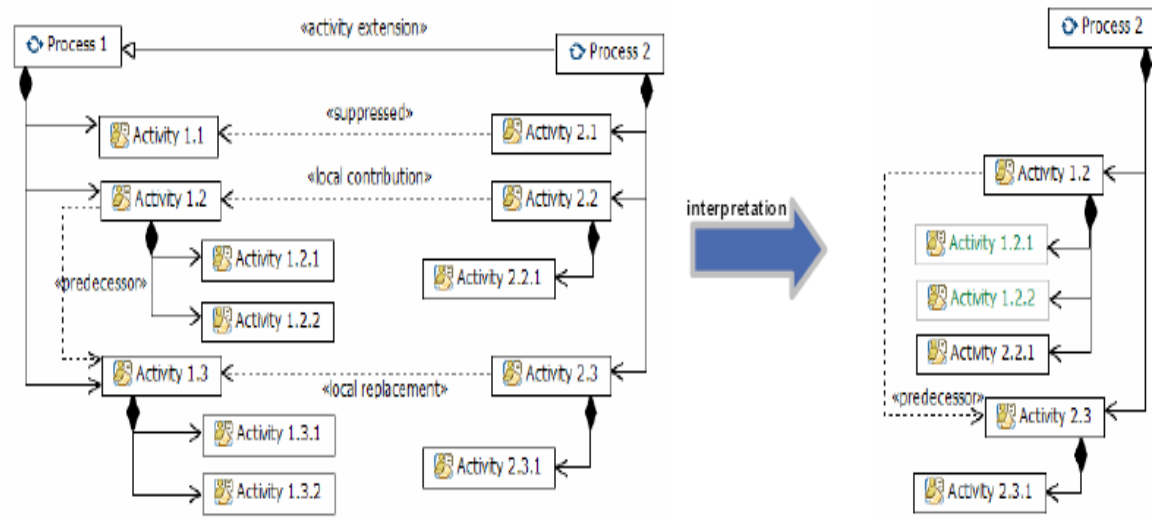


FIG. 2 Illustration du mécanisme de réutilisation [OMG-SPEM 2.0, 2008]

j) ProcessParameter

Description

ProcessParameter est un *WorkDefinitionParameter* qui spécifie les relations entre une activité et ses produits. Une instance de *ProcessParameter* connecte une instance du concept *Activity* à zéro ou une instance de *WorkProductUse*.

Généralisation

- WorkDefinitionParameter

Association

- parameterType : WorkProductUse [0..1]. Cette association connecte une instance de *ProcessParameter* à zéro ou une instance de *WorkProductUse*.

k) WorkProductUse

Description

WorkProductUse représente soit un artéfact consommé, modifié ou produit par une activité , soit un artéfact qui est contenu dans une activité via la composition « *nestedBreakdownElement* ». Si *WorkProductUse* est contenu dans une activité A1, alors il sera utilisé par une sous-activité de A1 comme entrée et/ou sortie. Par conséquent, il se sera lié à un rôle contenu dans A1 qui en est sera le responsable via *ProcessResponsibilityAssignment*.

Généralisation

- BreakdownElement

l) ProcessPerfomer

Description

ProcessPerformer est un *WorkDefinitionPerformer* qui spécifie les relations entre une activité et ses rôles. Une instance de *ProcessPerfomer* connecte au plus une instance du concept *Activity* à une ou plusieurs instances de *RoleUse*.

Généralisation

- WorkDefinitionPerformer

Associations

- linkedRoleUse : RoleUse [1..*]. Cette association connecte une instance de *ProcessPerformer* à une ou plusieurs instances de *RoleUse*.
- linkedActivity : Activity [0..1]. Cette association connecte une instance de *ProcessPerformer* à zéro ou une activité. L'association *linkedActivity* est un sous-ensemble de l'association *linkedWorkDefinition* liée à *WorkDefinition*.

m) RoleUse

Description

RoleUse représente soit le rôle joué par une personne physique qui réalise une activité, soit un rôle qui est contenu dans une activité via la composition « *nestedBreakdownElement* ». Si *RoleUse* est contenu dans une activité A1, alors il sera utilisé par une sous-activité de A1 comme rôle via *ProcessPerformer*. Par conséquent, il sera le responsable des produits de la sous-activité de A1 qui l'utilise comme rôle.

Généralisation

- BreakdownElement

n) ProcessResponsibilityAssignment

Description

ProcessResponsibilityAssignment est un *BreakdownElement* qui représente une relation entre une instance de *WorkProductUse* et une ou plusieurs instances de *RoleUse*. C'est une association qui définit les responsables des produits élaborés au cours d'un développement. Ces responsables sont spécifiés par *RoleUse*.

Généralisation

- BreakdownElement

Associations

- linkedRoleUse : RoleUse [1..*]. Cette association connecte une instance de *ProcessResponsibilityAssignment* à une ou plusieurs instances de *RoleUse*.
- linkedWorkProductUse : WorkProductUse [1]. Cette association connecte une instance de *ProcessResponsibilityAssignment* à une instance de *WorkProductUse*.

o) WorkProductUseRelationship

Description

WorkProductUseRelationship est un concept général qui permet de spécifier les relations possibles entre les produits d'un développement. Le mécanisme « Kind » de SPEM permettra de spécifier concrètement les relations entre les produits d'un développement.

Généralisation

- BreakdownElement

Associations

- source: WorkProductUse [1]. Cette association connecte une instance de *WorkProductUseRelationship* à une instance de *WorkProductUse* (la source).
- target: WorkProducUse [1]. Cette association connecte une instance de *WorkProductUseRelationship* à une instance de *WorkProductUse* (la cible).

p) Milestone

Description

Milestone représente un événement significatif dans un projet de développement. Il définit à la fois un événement et le moment où celui-ci doit se produire dans le projet. Un prise de décision majeure au cours d'un projet, la fin d'un livrable, la fin d'une phase de développement, sont des illustrations de *Milestone*. Il est modélisé comme un *WorkBreakdownElement,* il peut donc être contenu dans une activité et peut avoir des précédences.

Généralisation

- WorkBreakdownElement

Association

- requiredResults: WorkProductUse [0..*]. Cette association connecte une instance de *Milestone* et une ou plusieurs instances de *WorkProductUse*. Ces instances de *WorkProductUse* seront les produits utilisés dans un *Milestone*.

ANNEXE B : LE PAQUETAGE BEHAVIORSTATEMACHINES D'UML

Le paquetage UML 2.2 BehaviorSateMachines (voir FIG. 3) décrit les concepts qui spécifient une machine à états d'UML. Nous donnons ci-dessous la description complète de chaque concept de ce paquetage. Ces descriptions sont tirées de la spécification d'UML 2.2 Superstructure [OMG-UML 2.2, 2009].

a) StateMachine

Description

Une machine à états est utilisée pour décrire le comportement d'une partie d'un système. Le comportement est modélisé par un graphe de nœuds d'états interconnectés par un ou plusieurs arcs de transitions. Ces transitions sont déclenchées par l'arrivée d'événements (triggers). Ces événements sont soit des événements d'appels appelés opérations, des événements de type signal tels que la communication asynchrone à sens unique entre deux objets, des événements de changement (*When* <*condition_booléeenne*>), et des événements temporels (*When (date* = <*date*>). Une machine à états exécute une série d'actions au franchissement de chaque transition. La machine à états exécute une série d'activités/actions au franchissement de chaque transition.

