HAL
open science

Codèle : Une Approche de Composition de Modèles pour la Construction de Systèmes à Grande Échelle

Thi Thanh Tam Nguyen

To cite this version:

Thi Thanh Tam Nguyen. Codèle : Une Approche de Composition de Modèles pour la Construction de Systèmes à Grande Échelle. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2008. Français. NNT : . tel-00399655

HAL Id: tel-00399655

https://theses.hal.science/tel-00399655

Submitted on 27 Jun 2009

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 JOSEPH FOURIER - GRENOBLE I

THÈSE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier de Grenoble

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée et soutenue publiquement par

NGUYEN Thi Thanh Tam

le 22 Décembre 2008

Codèle : Une approche de composition de modèles pour la Construction de Systèmes à Grande Échelle

---

Directeur de thèse

Jacky ESTUBLIER

JURY

Joelle Coutaz, Professeur à l'Université Joseph Fourier	Président
Pierre-Alain Muller, Professeur à l'Université de Haute-Alsace	Rapporteur
Michel Riveill, Professeur à l'Université de Nice	Rapporteur
Mireille Blay-Fornarino, Maître de Conférences à l'Université de Nice	Examinateur
Hervé Verjus, Maître de Conférences à l'Université de Savoie	Examinateur
Jacky Estublier, Directeur de recherche au CNRS	Directeur de thèse

Thèse préparée au sein du Laboratoire Informatique de Grenoble (LIG) - Équipe ADÈLE

Remerciements

Je remercie toutes les personnes qui ont offert leur soutien et leur aide pour cette thèse. Parmi celles-ci, je tiens à remercier plus particulièrement :

Les membres du jury qui m'ont fait l'honneur de participer à ma soutenance.

J'adresse un grand merci à Monsieur Jacky Estublier, mon directeur de thèse, pour tous ses conseils et son soutien tout au long de ma thèse ainsi que pour sa disponibilité et sa précieuse relecture pendant la rédaction de ce manuscrit. Sans sa gentillesse et sa patience, je ne serais pas arrivée au bout de ce long chemin.

Je tiens à remercier profondément Jean-Marie Favre, mon encadrant de DEA. Les nombreuses connaissances du monde IDM que j'ai apprises pendant ce stage sont les premières pierres de mon chemin de thèse. Je le remercie également pour les discussions et les remarques très constructives aux sujets des modèles et métamodèles.

Je veux remercier également tous mes amis de l'équipe Adèle, spécialement mon groupe Mélusine, pour toutes les discussions que j'ai eu la chance d'avoir avec eux. Je pense ici en particulier à German Vega qui m'a rendu de nombreux services grâce à ses innombrables remarques et suggestions. Je lui suis très reconnaissante du point de vue académique et professionnel. Je pense aussi chaleureusement à Stéphane, Anca, Cristina pour leur amitié et leur partage très agréable des moments professionnels. Merci Thomas pour ses efforts de relecture et de corrections linguistiques attentives et rapides pendant le dernier temps critique de ma rédaction. Merci Stéphanie et Etienne pour leur aide technique de rédaction sur Latex, sans eux le manuscrit ne peut pas être comme il est. Merci Ada pour sa traduction anglaise de la résumé. Merci Adele.

Je veux remercier aussi mes amis vietnamiens à Grenoble pour leur partage des moments quoiditiens. Je pense chaleuresement à Hong-Thai, Phuong-Anh, Thuy-Hien, Thi-Thai. Je pense aussi à Ngoc-Anh, Doan-Hiep, mes chers amis au Viet Nam.

Enfin, je veux adresser toute ma gratitude à ma famille, à mes parents et mon petit frère Dua, qui m'ont toujours soutenu, malgré l'éloignement géographique. Je veux leur adresser cette thèse pour les remercier profondément pour leur inépuisable amour. (Cam on Ba, Ma và em Dua. Con xin danh tang ket qua nay cho gia dinh.)

Résumé

Depuis "toujours", en Génie Logiciel comme dans toutes les ingénieries, afin réduire la complexité et pour améliorer la réutilisation, le produit à construire est divisé en parties construites indépendamment et ensuite assemblées. L'approche récente de l'Ingénierie Dirigée par les Modèles (IDM, ou MDE pour Model-Driven Engineering), fait de même, en proposant "simplement" que les parties à construire et à assembler soient des *modèles* et non pas des programmes. C'est ainsi que le problème de la *composition de modèles* est devenu un thème important de l'IDM, et le sujet de cette thèse.

En effet, un système logiciel réel est bien trop complexe pour pouvoir être décrit par un seul modèle. De nombreux modèles devront être créés pour le spécifier, soit à divers niveaux d'abstraction, soit selon divers points de vue, soit encore selon des domaines fonctionnels différents et complémentaires.

Dans ce travail, nous partons de l'hypothèse que de tels domaines métiers existent. Un domaine est un champ d'expertise avec 1) les savoir-faire et les connaissances capturés et formalisés sous la forme d'un langage de modélisation dédié au domaine (un Domain-Specific Modeling Language (DSML)), et 2) des outils et des environnements supportant le développement d'applications dans ce domaine. Une application, dans un domaine, est décrite par un modèle (conforme au métamodèle du domaine). Dans ce travail, nous faisons aussi l'hypothèse que ces domaines sont *exécutables* ; les modèles du domaine sont exécutés par une machine virtuelle du domaine.

En IDM, comme dans les autres approches du Génie Logiciel, la réutilisation impose que le processus d'assemblage des parties puisse se faire sans avoir à modifier celles-ci, ni bien sur l'environnement qui les a produit ou qui les exécute. Appliqué à notre contexte, cela signifie qu'il faut être capable de composer des modèles sans les modifier et sans modifier ni l'éditeur ni la machine virtuelle du domaine dont ils sont issus. C'est le problème que nous abordons dans notre travail.

Nous montrons que pour atteindre cet objectif, il faut d'abord composer les domaines c'est à dire composer leurs métamodèles et leurs machines virtuelles, puis les modèles. Nous montrons dans ce travail comment ces compositions peuvent être réalisées sans modifier les éléments composés, en utilisant des relations dites *horizontales* entre les métamodèles, et des liens entre les modèles.

Cette approche est validée d'abord par la réalisation d'un certain nombre de domaines composites, et par de nombreuses compositions de modèles opérationnelles. Une partie importante du travail a consisté à définir Codèle, un langage de composition de modèles et de métamodèles, et à réaliser l'environnement d'assistance correspondant. Codèle assiste les ingénieurs pour composer de façon systématique les domaines, et rend automatique ou semi-automatique la composition de tous modèles provenant de ces domaines. Nous présentons l'outil Codèle et nous évaluons son usage dans divers projets.

Mots clés : Ingénierie dirigée par les modèles, composition de modèles, composition de métamodèles, modèles exécutables, langages de modélisation dédiés aux domaines, développement d'applications à grande échelle, Codèle.

Abstract

Since "always", in Software Engineering as in all other engineering fields, the product to be built is divided into parts that are independently built and subsequently assembled. This procedure reduces the complexity and improves the reusability of the products built. Model-Driven Engineering (MDE) is a recent engineering initiative that adopts this approach. MDE "simply" proposes that the parts to be built and assembled be *models* rather than software programs. In this context, the problem of *model composition* has become an important theme in the MDE domain and constitutes the subject of this thesis.

Indeed, a real software system is much too complex to be described in a single model. Multiple models should be created for the system specification. Such models could represent different system abstraction levels, different system view-points, or different and complementary functional domains.

In the presented work, we start from the hypothesis that such application domains do exist. A certain domain is a specialised field containing: 1) the know-how and the knowledge captured and formalised via a Domain-Specific Modeling Language (DSML); and 2) tools and environments for supporting the development of applications in the corresponding domain. In a certain domain, an application is described via a model (that conforms to the domain-specific meta-model). In the presented work, we also make the hypothesis that the considered domains are *executable* - the domain's models are executed by a domain-specific virtual machine.

In MDE, as in all other Software Engineering approaches, reutilisation imposes that the assembly process can be performed without having to modify the parts concerned or their production or execution environment. In our context, this means that model composition must be possible without modifying the concerned models and their domain-specific development editor and virtual machine. This is the problem that our work addresses.

We show that for reaching this goal we must first compose the concerned domains and then their specific models. In other words, we must first compose the domains' meta-models and virtual machines. In this work, we show how such compositions can be performed without modifying the composed elements, by using *horizontal* relations between the meta-models and links between the models.

This approach is validated via the creation of several composed domains and numerous composed functional models. An important part of this work consisted in defining Codèle - a model and meta-model composition language, and providing the corresponding support environment. Codèle assists engineers in performing system compositions in a systematic manner and renders automatic or semi-automatic all model composition in these domains. We present the Codèle tool and we evaluate its use in various projects.

Key words : Model Driven Engineering, model composition, meta-model composition, executable models, domain-specific modelling languages, large-scale application development, Codèle.

Table des matières

1 Introduction **9**
1.1 Contexte et problématique . . . 9
1.2 Objectifs de la thèse . . . 10
1.3 Contribution de la thèse . . . 10
1.4 Plan de la thèse . . . 11

2 Composition en Génie Logiciel **13**
2.1 Introduction . . . 13
2.2 Composition des logiciels . . . 14
2.2.1 Grain de composition . . . 14
2.2.2 Protocoles de communication . . . 15
2.2.3 Mécanisme de composition . . . 15
2.3 L'approche modulaire . . . 16
2.3.1 Grain de composition . . . 16
2.3.2 Protocole de communication . . . 16
2.3.3 Mécanisme de composition . . . 16
2.3.4 Synthèse . . . 18
2.4 L'approche de Description Architecturale . . . 18
2.4.1 Grain de composition . . . 19
2.4.2 Protocole de communication . . . 19
2.4.3 Mécanisme de composition . . . 20
2.4.4 Synthèse . . . 20
2.5 L'ingénierie logicielle à base de composants . . . 21
2.5.1 Grain de composition . . . 21
2.5.2 Protocole de communication . . . 22
2.5.3 Mécanisme de composition . . . 22
2.5.4 Synthèse . . . 24
2.6 La Programmation Orientée Aspects . . . 24
2.6.1 Grain de composition . . . 25
2.6.2 Protocole de communication . . . 25
2.6.3 Mécanisme de composition . . . 25
2.6.4 Synthèse . . . 26
2.7 Bilan . . . 26

3 L'ingénierie des modèles **29**
3.1 Modèle . . . 29
3.1.1 Classification des modèles . . . 30
3.1.2 Modèles et séparation des préoccupations . . . 31

3.1.3 Modèles et Espaces technologiques . . . 32
3.2 Langage, Langages de modélisation, Métamodèles . . . 33
3.2.1 Langage . . . 33
3.2.2 Langages de modélisation . . . 34
3.2.3 Métamodèles . . . 34
3.3 Modèles exécutables . . . 35
3.3.1 UML exécutable . . . 36
3.3.2 Conclusion . . . 37
3.4 Conclusion . . . 37

4 Composition de Modèles 39
4.1 Introduction . . . 39
4.2 Composition de modèle . . . 40
4.2.1 Une définition informelle . . . 40
4.2.2 Classification des approches . . . 40
4.3 Les approches . . . 44
4.3.1 Atlas Model Weaver . . . 44
4.3.2 Eclipse Modeling Framework . . . 47
4.3.3 Epsilon Merging Language . . . 48
4.3.4 Generic Modeling Enviroment . . . 50
4.3.5 Kompose . . . 52
4.3.6 Xactium XMF/XMF-Mosaic . . . 54
4.4 Bilan . . . 57

5 Architecture Mélusine : Contexte et Objectifs 61
5.1 Introduction . . . 61
5.2 Architecture Mélusine . . . 62
5.2.1 Présentation . . . 62
5.2.2 La première génération . . . 63
5.2.3 La deuxième génération . . . 65
5.2.4 Synthèse . . . 66
5.3 Domaines . . . 67
5.3.1 Métamodèle . . . 67
5.3.2 Modèles et éditeur de modèles . . . 70
5.3.3 Machine virtuelle . . . 71
5.4 Composition de domaines . . . 72
5.4.1 Notre démarche . . . 72
5.4.2 Composition de métamodèles . . . 73
5.4.3 Composition de modèles . . . 75
5.4.4 Composition de machines virtuelles . . . 76
5.4.5 La réalisation de composition de domaines . . . 77
5.5 Objectifs de la thèse . . . 77
5.6 Synthèse . . . 78

6 Un langage de composition de modèles 79
6.1 Introduction . . . 79
6.2 Notre approche . . . 80
6.2.1 Élements de composition . . . 80
6.2.2 Mécanisme de composition . . . 83
6.2.3 Une classification rapide . . . 84

6.2.4 Une approche générique . . . 84
6.3 Un scénario de composition . . . 87
6.3.1 Domaine de Gestion de Procédés . . . 87
6.3.2 Composition des Domaines de Procédés et de Produits . . . 89
6.4 Un langage de composition de modèles : vue globale . . . 92
6.5 Composition de métamodèles : Relations horizontales . . . 94
6.5.1 Composition structurelle (pour les modèles) . . . 95
6.5.2 Composition comportementale (pour les modèles) . . . 96
6.6 Composition de modèles : Les Correspondances . . . 99
6.7 Synthèse . . . 99

7 Codèle : Un outil de composition de modèles exécutables 101
7.1 Les modules de Codèle . . . 101
7.1.1 Composeur de métamodèles . . . 101
7.1.2 Composeur de modèles . . . 105
7.1.3 Générateur . . . 106
7.1.4 Exécution des correspondances . . . 113
7.1.5 Synthèse sur l'outil . . . 115
7.2 Codèle : Mise en ouvre . . . 115
7.2.1 Composeur de métamodèles . . . 115
7.2.2 Composeur de modèles . . . 116
7.2.3 Générateur . . . 117
7.3 Codèle : Validation . . . 117
7.4 Conclusion . . . 118

8 Conclusion 119
8.1 Synthèse de notre approche . . . 119
8.2 Contributions et résultats obtenus . . . 120
8.3 Perspectives . . . 121
8.4 Conclusion . . . 121

Références 121

Annexes 129
A.1 Génération de code pour les relations dynamiques . . . 129
A.1.1 Automatic_New . . . 129
A.1.2 Automatic_Selection . . . 130
A.1.3 Automatic_Selection.Automatic_New . . . 131
A.1.4 Interactive_Selection . . . 132
A.1.5 Interactif_Selection.Automatic_New . . . 133

TABLE DES MATIÈRES

Table des figures

2.1 Exemple de MIL 17

3.1 Modèle et ReprésenteDe 30
3.2 Espaces technologiques 33
3.3 Langage de modélisation 35

4.1 Processus de composition de modèles 40
4.2 Classification des approches de composition de modèles 45
4.3 Règles de composition dans EML 49
4.4 Un exemple de la syntaxe concrète textuelle de Kompose 53
4.5 Une mapping de synchronisation de XMF 56

5.1 Fédération 64
5.2 Domaine 66
5.3 Métamodèle du Domaine de Produit 69
5.4 Concrétisation des éditeurs et de la machine virtuelle 70
5.5 Un exemple d'éditeur de modèles 71
5.6 La composition de domaines 72
5.7 Composition de modèles par relations 73
5.8 Patrons simples de composition de modèles par relations 73
5.9 Transformation d'une relation n-aire en relations binaires 74
5.10 Les instances d'une relation 75
5.11 L'extension par l'instrumentation 76

6.1 Éléments de composition. 80
6.2 Formalisme de composition de métamodèles. 81
6.3 Formalisme de composition de modèles 81
6.4 Langage de composition de modèles 82
6.5 Processus de création de modèle composite. 84
6.6 Vision globale de notre approche. 85
6.7 Métamodèle de Gestion de Procédés 87
6.8 Un procédé de Développement Logiciel simple. 88
6.9 Composition des Domaines de Procédés et de Produits. 90
6.10 Vision globale du Métamodèle de Codèle 92
6.11 Métamodèle de Codèle. 94
6.12 Le concept de Correspondance. 99

7.1 Architecture générale du composeur de métamodèles 102
7.2 Réification automatique des métamodèles par les chargeurs. 103

7.3 Les métamodèles de Procédé et de Produit en langage Codèle. 103
7.4 Spécification des relations horizontales. 104
7.5 Architecture générale du composeur de modèles. 105
7.6 Composeur de modèles. 106
7.7 Architecture générale du générateur. 106
7.8 Patron du fichier AspectJ de la relation horizontale. 108
7.9 Exemple : Fichier Data_Product.aj 109
7.10 Établissement d'une correspondance statique 110
7.11 Patron du code de persistance. 111
7.12 Un exemple de persistance des correspondances 112
7.13 Patron du code de synchronisation. 112
7.14 Exécution du procédé SoftwareDevelopmentProcess - étape 1. 113
7.15 Exécution du procédé SoftwareDevelopmentProcess - étape 2. 114
7.16 Exécution du procédé SoftwareDevelopmentProcess - étape 3. 115
7.17 Métamodèle de Codèle en Ecore. 116
7.18 Génération de code par les patrons JET. 117

Chapitre 1

Introduction

1.1 Contexte et problématique

Il n'est plus besoin de justifier que la complexité et la taille des logiciels s'accroît, alors que les contraintes de qualité et de délais se durcissent. Devant le défi permanent de la complexité, les principes de solution ne sont pas légion, ils sont connus "de toute éternité" : diviser pour régner, abstraire pour comprendre. Le défi de la qualité et des délais passe, lui, par la réutilisation. Par contre, la déclinaison de ces principes est fortement tributaire des technologies utilisées.

En terme de Génie logiciel, "diviser" se décline dans les diverses manières de décomposer une application logicielle en problèmes "indépendants" résolus dans des parties (programmes, modules, services etc.) indépendantes ; et par la manière de recomposer ces parties pour obtenir le logiciel voulu.

De son côté, la réutilisation impose que la composition se fasse (en grande partie) en terme de parties existantes (programmer en réutilisant) ; et inversement que les parties existantes soient suffisamment flexibles pour satisfaire aux besoins d'un grand nombre de décomposition (programmer pour réutiliser).

La flexibilité des parties réutilisables a, très vite, pris la forme de programmes qui, pour être largement réutilisables, interprètent des informations de "configuration". Ces informations peuvent être des paramètres peu structurés (e.g, le fichier "config" de Apache), un modèle formel et explicite (le schéma de données d'une base de données), ou être implicites (les informations de style pour Latex ou MSWord).

A la lumière de l'IDM, on voie rapidement que ces programmes définissent, implicitement, des domaines fonctionnels (serveur web, base de données, éditeur de texte) et que les informations de configuration constituent un modèle qui définit une application particulière dans ce domaine. Le "programme" peut alors être vu comme un interpréteur du modèle tel que le couple interpréteur + modèle soit une application particulière du domaine.

Dans ce travail, nous généralisons cette approche en définissant un concept de domaine plus général, et en nous appuyant sur une vision IDM plus rigoureuse. Nous considérons d'une part que les domaines sont décrits par des métamodèles auxquels les modèles sont conformes ; et d'autre part nous proposons une façon systématique de dériver l'interpréter à partir du métamodèle.

Pour ce qui est de la composition, la technologie habituelle consiste à utiliser des appels de méthode entre les parties réutilisables. Le client doit alors connaître explicitement l'API du serveur ce qui introduit une dépendance. En nous appuyant sur une vision IDM, nous considérons que les relations entre parties ne sont pas des relations entre programmes (appels de méthodes) mais des relations entre modèles (composition de modèles).

Pour que cette vision soit réaliste en pratique, il faut respecter certaines propriétés "habituelles" ; par exemple pour (ré)utiliser un serveur Apache, une base de données ou un texte Word existants, il n'est nul besoin de modifier le programme (Apache, Oracle, MSWord), ni même sa configuration. En terme IDM, la composition doit se faire sans modifier le métamodèle, l'interpréteur ni même les modèles à composer. La réutilisation est à ce prix.

Nous souhaitons aller plus loin et éliminer la dépendance entre client et serveur. Pour ce faire, le domaine "client" est vu comme un domaine autonome qui ne connaît pas le ou les domaines "serveur". C'est la composition des domaines qui exprime la dépendance entre les domaines. De ce fait un domaine peut être composé avec d'autres domaines, à posteriori, et de diverses manières sans avoir à modifier le code de son interpréteur, ni ses modèles. Par exemple une application devrait pouvoir être composée avec d'autres applications (serveurs web, bases de données ou éditeurs) sans aucun impact sur son code et son fonctionnement. Le choix des applications "serveur" devrait pouvoir être modifié à tout moment (autre serveur, autre bases de données etc.) toujours sans impact sur le client.

Des initiatives existent qui vont dans ce sens, mais dans des domaines fonctionnels limités, comme Windows OLE [1] qui permet de faire interopérer diverses application Windows (e.g. Word et Excel) ou externes (e.g. Word et EndNotes) sans que ces applications se connaissent explicitement, sans modifier les programmes, et sans modifier les modèles. Nous proposons une démarche, des concepts et des outils qui généralisent cette approche, basé sur les domaines, la composition de métamodèles, et la composition de modèles.

Notre démarche est une approche de réutilisation de niveau élevé. Le domaine est l'élément de composition de gros grain et de haut niveau d'abstraction. Réutiliser les domaines permet de réutiliser non seulement une application mais une famille d'applications. De plus, le développement basé sur des éléments de gros grains permet de manipuler le système au travers de peu d'entités ce qui simplifie le travail de développement et rend le système plus compréhensible.

1.2 Objectifs de la thèse

La démarche décrite ci-dessus, dans ses grandes lignes, a été initiée au début des années 2000, et à donné lieu à plusieursthèses. La décomposition et la formalisation des domaines ont été résolu dans [Vil03][Le04][Veg05]. La technique de composition a initialisé mais la réalisation est restée ad-hoc. Cette thèse continue ces traveaux sur l'axe composition. Nous proposons une approche de composition de modèles dont les principaux objectifs sont

- Un langage de composition de modèles et
- Les outils de composition de modèles assistant le développeur pour construire des applications basées sur la composition de modèles.

1.3 Contribution de la thèse

Les contributions de la thèse peuvent être résumés dans les 4 points suivants :

- Une étude des mécanismes de composition existants en Génie Logiciel .
- Une étude des mécanismes de composition de modèles existants.
- La proposition d'un langage de composition de modèles.
- Un environement supportant la composition de modèles.

1. Object Linking and Embedding.

1.4 Plan de la thèse

Le manuscrits contient deux grandes parties : l'état de l'art et la contribution. L'état de l'art se divise 4 chapitres :

- Chapitre 2 : présente un survol rapide des approches de composition dans l'ingénierie du logiciel. Nous étudions 4 approches du Génie Logiciel : l'approche modulaire, l'approche de description architecturale, l'approche à base de composants et la programmation orientée aspects.
- Chapitre 3 : étudie le modèle et l'ingénierie des modèles. L'objectif est de construire un vocabulaire IDM utilisé fréquemment dans ce document.
- Chapitre 4 : est une étude de l'état de l'art dans la composition des modèles.
- Chapitre 5 : présente de la démarche abordée ci-dessus, c'est le contexte de notre travail. Mélusine est une démarche et en même temps une architecture pour formaliser les domaines et un environement pour concevoir et exécuter les domaines.

La contribution a 2 chapitres :

- Chapitre 6 : présente notre approche de composition de modèle et la proposition d'un langage de composition de modèle appelé Codèle.
- Chapitre 7 : décrit un environement réalisant cette proposition.

Chapitre 2

Composition en Génie Logiciel

2.1 Introduction

La composition est un concept fondamental du Génie Logiciel. Il concerne la construction des systèmes à partir d'unités logicielles séparées. Ce concept n'est pas nouveau. Il est utilisé dans plusieurs approches du Génie Logiciel :

- **L'approche modulaire** : née au début des années 70, le concept de *module* et la composition des modules sont les principales entités dans la construction des logiciels. Cette approche a donné lieu aux Langages d'Interconnection de Modules (MILs [1]).
- **L'approche architecturale** : née au début des années 90, elle utilise la composition comme discipline d'architecture logicielle. Les Langages de Description d'Architecture (ADLs [2]) permettent de concevoir des systèmes en terme des *briques architecturales* composées.
- **L'approche à base de composants (CBSE [3])** : émerge au milieu des années 90. Les systèmes sont construits par l'assemblage des briques logicielles (appelées *composant*) préfabriquées (probablement déjà déployées) au travers de la connexion (statique ou dynamique) de leurs interfaces sans avoir besoin d'accéder au détail de leurs implémentations lors de la composition.
- **La programmation orientée aspects (AOP [4])** : apparait à la fin des années 90. Elle introduit un nouveau concept appelé *aspect* représentant les préoccupations transversales d'un système global. La construction du système consiste à composer des aspects dans un programme noyau contenant le code métier essentiel du système.

Ce chapitre étudie la composition dans les approches de Génie Logiciel. L'objectif n'est pas de faire un panorama complet de toutes les approches de Génie Logiciel utilisant la composition, ni une étude exhaustive de chaque approche mais simplement de réviser certaines approches entre eux, sous la perspective de composition pour pouvoir savoir comment ce concept a été utilisé en Génie Logiciel. Les leçons retenues sont intéressantes pour discuter de la composition de modèles dans les prochains chapitres.

Ce chapitre va parcourir 4 approches qui ont eu de grands impacts en Génie Logiciel. Avant d'avancer dans les approches, nous voulons clarifier quelques les critères selon lesquels nous allons les étudier. Ces critères concernent seulement la composition, ils nous permettent d'ignorer les autres détails moins pertinents de l'approche. Les critères proposés sont suivants :

- Grain de composition.

1. Module Interconnection Languages.
2. Architecture Description Languages.
3. Component-Based Software Engineering.
4. Aspect-Oriented Programming.

- Protocole de communication.
- Mécanisme de composition.

La section 2.2 explique les critères. La section 2.3 présente la notion de module et la composition par modules. Les langages d'interconnexion de modules sont également étudiés dans cette section. La section 2.4 étudie les langages de description d'architecture (ADLs) et l'approche de description architecturale. La section 2.5 présente la notion de composant et la composition de composants. La section 2.6 se concentre sur l'AOP et la composition d'aspects. La section 2.7 fait un bilan sur les approches.

2.2 Composition des logiciels

2.2.1 Grain de composition

La composition est toujours inhérente à la phase de décomposition précédente. La décomposition permet de diviser un système en des entités que nous appelons *grain de composition*.

La nature des grains dépend de la vision de modularisation de chaque approche, e.g, en fonction de la frontière qui sépare naturellement les objets de la réalité (modularisation orientée-objet), ou en fonction des préoccupations (modularisation orientée-aspect), des fonctionnalités fournies (modularisation orientée-modulaire, -composant, -service), ou de la vision des acteurs qui participent dans le système (modularisation orientée-vue) etc.

Chaque approche formalise les grains par ses propres notions de modularisation (e.g, *module*, *classe*, *composant*, *service* etc.) ; dans une approche, il peut exister plusieurs vocabulaires pour la même notion, par exemple, *composant*, *bean*, *bundle* sont les différents vocabulaires du grain de composition dans CBSE.

Propriétés du grain de composition Ses deux premières propriétés sont *composables* et *réutilisables*. Outre ceux-ci, les grains sont associés aussi aux propriétés suivantes :

- **Granularité** : Les grains pourraient être très *fins*(e.g., les objets) ou très *gros* (e.g., les applications, les sous-systèmes). Le degré de granularité dépend du niveau d'abstraction du grain. Souvent plus celui ci est gros et plus le composant est abstrait. Il peut englober un grand nombre des ressources et des implémentations complexes mais exhiber seulement des interfaces ou des descriptions spécifiant succinctement ses fonctionnalités.
- **Cohésion** : C'est la propriété mesurant le degré de proximité fonctionnelle des opérations exposées par l'unité. Les grains *cohésifs* peuvent se focaliser rigoureusement sur l'implantation d'une logique métier. Au contraire, les grains *moins cohésifs* sont moins liés à une fonctionnalité précise, leurs opérations sont plus hétérogènes.
- **Couplage** : C'est la propriété mesurant le dégré d'attachement de deux unités. Souvent on parle de *couplage fort* et le *couplage faible*. Dans le couplage forte, il est difficile à comprendre les unités en isolation ; changer une unité va forcer de changer toutes les unités associées ; la réutilisation de ces unités est difficile. En revanche, le couplage faible permet de surmonter tous ces inconvénients.
- **Encapsulation** : Cette propriété concerne la modularisation *basée sur le masquage d'information* proposée par Parnas au début des années 70 [Par72]. L'idée de masquage d'information est de cacher les décisions de conception susceptibles de changer pour pouvoir protéger les autres parties du programme lorsqu'une modification doit être réalisée. La modularisation basée sur le masquage d'information (encapsulation) consiste à isoler les décisions de conception dans des modules séparés. Les autres parties du système reconnaissent l'existence d'un module seulement grâce à des descriptions qui exposent ses éléments stables et

ignorent complètement son implémentation qui rassemble les détails susceptibles de changer.

- **Abstraction** : L'abstraction consiste à retenir uniquement les informations pertinentes dans un but particulier en ignorant les détails moins pertinents. Les unités n'exposent pas tous leurs détails mais seulement les parties qu'ils veulent publier. Dans plupart des cas, le grains de composition, correspond aux éléments publiés dans leurs interfaces. Les grains plus gros sont souvent plus abstraits.

Les caractéristiques attendues d'un grain de composition

- gros grain : La tendance actuelle est d'utiliser des grains plus gros pour pouvoir manipuler le système au niveau d'abstraction élevé avec moins d'entités à manipuler.
- grande cohésion : Les grains fortement cohésifs favorisent la compréhension, la réutilisabilité, la maintenance de système.
- couplage faible .
- avoir une interface.
- avoir un haut niveau d'abstraction.

2.2.2 Protocoles de communication

Les grains ont besoin de communiquer pour travailler ensemble. La communication concerne généralement l'envoie et la réception de données, d'événements, ou de messages. Dans [SG94], Shaw et Garlan ont présenté certains protocoles de communication typiques :

- **Appel de procédure** : Les flots de contrôle passent entre les entités qui se connaissent (e.g, appel de procédure local (dans un même espace de nom) ou appel de procédure à distance (entre différents espaces de nom)).
- **Flots de données** : Les processus indépendants interagissent au travers des flux de données (e.g, pipes et sockets Unix). La disponibilité des données provoque le calcul.
- **Déclenchement implicite (*triggering implicit*)** : L'occurrence d'un événement provoque le calcul bien qu'il n'existe pas d'interaction explicite entre les unités (e.g, système d'événement basé sur le patron observable/observateur, ramasse-miettes automatique (*automatic garbage collection*)).
- **Diffusion de message (*message passing*)** : Les processus indépendants interagissent par la diffusion discrète et explicite des données. La diffusion peut être synchrone ou asynchrone (e.g, TCP/IP).
- **Données partagées** : Les composants travaillent en concurrence dans un même espace de données (e.g, système à tableau noir, bases de donnée multi-utilisateurs).

2.2.3 Mécanisme de composition

C'est la façon d'assembler des grains de composition. Il faut distinguer ce critère de celui de protocole de communication. Ce dernier concerne l'interaction entre deux grains alors que le premier concerne la structure d'assemblage des grains. Les mécanismes de composition peuvent être classés selon le type de réutilisation : boîte blanche ou boîte noire.

Quelques mécanismes de composition de boîte blanche sont :

- **Héritage** : La sous classe est une composition entre une partie de données et de comportements de sa super classe et ses propres données et comportements.
- **Composition invasive** [5] : les données et les comportements étendus sont introduits dans un composant à certains points de tissage(*weaving point*) via une transformation.

5. Invasive Composition.

- **Super-imposition**[6] : La structure des composants logiciels sont fusionnées [AL08]
- **Méta-programmation**[7] : Deux programmes peuvent être composés via les manipulations d'un méta-programme. Un méta-programme est un programme capable de lire, manipuler et écrire les codes des programmes. Le méta-programme fonctionne au niveau des packages, des classes, des attributs, des expressions, des instructions etc. Ses opérations typiques sont l'ajout, la suppression, le déplacement, le remplacement de codes de programme.

A propos de la composition de type boîte noire, la philosophie commune est de relier les interfaces des composants. Les mécanismes proposés pour exprimer la structure d'assemblage des grains sont :

- Utilisation des langages ADLs.
- Utilisation des contrôles externes tels que la chorégraphie ou l'orchestration.
- Utilisation des langages de *script* dédiés à la composition. Ces derniers permettent d'exprimer les compositions à un niveau d'abstraction élevé.
- Utilisation des langages de programmation.

Notons que les listes ci-dessus ne sont pas exhaustives.

2.3 L'approche modulaire

2.3.1 Grain de composition

Le *module* est le grain de composition. Du point de vue conceptuel, un module est défini comme le responsable d'une tâche (*responsibility assignment*) [Par72]. La construction d'un système consiste à mettre ensemble des modules qui remplissent différentes tâches. Un module est caractérisé par les points suivants :

- un module est associé à un ensemble d'interfaces : ses interfaces exposent les éléments (*resources*) fournis et requis par le module. Ces ressources pourraient être les variables globales ou les procédures avec les paramètres et sans l'implémentation.
- un module a une partie d'implémentation qui est un ensemble de sous programmes et de structures de données qui sont accessibles au travers des interfaces.
- un module peut être compilé séparément : ceci permet le travail en parallèle et permet de remplacer facilement un module par un autre dans un système.

2.3.2 Protocole de communication

Généralement, un module communique avec un autre au travers des appels de procédure et de l'accès aux variables globales déclarées dans les interfaces de l'autre.

2.3.3 Mécanisme de composition

L'idée de l'approche modulaire est *l'assemblage de modules* au travers de leurs interfaces. Ce travail a donné lieu aux langages d'interconnexion de modules (MILs).

Un langage d'interconnexion de modules permet de décrire de manière formelle la structure du système logiciel en terme des spécifications d'interconnexions de modules ("*MIL code listings*"). Une spécification d'interconnexions de modules définit quels sont les modules participants à un système et comment ces modules coopèrent, i.e. la structure du système. Ils ne définissent pas la fonctionnalité du système ni l'implémentation de ces fonctionnalités i.e. l'implémentation des modules.

6. Super-imposition.
7. Meta-programming.

Les langages d'interconnexion de modules viennent de l'idée de la programmation globale ("in-the-large") et la programmation détaillé ("in-the-small")[DK76] à la fin des années 70. La programmation détaillée concerne la construction des modules individuels tandis que la programmation globale concerne la structuration des modules dont les MILs sont issus.

Ci-dessous est un exemple du code MIL extrait de [PDN86]. Il présente 4 primitives basiques des MILs : *provide*, *require*, *has-access-to* and *consists-of*. Ces primitives peuvent être précédées du qualificatif *must* pour indiquer des ressources optionnelles et celles obligatoires.

```
module ABC                                  Nom du module
    provides a,b,c                          Ressources fournies
    requires x,y                            Ressources requises
    consist-of function XA, module YBC      Constituants

    function XA
        must-provide a
        requires x
        has-access-to module Z
        real x, integer a
    end XA

    module YBC                              Module imbriqué
        must-provide b,c
        requires a,y
        real y, integer a,b,c
    end YBC

end ABC
```

FIGURE 2.1 – Exemple de MIL [PDN86]

Lorsqu'un système a été bien analysé, évalué et conçu, la spécification MIL peut être écrite. Elle nous permet de vérifier la complétude du système avant de l'implémenter. Pendant l'implémentation, elle doit être maintenue et employée éventuellement pour la maintenance du système.