Généralisation

- Behavior (from UML 2.2 :: CommonBehaviors :: BasicBehaviors)

Associations

- region : Region [1..*]. Cette association spécifie les régions qui sont directement contenues dans une machine à états.
- connectionPoint [*] : Pseudostate. Cette association spécifie les points de connexion d'une machine à états. Ces points de connexion sont des nœuds d'états spéciaux tels que : état initial, état historique (*deepHistory*, *shallowHistory*), nœud de parallélisme (*fork*), nœud de jointure ou d'union (*join*), nœud de jonction (*junction*), nœud de décision (*choice*), état entrant (*entryPoint*), état sortant (*exitPoint*), et nœud de terminaison (*terminate*).
- subMachineState [*] : State. Cette association spécifie les états d'une machine à états.

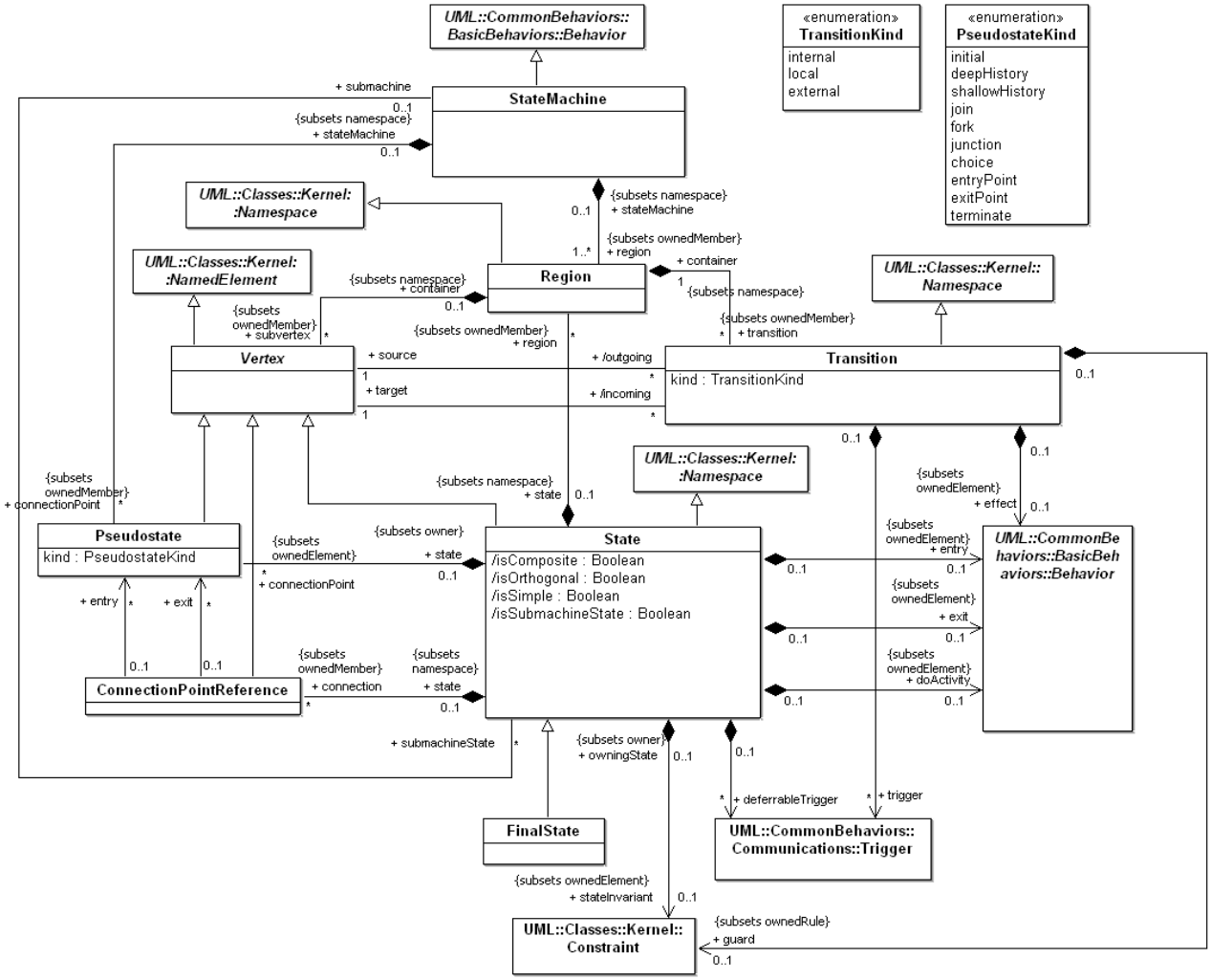


FIG. 3 Paquetage UML 2.2 BehaviorStateMachines [OMG-UML 2.2, 2009]

b) Region

Description

Une région est la partie orthogonale d'un état composite ou d'une machine à états. Il contient des états et des transitions.

Généralisation

- Namespace (from UML 2.2 :: Classes :: Kernel)

Associations

- stateMachine : StateMachine [0.1]. Cette association spécifie la machine à état qui contient la région. Si une région est contenue dans une machine à états, elle ne pourra ne pas être contenue dans un état.
- state : State [0..1]. Cette association spécifie l'état composite qui contient la région. Si une région est contenue dans un état, composite elle ne pourra ne pas être contenue dans une machine à états.

- transition : Transition [*]. Cette association spécifie l'ensemble des transitions contenues dans la région.
- subvertex : Vertex [*]. Cette association spécifie l'ensemble des Vertex contenues dans une région.

c) Vertex

Description

Vertex est une généralisation abstraite des nœuds d'états. IL est la source ou la destination d'une transition.

Généralisation

- NamedElement (from UML 2.2 :: Classes :: Kernel)

Associations

- outgoing : Transition [*]. Cette association spécifie les transitions sortantes (i.e. transitions qui partent du Vertex)
- incoming: Transition [*]. Cette association spécifie les transitions entrantes (i.e. transitions qui arrivent au Vertex)
- container : Region [0.1]. Cette association spécifie la région qui contient le Vertex.

d) State

Description

Un état modélise une situation dans laquelle un invariant devient vrai. Cet invariant peut représenter une situation statique telle qu'un objet qui attend qu'un événement externe se produise pour entrer dans un état.

Généralisation

- Vertex

Attributs

- isComposite : Boolean. Cette association spécifie si l'état est composite ou non. Un état composite est un état qui contient au moins une région.
- isOrthogonal : Boolean. Cette association spécifie si l'état est orthogonal ou non. Un état orthogonal est un état composite qui contient au moins deux régions.
- isSimple : Boolean. Cette association spécifie si l'état simple ou non. Un état simple ne contient pas de région et ne référence aucune machine à état.
- isSubmachineState : Boolean. Cette association spécifie si l'état référence une machine à état ou non.

Associations

- connection : ConnectionPointReference [*]. Cette association spécifie les références des points de connexion d'un état. Ces références pointent vers des instances de Pseudostate.
- connectionPoint : Pseudostate [*]. Cette association spécifie les points de connexion d'un état composite. Ces points de connexion représentent des instances de Pseudostate dont l'attribut *kind* est entryPoint ou exitPoint (voir WF07).
- entry : Behavior [0..1]. Cette association spécifie l'activité qui s'accomplit quand on entre dans l'état.
- exit : Behavior. Cette association spécifie l'activité qui s'accomplit quand on sort de l'état.
- doActvity : Behavior [0..1]. Cette association spécifie une activité qui commence dés que l'activité entry est terminée. Si une transition se déclenche, l'activité « do » est interrompue et l'activité exit de l'état s'exécute.
- region : Region [*]. Cette association spécifie les régions qui sont contenues dans un état.
- subMachine : StateMachine. Cette association spécifie la machine à état qui est référencée par l'état (qui doit être insérée à la place de l'état).
- stateInvariant : Constraint [0..1] Cette association spécifie la condition qui est toujours vraie quand l'état devient l'état courant.

Règles de bonne modélisation

- WF01: Un état ne peut pas voir à la fois des régions et une sous-machine à état. isComposite **implies not** isSubmachineState
- WF02 : Un état simple est un état qui ne contient aucune région.
  isSimple **implies** region.isEmpty ()
- WF03 : Un état composite est un état qui contient au moins une région.
  isComposite **implies** region.notEmpty ()
- WF04 : Un état orthogonal est un état composite qui contient au moins deux régions.
  isOrthogonal **implies** (region→size () > 1)
- WF05 : Seuls les états dont la propriété isSubmachineState est vraie peuvent référencer une machine à états. isSubmachineState **implies** (submachine.notEmpty ())
- WF06 : Seuls les états composites peuvent avoir des points de connexion. connectionPoint→notEmpty () **implies** isComoposite
- WF07 : Les points de connexion d'un état doivent être des instances de *Pseudostate* dont la valeur de *kind* est entry ou exit.
  connectionPoint→**forAll** (cp|cp.kind = #entryPoint **or** cp.kind = #exitPoint)
- WF08 : Seuls les états dont la propriété *isSubmachineState* est vraie peuvent avoir des références de points de connexion.
  isSubmachineState **implies** (connection→notEmpty ( ))
- WF09 : Une machine à états M1 référencée par un état E1 est la même que celle qui

contient les instances de *Pseudostate* (entry, exit) connectées à l'état E1 via *ConnectionPointReference*.
self.isSubmachineState **implies** (self.connection→**forAll** (cp | cp.entry→forAll (p | p.statemachine = self.submachine) **and** cp.exit→**forAll** (p | p.statemachine = self.submachine)))

e) Pseudostate

Description

Les instances de *Pseudostate* sont utilisées pour connecter des transitions multiples qui peuvent prendre des chemins différents. Par exemple le nœud de parallélisme (*fork*) peut être utilisé pour spécifier deux comportements qui s'exécutent de façon parallèle et indépendante.

Généralisation

- Vertex

Attribut

- kind : PseudostateKind. Cet attribut spécifie le type de PseudoSate. La valeur du type est déterminée par l'énumération *PseudostateKind*. La valeur par défaut est « *initial* ».

Associations

- stateMachine : StateMachine [0..1]. Cette association spécifie la machine état dans laquelle sont définies les instances de *Pseudostate* (*entryPoint* ou *exitPoint)*.
- state : State [0..1]. Cette association spécifie l'état qui contient l'instance du *Pseudostate*.

Règles de bonne modélisation

- WF01: Un état initial doit avoir au plus une transition sortante. (self.kind = #initial) **implies** (self.outgoing→size <= 1)
- WF02 : Une transition sortante d'un état initial est automatique (i.e. elle n'a ni déclencheur, ni garde). (self.kind = PseudostateKind::initial) **implies** (self.outgoing.guard→isEmpty() **and** self.outgoing.trigger→isEmpty())
- WF03 : Dans une machine à états complet, un nœud de choix doit avoir au moins une transition entrante et une transition sortante. (self.kind = #choice) **implies** ((self.incoming→size >= 1) **and** (self.outgoing→size >= 1))
- WF04 : Dans une machine à états complet, un nœud de jonction doit avoir au moins une transition entrante et une transition sortante. (self.kind = #junction) **implies** ((self.incoming→size >= 1) **and** (self.outgoing→size >= 1))
- WF05: Les transitions entrantes d'un nœud de jointure (*join*) doivent provenir des états de différentes régions d'un état orthogonal. (self.kind = #join) **implies** self.incoming→**forAll** (t1, t2 |t1<>t2 **implies** (self.stateMachine.LCA (t1.source, t2.source).container.isOrthogonal))
- WF06: Les transitions sortantes d'un nœud de parallélisme (fork) doivent avoir comme cible des états qui sont dans des régions différentes d'un état orhogonal.

((self.kind = #fork) **implies** self.outgoing→**forAll** (t1, t2 | t1<>t2 **implies** (self.stateMachine.LCA (t1.target, t2.target). container.isOrthogonal))

- WF07 : Dans une machine à états complet, un nœud de jointure doit avoir au moins deux transitions entrantes et exactement une transition sortante. (self.kind = #junction) **implies** ((self.incoming→size >= 2) **and** (self.outgoing→size = 1))
- WF08 : Dans une machine à états complet, un nœud de parallélisme doit avoir au moins deux transitions sortantes et exactement une transition entrante. (self.kind = #fork) **implies** ((self.incoming→size =1) **and** (self.outgoing→size >= 2))
- WF09: Un état historique-profond ou historique doit avoir au maximum une transition sortante. ((self.kind = #deepHistory) **or** (self.kind = #shallowHistory)) **implies** (self.outgoing→size <= 1)

f) FinalState

Description

C'est un état spécial qui marque la fin d'un comportement contenu dans une région. Si la région est contenue dans une machine à états et que tous les comportements des autres régions atteignent le nœud final, alors le comportement de la machine à états sera donc terminé.

Généralisation

- State

Règles de bonne modélisation

- WF01: Un état final n'a pas de transitions sortantes. self.outgoing→size ()=0
- WF02 : Un état final ne doit pas contenir des régions. self.region→size ()=0
- WF03 : Un état final ne peut pas référencer une sous-machine à états. self.submachine→isEmpty ()
- WF 04 : Un état final n'a pas de comportement d'entrée. self.entry→isEmpty ()
- WF 05 : Un état final n'a pas de comportement de sortie. self.exit→isEmpty ()
- WF06 : Un état final n'a pas de comportement (doActvity). self.doActivity→isEmpty()

g) Transition

Description

Une transition est une relation directe entre un état source et un état cible.