En résumé, la composition de modules par les MILs possède des caractéristiques suivantes :

- L'assemblage de modules par le liaison statique d'interfaces (*"inteface static binding"*).
- S'effectue après la phase de conception des modules.
- Les spécifications MIL modélisent la composition de modules.

Les exemples des langages MILs sont MIL-75, INTERCOL, Thomas's MIL, Cooprider's MIL [8]. Les exemples des langages supportant le concept de module sont C, Ada, Modula-2 etc. Ces derniers ne sont pas les MILs mais ils ont des supports comme MILs permettant la composition des modules définis dans ces langages.

Ci-dessous nous présentons Ada et MIL-75, le premier est l'exemple pour les langages supportant le concept de module et le second est l'exemple pour les langages MILs.

Ada Ada est un langage de programmation fournissant du support pour le concept de module. Dans Ada, les modules sont représentés par la notion de *package* (**grain de composition**). Un package Ada comportent deux parties : une spécification de package dont le rôle est est identique à celui d'une interface et un corps du package qui contient la partie d'implémentation. La spécification de package est une liste de déclarations de fonctions, de procédures et de types de données

8. cités depuis [PDN86].

du package dont les implémentation sont définies dans le corps du package. Le package peut utiliser un autre package par déclarer explicitement dans son interface l'importation du package qu'il veut avec l'aide de clause *with* (**mécanisme de composition**). Les procédures Ada peuvent être invoquées mutuellement par des appels de procédure (**protocoles de communication**).

MIL-75 MIL-75 est le premier langage d'interconnection de module développé par DeRemer et Kron [DK76]. L'idée au début est de décomposer le système sous forme d'un arbre dont chaque *noeud* est un module (**grain de composition**). Les noeuds sont connectés par des *arcs* (**mécanisme de composition**). Un arc relie un noeud parent et ses fils. Le noeud parent est représenté par la notion de *system* tandis que leurs fils sont représentés par la notion *subsytem*. MIL-75 permet aux modules de communiquer par l'accès à leus resources sous le contrôle des règles d'accès, par exemple, deux modules de frères et soeurs peuvent accéder aux resources de l'autre s'il y a un lien d'accès (*accessibility link*) qui les relie (**protocoles de communication**).

2.3.4 Synthèse

L'approche modulaire permet de construire les systèmes par l'assemblage des modules. L'idée est de connecter ces modules à travers leurs interfaces. Ce travail a donné lieu les langages MILs qui permettent de décrire la structure des systèmes en terme des spécifications MIL statiques. Ces dernières peuvent éventuellement être utilisées pour l'implantation du système.

Points positifs L'intérêt principal de l'approche modulaire est de faciliter la construction de systèmes en permettant 1) d'écrire le module avec peu de connaissance sur le code d'autres modules et 2) de remplacer un (ou plusieurs) module(s) sans re-assembler tout le système [Par72]. La construction du système est donc plus compréhensible, gérable, et maintenable [Par72].

Points négatifs Une des limites de l'approche est de ne pas permettre de personnaliser un module, c'est-à-dire ils ne permettent pas de faire des variants d'un module existant [Cer04].

En ce qui concernent les langages MILs, il ont comme principaux défauts d'être limités en ce qui concerne les vérifications possibles, et de devoir être maintenus à la main durant toute la vie du système. Le rapport qualité/prix (vérification/effort de maintenance du MIL) est jugé trop faible par les industriels. Pour cette raison, la génération suivante essaye d'améliorer les vérifications et validations permises par le langage, ce furent les ADL.

2.4 L'approche de Description Architecturale

L'approche de Description Architecturale est le successeur de l'approche modulaire. Il se concentre sur la modélisation de l'architecture d'un logiciel en terme d'un "*...abstract system specification consisting primarily of functional components described in terms of their behaviours and interfaces and component-component interconnections*" [Hayes-Roth, 1994] [9].

Dans l'approche de Description Architecturale, les architectures sont exprimées par les langages de description d'architecture (ADL [10]). Un ADL fournit des notations formelles permettant de spécifier les briques architecturales. Une brique architecturale est une unité logicielle conceptuelle qui représente des parties d'un système indépendamment de leur implémentation. Le système est construit par l'assemblage de ces briques. Ceci permettra alors la conception d'applications en se détachant des détails techniques propres à l'environnement.

9. citée dans le site web http://www.sei.cmu.edu/architecture/published_definitions.html
10. Architecture Description Language.

2.4.1 Grain de composition

Les briques architecturales sont les grains de composition de cette approche. En général, les ADLs acceptent trois notions formelles de briques architecturales suivantes : *composant*, *connecteur*, et *configuration* [MT00].

- **Composant** : est une unité de calcul ou un entrepôt de données (par exemple un client, un serveur, une base de données, etc.) Il expose des interfaces qui spécifient les services (messages, opérations et variables) qu'il fournit.
- **Connecteur** : modélise les interactions entre les composants à travers leurs interfaces ainsi que les règles qui gouvernent ces interactions.
- **Configuration** : représente un graphe de composants connectés au travers des connecteurs.

Cependant, comme le terme "Langage de Description d'Architecture" couvre une famille de langages vaste [11], et dans la réalité, les ADLs sont élaborés de façon ad-hoc pour répondre à des besoins particuliers, donc chaque ADL définit ces trois notions de sa propre manière. Par exemple, dans Darwin [MK96], les composants sont représentés sous forme de *services*, tandis que dans Rapide [12] [Luc97], ce sont des *modules*. L'existence de ces trois notions n'est pas forcément explicite. Par exemple UniCon [13] [SDK+95a] ou Darwin [MK96] n'ont pas le concept explicite de *configuration*, mais considèrent la configuration comme étant des *composants composites* ; d'autres concepts, par exemple le *connecteur*, n'est pas explicite dans Darwin et Rapide, ils sont considérés comme des flux causals d'événements et des appels de fonction (dans Rapide) ou la liaison de services (*service binding*) (dans Darwin).

Certains ADLs ajoutent en plus leurs propres notions. Par exemple, ACME [14] [GMW00] ajoute les concepts : *système*, *port*, *rôle*, *représentation* et *rep-maps* (*representation map*). Aesop [15] [Gar95, GAO94] ajoute des concepts : *port*, *rôle*, *représentation* et *associations* (*binding*). Ces concepts répondent à des besoins particuliers.

Les composants et les connecteurs peuvent avoir des types. Ceci donne la notion de *style d'architecture*. Un style d'architecture est la définition d'architecture d'une famille de logiciels qui partagent un ensemble des propriétés communes, par exemple les systèmes de réservation de billets d'avion en ligne, les systèmes de gestion de projets de l'entreprise etc. Le style d'architecture fournit un vocabulaire commun pour une famille d'application en définissant des types de composants, de connecteurs, des propriétés et des contraintes partagées par toutes les configurations (i.e. applications) appartenant à ce style. A titre d'exemple, C2 [16], Aesop sont les ADLs fournissant la notion de style d'architecture.

2.4.2 Protocole de communication

Les protocoles de communication souvent utilisés dans les ADLs sont : appels de méthode (RPC [17], RMI [18] etc.), flots de données (pipes et sockets), liaison de services, diffusion d'événement etc.

Dans les ADLs utilisant le concept de connecteur, ceux-ci sont chargés de décrire la communication entre les composants. Le protocole de communication est défini par le type de connecteur. Par exemple, les connecteurs de type RPC utilise le protocole RPC. Un ADL peut supporter

11. un Langage de Description d'Architecture peut être considéré comme le langage de modélisation d'Architecture Logicielle dédié à un domaine spécifique.
12. http://pavg.stanford.edu/rapide/
13. http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/index.html
14. http://www.cs.cmu.edu/ãcme
15. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html
16. http://www.ics.uci.edu/pub/arch/
17. Remote Procedure Call.
18. Remote Method Invocation.

plusieurs types de connecteur différents (e.g., UniCon, Olan). Il est capable aussi de définir de nouveaux types de connecteur (e.g., Wright).

2.4.3 Mécanisme de composition

Les composants sont connectés soit par les connecteurs, soit par la liaison d'interface directe. Dans le premier cas, le composant fournit et/ou requiert un ou plusieurs ports. Le connecteur relie les ports des composants. L'exemple de ce type de composition est Wright, Unicon, ACME. Dans le deuxième cas, le composant fournit et/ou requiert une ou plusieurs interfaces. Une liaison est un chemin de communication entre une interface fournie et une interface requise. L'exemple de ce type de composition est Rapide et Darwin.

Wright Wright est un ADL pour usage général (*general-purpose ADL*) développé à Carnegie Mellon University par Allen et Garlan [All97]. Ce langage utilise trois notions architecturales classiques : *composants*, *connecteurs*, et *configurations* (**grain**). Un point particulier de Wright par rapport des autres ADLs est qu'il permet à l'utilisateur de spécifier formellement ses protocoles de communication entre ses composants. Ces protocoles sont formalisés en terme de ses types de connecteur. Ces derniers sont décrits par un langage formel à la CSP (CSP [19] est une notation algébrique permettant de spécifier les processus d'interaction [HH85]). En fonction des types de connecteur différents, les composants auront des comportements de communication différents (par exemple du style *pipe-filtre* ou *client-serveur* etc.)(**protocole de communication**).

Le mécanisme de composition dans Wright est celui classique des ADLs : les composants sont connectés par des connecteurs et tous sont encapsulés dans la configuration (**mécanisme de composition**). La configuration est une collection d'instances de composants combinés à travers leurs connecteurs.

Unicon Unicon est un ADL pour usage général (*general-purpose ADL*) développé par Shaw et al [SDK+95b]. Dans Unicon, les systèmes logiciels sont décrits par deux types de briques architecturales : *composants* et *connecteurs* (**grain**). Sa particularité est la capacité de générer le glue code à partir de la spécification des connecteurs. Contrairement à Wright, Unicon ne supporte qu'un ensemble de types de connecteur primitifs prédéfinis. Cette collection n'est pas facilement extensible [20]. Donc, en général, les protocoles de communication de Unicon se limitent à quelques types atomiques [21] tels que *pipe*, *FileIO*, *ProcedureCall*, *RemoteProcCall* etc. (**protocole de communication**).

Les composants dans Unicon sont connectés par des connecteurs (**mécanisme de composition**). Unicon n'a pas le concept de configuration explicite comme Wright, par contre le composant composite est considéré comme la configuration. La configuration décrit la combinaison des instances de composants et de connecteurs à l'aide des instructions *uses*, *connect*, *bind*, *establish*.

2.4.4 Synthèse

L'approche de Description Architecturale se concentre sur la conception de systèmes par assemblage de briques architecturales. Trois notions de briques architecturales acceptées populairement sont *composant*, *connecteur* et *configuration*. Elles sont fournies par les ADLs. Le mé-

19. Communicating Sequential Processes.
20. Unicon fournit des supports très limités pour définir de nouveaux types de connecteur, qui doivent être implémentés à la main par des experts.
21. Il y a sept types de connecteur prédéfinis dans Unicon. Leurs détails sont dans [Zel96].

canisme de composition est par connecteur ou par la liaison d'interface directe. Les connecteurs et la liaison expriment la communication entre les composants.

Points positifs L'intérêt majeur des ADLs se situe principalement lors de l'analyse des systèmes [BFCD+06]. Les briques architecturales modélisent des parties d'un système de manière abstraite sans définir leur implémentations donc peuvent être utilisées dès la phase de l'analyse du système pour créer la description d'architecture. Cette description peut permettre par la suite de faire des simulations relatives au système à construire ou bien éventuellement elle permet à d'autres personnes d'analyser le système. L'analyse d'un système grâce à sa description d'architecture est très pratique lorsqu'elle donne une vue globale du système et permet de raisonner sur le système à un niveau d'abstraction supérieur à celui qui peut être obtenu à partir de la lecture directe du code [Cer04, MT00]. La description d'architecture peut être utilisé ensuite dans la construction, le déploiement, ou la gestion de configuration du système [MT00].

Points négatifs La diversité des ADLs est un inconvénient grave car elle a créé des problèmes d'incompatibilité, de portabilité, de multiplicité, de complexité, d'absence de standardisation [Mar04]. Les ADL récents comme ACME ont essayé de résoudre le problème d'incompatibilité en permettant de faire la traduction entre les ADLs. Cependant, cette traduction dépend de l'existence d'outils "traducteurs" entre ces ADLs.

Un autre problème des ADLs est la standardisation. Pendant longtemps, les ADLs ont souffert d'être une approche très académique, peu utilisée par les industriels du génie logiciel [Bar05]. Des travaux récents tels que le langage UML [22] de l'OMG [23] a montré le besoin pour les industriels de disposer d'un langage de description d'architecture standard [Bar05].

Le destin des ADLs fut similaire à celui des MILs, et ce pour les mêmes raisons. Le rapport qualité/prix n'est pas suffisant : les vérifications permises ne compensent pas l'effort de conception et surtout de maintenance de la description architecturale tout au long de la vie d'un système.

2.5 L'ingénierie logicielle à base de composants

L'ingénierie logicielle à base de composants (CBSE) s'appuie sur la construction des systèmes complexes par l'intégration de composants logiciels préfabriqués au lieu de développer "from scratch". Le principe de cette approche est simple : "*reuse but do not reinvent (the wheel)*" [Wan00]. "*Components are for composition. Nomen est omen*" [Szy98].

2.5.1 Grain de composition

Le grain de composition est les composants. Cependant, il n'existe pas de consensus sur la définition de ce concept. Plusieurs définitions étaient proposées. On peut en citer quelques unes :

> *"A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction." [Boo87]*
>
> *"Un composant (de composition) est un artefact qui permet de grouper et d'isoler un sous graphe du modèle d'objets, en définissant de façon explicite ses responsabilités et ses besoins par rapport au reste de l'application, ce qui lui permet d'évoluer de manière indépendante."* [Vil03]

22. Unified Modeling Language.
23. Object Management Group.

> *"A software component is a unit of composition with contractually specified and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition."*[Szy98]

Ces définitions concordent sur le fait qu'un composant est une unité de composition qui interagit avec l'extérieur en définissant de manière abstrait les services qu'elle offre et ceux qu'elle requiert. Pour certaines approches, le composant est caractérisé par les unités déployées (voir la définition de Szyperski).

Le terme de composant est en fait utilisé pour parler à la fois des instances et des classes [BFCD+06, Cer04]. Une classe de composant peut être vue sous deux visions : interne et externe [Cer04]. La vue externe se compose d'un ensemble des interfaces fonctionelles fournies et requises, et des propriétés de configuration qui sont des valeurs initiales qui servent à configurer l'instance de composant lors de sa création à l'exécution [Cer04]. La vue interne se compose d'un ensemble d'interfaces de contrôle, de dépendances et propriétés de déploiement. L'interface de contrôle contient des méthodes permettant de gérer le cycle de vie des instances du composant pendant l'exécution. Elles sont normalement appelées par l'environnement d'exécution, et non pas par les clients du composant. Les dépendances définissent les conditions qui doivent être satisfaites au déploiement par exemple une dépendance envers une version particulière de l'environnement d'exécution. Les propriétés de déploiement sont des variables communes de la classe de composant qui sont utilisées pour configurer des caractéristiques communes à toutes les instances [Cer04].

La composition des composants dépend du *modèle à composants*. Un modèle à composants est une infrastructure qui "*consiste en un ensemble de conventions à respecter dans la construction et l'utilisation des composants.*" [Le04]. Parmi les modèles à composants représentatifs, on peut citer CCM[24] de OMG [OMG06], COM[25] de Microsoft [Box97], Java Beans et Enterprise Java Beans de Sun [BMH06], Fractal[26], OSGi[27] etc. Ces modèles fournissent les notions permettant d'exprimer les connexions entre les composants. Les concepts pris en charge sont généralement les *connecteurs*, les *conteneurs*, et les *structures d'accueil*. Le concept de connecteur est similaire celui dans les ADLs. Le conteneur encapsule les composants, et prend en charge des services non fonctionnels du système tels que la sécurité, la persistance, la transaction etc. La structure d'accueil prend en charge le déploiement et l'exécution des composants. Cependant, dans la réalité, en raison de la multiplicité des modèles à composants, ces concepts (et même le concept de composant) ont plusieurs vocabulaires différents. Par exemple, un conteneur peut être une *membrane*, un *contrôleur* etc. Une structure d'accueil peut être un *framework*, un *serveur d'application* etc. Un *composant* peut être, un *bundle* OSGi, un *bean* EJB etc.

2.5.2 Protocole de communication

Ceci dépend des technologies de composant, par exemple, CORBA ou DCOM utilisent RPC, EJB utilise RMI, Web Services utilise SOAP-RPC[28] etc.

2.5.3 Mécanisme de composition

Cervantes [Cer04] a identifié 4 types de composition des instances de composant à partir des éléments présents sur leur vue externe :

24. CORBA Component Model.
25. Component Object Model.
26. http://fractal.objectweb.org/
27. http://www.osgi.org
28. un type de protocole RPC basé sur XML. Les appels de méthode entre les composants sont sérialisés/transférés/désérialisés sous forme de représentations XML.

- Composition visuelle : l'environnement permet d'assembler des instances de composants de façon interactive, par exemple Bean Builder [29] de Sun.
- Composition déclarative : par utilisation de langages décrivant les relations entre des concepts propres à la composition, tels que "composant" ou "connecteur" (i.e. l'utilisation des ADLs), par exemple FractalADL [30] de Fractal.
- Composition impérative par langage de programmation : l'utilisation d'un langage de programmation "classique", tel que Java, pour réaliser la composition des composants.
- Composition impérative par langage de script : dans ce type de composition, un langage (e.g. CorbaScript [31] de Corba ou Visual Basic Script [32] de COM) pouvant contenir des concepts spécifiques à la composition, et donc de plus haut niveau d'abstraction, est utilisé.

La composition visuelle et déclarative souvent donnent lieu aux assemblages de composants statiques alors que la composition impérative permet de réaliser les assemblages dynamiques (i.e. il permet la création et destruction d'instances et de connexions pendant l'exécution). Ceci signifique que ces types de composition distinguent le moment de la composition : lors de l'analyse (approche déclarative), lors de la programmation (visuelle) et lors de l'exécution(approche impérative).

Fractal Fractal est un modèle à composants générique, extensible pour la construction des systèmes distribués reconfigurables et faciles à adapter. Il est développé par France Telecom et INRIA, France depuis 2001 [CS06].

Fractal utilise les concepts classiques de CBSE : *composant*, *interface* et *liaison* (*binding*) [CS06] (**grain de composition**).

Un composant est divisé en deux parties : une *membrane* et un *contenu*. Une membrane est un ensemble de *contrôleurs* chargés de contrôler le contenu (e.g., interception des messages, modification des paramètres des messages, etc.).

L'interface du composant est divisée en deux types : requise (ou client) et fournie (ou serveur).

La liaison est le chemin de communication entre les interfaces. Son implémentation doit se baser sur certains protocoles de communication . Cependant, le modèle de Fractal n'impose pas l'implémentation de ces liaisons (**protocoles de communication**). Ceci dépend des approches qui implémentent le modèle Fractal. Actuellement, il y a des outils tels que Fractal RMI [33] qui fournit une usine de liaison (*binding factory*) permettant de créer les liaisons synchrones distribuées entre les composants Fractal basées sur Java RMI [CS06].

La composition des composants dans Fractal est la liaison des interfaces. Ceci peut être réalisé de façon déclarative à travers le langage Fractal ADL [34] [Cer04]. La composition visuelle est supportée dans un outil appelé FractalGUI [35] [Cer04] (**mécanisme de composition**).

JavaBeans JavaBeans [36] est un modèle à composants pour développer les applications par l'assemblage visuel de composants Java. Il est développé par Sun dont l'objectif était de développer une technologie de composition visuelle des composants Java similaire à VisualBasic de Microsoft.

Un composant JavaBean, appelé un *bean* (**grain de composition**), est une classe Java dont l'implémentation est écrite en se basant sur certaines conventions par exemple un bean doit implémenter l'interface de sérialisation, il doit avoir des méthodes d'accès *get/set*, son constructeur

29. https://bean-builder.dev.java.net/guide/tutorial.html
30. http://fractal.objectweb.org/tutorials/adl/index.html
31. http://sourceforge.net/projects/wcs/
32. http://msdn.microsoft.com/en-us/library/t0aew7h6.aspx
33. http://fractal.objectweb.org/fractalrmi/index.html
34. http://fractal.objectweb.org/tutorials/adl/index.html
35. http ://fractal.objectweb.org/current/doc/javadoc/fractal-gui/overview-summary.html
36. http://java.sun.com/docs/books/tutorial/javabeans/

n'a pas des paramètres (pour qu'il puisse instancier lui même) etc. Les beans peuvent communiquer au travers d'un modèle de communication par événement (patron émetteur/écouteur) (**protocole de communication**).

La composition des *beans* est principalement réalisée de façon visuelle dans un environnement dédié (Bean Builder [37]) [Cer04]. Cet environement expose un catalogue de composants Java, et permet l'utilisateur d'instancier ces composants, configurer et mettre ensemble des instances de ces composants. Cependant, le modèle à composants JavaBeans ne spécifie pas la manière de conditionner un assemblage. La composition peut aussi être réalisée de façon impérative (i.e. l'utilisateur écrit le code pour instancier, configurer et connecter ses instances de composants) (**mécanisme de composition**).

2.5.4 Synthèse

L'approche à base de composants promeut fortement la notion de composition. Les composants sont les grains de composition. Les applications sont construites par assemblage de composants au travers de leurs interfaces. Il y a deux types d'assemblage soit par liaisons directes entre les interfaces, soit par des connecteurs qui les relient indirectement. L'assemblage des composants peut être réalisé de plusieurs façons : par les ADLs, par les langages de script, par les langages de programmation ou par les outils visuels.

Points positifs Le principal intérêt de l'approche à base de composants est la réutilisation élevée. Ceci permet de réduire considérablement le temps de développement d'application.

Points négatifs Le fait que le concept de composant ne soit pas clairement défini a pour conséquence de rendre vague le concept de composition. Ceci rend difficile la définition de mécanismes de composition standards efficaces. De plus, Cervantes [Cer04] a noté que la composition de composants est une activité fondamentale et importante dans l'approche à composant mais qu'il n'y a pas beaucoup d'efforts de recherche permettant de réaliser la composition, comparée aux efforts pour définir de nouveaux modèles à composants. Ceux-ci sont les facteurs qui limitent l'amélioration de la productivité et de la réutilisation de l'approche.

2.6 La Programmation Orientée Aspects

Dans une application orientée objet, il est fréquent que les fonctionnalités de l'application soient disséminées dans différents endroits et ne bénéficient pas d'une encapsulation adéquate tant au niveau des modèles de conception que des langages de programmation. Une telle fonctionnalité est appelée une préoccupation transversale (*crosscutting concern* en anglais) [KLM+]. Les fonctionnalités transversales éparpillées dans le code deviennent, au fur et à mesure que l'application évolue, difficiles à identifier, à comprendre et à faire évoluer [KLM+][Wam03].

La Programmation Orientée Aspects vise à résoudre ce problème en proposant d'écrire le programme en deux parties : une partie fonctionnelle qui encapsule le code métier noyau de l'application, et une partie secondaire qui regroupe des fonctionnalités transversales disséminées. La construction de l'application consiste à assembler ces deux parties (appelée le *tissage* dans la terminologie AOP)

37. https://bean-builder.dev.java.net/guide/tutorial.html

2.6.1 Grain de composition

Une application orientée aspect se fonde sur un programme principal, appelée le *programme de base*, et un ensemble d'*aspects*. Le programme de base détermine la sémantique métier de l'application. Il sert de référentiel pour introduire les points servant à ancrer les aspects. Les aspects sont ceux qui capturent des fonctionnalités transverses de l'application. Ils étendent la sémantique du programme principal en y "ajoutant" des sémantiques supplémentaires.

Un aspect est caractérisé par deux éléments : (1) le code ajouté et (2) la localisation. Le code ajouté est la partie sémantique additionnelle. Il peut être les codes insérés définis dans les gréffons (*advice* en anglais) ou des introductions (*intertype declarations*). Ce dernier permet d'ajouter des attributs, des méthodes, des références de sous-classage. De ce point de vue, AOP peut être classé dans la catégorie des approches de composition invasive (boîte blanche). La localisation est l'endroit dans le programme de référence où l'aspect doit être attaché "tissé".

2.6.2 Protocole de communication

Le protocole de communication de l'AOP est les appels d'événement invoqués implicite entre le programme de base et les aspects.

2.6.3 Mécanisme de composition

Il s'agit du tissage. Lors de la programmation, Blay-Fornarino et al. a mentionné deux approches pour "tisser" un aspect à un programme de base [BFCD+06]. La composition des aspects peut être décrite soit par :

- L'annotation du programme de base afin de préciser les points où les aspects doivent s'appliquer, soit par
- Un langage tiers définissant les relations exactes qui existent entre le programme de base et les aspects. Dans cette approche, le code du programme de base et le code des aspects sont totalement séparés. Un langage tiers permet d'établir les relations entre eux. Cette approche est celle suivie par AspectJ [38].

Les aspects peuvent être "tissés" à la compilation et/ou à l'exécution. Le tissage à la compilation est statique, un compilateur va générer le nouveau programme en insérant les aspects au programme de base ; le tissage à l'exécution est dynamique, un interpréteur va ajouter les aspects au programme de base dynamiquement lors de l'exécution [BSL01]. JAC [39] est un framework de construction des applications distribuées Java par aspects qui offre la capacité de tisser les aspects à l'exécution, alors que AspectJ utilise le tissage à la compilation. Une autre approche de tissage "hybride" entre statique et dynamique, est décrite dans [BSL01][DLBS]. Cette approche permet de "tisser" les aspects en deux phases à la compilation et aussi à l'exécution. Un des rares systèmes représentant de cette approche, est JAVASSIST [40].

Conflits et la Résolution des conflits Un des problèmes fondamentaux de l'AOP lors de la composition des aspects au programme de base provient des conflits potentiels qui apparaissent quand plusieurs aspects qui interfèrent sont tissés au même endroit du programme de base [ZCvdBG06], (e.g., les aspects changent l'état du programme de base simultanément).

Certaines techniques ont été fournies pour résoudre ou réduire les conflits. Une approche simple est de contrôler d'ordre de tissage des aspects. Dans AspectJ, l'ordre de tissage des aspects est guidé par les déclarations de précédent simple. Une autre proposition de même nature est

38. http://www.eclipse.org/aspectj/
39. Java Aspect Components. http://jac.objectweb.org/
40. citée depuis [BSL01]

dans [RGF+06], elle diffère par deux points : 1) l'aspect est tissé avant un autre, et 2) l'aspect est tissé après un autre. Deux directives : précédent (`precedes`) et suivant (`follows`) sont fournis pour déclarer ces types d'ordre de tissage.

D'autres techniques de contrôle de conflits plus complexes sont décrites dans [DFS02, DSB+05, LJW04, NBA05]. Dans [DSB+05], la sémantique des greffons (*advices*) est définie en terme de contrats. Basé sur ces contrats, les conflits peuvent être détectés lors de l'analyse des greffons tissés au même endroit. [DFS02, LJW04] sont de la même catégorie, basé sur les contrats. [NBA05] étend l'approche de contrôle de d'ordre de tissage par des déclarations simples pour introduire plus de types de relations et de dépendances d'ordre complexes entre les aspects. Toutefois, il est clair que l'ordre de tissage ne peut résoudre que quelques cas d'interférence simple ; le problème général des interférences sémantiques entre des aspects conçus et développés indépendamment reste entier.

Tisseurs A propos des tisseurs de l'AOP, on peut citer AspectJ, JAC, HyperJ [41], JBoss AOP [42] pour le langage Java ; AspectC++ [43] pour C++. D'autres tisseurs pour PHP, C# peuvent être trouvés dans [Wik].

2.6.4 Synthèse

L'approche AOP permet d'encapsuler des codes transversaux d'une application dans les unités logicielles séparées appelées *aspects* et de tisser ces unités au sein du programme de base. Le tissage est réalisé soit à la compilation, soit à l'exécution ou soit dans les deux phases. Les conflits entre aspects tissés au même endroit peuvent être détectés et contrôlés par les contrats de tissage et/ou par les contrôles d'ordre de tissage.

Points positifs L'intérêt de l'AOP est d'éviter des codes redondants qui apparaissent fréquemment à plusieurs endroits dans l'application, et ainsi d'augmenter la réutilisation de ces codes par leur composition dans plusieurs applications différentes.

Points négatifs Une limite de l'approche AOP est le code résultant est parfois très compliqué, difficile à comprendre, à tester, à déboguer.

De plus, bien qu'en principe, le tissage des aspects puisse être réalisé sans avoir besoin des connaissances de source code du programme de base, mais dans la pratique, afin de construire des "bons" aspects et les inserer aux "bons" endroits, les développeurs devraient savoir certaine connaissance du code du programme de base. Ceci est parfois impossible à cause du problème de patrimoine (legacy). Il restreint donc l'application de l'AOP dans le domaine des compositions de type boîte noire qui est en réalité très vaste.

Finalement, les mécanisme de contrôle de l'ordre de tissage actuel ont besoin encore de beaucoup d'améliorations pour mieux gérer les anomalies, les conflits lors du tissage.

2.7 Bilan

Dans ce chapitre, nous avons étudié 4 approches du Génie Logiciel sous la perspective de composition. Le tableau 2.1 résume ces approches.

41. http://www.alphaworks.ibm.com/tech/hyperj
42. http://www.jboss.org/jbossaop/
43. http://www.aspectc.org/

	Modulaire	**Architecturale**	**Composant**	**AOP**
Grain de composition	module	composant	composant	programme de base et aspects
Protocole de communication	appels de fonction et l'accès des variables globales	défini par les types de connecteur et/ou par les liaisons	défini par les types de connecteur et/ou par les liaisons	appel d'événement entre le programme de base et les aspects
Mécanisme de composition	par connexion des modules après la conception du système.	par connexion des composants lors de l'analyse du système.	par connexion des composant lors de l'analyse (déclaratif), lors de la conception (langage de script et de programmation), et lors de l'exécution (visuelle)	par tissage à la compilation (statique), ou à l'exécution (dynamique)

TABLE 2.1 – Bilan des approches

Cette étude nous permet de faire une remarque : Il y a deux catégories de composition : la composition boîte blanche qui intervient dans la structure interne des composants, et la composition de boîte noire qui compose les composants tels quels. Nous pouvons trouver ceci dans la composition de modèles où il existe aussi deux types de composition : l'un permet de composer les modèles tels quels, et l'autre les compose mais leur structure est transformée. Cependant, avant de discuter en détail dans le chapitre 4, nous allons préciser le concept de modèle. Le concept de modèle et l'Ingénierie Dirigées par les Modèles (IDM) sont étudiés dans le chapitre suivant.

Chapitre 3

L'ingénierie des modèles

Depuis l'apparition du MDA [1] en Novembre 2000, le développement de logiciel évolue du paradigme centré code à celui centré modèle. Cette évolution prometteuse a attiré immédiatement l'attention des milieux académiques et industriels. Le concept de modèle devint alors un sujet de recherche intensif bien que ce concept fut très ancien et qu'il ait été utilisé au début des années 90, dans les approches de modélisation Booch [Boo93], OMT [RBP+91], Merise [RM89]. Le fait que ce concept revienne au premier plan a conduit à se poser le problème de savoir ce qu'est un modèle, et surtout ce qu'est un modèle pour l'IDM [2].

L'objectif de ce chapitre est d'étudier les notions dans l'ingénierie des modèles que nous allons utiliser fréquemment tout au long du reste du document. La section 3.1 étudie le concept de *modèle*. La section 3.2 parle de trois concepts : *langage*, *langage de modélisation* et *métamodèle*. La section 3.3 discute du *modèle exécutable*. La section 7.4 est une brève conclusion.

3.1 Modèle

Plusieurs définitions sont proposées, et il n'existe pas à ce jour une définition universelle. Nous donnons ci-dessous trois définitions représentatives.

> "*A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.*" [BG01]
>
> "*A model is a set of statements about some system under study (SUS).*" [Sei03]
>
> "*A model is an abstraction of a (real or langague based) system allowing predictions or inferences to be made.*" [Küh06]

Malgré les différences de formulation, ces définitions sont d'accord sur certaines propriétés [Küh06, Veg05] :

- **Représentation** : Il existe un système original qui est l'objet à étudier et que l'on veut représenter par un modèle. Cette caractéristique est fondamentale ; un modèle ne peut exister sans le système qu'il représente.
- **Simplification** : Le modèle donne une vue simplificatrice du système. Certaines caractéristiques du système sont représentées, d'autres non ; tout dépend du but et de l'utilisation du modèle. Un modèle ne représente pas toutes les propriétés du système. Il n'est donc pas une copie du système [Küh06].

1. Model-Driven Architecture
2. Ingénirie Dirigée par les Modèles

- **Pragmatisme** : Un modèle peut remplacer un système pour un propos donné. Les informations obtenues en consultant le modèle devraient être conformes à celles que on aurait obtenues en consultant le système. Cette caractéristique assure l'utilité du modèle car elle permet d'obtenir les informations souhaitées plus rapidement et plus simplement qu'en interrogeant le système.

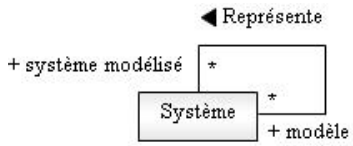


FIGURE 3.1 – Modèle et RepréseneteDe

Le fait qu'un modèle représente un système introduit une relation *Représente* entre eux. Il est important de noter que les notions de système original et de modèle sont relatives [Fav04b]. Un modèle peut représenter un système mais à la fois être un système représenté par un autre modèle. Le modèle et le système étudié ont donc deux rôles complémentaires. Favre [Fav04b] a synthétisé les concepts de *modèle*, *système étudié* et *Représente* en un diagramme de classe comme la figure 3.1.

3.1.1 Classification des modèles

Les modèles, dans le sens le plus général, représentent un spectre très vaste de systèmes, comme énoncé Bezivin "tout est modèle" [Béz05]. Ce slogan vient après le slogan célèbre "tout est objet" de la programmation orienté objet (POO). Dans la POO, le concept de classe est utilisé pour classifier des objets. De la même manière, dans le monde IDM, les modèles sont classifiés selon les langages de modélisation dans lesquels les modèles sont écrits (nous allons étudier ce concept dans la prochaine section).

Cette classification est ainsi basée sur les formalismes dans lesquels sont exprimés les modèles. Néanmoins, un modèle peut être classifié en se basant sur d'autres critères, par exemple :

- **La nature du modèle** : Physique, abstrait ou numérique.
  Favre [Fav04b] distingue les systèmes physique, abstrait ou numérique. Puisqu'un modèle est lui-même est un système, il peut également être physique, abstrait ou numérique. Cette classification, comme l'indique Favre, est relative. Le but est simplement de distinguer les modèles numériques utilisés en informatique, par rapport aux autres.
- **La motivation de modélisation** : descriptive ou spécification.
  Ce critère de classification est un raffinement de la relation de représentation. Un modèle peut être utilisé pour *décrire* un système existant ou pour *spécifier* un système à construire. Dans [Sei03], Seidewitz a proposé de distinguer les modèles descriptifs et les spécification. La différence entre ces deux types est que dans le premier cas, le modèle est construit par l'observation d'un système existant alors que dans le deuxième cas, le système est construit par l'observation d'un modèle [Béz05]. Une analyse du domaine peut être considérée comme un modèle descriptif, par contre, un cahier de charges des besoins de l'utilisateur est une spécification.
- **La nature de l'évolution du modèle** : statique ou dynamique.
  Bézivin catégorise les modèles en statique et dynamique selon le changement d'état du modèle au cours du temps [Béz05]. Un modèle est dit *statique* si son état est constant ; il est dit *dynamique* si son état évolue. Il utilise cette distinction pour remarquer l'usage répandu en informatique de modèles statiques et de systèmes dynamiques : le modèle en

soi ne change pas, mais il représente l'évolution du système modélisé dans le temps [Veg05]. A titre d'exemple, le source code d'un programme peut être considéré comme un modèle statique, représentant le comportement d'un système dynamique réel tel qu'une transaction de virement bancaire.