Généralisation

- Namespace (from UML 2.2 :: Classes :: Kernel)

Attribut

- kind : TransitionKind. Cet attribut définit les types de transitions spécifiés par

l'énumération *TransitionKind*. Une transition externe modifie l'état actif. C'est le type de transition le plus répandu. Elle est représentée graphiquement par une flèche qui lie les deux états. Une transition interne ne possède pas d'état cible : l'état actif reste le même à la suite de son déclenchement.

Associations

- trigger:Trigger [*]. Cette association spécifie les événements déclencheurs d'une transition. Ces événement sont soit des événements d'appels (*opérations*), des événements de type signal (*communication asynchrone à sens unique entre deux objets*), des événements de changement (*when <condition_booléeenne>*) ou des événements temporels (*when (date = <date>*).
- guard : Constraint [0..1]. Cette association spécifie les contraintes associées au déclenchement d'une transition. Une garde est évaluée quand un événement arrive. Quand la garde devient vraie, la transition est activable ou franchissable ou franc, si non elle est infranchissable. Les gardes sont des expressions exprimées parfois en OCL. Elles sont sans effet de bord car elles ne modifient pas l'état d'un objet.
- effect : Behavior [0..1]. Cette association spécifie optionnellement un comportement (un ensemble d'actions) qui peut être exécuté une fois la transition déclenchée.
- source : Vertex [1]. Cette association spécifie l'état source d'une transition.
- target : Vertex [1]. Cette association spécifie l'état cible d'une transition.

Règles de bonne modélisation

- WF01: Une transition qui part d'un nœud de parallélisme n'a ni garde ni événement déclencheur. (source.oclIsKindOf (Pseudostate) **and** source.kind =#fork) **implies** (guard→isEmpty () **and** trigger→isEmpty ()).
- WF02: Une transition qui arrive à un nœud de jointure n'a ni garde ni événement déclencheur. (target.oclIsKindOf (Pseudostate) **and** target.kind=#join) **implies** (guard→isEmpty () **and** trigger→isEmpty ()).
- WF03: Les transitions sortantes d'un nœud de parallélisme ont pour cible des états. (source.oclIsKindOf(Pseudostate) **and** source.kind=#fork) **implies** (target.oclIsKindOf(State))
- WF04: Les transitions entrantes d'un nœud de jointure ont pour source des états. (target.oclIsKindOf(Pseudostate) **and** target.kind=#join) **implies** (source.oclIsKindOf(State))
- Les transitions sortantes d'un Pseudostate ne doivent pas avoir de déclencheurs (exceptées celles provenant de l'état initial). (source.oclIsKindOf(Pseudostate) **and** (source.kind <> #initial)) **implies** trigger→isEmpty()

ANNEXE C: LE PAQUETAGE QVTBASE DE MOF 2.0 QVT

La spécification de MOF 2.0 QVT [OMG-QVT 1.0, 2008] propose huit paquetages (voir Figure III.6). Dans cette section, nous ne décrivons que les concepts du paquetage *QVTBase* sur lequel reposent les paquetages *QVTCore*, *QVTRelation* et *QVTOperational*. Pour plus d'information sur les autres paquetages de la spécification de MOF 2.0 QVT, le lecteur pourra consulter [OMG-QVT 1.0, 2008].

Le paquetage *QVTBase* de MOF 2.0 QVT (FIG. 4) contient un ensemble de concepts de base dont une partie provient de la spécification d'EMOF et d'OCL. Les concepts de *QVTBase* structurent les transformations, leurs règles, et leurs modèles d'entrée et de sortie. Nous donnons ci-dessous la description complète de chacun de ces concepts. Ces descriptions sont tirées de la spécification de MOF 2.0 QVT [OMG-QVT 1.0, 2008]

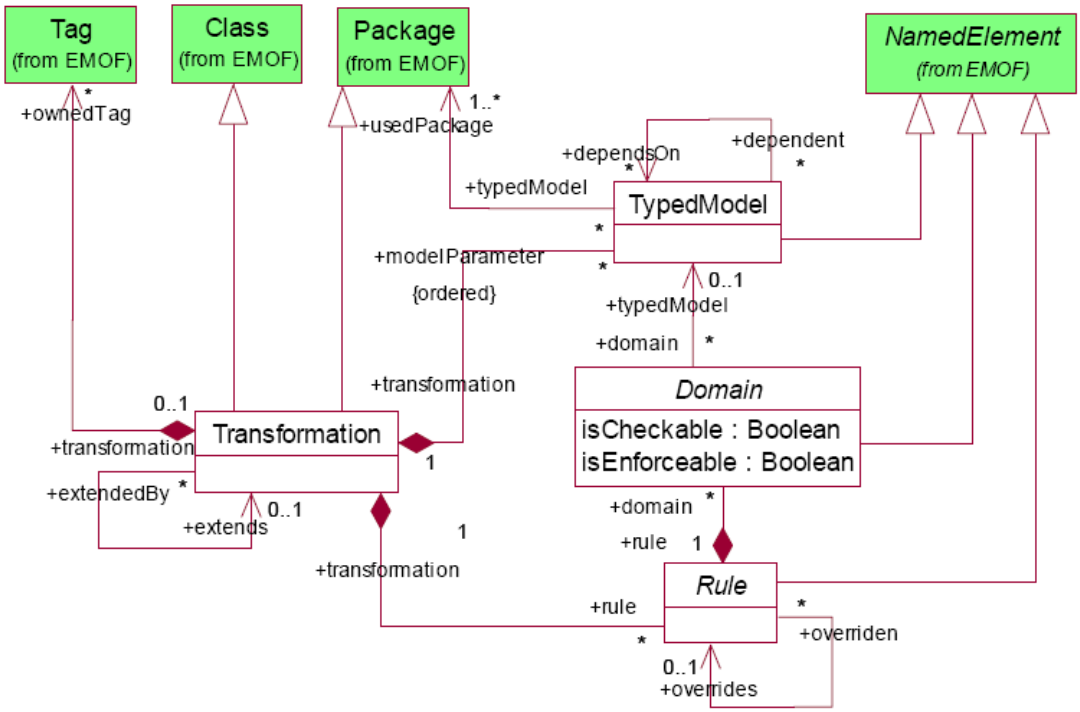


FIG. 4 Paquetage *QVTBase* [OMG-QVT 1.0, 2008]

a) Transformation

Description

Une transformation dans QVT définit comment un ensemble de modèles sources peut être transformé en un ensemble de modèles cibles. Il contient un ensemble de règles qui spécifie le comportement d'exécution de la transformation. La transformation est exécutée sur un ensemble de modèles dont les types sont définis par les paquetages associés à ces modèles. Syntaxiquement, le concept *Transformation* est à la fois une spécialisation des concepts *Package* et *Class*. Définie comme un paquetage, la transformation fournit un espace de nommage à ses règles. Définie comme une classe, elle peut définir des propriétés et des opérations. Les propriétés servent à spécifier les valeurs des paramètres de la configuration de l'environnement d'exécution de la transformation. Les opérations servent à implémenter les fonctions utilitaires requises par l'environnement d'exécution.

Généralisation

- Package, Class (from EMOF)

Associations

- modelParameter: TypedModel [*]. Cette association spécifie un ensemble ordonné de modèles typés qui participent à une transformation.
- rule : Rule [*]. Cette association spécifie les règles QVT contenues dans une transformation. Ces règles décrivent la sémantique d'exécution d'une transformation.
- ownedTag : Tag [*]. Cette association spécifie les tags d'une transformation dont leurs valeurs peuvent être utilisées pour configurer l'environnement d'exécution.
- extends : Transformation [0..1]. Cette association spécfie la transformation qui est étendue.
- extendedBy : Transformation [*]. Cette association spécifie les transformations qui étendent une autre transformation. Les règles de la transformation étendue sont incluses dans les règles de chaque transformation qui l'étend afin de spécifier plus tard leur comportement d'exécution.

b) TypedModel

Description

TypedModel spécifie un paramètre c'est-à-dire un modèle typé d'une transformation QVT. Un modèle typé est un modèle conforme à un autre modèle appelé métamodèle. Les métamodèles d'une transformation QVT sont définis par le concept *Package*. Une transformation QVT est exécutée dans une direction particulière en sélectionnant un modèle comme cible de la transformation parmi l'ensemble des modèles typés.

Généralisation

- NamedElement (from EMOF)

Associations

- transformation: Transformation [1]. Cette association spécifie la transformation qui contient les modèles typés.
- usedPackage : Package [1..*]. Cette association spécifie les paquetages qui définissent les métamodèles de la transformation.
- dependsOn : TypedModel [*]. Cette association spécifie les modèles sur lesquels dépendent d'autres modèles.
- dependent: TypedModel [*]. Cette association spécifie les modèles qui dépendent d'autres modèles.

c) Domain

Description

Domain spécifie l'ensemble des éléments de modèle qui participent à une règle d'une transformation. *Domain* est une méta-classe abstraite dont ses sous-classes concrètes spécifient les éléments de modèle sous forme de graphe, de variable ou de contraintes, ou tout autre mécanisme qui permet de représenter les éléments d'un modèle.

Généralisation

- NamedElement (from EMOF)

Attributs

- isCheckable : Boolean. Si un domaine est « checkable » si les éléments de modèles du domaine existent dans le modèle cible. Si non la règle associée au domaine reporte les erreurs.
- isEnforceable : Boolean. Si un domaine est « enforceable » alors les éléments de modèles du domaine doivent exister dans le modèle cible quand la transformation est exécutée.

Associations

- rule : Rule [1]. Cette association spécifie la règle qui contient le domaine.
- typedModel : TypedModel [0..1]. Cette association spécifie le modèle typé qui contient les éléments de modèle spécifiés par le domaine.

d) Rule

Description

Les règles décrivent la sémantique d'exécution d'une transformation c'est-à-dire comment les éléments d'un modèle source seront transformés en des éléments d'un modèle cible. *Rule* est une méta-classe abstraite dont ses sous-classes concrètes spécifient la sémantique d'exécution exacte d'une transformation. Une règle est soit déclarative ou impérative. Elle peut être unidirectionnelle (Mapping) ou bidirectionnelle (Relation).

Généralisation

- NamedElement (from EMOF)

Associations

- domain : Domain [*]. Cette association spécifie les domaines contenus dans une règle.
- transformation : Transformation [1]. Cette association spécifie la transformation qui contient la règle.
- overrides : Rule [0..1]. Cette association spécifie la règle qui a été remplacée.
- overriden : Rule [*]. Cette association spécifie les règles qui remplacent la règle.

ANNEXE D: LES METAMODELES DU PROCESSUS UWE

Les approches de modélisation des systèmes web sont basées sur la séparation des préoccupations. Ces préoccupations sont : les exigences, le contenu, la structure hypertexte, la présentation et le processus métier. Le processus UWE propose un panorama de concepts qui permettent de modéliser ces préoccupations.

La section a) décrit les paquetages du métamodèle UWE. Ce métamodèle offre les concepts qui permettent de décrire les différentes préoccupations des applications web. La section b) décrit le métamodèle UWE sous forme de profil UML. La section c) décrit le métamodèle WebRE. Ce métamodèle décrit les concepts utilisés par le modèle des exigences. Il est composé de deux paquetages : WebRE Structure et WebRE Behavior. La section d) décrit le métamodèle WebRE sous forme de profil UML.

a) Le métamodèle UWE

Le métamodèle UWE est une extension du métamodèle UML 2. Il est composé des paquetages *Core* et *Adapivity*. La séparation des préoccupations est spécifiée par *Core* qui contient les paquetages suivants : *Requirements*, *Content*, *Navigation*, *Presentation*, et *Process*. Le métamodèle UWE est aussi défini sous forme de profil UML.

Le paquetage *Requirements* contient les extensions des cas d'utilisation UML et activités d'UML. Ce paquetage discerne les cas d'utilisation afférents à la navigation et ceux relatifs à l'aspect processus (activités/actions utilisateur). Les autres concepts du paquetage *Requirements* sont décrits dans le métamodèle WebRE (section c).

Le paquetage *Content* décrit le contenu d'une application web. La modélisation du contenu des applications web dans le processus UWE ne diffère pas de celle des autres applications. C'est ainsi que dans UWE, les diagrammes de classes sont utilisés pour décrire la structure du contenu tandis que les machines à états et les diagrammes de séquence décrivent le comportement du contenu de l'application web.

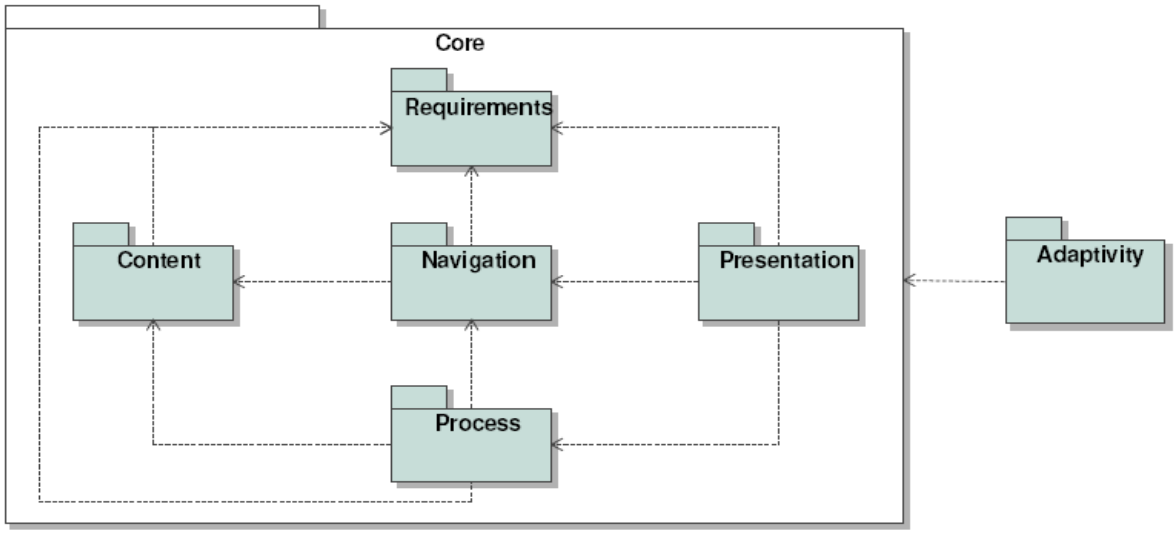


FIG. 5 Les paquetages du métamodèle UWE [Kroiß et al., 2008]

Le paquetage *Navigation* (voir Fig. 6) décrit les modèles de navigation d'une application web. Un modèle de navigation permet de spécifier les nœuds (unité d'information d'un système web) et leurs liens. Un lien permet de connecter un nœud (la source) à plusieurs nœuds (les cibles). Exemple de nœud : une page web vue comme une boîte noire. Exemple de lien : un lien hypertexte.