- **L'usage du modèle** : contemplatif ou productif.
  Cette distinction différencie les modèles produits par des méthodes de modélisation à la Merise qui sont en général contemplatives, interprétées par les humains, par opposition aux modèles productifs, outillables, interprétables par des machines dans l'IDM. Un modèle dans l'IDM est non seulement une documentation mais un artéfact central participant au processus de développement logiciel.
- **Leur rôle** : produit ou processus.
  L'idée de séparer les données à manipuler (produit) de la manipulation (processus) de ces données est très ancienne dans l'ingénierie de logiciel. On peut voir cette idée à plusieurs endroits, par exemple, dans la programmation procédurale où la partie des donnés et des contrôles sont séparés en variables globales et procédures. On peut voir aussi dans CBSE où plusieurs systèmes sont développés en procédé/composant. Produit et processus sont donc deux faces d'un système. De ce point de vue, Bézivin [Béz05] a fait la différence entre les modèles de produit et les modèles de processus. Dans le développement, le mariage entre ces deux modèles est une des façons de construire un système. De ce point de vue, encore une fois, on retrouve le problème de la composition de modèles.
- **Le niveau d'abstraction** : PIM [3] ou PSM [4].
  Les modèles peuvent représenter les systèmes à des degrés d'abstraction différents. Ils peuvent être très abstraits ou très détaillés ; très conceptuels ou très techniques. Pour distinguer les modèles selon les différences de niveaux d'abstraction, le standard MDA introduit les concepts de modèles indépendants de la plate-forme (PIM) et dépendants de la plate-forme (PSM).
  Cette distinction se fonde sur une l'idée de base du MDA selon laquelle le cycle de développement du logiciel est un processus de transformation progressive des modèles métiers, au niveau d'abstraction haut et qui ne dépendent d'aucune technologie d'implémentation, vers des modèles plus spécifiques aux plates-formes techniques. Le code exécutable est le résultat final de ce processus de raffinement.
  Bien que le principe de MDA soit facile à comprendre, sa réalisation n'est pas évidente. Ceci vient de l'ambiguïté de la définition du concept de plate-forme. Cette notion est assez vague et très dépendante du contexte, ce qui rend les notions de PIM et PSM controversées. Malgré ça, l'idée de raffinement les modèles jusqu'au code exécutable du MDA est intéressante. Ceci donne une nouvelle conception du développement logiciel. Néanmoins, PIM, PSM et la transformation de modèles est hors du cadre de ce travail.

D'autres critères de classification peuvent être trouvé dans [BGMR03]. Notons que, parmi les critères que nous avons présentés, les trois premiers concernent tous les modèles, alors que les trois derniers sont plutôt destinés aux modèles informatiques.

3.1.2 Modèles et séparation des préoccupations

"*A model is an artifact ... that represents a given aspect of a system*" [BJPV04]. En effet, comme montré dans la figure 3.1, un système peut être représenté par plusieurs modèles. Ceci introduit naturellement l'idée de modélisation par aspect. Divers aspects du système peuvent être capturés dans différents modèles correspondant aux différentes vues sur ce système. Dans

3. Platform Independent Model.
4. Platform Specific Model.

le contexte du développement de systèmes à grande d'échelle, ceci est significatif. Un modèle représentant tout, risque être difficile à gérer et d'être ni intuitif ni utile (imaginez une carte à l'échelle 1/1).

Le principe de la séparation des préoccupations au niveau modèle est identique à celui au niveau code (l'approche AOP typique). L'objectif est de gérer la complexité du système. Il est important cependant de noter que les modèles eux-mêmes visent aussi le même but par l'augmentation du niveau d'abstraction. La modélisation par aspect fournit donc une haute capacité à gérer la complexité (par la séparation des préoccupations et par abstraction).

La séparation des préoccupations conduit naturellement au besoin de l'intégration des modèles. L'intégration des modèles se fonde sur le tissage et la composition de modèles. Malheureusement, les concepts et les outils permettant tissage et composition de modèles sont insuffisants ; il est critique d'améliorer cet aspect.

3.1.3 Modèles et Espaces technologiques

La section 3.1.1 montre l'existence de nombreux types de modèles. Même si on se restreint au monde informatique, les modèles peuvent être catégorisés en différents groupes. Une autre façon de faire est de s'appuyer sur les espaces technologiques.

La notion d'Espace Technologique concerne "*a working context with a set of associated concepts, body of knowledge, tools, required skills, and various other possibilities.*" [KBA02]. Les espaces techniques sont souvent liés à un domaine de recherche. Dans [FEBF06], Favre cite quatre espaces technologiques : l'espace technique des modèles (Modelware), défini par des métamodèles MOF [5] [Gro06], Ecore [BSM+03] et de nombreux DSL [6] ; l'espaces techniques des grammaires (Grammarware) défini par des langages de définition de grammaire comme BNF [7] ou EBNF [8] ; l'espaces techniques des documents (Docware), défini par des langages de définition de schéma comme XSchema [9] ou DTD [10] ; et l'espace des bases de données, défini par les langages de définition de schéma comme l'algèbre relationnelle ou entité-relation.

Les modèles peuvent être classifiés selon leur espace technologique. Par exemple, un programme Java appartient à l'espace technologique des grammaires, un document XML [11] appartient à l'espace technologique des documents et un diagramme de classe d'UML appartient à l'espace technologique des modèles.

Il n'y a pas néanmoins un espace technologique dominant. Tous existent et ne sont pas isolée. L'IDM a indiqué des possibilités d'établir des ponts entre les espaces afin de permettre aux utilisateurs d'un espace de pouvoir profiter des outils et des techniques des autres espaces.

Du point de vue pratique, les ponts entre les espaces, dans le sens de l'IDM, sont des transformations de modèles. Cependant, dans un sens plus large, ces ponts peuvent être considérés aussi comme des compositions de modèles car il existe depuis longtemps la situation dans laquelle un programme java effectue des requêtes sur une base de donnés et sauvegarde les informations en XML. Celui-ci est un exemple typique de l'idée de la composition des modèles inter espaces.

5. Meta Object Facility
6. Domain-Specific Language
7. Backus-Naur Form
8. Extended Backus-Naur Form
9. XML Schema
10. Document Type Definition
11. Extensible Markup Language

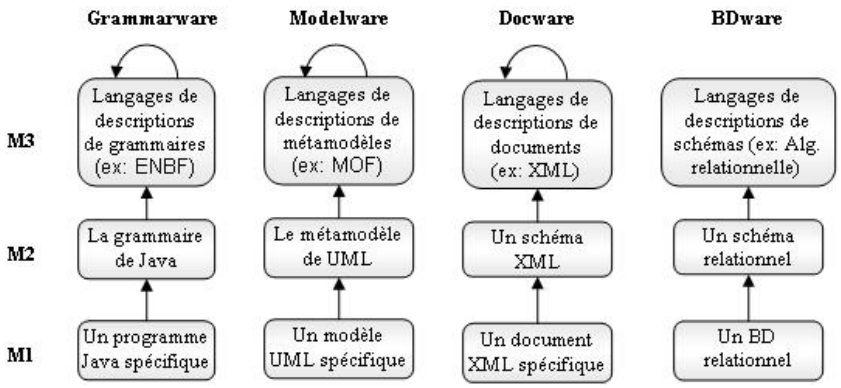


FIGURE 3.2 – Espaces technologiques [FEBF06]

3.2 Langage, Langages de modélisation, Métamodèles

3.2.1 Langage

Le langage est la façon fondamentale de communiquer. Les langages naturels sont les moyens pour la communication entre les humains. Les langages de programmation sont les moyens pour communiquer entre humains et machines.

Il est important de noter d'abord que le terme langage présenté dans cette section ne concerne que les langages informatiques.

Dans la théorie des langages de programmation, un langage est définit par 3 facettes principales : la syntaxe abstraite, la syntaxe concrète et la sémantique.

La syntaxe abstraite

> "*The abstract syntax of a language describes the vocabulary of concepts provided by the language and how they may be combined to create models. It consists of a definition of the concepts, the relationships that exist between concepts and well-formedness rules that state how the concepts may be legally combined.*" [CESW04]

D'abord, comme l'indiqué dans la définition, une syntaxe abstraite fournit un ensemble de concepts basiques permettant de définir la structure des programmes (ou modèles). Ensuite, elle dispose d'un ensemble des règles de validation syntaxique à imposer lors de la création des programmes afin d'assurer que la structure du programme créé est syntaxiquement correcte. A titre d'exemple, des concepts comme *classe* et *objet* font partie de la syntaxe abstraite de n'importe quel langage orienté objet tels que Java ou C++. Il est important à souligner que la syntaxe abstraite traite seulement la forme et la structure des concepts du langage sans prendre en compte leur représentation et leur sens (*meaning*).

La syntaxe concrète

La syntaxe concrète est la notation permettant la présentation et la construction des modèles sous une forme humaine lisible. Pour qu'un langage soit "lisible", il y a en général deux types de syntaxe concrète : textuelle et visuelle.

La syntaxe textuelle permet d'écrire les modèles ou les programme dans une forme de texte, alors que la syntaxe visuelle présente souvent les modèles ou les programmes sous forme de dia-

gramme. Le bénéfice principal de la syntaxe textuelle est de permettre d'exprimer des expressions complexes, alors que le bénéfice de la syntaxe visuelle est l'intuitivité.

La sémantique du langage

La syntaxe abstraite définit des concepts du langage mais sans préciser leur sens. La sémantique du langage va compléter cette partie. En effet, il faut être clair entre ce que le langage représente (quoi) et la signification de ceux-ci (comment). Selon [CESW04], il y a plusieurs types de sémantiques :

- **La sémantique de traduction** : La sémantique du langage peut être obtenue par la traduction des concepts du langage vers les concepts d'un autre langage qui a déjà une sémantique précise. Ce type de sémantique peut être trouvé dans la traduction des langages de programmation de haut niveau vers des langages machines. L'avantage de ce type de sémantique est qu'il permet d'obtenir directement une sémantique exécutable pour un langage à travers de la traduction. Cependant, son inconvénient est le risque de perte d'informations du langage original au cours du processus de traduction, et de lier un langage à une machine, ce qui empêche son exécution sur d'autres machines.
- **La sémantique opérationnelle** : La sémantique opérationnelle décrit la sémantique dynamique du langage. C'est-à-dire, elle décrit la manière d'exécuter un modèle (ou programme). Typiquement, la sémantique opérationnelle est exprimée en termes d'opérations associées aux concepts du langage. Par exemple, les opérations *run()*, *start()*, *stop()* du concept *Thread* dans le langage Java décrivent la sémantique opérationnelle de ce concept en indiquant la manière d'exécuter un *thread* dans un programme Java.
- **La sémantique extensionnelle** : La sémantique d'un langage est définit comme une extension d'un autre langage. Les concepts du nouveau langage héritent la sémantique des concepts d'un langage existant. Cette idée est très proche avec la notion de profil dans UML. Le bénéfice est la réutilisation des langages existants avec un minimum effort.
- **La sémantique dé notationnelle** : L'objectif de la sémantique dé notationnelle est d'associer des objets mathématiques tels que des nombres, des tuples, des fonctions aux concepts du langage. Par exemple, l'ensemble des nombres entiers de 0 à l'infini peut être associé au concept *Integer* du langage Java. Le concept est dit "dénoter" les objets mathématiques, alors que les objets sont appelés "dénotation" du concept.

3.2.2 Langages de modélisation

Dans la théorie des langages de programmation, un langage est défini comme un ensemble de phrases.

Cette définition est transposée dans [BBB$^+$] aux concepts de base de l'IDM, et on définit un langage de modélisation comme un ensemble de modèles.

> "*A modelling language is a set of model.*" [Fav04a]

La figure 3.3 exprime le concept de langage de modélisation en l'intégrant avec d'autres concepts définis précédemment.

3.2.3 Métamodèles

Un langage de modélisation est lui-même un système. Par conséquent, il peut devenir le sujet d'une modélisation. Cette idée introduit naturellement la notion de métamodèle. Un métamodèle est défini donc comme le modèle d'un langage de modélisation [Fav04a].

Puisqu'un système peut être représenté par plusieurs modèles, un langage de modélisation peut donc avoir plusieurs métamodèles. Comme nous l'avons indiqué précédemment, un langage

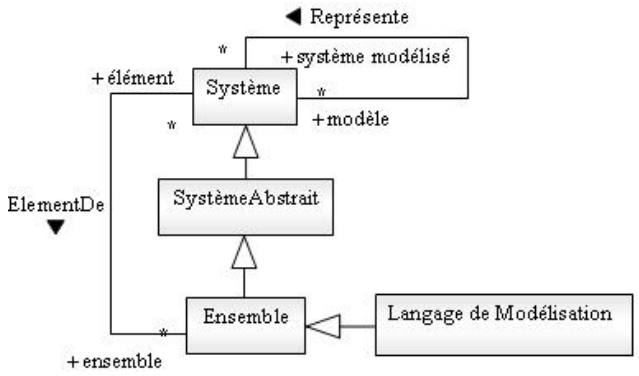


FIGURE 3.3 – Langage de modélisation

a plusieurs facettes : la syntaxe abstraite, la syntaxe concrète, la sémantique. Il est possible, pour un même langage, d'avoir plusieurs métamodèles dont chacun représente une (ou une partie de) facette du langage. On peut avoir un (ou plusieurs) métamodèle(s) de la syntaxe abstraite, un (ou plusieurs) métamodèle(s) de la syntaxe concrète, un (ou plusieurs) métamodèle(s) de la sémantique ; en revanche, un même métamodèle peut aussi représenter plusieurs facettes en même temps : par exemple, il modélise la syntaxe abstraite et la sémantique (c'est le cas d'un programme Java), il modélise la syntaxe concrète et la syntaxe abstraite (c'est le cas les codes sources des éditeurs EMF [12] [BSM+03]), ou il modélise les trois à la fois. Le concept de métamodèle pose la question de la relation entre le modèle et son métamodèle. En fait, cette relation est dérivée de deux relations : celle entre le modèle et le langage de modélisation (la relation de dénotation) et celle entre le langage de modélisation et le modèle qui le présente (la relation de représentation) [Fav04a]. Cette relation dérivée est identifiée comme la relation de conformité d'un modèle vis-à-vis de son métamodèle.

3.3 Modèles exécutables

De notre point de vue, la notion de modèle exécutable est liée à la notion de machine d'exécution. Ce point de vue est assez évident dans le monde de la programmation où on peut aisément dire que les programmes Java sont exécutés par la machine virtuelle Java. En transposant cette vision dans le monde des modèles, nous pouvons dire qu'un modèle est exécutable lorsqu'il existe une machine capable d'exécuter ce modèle. Nous donnons ci-dessous quatre exemples pour démontrer cette idée.

- **Exemple 1** : Les modèles de transformation d'ATL [**?**]. Ils sont les modèles exécutables. Ces modèles sont exprimés par le langage ATL, et sont exécutés par le moteur ATL. La construction *helpers* du langage ATL est le moyen permettant de décrire les comportements dynamiques des transformations ATL.
- **Exemple 2** : Les modèles de Kermeta [MFJ05]. Ils sont exprimés comme une combinaison entre le langage Ecore et un langage d'action impérative de Kermeta. Ces modèles sont exécutés par le moteur Kermeta.
- **Exemple 3** : Les modèles de Xactium XMF [CESW04]. Le langage XCore modélise sa partie statique, alors que le langage XOCL, XSync et XMap modélisent sa partie dynamique. Ces modèles sont exécutés par le moteur XMF.

12. Eclipse Modeling Framework

- **Exemple 4** : Les modèles UML exécutables [MB02]. Ils peuvent être rendu exécutables par les machines virtuelles UML [RFBLO01, SM05].

UML et les modèles UML sont très populaires et bien connus dans la communauté de modélisation. Nous allons donc présenter ci-dessous UML exécutable comme un exemple représentatif de l'idée de modèle exécutable.

3.3.1 UML exécutable

UML traditionnel est connu depuis longtemps comme un langage de modélisation orienté objet "universel". Il semble efficace dans la modélisation d'aspects structurels du système, mais reste encore faible pour la modélisation d'aspects comportementaux. En effet, UML a proposé plusieurs types de diagrammes spécifiant les comportements tels que le diagramme de collaboration, d'activité, d'état etc. Cependant, leur efficacité est encore loin de ce qu'on attend. Ils ne peuvent pas exprimer suffisamment de détail des comportements d'un objet individuel ni les interactions entre les objets. De ce fait, les modèles UML traditionnels restent encore les spécifications rudimentaires, contemplatives. Ils ne sont pas des artefacts exécutables comme le code.

Dans le contexte de l'IDM actuel, l'augmentation de la contribution des modèles dans le développement est un des verrous critiques. Dans ce mouvement, UML traditionnel envisage à rendre ses modèles plus utilisables. Pour cela, une des possibilités est de les rendre exécutables.

UML exécutable vient donc dans ce contexte. Il s'agit d'un nouveau domaine de recherche rassemblant des travaux autour du sujet de rendre exécutables les modèles UML.

Une démarche générale des approches dans ce domaine est :

- D'abord, modéliser (à haut niveau d'abstraction) des comportements du système. Le but est d'enrichir la sémantique d'exécution des modèles UML.
- Puis, rendre exécutable ces modèles.