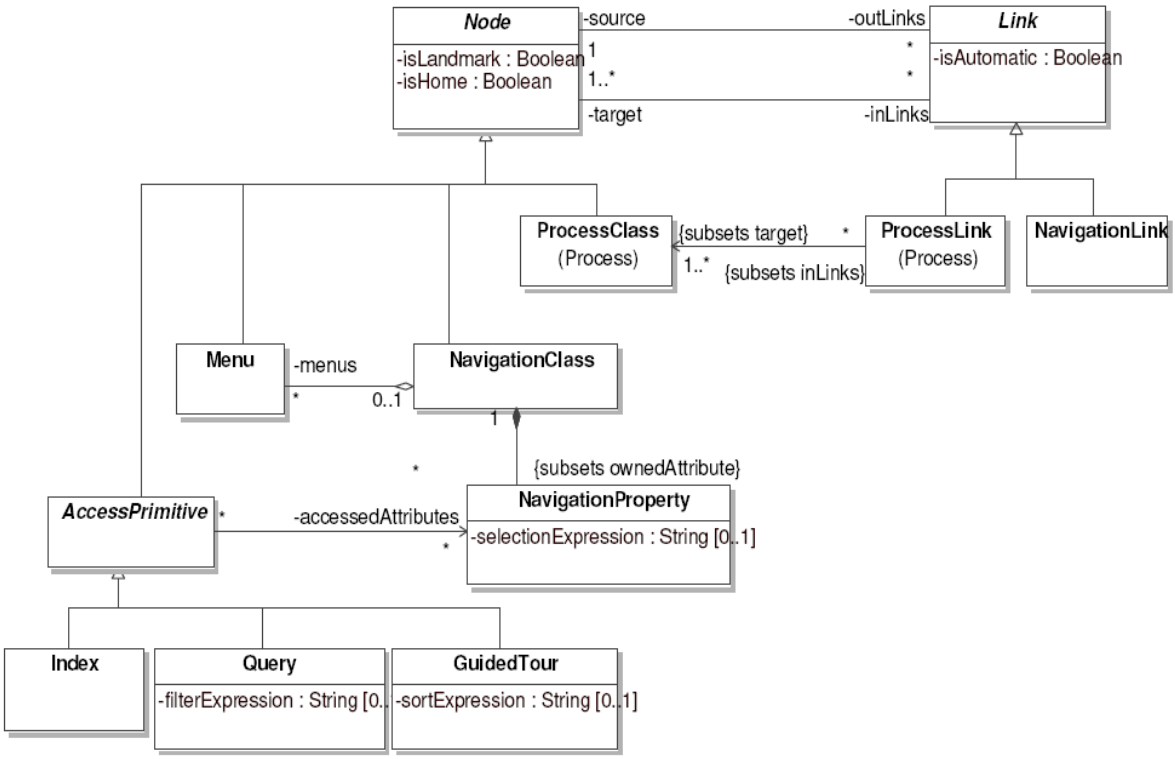


FIG. 6 Le paquetage « Navigation » [Kroiß et al., 2008]

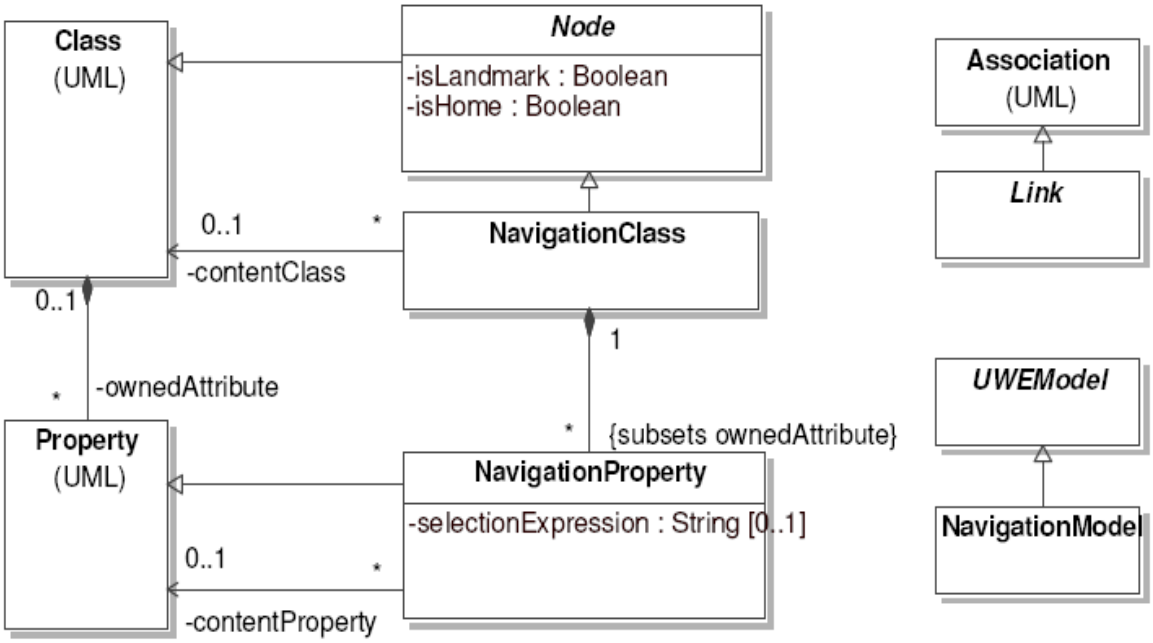


FIG. 7 Relations entre le paquetage « Navigation » et le métamodèle UML [Kroiß et al., 2008]

Le paquetage *Presentation* (voir FIG. 8 et FIG. 9) fournit les concepts qui permettent de décrire les interfaces utilisateur (zone de saisie, label, image, bouton, formulaire, etc.) d'un système web. Il décrit la structure basique des interfaces utilisateurs (c'est-à-dire quelle interface utilisateur sera utilisée pour présenter un nœud du modèle de navigation). Le modèle de présentation ne permet pas de décrire certains aspects spécifiques aux interfaces utilisateur telles que l'utilisation des couleurs, la police, et l'emplacement de ces interfaces sur une page web. Les interfaces utilisateurs sont aussi indépendantes d'une technologie donnée.

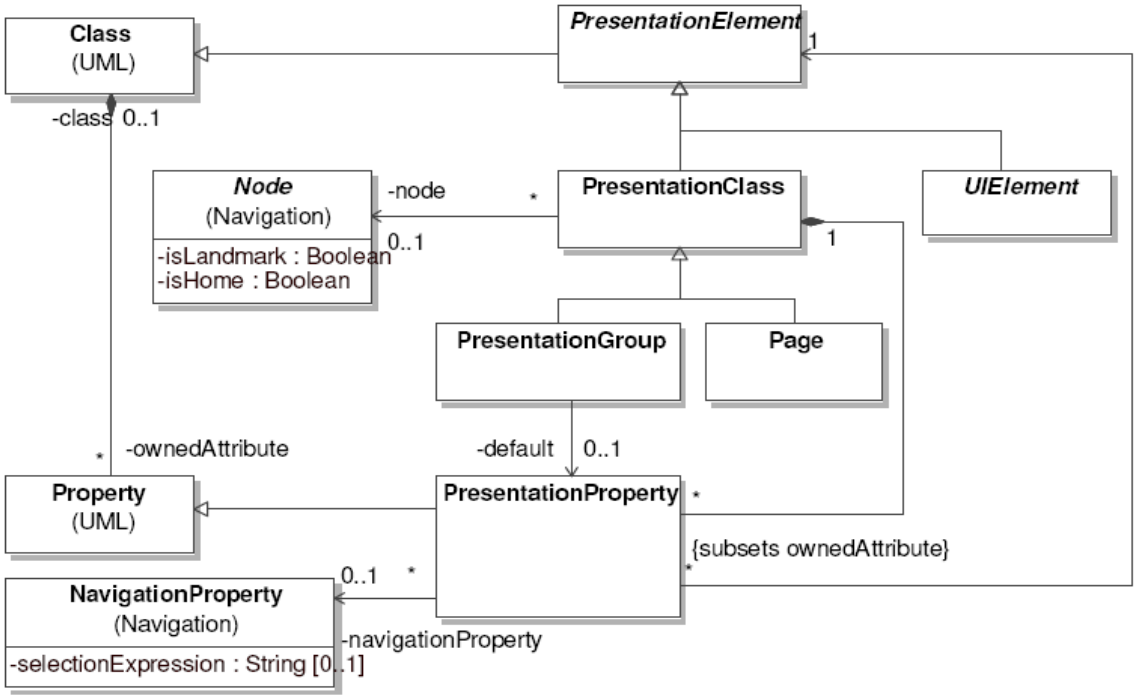


FIG. 8 Le noyau du paquetage « Presentation » [Kroiß et al., 2008]

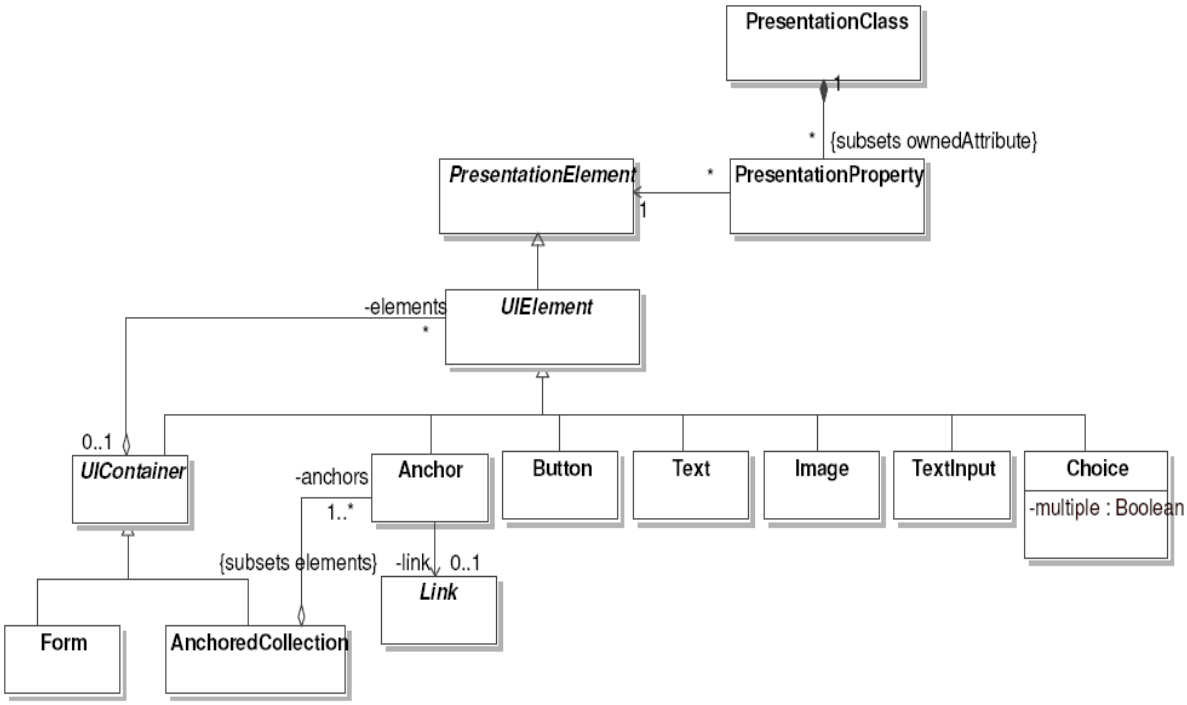


FIG. 9 Les éléments du paquetage « Presentation » [Kroiß et al., 2008]

Le paquetage *Process* (FIG. 10) fournit les concepts qui permettent d'intégrer le processus métier dans un modèle de navigation Cette intégration se déroule en trois activités :

Intégration du processus métier dans le modèle de navigation. Cette activité permet de lier chaque activité/action du processus spécifiée par le concept *ProcessClass* à un nœud du modèle de navigation.

Définition d'une interface utilisateur qui supporte les activités et les actions du processus métier. Pour chaque activité et action du processus, une interface utilisateur lui est associée dans le modèle de présentation.

Définition d'un comportement. Cette activité se fait par le bais des diagrammes d'activités qui décrit l'enchainement des activités et des actions d'un utilisateur.