Nous allons détailler ces points ci dessous :

Modélisation des comportements d'un modèle UML

La signature des méthodes modélisées dans les diagrammes de classe UML n'est pas la spécification comportementale suffisamment riche pour pouvoir interpréter exactement ce qu'il va faire le modèle. Elle illustre le comportement, mais ne le spécifie pas [RFBLO01]. Le but maintenant est qu'il faut spécifier plus précisément ces comportements.

Plusieurs approches proposent de créer des profils pour spécifier les comportements des modèles UML [JZM08, LRdS04, MB02, PS04, RFBLO01].

Un de premiers travaux parmi eux est [MB02]. Les auteurs ont proposé un profil dans lequel les modèles UML exécutables sont modélisés par les diagrammes de classes, les diagrammes d'état (*statechart*) et un langage d'action. Concrètement :

- La structure du système est modélisée par le diagramme de classe.
- Les comportements sont distingués en deux cas pour modéliser : ceux d'un objet individuel et ceux représentant l'interaction entre les objets.
  - Les comportements d'un objet individuel sont modélisés par le diagramme d'état. Un diagramme d'état décrit l'espace d'états des instances d'une classe et les transitions possibles entre ces états. Chaque transition d'état est associée à des événements qui le provoquent et à des procédures qui sont un ensemble des actions s'effectuant lors de la transition. Ces actions sont écrites dans un langage d'action.
  - Les interactions entre les objets sont modélisées par le diagramme de collaboration.

Notons qu'il est possible d'utiliser différents moyens pour modéliser ces genres de comportement. Par exemple, à propos du comportement d'un objet individuel, au lieu d'utiliser le diagramme d'état UML, on peut utiliser un modèle de Petri net comme la proposition dans [BB01].

A propos du langage d'action, une proposition populaire est le langage Action Semantic. Cependant, les langages de programmation comme C, C++, Ada peuvent être aussi utilisés pour le même but [JZM08].

Les interactions entre les objets peuvent être modélisées par le langage OCL [JZM08, RFBLO01] au lieu de l'utilisation de diagramme de collaboration.

En résumé, pour modéliser un modèle UML exécutable, il faut spécifier les éléments suivants :

- Sa structure (par les diagrammes de classes par exemple).
- Les comportements d'un objet individu (par les diagrammes d'état, le modèle Petri net etc.).
- Les interactions entre les objets (par le diagramme de collaboration, par OCL etc.).

Exécution d'un modèle UML exécutable

Une fois les modèles réalisés, il est important de les rendre exécutables. Les approches UML exécutables actuelles proposent soit de construire les machines virtuelles UML exécutable capable interpréter ces modèles [RFBLO01, SM05] ; soit de les transformer en code.

3.3.2 Conclusion

Pour conclure cette section, l'idée importante à retenir à propos de la notion de modèle exécutable est qu'il faut avoir des machines d'exécution. Les modèles ne sont que les spécifications, ils ne peuvent pas s'exécuter directement. L'interprétation d'un modèle demande l'intervention d'un moteur d'exécution, ou une chaîne de transformations (compilation) aboutissant automatiquement à un programme dans un langage exécutable.

3.4 Conclusion

Dans ce chapitre, nous avons abordé les terminologies qui seront utilisées fréquemment dans ce document, y compris *modèle*, *métamodèle*, *langage (de modélisation)*. Nous avons introduit aussi une vision de la notion de *modèle exécutable*.

Dans le chapitre qui suit, nous allons étudier de *composition de modèles*.

Chapitre 4

Composition de Modèles

4.1 Introduction

La composition de modèles est un thème de recherche nouveau dans l'IDM. Les travaux sont en cours du développement et de l'évolution. Donc, il n'existe pas encore à ce jour une fondation mature pour ce sujet.

Notre objectif dans ce chapitre est donc seulement d'étudier les approches de composition de modèles actuelles en analysant et identifiant 1) *quels* sont les éléments participant au processus de composition, et 2) *comment* la composition de modèles est réalisée dans ces approches. Le but final est d'aboutir à une compréhension de ce qui est fait pour la composition de modèles dans ces approches. Cet étude va être la base nous permet de situer notre travail par rapport aux travaux actuels.

Nous reprenons les critères établis dans le chapitre 2 pour l'étude dans ce chapitre.

- A propos du *grain de composition*, il s'agit bien évidemment du modèle. Une étude général de ce concept a été présentée dans le chapitre 3. Dans ce chapitre, ce critère est utilisé seulement pour préciser les modèles composés sous le point de vue de la composition de modèles (e.g. les types de modèles composés, le nombre de modèles en entrée, le rôle du modèle, le langage de modèles etc.).
- Ce qui est important à étudier est le *mécanisme de composition*. En ce qui concerne ce point, nous étudierons deux facettes suivantes : 1) les éléments (à part des modèles composés) nécessaires à la composition et 2) le processus de créer un modèle composite. Pour 1), souvent le processus de composition a besoin d'éléments supplémentaires, ceux-ci peuvent être les *spécifications de composition*, ou les *relations* utilisées à établir entre les modèles, étudier ces éléments est donc important. De plus, il est indispensable d'étudier les langages qui les créent. Pour 2), les diverses façons de créer le modèle composite (e.g, exécuter la spécification modélisée pour produire le modèle composite ou établir des relations entre les modèles) devrait être étudiées.
- Par contre, le *protocole de communication* et l'*adaptation* (de modèles pour les composer) ne seront pas vraiment abordés car, suite à notre observation, ces points sont très peu abordés dans les approches. Nous mentionnerons simplement les approches qui parlent explicitement de ces points.

Le chapitre est organisé de la manière suivante : la section 4.2 présente une définition informelle de la composition de modèles (paragraphe 4.2.1) et une classifications des approches de composition de modèles(paragraphe 4.2.2). Six approches de composition de modèles seront étudiées dans la section 4.3. La section 4.4 fait un bilan du chapitre.

4.2 Composition de modèle

4.2.1 Une définition informelle

> "*Model composition is an operation that combines two or more models into a single one.*" [DDFBV06]
>
> "*Model composition in its simplest form refers to the mechanism of combining two models into a new one.*" [AVP07]

Ces deux définitions peuvent être traduites diagrammatiquement comme le schéma 4.1. Conformémant à celui-ci, on peut dire que la composition de modèle est un processus qui prend deux ou plusieurs modèles en entrée, les intègre au travers d'une opération de composition et produit un modèle composite en sortie.

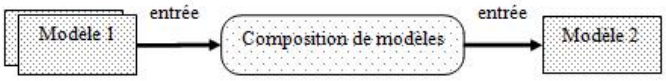


FIGURE 4.1 – Processus de composition de modèles : vue simple

Néanmoins, ce schéma est très abstrait. Aucune hypothèse sur les modèles en entrée, sur en sortie, ni sur l'opération de composition est exprimée. Dans la pratique, chaque approche doit préciser ces hypothèses pour son contexte de travail. Celles-ci consistent également les différences permettant de classifier les approches.

4.2.2 Classification des approches

A. Classification selon les caractéristiques des modèles en entrée

- **La quantité des modèles en entrée** : *deux* ou *plusieurs*.
  Un processus de composition peut recevoir *deux* ou *plusieurs* modèles en entrée. Dans la plupart des cas, chaque composition prend deux modèles, mais il existe des cas comme les éditeurs EMF [BSM+03] où on peut composer plusieurs modèles à la fois (par l'importation des modèles sous forme de resources au même éditeur).
- **Le rôle des modèles composés** : *symétrique* ou *asymétrique*.
  Certaines approches ne font pas la différence entre les modèles composés (*symétrique*) [Ber03] mais certaines autres distinguent les modèles de base et les modèles d'aspect (*asymétrique*) [RGF+06].
- **Le type des modèles composés** : *structurel* ou *comportemental*.
  Certaines approches se focalisent sur la composition de *modèles structuraux* (schémas de base de données [Ber03], diagrammes de classes UML [Cla02] etc.), certaines autres s'intéressent aux *modèles comportementaux* (modèles de Petri Net, diamgrammes d'état UML [NSC+07] etc.).
  Notons que, dans la première catégorie, certaines approches ne précisent pas les types des modèles composés (ceux-ci peuvent être un ensemble des objets, les schémas de base de donnée, les modèles UML etc. [Ber03, DDFBV06].) alors que d'autres se restreintes aux modèles UML [Cla02].
- **L'hétérogénéité des modèles source** : Composition de *même métamodèle* ou *différents métamodèles*.
  Le premier cas est dite la composition *endogène*, alors que le deuxième est dite la composition *exogène*.

B. Classification selon les caractéristiques des modèles en sortie

- **Le quantité des modèles en sortie** : *un* ou *plusieurs*.
  Le résultat final de l'opération de composition peut être soit *un* modèle composite [Cla02, RGF+06], soit un *ensemble* comprenant les modèles d'entrée plus une troisième partie qui les intègre [BG04].

C. Classification selon les caractéristiques de l'opération de composition

- **Le type de composition** : *par opérateurs* ou *par relations*.
  Nous avons identifié deux types de composition : *par opérateurs* tels que la fusion, le remplacement, l'union, le tissage etc. et par l'*établissement des relations* telles que l'association, l'agrégation, l'héritage.
  Dans le premier type, la composition est réalisée en exécutant les opérateurs de composition sur les modèles sources ; alors que dans le second type, les modèles source sont composés en utilisant les relations pour les connecter.
  La différence est que les opérateurs ne sont pas une partie du modèle final ; tandis que les relations font vraiment partie de ce modèle.
- **Les éléments de composition** : ce sont les éléments supplémentaires participant aux composition. Il y a deux axes de classification : le *type* et le *formalisme* de ces éléments.
  - **Le type d'éléments de composition** : *spécifications de composition* ou *relations*
    Correspondant aux types de composition identifiés ci-dessus, nous distinguons deux types d'élément de composition possibles : les *spécifications de composition* et les *relations*.
    Les relations expriment les liens de connexion entre les modèles. Elles sont à la fois les éléments de composition et les éléments du modèle final.
    Les spécifications de composition sont les éléments de composition mais par contre elles ne sont pas des éléments du modèle final. Leur rôle est de modéliser les connaissances nécessaires au processus de composition. En général, un processus de composition a besoin de connaître :
    - *Quoi* : spécifie quels éléments de modèle seront composés (e.g., une classe, un attribut, un méthode etc.) et leurs correspondances.
    - *Comment* : spécifie des opérateurs de composition (e.g., correspondance (*match*), remplacement (*replace*), redéfinition (*override*) etc.) qui vont être utilisés pour composer des éléments indiqués.
      Notons que, ces informations (i.e. *quoi* et *comment*) peuvent être modélisées par une ou plusieurs spécifications de composition à la fois. Ceci dépend de chaque approche. Par exemple, dans EML [KPP06c], ces informations sont décrites dans les *spécifications des règles de composition* (règles de correspondance, règles de fusion et règles de transformation), tandis que dans AMW [DDFBV06] elles sont réparties dans les *modèles de tissage* et les *transformations* [BBF+06].
  - **Le langage de composition)** :
    Les éléments de composition ont besoin de formalismes pour les exprimer. Ces formalismes sont très variés car chaque approche a ses propres éléments de composition. Ils peuvent être un langage de tissage [DDFBV06], un métamodèle des règles de composition [KPP06c], un profil UML pour la composition de modèles [Cla02] etc.
    Malgré leur diversité, on peut généralement évaluer un formalisme de composition sur deux points : les abstractions de composition qu'il fournit et son extensibilité. Le premier point concerne ses primitives alors que le second point concerne sa capacité à permettre à l'utilisateur de définir ses propres abstractions de composition.
    La classification sur les formalismes se base donc sur deux points : les abstractions de

composition prédéfinies et l'extensibilité du langage.

· **Abstractions de composition prédéfinies** :
De manière détaillée, il y a plusieurs variations autour ce point qui font les différences entre les approches. Par exemple, nous pouvons faire la classification selon les trois axes suivants :

∗ **La variété de *types* des abstractions** : {correspondance, tissage, la fusion, remplacement, surcharge, ...}
Dans le cas de la composition par relations, ces abstractions sont les types de relation tels que l'association, l'héritage, l'agréation.
Dans le cas de la composition par opérateurs, ces abstractions sont les opérateurs de composition primitifs tels que la correspondance (*matching*), le tissage (*weaving*), la fusion (*merging*), le remplacement (*replace*), le surcharge (*override*) etc.

∗ **La variété de *quantité* des abstractions** :
Le nombre des abstractions est différent selon les approches. Par exemple, AMW [BBF+06] propose un opérateur prédéfini : la correspondance (*match*) alors que EML [KPP06c] propose trois opérateurs : la correspondance (*match*), la fusion (*merge*), et la transformation (*transform*).

∗ **La variété d'*implémentation* des abstractions** :
Plusieurs approches peuvent proposer un même type d'abstraction de composition mais l'implémenter différemment. A titre d'exemple, prenons la fusion de modèles (structurels, orientés objet), cette abstraction est implémentée dans plusieurs approches mais chacune a sa version.
Généralement, le processus de fusion de modèles peut être divisé en trois stades : avant, pendant et après la fusion. La description d'une telle fusion est comme suit :

1. Avant la Fusion : il est probablement nécessaire de transformer (ou adapter) les modèles pour que la fusion puisse être réalisée sans conflit.
2. Pendant la Fusion : il y a deux phases :
   - Phase de correspondances : établit les correspondances entre les éléments de modèles. La découverte des correspondances peut être automatique, manuelle ou semi-automatique.
   - Phase de fusion : intègre les éléments en correspondance. Il est probable d'avoir les conflits (e.g, les éléments correspondants n'ont pas le même type). La résolution de conflit est soit automatique (basée sur certaines stratégies de résolution prédéfinies) soit manuelle (e.g, faire apparaître un panel pour que l'utilisateur résoudre le problème).
3. Après la Fusion : il est nécessaire de restaurer la cohérence du modèle final.

Cependant, dans la réalité, l'implémentation des approches ne respectent pas toujours ce processus. Certaines approches proposent explicitement l'adaptation [FBFG07] mais certaines autres non [KPP06c]. La manière de découvrir des correspondances est faite automatiquement dans [KPP06c, FBFG07] ; mais manuellement dans [BBF+06] ; et semi-automatiquement dans [NSC+07]. De la même manière, la résolution de conflit peut varier entre manuel et automatique, même les stratégies de résolution sont différentes. De notre point de vue, la variété dans l'implémentation des abstractions est un critère de classification important des approches.

· **Extensibilité du formalisme** : *extensible* ou *non extensible*.
Les abstractions prédéfinies avec les sémantiques prédéfinies permettent la réalisation d'un grand nombre de scénarios de composition. Cependant, dans le cas général, ceci ne veut pas dire que la composition se limite à ces actions. L'utilisateur a probablement besoin de définir ses propres sémantiques de composition. Ceci entraîne la nécessité

d'extension du formalisme de composition par la définition de nouvelles abstractions ou la redéfinition de la sémantique d'une abstraction prédéfinie.
Nous distinguons ainsi les formalismes *extensibles* et *non extensibles*. Le premier type est plus flexible que le deuxième. Il permet de réaliser les compositions de manière particulière.
Dans le cas extensible, nous distinguons aussi deux manières d'extension du langage : la définition de nouvelles abstractions ou la redéfinition de la sémantique d'une abstraction prédéfinie. La première est fournie dans AMW [DDFBV06] ; alors que la deuxième est fournie dans EML [KPP06c].

- **La manière de créer un modèle composite** : l'*exécution des opérateurs* ou l'*établissement des relations*.
  Correspondant aux deux types de composition par opérateur et par relations, il y a deux manières de créer un modèle composite : soit par l'exécution des opérateurs sur les modèles source ; soit par l'établissement des relations entre eux.

D. Autres caractéristiques de classification

- **L'effet de composition** : structure de modèle *transformée* ou *préservée*.
  Les compositions peuvent transformer la structure des modèles source ou la préserver.
  - *Composition poids lourd* : La composition transforme la structure des modèles composés (e.g., la fusion des modèles)
  - *Composition poids léger* : La composition ne transforme pas la structure des modèles composés (e.g., établir des relations entre les modèles)
- **L'orientation de composition** : Composition orientée transformation ou composition pure.
  Ceci est assez lié à la caractéristique de l'effet de composition mentionné ci-dessus. Dans le cas de tissage, ou de fusion où il y a une transformation effectuée sur les éléments sources pour produire les éléments cibles, la composition est dite orienté transformation, alors que dans le cas d'établissement des relations, la composition est dite pure composition.
- **Le domaine de recherche** : actuellement il y a trois domaines majeurs qui travaillent activement sur la composition de modèles :
  - *Modélisation orientée aspect (AOM*[1]) : L'originalité de AOM est l'application du principe de séparation de préoccupation à la modélisation. L'opération de composition centrale de AOM est le tissage. Il consiste à composer les modèles d'aspects à un modèle de base. La relation entre le modèle d'aspect et le modèles de base est relatif. Un modèle peut être un aspect et une base à la fois. De ce fait, deux types de tissage ont été identifiés : aspect avec base (asymétrique) et base avec base (symétrique). Le premier vient de AOP tandis que le deuxième est inspiré de SOP [2].
  - *Gestion de modèle*[3] : Ce domaine émerge dans le contexte IDM. Il s'intéresse aux platformes IDM fournissant les opérateurs génériques de manipulation de modèles tels que la fusion, la comparaison, la différence, la génération etc. En ce qui concerne de la composition, ces opérateurs peuvent divisés en trois groupes :
    - Pour la découverte des correspondances : tels que *match* [Ber03, BCE+06], *relate* [KDDF06], *compare* [KPP06a].
    - Pour l'intégration de modèles : tels que *merge* [KPP06a, Ber03, BCE+06]. *compose* [Ber03], *weaving* [RKRS05].

1. Aspect-Oriented Modelling.
2. Subject-Oriented Programming.
3. Model Management.

· Pour la liaison de modèles, i.e. relier les modèles sans changer leur structure : *sewing* [RKRS05].

Cetaines plateformes de gestion de modèles ont été développées dans ce domaine, telles que AMMA [DDFJ05], Rondo [MRB03], EOL [KPP06b], MOMENT [BCR05].

– *Meta-modélisation*[4] : Les approches de méta-modélisation permettent la définition des métamodèles. Ces approches peuvent posséder eux même les mécanismes de composition permettant de composer les métamodèles. Comme la relation entre modèle/métamodèle est relative [Fav04b], il est possible de croire que si ces mécanismes sont applicables au niveau métamodèle, leur *principe* devraient être applicables également au niveau de modèle.

De plus, lors une approche permet la composition de métamodèles, il est important aussi d'étudier si l'approche est capable ou non de composer les modèles crées par ces métamodèles. De notre observation, certaines approches le permettent [CESW04][BSM+03] ; d'autres non [GME].

La figure 4.2 donne une vue synthétisée de la classification des approches de composition de modèles.

Dans le reste du chapitre, nous allons présenter six approches représentatives de composition de modèles.

- Atlas Model Weaver [DDFBJ+05, DDFBV06, DDFJ05, BBF+06].
- Eclipse Modeling Framework [BSM+03].
- Espilon Merging Language [KPP06c, KPP06a, KPP06b].
- Generic Modeling Enviroment [GME, Dav, KML+04, LMV01, MBL07].
- Kompose [FBFG07, FRF+].
- Xactium XMF/XMF-Mosaic [CESW04].

Une classification rapide de ces approches selon les domaines auquel ils appartiennent est dans le tableau 4.1

Modélisation orientée aspect	**Gestion de modèle**	**Métamodélisation**
Kompose	Atlas Model Weaver, Espilon Merging Language,	Eclipse Modeling Framework, Generic Modeling Enviroment, Xactium XMF/XMF-Mosaic

TABLE 4.1 – Les approches de composition de modèles

4.3 Les approches

4.3.1 Atlas Model Weaver

Atlas Model Weaver (AMW) est un module de AMMA (ATLAS Model Management Architecture) [DDFJ05] - une plateforme de gestion de modèles générique. Ce module est destiné particulièrement à la création de relations (i.e. des liens) entre les éléments de modèles (ou métamodèles).

4. Metamodeling.

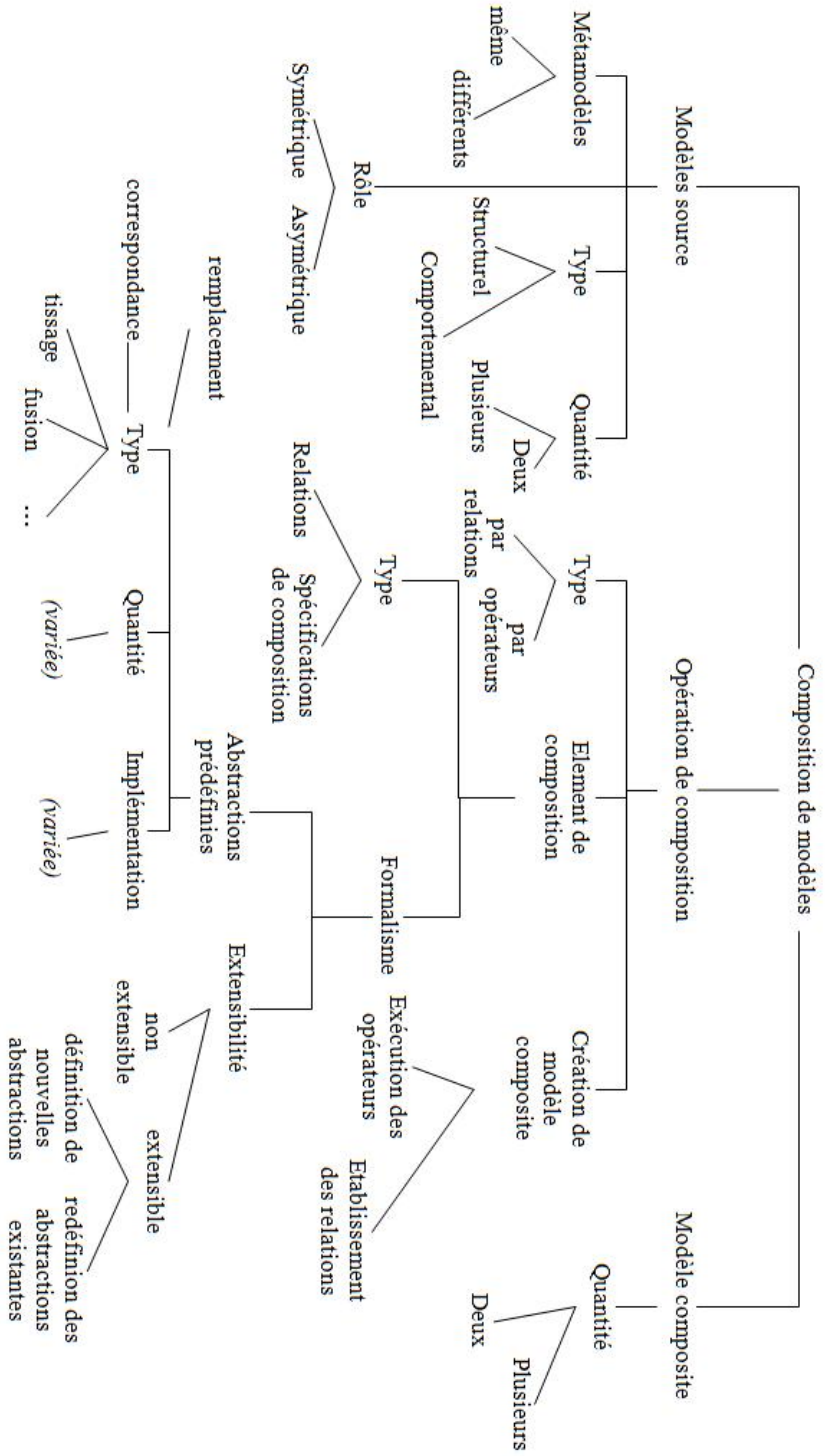


FIGURE 4.2 – Classification des approches de composition de modèles

A. Modèles

Les modèles composés sont structuraux, orientés objet. A présent, AMW utilise EMF comme le *handler* de ses modèles. Le type de composition est par opérateur.

B. Élements de composition : les modèles de tissage et les transformations HOT

AMW utilise un langage, appelé le langage de tissage (weaving language) (**formalisme**), qui a une partie core fournissant les concepts génériques de base permettant de créer les liens structurels entre les modèles. Ces liens sont sauvegardés dans les modèles de tissage (weaving models) (**élément de composition**). Le métamodèle de tissage de base de AMW contient les concepts de tissage de base. `WElement` est l'élément basique de tous les éléments du métamodèle de tissage ; `WModel` représente la racine du modèle de tissage. `WLink` représente des liens entre les éléments des modèles. `WLinkEnd` indique le type des éléments pouvant être composés.

Les liens crées par les concepts core n'ont aucune sémantique. Pour la tâche de composition, AMW fournit la capacité d'étendre le métamodèle de tissage de base afin de construire les nouveaux concepts de tissage spécifiques au domaine de composition de modèle (i.e. fusion, remplacement, union, surcharge etc.)(**extensibilité du formalisme**). C'est les concepts `WLink` et `WLinkEnd` que l'on peut étendre pour ajouter les nouveaux types de relation de composition (par exemple fusion, remplacement etc.) et également définir les types d'éléments que les nouvelles relations peuvent relier (par exemple au lieu de relier des `EObjets` de `EClass`, ce sera peut être relier les objets des classes de l'utilisateur `Personne` et `Etudiant`). Cette extension permet d'avoir un langage de tissage dédié à la composition de modèles. De façon similaire, par étendre le métamodèle de tissage de base, on peut créer aussi une famille des métamodèles de tissage pour le traçage l'évolution, la comparaison, la traduction et même la transformation de modèles etc.

C. Mécanisme de composition : tissage et transformation

Les modèles de tissage eux même ne sont pas le modèle composite résultat. Ce sont les abstractions de l'opération de composition mais pas le modèle composite final. Cependant, ces abstractions ne sont pas exécutables. Pour obtenir le modèle composite, il faut transformer ces abstractions sous forme d'un programme exécutable (vers un langage de programmation par exemple), puis l'exécution de ce programme va produire le modèle composite résultat (**manière de créer le modèle composite**). Dans le cas de AMW, AMW propose de transformer les modèles de tissage en programmes ATL (ATL est un langage de transformation du plateforme AMMA). Le processus de transformation d'un modèle de tissage vers le programme ATL est réalisé par une autre transformation écrite aussi en ATL. Ce qu'on appelle une transformation d'ordre supérieur (HOT - higher-order transformation) (**élément de composition**). Un HOT est une transformation qui soit prend une transformation en entrée, soit produit une transformation en sortie, soit les deux.

Pour chaque métamodèle de tissage étendu (pour la composition), il y a un HOT différent. Ce HOT est unique et ne se change pas. Par contre, les modèles de tissage peuvent être changés. Chaque fois que les modèles de tissage sont changés, la translation de nouveaux modèles de tissage vers les nouveaux programmes ATL va s'effectuer par l'application de même HOT. L'exécution des nouveaux programmes ATL va produire les nouveaux modèles composites.

La transformation de modèle de tissage vers le code ATL n'est qu'une possibilité pour obtenir le modèle de composition exécutable. Ceci a pour but de profiter des facilités fournies par la plateforme AMMA (dans ce cas, c'est le langage ATL). Cependant, il est possible, au lieu d'utiliser ATL, d'utiliser les autres langages de transformation comme XSLT pour traduire les modèles de

tissage. De plus, le programme résultat n'est pas forcément en ATL. Il peut être dans un langage d'exécution quelconque par exemple Java ou C.

4.3.2 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [BSM+03] est un framework de modélisation open-source, intégré dans Eclipse. Il permet de spécifier les modèles structurels, simples, indépendants des plateformes et à partir desquels le code peut être généré. Un des supports par défaut de EMF est la génération de code d'éditeurs pour les modèles. Les éditeurs de EMF sont capables de composer les modèles par références.

A. Modèles

Les modèles composés sont en Ecore. Le type de composition est par relations.

B. Éléments de composition : Les références

Les références (**éléments de composition**) sont créées par les éditeurs EMF générés à partir des métamodèles. Ces métamodèles sont définis par un métamodèle étendu du standard MOF [5], appelé Ecore (**formalisme**). Un métamodèle (i.e. un modèle en Ecore) est décrit par des objets `EClass`. Ces objets peuvent avoir des relations (`eReferences`) et des attributs (`eAttributes`). Il est possible de créer les références inter métamodèles, i.e. les références relient deux concepts de deux métamodèles différents. Ces références sont définies par le concept `EReference` dans le métamodèle Ecore. L'instanciation de références au niveau de métamodèles permet de créer les liens entre les modèles (**composition par relations**). Les modèles et les liens inter modèles font ensemble le modèle composite.

Du point de vue conceptuel, une référence inter métamodèle ressemble aux autres références définies dans le même métamodèle. C'est-à-dire qu'elle est définie aussi par le type `EReference` du métamodèle Ecore. Elle peut être unidirectionnelle ou bidirectionnelle, avec la cardinalité 1 ou multiple. Elle peut être aussi du type de containement ou non. Conceptuellement, il n'y a pas de différence entre les types de références, quelles soient inter métamodèles ou non, sauf que des références inter métamodèles relient deux concepts de deux métamodèles différents alors que les autres relient des concepts dans le même métamodèle.

Du point de vue technique, des relations inter métamodèles sont représentées comme des proxis. Les métamodèles référencés seront chargés lorsqu'ils sont demandés (*lazy loading*).

C. Mécanisme de composition : l'établissement des références

Le processus de composition de modèles par les éditeurs EMF peut être résumé comme suivant :

- D'abord, nous établissons les références entre les métamodèles. Pour ce faire, il faut charger les métamodèles dans un éditeur de Ecore. EMF gère les métamodèles comme des resources. Une resource est un document persistant contenant les objets du modèle. La fonctionnalité *Charger Ressource* est fournie pour importer des métamodèles. Une fois que les métamodèles sont chargés dans un même éditeur Ecore, on peut les composer. La composition consiste à créer des références entre les métamodèles importés. L'éditeur qui compose est celui du métamodèle composite.
- Une fois le métamodèle composite obtenu, nous utilisons EMF pour générer les éditeurs à partir de ce métamodèle. Nous allons générer, pour chaque sous métamodèle composé, un

5. Meta Object Facility.

éditeur séparé. Dans ces éditeurs, nous pouvons construire des modèles distincts conformes à ces différents métamodèles composés. Puis, nous générons un troisième éditeur pour le métamodèle composite (en fait, c'est l'éditeur pour la partie des références établies entre les métamodèles qui évidemment ne sont pas connues par les éditeurs des métamodèles composés). Apparemment, du point de vue de l'interface, ces éditeurs sont séparés, mais ils peuvent communiquer parce qu'ils partagent et manipulent un même modèle de donnée (i.e. le métamodèle composite). Dans l'éditeur composite qui vient d'être généré, en utilisant le même mécanisme de chargement de ressources [6], nous pouvons réimporter les modèles déjà créés dans différents éditeurs et instancier les références entre eux. Le résultat est un modèle composite.

4.3.3 Epsilon Merging Language

Epsilon Merging Language (EML) est un langage basé sur des règles (*rule-based language*), indépendant des métamodèles, permettant de spécifier la composition de modèles par fusion.

EML est construit au dessus de EOL [7] qui est un langage générique développé sur OCL supportant les tâches de gestion de modèles génériques telles que la création, la navigation et la modification des modèles. EOL peut être réutilisé pour construire les langages de gestion de modèles centrés sur des tâches spécifiques (*task-specific languages*) comme la fusion, la comparaison, la transformation des modèles. Actuellement, il y a trois langages de gestion de modèles spécifiques implémentés sur EOL. Il s'agit du langage de comparaison de modèles - ECL [8], de fusion de modèles - EML [9], de transformation de modèles - ETL [10], et un prototype du langage de transformation de modèle textuel - EGL [11] - est en cours de développement. La discussion dans cette section se concentre sur EML, le langage réalise la composition de modèles par le mécanisme de fusion.

A. Modèle

Les modèles composés sont en EOL. Le type de composition est par fusion.

B. Éléments de composition : Les spécifications EML

EML (**formalisme**) est basé sur des règles. Une spécification EML (**éléments de composition**) est un ensemble des règles décrivant comment composer les modèles. Il y a trois types de règle dans EML : comparaison, fusion et transformation (**abstractions prédéfinies**).

Une règle de comparaison déclare un nom et deux paramètres du type des méta-classes des instances à comparer. Le corps de la règle est divisé en deux parties : une partie comparaison (`compare`), et une partie pour vérifier la conformité (`conform`). Ces deux parties contiennent les critères qui déterminent si les instances correspondents sont conformes. A l'exécution, les règles de comparaison sont appliquées sur toutes les instances de deux modèles, la partie de comparaison est appliquée avant la partie de vérifier la conformité. Autrement dit, la partie `conform` raffine la partie `compare`.

Une règle de fusion déclare un nom et deux paramètres représentant les instances à fusionner. La différence par rapport à la déclaration d'une la règle de comparaison est que la règle de fusion

6. Ce mécanisme est utilisé à la fois pour la composition de métamodèles et de modèle dans le cas EMF. Cette approche considère tous les métamodèles et modèles comme des ressources et les traite de la même manière.
7. Epsilon Object Language.
8. Epsilon Comparaison Language.
9. Epsilon Merging Language.
10. Epsilon Transformation Language.
11. Epsilon Generation Language.

déclare en plus une variable pour l'objet fusionné, retourné par la règle, créé dans le modèle cible. La sémantique de fusion exprimée dans le corps de la règle est définie par la personne qui écrit la spécification EML. À l'exécution, les règles de fusion ne s'appliquent que sur les paires d'instances qui ont été déterminés comme correspondants et conforme à la fois. Autrement dit, l'entrée de l'exécution des règles de fusion est les paires d'instances catégorisées par l'exécution des règles de comparaison.

Une règle de transformation par contre ne s'applique que sur les paires d'instances qui ne sont pas ni correspondants, ni conformes. Une règle de transformation déclare deux paramètres, l'un représente l'objet source et l'autre représente l'objet cible obtenu par la transformation. Similairement à la règle de fusion, la sémantique de la règle de transformation est définie par la personne qui écrit la spécification. Le corps de la règle est un ensemble d'expressions EOL. L'exécution des règles de transformation a besoin aussi des données produites par l'exécution des règles de comparaison.

Le code 4.3 montre un exemple de règle dans EML illustrant la comparaison, la fusion et la transformation sur deux classes (`Left!Class` et `Right!Class`). La règle de comparaison retourne vrai si et seulement si les deux classes sont soit concrètes, soit abstraites à la fois et leurs nom et leurs espaces de noms sont les mêmes. La règle de fusion crée, dans le modèle composite résultat, une classe dont le nom et l'espace de noms viennent de la classe de gauche, les propriétés sont l'union des propriétés de deux classes en entrée. La règle de transformation crée l'objet destination dont le nom, l'espace de nom et les propriétés proviennent de l'objet source.

```
rule MatchClasses
  match l: Left!Class
  with r: Right!Class
  compare
    return l.name = r.name and l.namespace.matches(r.namespace);

  conform
    return l.isAbstract = r.isAbstract;


  rule MergeClasses
  merge l: Left!Class
  with r: Right!Class
  into m: Merged!Class
    m.name := l.name;
    m.namespace := l.namespace.equivalent();
    m.feature := l.feature.includeAll(r.feature).equivalent();


rule ClassToClass
  transform source:Left!Class
  to target: Right!Class
    target.name := source.name;
    target.namespace := source.namespace.equivalent();
    target.feature := source.feature.equivalent();
```

FIGURE 4.3 – Règles de composition dans EML

Dans EML, les règles ont la capacité d'étendre les autres règles par la déclaration `extends`. Cette capacité permet à l'utilisateur de réutiliser des librairies de règles déjà construites et de

ce fait, d'être capable d'ajouter ses sémantiques de composition supplémentaires sans réécrire tous les codes de sa règle. De plus, la modularisation d'une spécification EML est mieux gérée également.