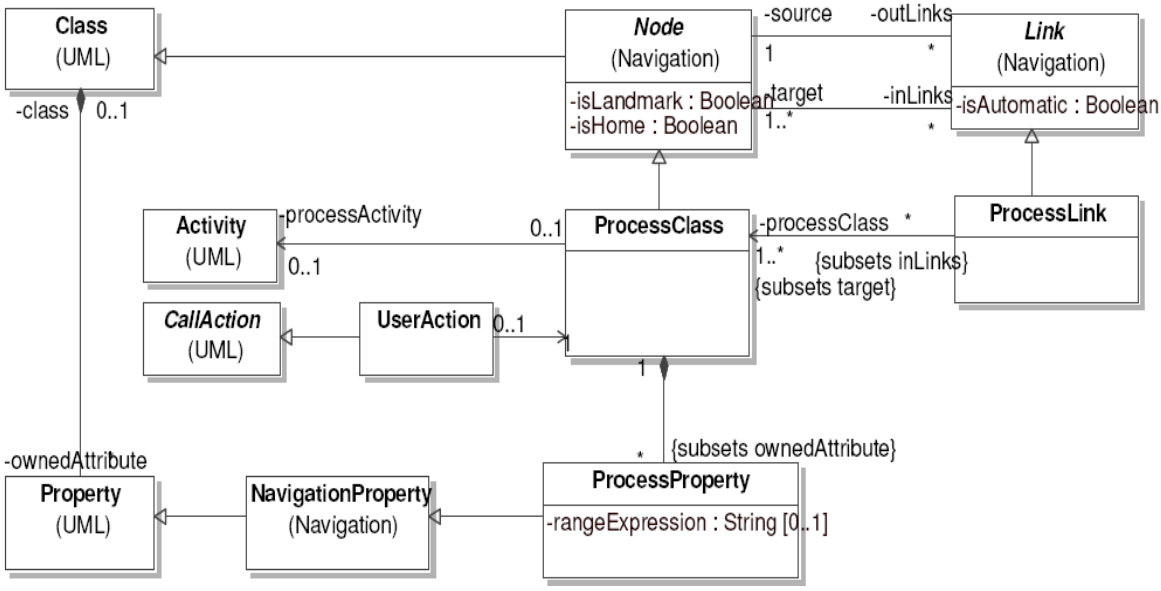


FIG. 10 Le paquetage « Process » [Kroiß et al., 2008]

b) Le profil UWE

UWE stereotype	UML base class	Used in	Icon
«anchor»	class	presentation model	
«anchored collection»	class	presentation model	
«button»	class	presentation model	
«choice»	class	presentation model	
«form»	class	presentation model	
«guided tour»	class	navigation model	
«image»	class	presentation model	
«index»	class	navigation model	
«menu»	class	navigation model	
«navigation class»	class	navigation model	
«navigation link»	association	navigation model	
«navigation property»	property	navigation model	
«page»	class	presentation model	
«presentation class»	class	presentation model	
«presentation group»	class	presentation model	

UWE stereotype	UML base class	Used in	Icon
«presentation property»	property	presentation model	
«process class»	class	navigation/process model	
«process link»	association	navigation model	
«process property»	property	navigation/process model	
«query»	class	navigation model	
«text input»	class	presentation model	
«text»	class	presentation model	
«user action»	action	process model	

FIG. 11 Les stéréotypes du profil UWE [Kroiß et al., 2008]

c) Le métamodèle WebRE du processus UWE

Le métamodèle WebRE décrit les concepts utilisés par le modèle des exigences. Il est composé de deux paquetages : WebRE Structure et WebRE Behavior.

Dans ce métamodèle un utilisateur spécifié par « *WebUser* » interagit avec le système web. Un utilisateur est associé à un cas d'utilisation spécifié par «*Navigation* ». Ce type de cas d'utilisation inclut une ou plusieurs actions spécifiées par « *Browse* » qui permettent de suivre un lien (de connaitre le nœud de départ et le nœud d'arrivée). Une action de type « *Search* » est un nœud spécial de type *« Browse »* permettanr de spécifier une requête d'un utilisateur sur un système web. Un cas d'utilisation spécifié par « *WebProcess »* inclut une ou plusieurs actions utilisateurs spécifiées par « *UserTransaction* ».

La source et la cible d'une action spécifiée par *Browse* sont des nœuds dont chacun est associé à une plusieurs pages spécifiées par « *WebUI* ». A l'inverse une page est associée à un ou plusieurs nœuds. Un nœud permet de montrer les différentes pièces d'une information, chacune étant associée à une ou plusieurs instances de la méta-classe «*Content* ».

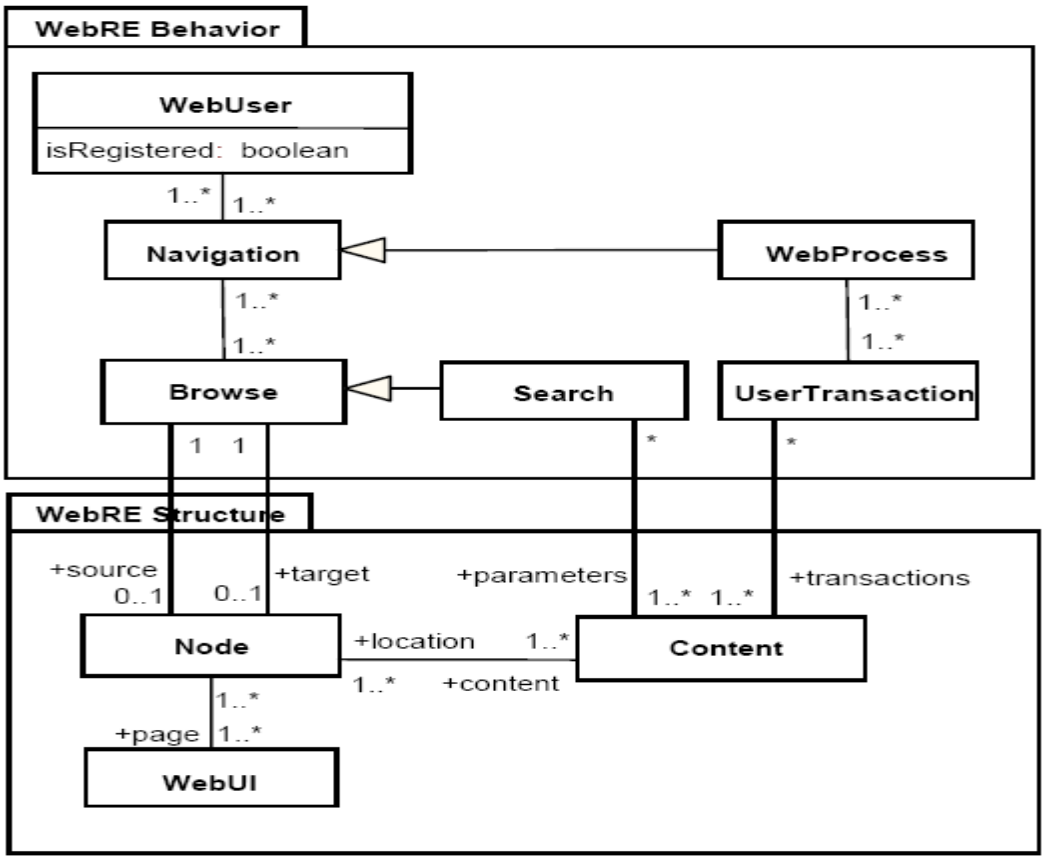


FIG. 12 Le métamodèle WebRE [Escalona et al., 2006]

d) Le profil WebRE

Metaclass	Stereotype	Icon
UseCase	«navigation»	□
UseCase	«Web process»	Σ
Action	«browse»	⇒
Action	«search»	?
Action	«user transaction»	⇔
Classifier	«node»	□
Classifier	«content»	○
Classifier	«webUI»	▣

FIG. 13 Les stéréotypes du profil WebRE [Escalona et al., 2006]