C. Mécanisme de composition : fusion

Le processus de composition de modèles dans EML est divisé en 2 phases principales :

- **Phase de comparaison** : cette phase exécute les règles de comparaison. Pour chaque paire d'éléments de deux modèles, le moteur EML va chercher une règle de comparaison non abstraite approprié aux types d'élément de cette paire, si une telle règle est trouvée, le moteur va l'exécuter pour déterminer si ses éléments sont correspondants et conformes ou non (les parties `compare` et `conform` de la règle doivent retourner vrai) ; si le nombre des règles applicables trouvées est supérieur 1, le processus de composition sera terminé après d'avoir envoyé un warning à l'utilisateur ; si la règle n'est pas trouvée, les stratégies de comparaison attachées à la spécification EML vont être utilisés. Les stratégies sont en fait des règles par défaut qui sont attachées à la spécification EML en terme d'algorithmes *pluggables*.
  Après l'exécution des règles de comparaison, les éléments des modèles sont classés en 4 catégories : 1) les éléments correspondants et conformes, 2) les éléments correspondants mais non conformes, 3) les éléments ni correspondants ni conformes, 4) les éléments sur lesquels il n'y a aucune la règle de comparaison approprié à appliquer. Ces paires d'éléments sont sauvegardées dans une trace des correspondances (*matchTrace*). Cette liste est utilisée comme les données en entrée de la phase de fusion.
- **Phase de fusion** : En se basant sur le résultat de comparaison, la phase de fusion réalise deux activités :

1. Les éléments qui correspondent et sont conformes seront fusionnés. Ceci est fait par les règles de fusion.
2. Les éléments qui ne correspondent pas et ne sont pas conformes seront transformés vers les éléments du modèle cible.

Dans le cas de la catégorie 2, le processus de composition sera annulé car EML ne suppose pas de composer les modèles en conflits. Dans le cas de la catégorie 4, un avertissement est lancé pour qu'avertir l'utilisateur que les éléments dans cette catégorie peuvent provenir du fait la spécification EML est incomplète.

4.3.4 Generic Modeling Enviroment

Generic Modeling Enviroment (GME) [GME, Dav, KML+04, LMV01, MBL07] est un environnement de modélisation générique pouvant être rendu spécifique en utilisant les paradigmes de modélisation (les langages de modélisation dédié aux domaines - DSMLs [12]) qui sont construits aussi par l'environnement GME générique lui-même. Un paradigme de modélisation formalise un métamodèle définissant la syntaxe, la sémantique et la présentation concrète du DSML.

Les environnements de modélisation spécifiques configurés permettent ensuite de créer les modèles. Ces modèles sont sauvegardés dans la base de données et puis sont utilisés pour générer ou synthétiser automatiquement les applications.

Tous les modèles définis par GME, quelque soit le paradigme, partagent un même ensemble de concepts de modélisation générique, indépendant des paradigmes et partagés par tous les environnements GME configurés. Ces concepts sont appelés FCOs (First Class Objects) (**formalisme**).

12. Domain-Specific Modeling Languages.

Ils définissent des modèles d'un point de vue générique de sorte qu'un modèle est un ensemble de parties composées sans préciser la nature de la partie. Cette nature va être précisée par les concepts métiers du paradigme. Les concepts de FCOs ont capable de composer/décomposer des parts permettant de créer et de gérer des modèles complexes.

A. Modèle

Les modèles composés sont en FCOs. Le type de composition est par relation de FCOs.

B. Éléments de composition : Les relations de FCOs

Nous présentons ci-dessous les concepts de FCOs et les techniques de composition/décomposition de modèles fournis par ces concepts (**abstractions de composition**). Les concepts seront regroupés selon une vision centré sur la composition de façon suivante : 1) les concepts les plus basiques, 2) les concepts de décomposition, 3) les concepts de composition.

- **Les concepts les plus basiques** :
  - `Model` : c'est le concept qui représente les modèles. Un modèle est un objet composite. Il contient d'autres objets (autres parts, selon la terminologie de GME) comme leurs structures internes. Les types d'objet acceptés comme le contenu d'un modèle peuvent être : des *atomes*, des *références*, des *ensembles* (*sets*), des *connections* ou même d'autres modèles.
  - `Atome` : représente un objet élémentaire, indivisible, qui ne contient pas d'autres parts.
- **Les concepts de décomposition** :
  - `Model Hierarchy` : Un modèle peut contenir d'autres modèles ; un modèle peut donc être structuré de façon hiérarchique. Les objets d'un modèle sont répartis dans des sous-modèles. Plusieurs détails sont cachés. La complexité d'un modèle complexe peut être donc géré.
  - `Aspect` : Une solution alternative pour gérer la complexité d'un modèle est d'utiliser des aspects. Le modèle est partitionné en aspects. Un aspect peut être visible ou caché ce qui permet de contrôler la visibilité du modèle en utilisant des vues différentes.
- **Les concepts de composition** :
  - `Connection` : Une connexion exprime une relation entre deux objets. Elle est peut être définie entre deux atomes, un atomes et un modèle, un modèle et un modèle, même entre des ensemble, des références à condition que ce soient des objets au même niveau hiérarchique ou au niveau plus bas. Une connexion peut être orienté ou non. Sa sémantique est définie par le paradigme de modélisation.
  - `Set` : Un set spécifie une relation entre un groupe d'objet. Les types de part qui peuvent participer dans un *set* sont les même que dans le cas des *connections*. La seule restriction pour des *sets* est que leurs membres doivent avoir un même parent et être visible dans même aspect. Une *connection* représente une relation binaire, un *set* par contre représente une relation n-aire.
  - `Reference` : Ce concept est similaire à un pointeur dans les langages de programmation. Une référence n'est pas un objet réel. Elle est simplement un moyen pour "pointer" vers un objet. Une référence est binaire. Elle établit une relation entre un modèle (qui contient cette référence) et l'objet référencé. Contrairement aux *connections* et aux *sets*, les références permettent non seulement d'associer deux objets d'une même hiérarchie mais encore de différentes hiérarchies de modèles. C'est un type de relation spécifique permettant à un modèle d'accéder directement aux parts contenu dans un autre modèle.

C. Mécanismes de composition : l'établissement des références

GME est une approche de métamodélisation, il fournit les techniques de composition de métamodèles et aussi celles de modèles.

- **Composition de métamodèles** : GME propose trois types de relations pour la composition de métamodèles : *équivalence*, *héritage d'implémentation* et *héritage d'interface* [LNK+01].
  - *Equivalence* : Cet opérateur concerne l'opération "union" de deux concepts définis dans deux paradigmes. Cette union produit un nouveau concept unique qui inclut tous les attributs et les associations des deux concepts unifiés.
  - *Héritage d'implémentation* et *Héritage d'interface* : ces deux opérateurs utilisent des mécanismes de spécification de classes typique d'UML. Dans l'héritage d'implémentation, le sous-concept peut hériter de tous les attributs du super concept et de toutes les relations de containement dans lesquelles le super concept joue le rôle de conteneur. L'héritage d'interface, par contre, ne permet pas d'hériter des relations.
    Ces opérateurs sont utiles dans le cas où on veut étendre un métamodèle (grâce aux mécanismes de spécification de classe) ou produire un troisième métamodèle unifiant les deux métamodèles composés.
- **Composition de modèles** : GME distingue deux cas : la composition de modèles du même métamodèle et la composition de modèles de différents métamodèles.
  La composition de modèles du même métamodèle peut être faite par le concept de référence de FCOs. Par contre, GME n'a pas capable de composer les modèles créés par différents métamodèles quoi qu'il fournisse la capacité de composer les métamodèles. Il est impossible d'importer des modèles existants et de les réutiliser par établissement de relations entre eux. Une fois qu'on a un nouveau paradigme composite, il n'est que possible de créer de nouveaux modèles.

4.3.5 Kompose

Kompose est un outil de composition implémentant l'approche de composition de modèles basé sur les signatures [FBFG07]. Cet outil est lancé récemment par l'équipe Triskell[13] , IRISA (Rennes, France). Au moment de la rédaction de cette thèse, Kompose est encore un prototype (version 0.0.1).

A. Modèle

Les modèles composés sont en Ecore. Le type de composition est la fusion. Notons que Kompose a commencé historiquement par la perspective de modélisation orientée aspect, donc il a distingué les modèles primaires aux les modèles d'aspect. Cependant, comme ses auteurs disent dans [FBFG07], cette distinction n'a pas de signification spéciale sauf pour avoir une bonne vision conceptuelle de la séparation des préoccupations. Dans la pratique, les supports de manipulations sur les deux types sont les mêmes.

B. Eléments de composition : les directives de composition

Kompose a un langage de composition de modèle permettant de décrire des spécifications de composition appelées directives de composition. Le concept central de ce langage est le Composeur (`Composer`). Un composeur représente une opération de composition effectuée sur deux métamodèles. Il est lui même l'objet racine qui rassemble toutes les directives de composition.

13. http://www.irisa.fr/activites/equipes/triskell

La structure d'un composeur comprends : un métamodèle primaire en entrée, un métamodèle aspect en entrée, le nom du métamodèle composite en sortie et l'ensemble des directives de composition.

Kompose fournit deux types de directives de composition : *pre-merge directive* et *pos-merge directive*. Les *pre-merge directives* sont appliquées sur les métamodèles avant de les composer, les *post-premege directives* sont appliquées sur le métamodèle composite avant de le produire. Les premières spécifient les modifications simples sur les modèles en entrée telles que le renommage, la suppression ou l'ajout d'éléments afin de forcer ou empêcher la fusion. Les deuxièmes réconcilient le modèle fusionné pour qu'il soit fiable et cohérent. Cinq types d'actions peuvent être définies dans une directive afin de désigner ce qu'elle va faire : Ajouter (`Add`), Supprimer (`Remove`), Créer (`Create`), Mettre (`Set`).

Le langage de composition de Kompose a une syntaxe concrète textuelle. Un exemple de cette syntaxe est dans la figure 4.4. Sa syntaxe abstraite est en Ecore. On peut "programmer" des programmes de composition soit en syntaxe textuelle à l'aide d'un éditeur textuel de Kompose, soit directement en syntaxe abstraite à l'aide de l'éditeur EMF.

```
// Primary Model URI
PM "platform:/resource/KomposeSamples/bank/Bank.ecore"

// Aspect Model URI
AM "platform:/resource/KomposeSamples/bank/BLP.ecore"

// Composed Model URI
CM "platform:/resource/KomposeSamples/bank/BankBLP.ecore"

// predirectives for primary model
PMPre

// predirectives for aspect model
AMPre
// Rename package BLP to Bank
BLP.name = "Bank"


// postdirectives
Post
Bank::Controller.eOperations - Bank::Controller::transfer
Bank::Controller.eOperations - Bank::Controller::withdraw
```

FIGURE 4.4 – Un exemple de la syntaxe concrète textuelle de Kompose

C. Mécanisme de composition : fusion

La composition de modèles dans Kompose est réalisé en deux phases : une phase de comparaison et une phase de fusion.

- **Phase de comparaison** : Les éléments de modèle qui décrivent les différentes vues du même concept sont identifiées automatiquement en se basant sur la comparaison de leurs signatures. Ils seront fusionnés (dans la phase de fusion) pour obtenir la vue intégrale du concept.
  Le mécanisme de détermination des éléments à fusionner se base sur la comparaison de leurs signatures. Une signature est un ensemble de valeurs extraites de certaines propriétés de l'élément du modèle. L'ensemble des propriétés du type de l'élément définit le type de la signature. Par exemple, le type de la signature d'une classe UML consiste en des propriétés

`name` et `isAbstract` ; la signature d'une classe concrète, par exemple `Etudiant`, sera peut être `name` = `Etudiant`, `isAstract` = `false`

- **Phase de fusion** : Les éléments de modèle ont la même valeur de signature vont être fusionnés. Dans le cas contraire, ils seront simplement copiés vers le modèle final.
  Pendant le processus de fusion, il est probable qu'auront lieu des conflits, par exemple deux éléments se correspondent aux signatures (e.g, deux classes de même nom) mais diffèrent par certains autres propriétés (e.g, l'une est abstraite, l'autre non). Kompose utilise des directives de composition pour forcer la fusion, empêcher la fusion ou redéfinir la stratégie de fusion par défaut (**adaptation de modèles sources**). Par exemple, dans le cas des classes abstraites et non abstraites ci dessus, les directives peuvent forcer la classe fusionnée finale de prendre la valeur `isAbstract = false`.

Kompose suppose que les modèles composés sont représentés par les objets dont les structures peuvent être obtenues par la réflexion, par exemple les modèles sont représentés par les instances de la classe `EMOF : :Objet`. L'algorithme de composition calcule la similarité des signatures des éléments et les fusionne en se basant sur des instances de métamodèle.

4.3.6 Xactium XMF/XMF-Mosaic

eXecutable Metamodelling Facility (XMF) [CESW04] est un framework de méta-modélisation inventé par l'entreprise Xactium [14] qui permet de concevoir facilement des DSMLs [15] exécutables. XMF-Mosaic [16] est l'environnement implémentant ce framework.

A. Modèle

Les modèles composés sont créés dans les langages qui sont modélisés par le langage de métamodélisation XMF (rappelons que XMF est une approche de métamodélisation). Le langage de métamodélisation XMF en fait se compose de plusieurs sous langages :

- XCore et XOCL fournissent la syntaxe abstraite et la sémantique du langage.
- XEBNF fournit la syntaxe concrète textuelle du langage.
- Un propre langage graphique fournit la syntaxe concrète diagrammatique du langage.

XCore et XOCL XCore est le coeur de XMF, il s'agit d'un langage noyau fournissant tous les concepts et primitives de méta modélisation orienté objet tels que *packages*, *classes*, *associations*, *objets* nécessaires pour concevoir la structure d'un langage. XCore est une extension de MOF. Cependant, contrairement à MOF, XCore est exécutable. L'exécutabilité de XCore vient de deux facteurs. D'une part, XCore utilise un langage de contrainte OCL extensible, appelé XOCL, qui permet de décrire la sémantique comportementale du langage ; d'autre part XCore définit explicitement un concept `Operation` qui est le moyen permettant de manipuler l'état des modèles ; ceci est un avantage sur MOF [17]. XCore et XOCL permettent de modéliser la syntaxe abstraite et la sémantique du langage XMF. Nous ajoutons que lors de la définition de la syntaxe abstraite, XOCL est utilisé aussi pour décrire des règles de validations (*well-formed rules*) pour que le langage puisse éliminer les modèles invalides.

XEBNF XMF utilise une extension du langage EBNF, appelé XEBNF - un langage d'analyseur syntaxique générique (*generic parser language*), pour définir la syntaxe concrète textuelle du

14. http://jessica.xactium.com/
15. Domain-Specific Modeling Languages.
16. http://www.ceteva.com/xmf.html
17. Dans MOF, les comportements sont décrits par un langage hybride entre OCL et l'anglais qui n'est pas exécutable

langage.

Notons qu'en se basant sur les standards tels que QVT, OCL, MOF, les langages de XMF sont indépendants des plateformes techniques, même l'implémentation du comportement du langage est réalisée dans un langage indépendant des plateformes (i.e. XOCL). Ceci donne une grande flexibilité à XMF. La flexibilité et l'exécutabilité sont les avantages principaux de XMF.

Les modèles composés dans XMF sont des diagrammes de classe UML. Le type de composition est par relations appelées *mappings*.

B. Elements de composition : Mappings - des relations entre les modèles

Dans XMF, un *mapping* est définit de la façon suivante :

> *A mapping is a relationship or transformation between models or programs written in the same or different languages.* [CESW04]

Dans cette définition, on peut voir qu'il y a deux visions pour le *mapping* : une vision de transformation spécifique du MDA ; l'autre vision est plus générique. Générique dans le sens où un mapping peut être utilisé pour plusieurs objectifs différents, non seulement la transformation. Par exemple, on peut maintenir la cohérence entre deux modèles, gérer l'échange d'événement et de données, gérer la synchronisation et la communication entre deux modèles. La vision de transformation est plutôt liée à l'approche MDA. Par contre, nous nous s'intéressons à la composition de modèles, donc la vision générique se place bien dans notre optique.

Correspondant aux deux visions ci-dessus, XMF propose deux types de mappings : *mapping unidirectionnel* et *mapping synchronisé*.

- **Mapping unidirectionnel** : il est basé sur la vision de transformation. Le mapping unidirectionnel prends un (ou un ensemble de) modèle(s) en entrée(s) puis génère un modèle en sortie. Afin de décrire ce mapping, XMF fournit XMap. XMap est un langage de mappings unidirectionnels déclaratif, exécutable, basé sur pattern-matching similaire aux langages de transformation comme ATL [18] [JK05]. Les mappings unidirectionnels sont souvent utilisés dans la génération de code, par exemple pour transformer un modèle vers Java ou C++. Dans l'optique de composition, nous ne nous intéressons pas à ce type de mapping.
- **Mapping synchronisé** : C'est un mapping destiné à gérer la synchronisation entre deux modèles. Comme nous avons dit ci-dessus, les mappings synchronisés peuvent être utilisés dans plusieurs buts ; la gestion de la cohérence est seulement une parmi plusieurs applications de ce type de mapping. D'autres applications typiques du mapping synchronisé sont la gestion de multiple modèles d'un système, le support du "round trip engineering" etc.
  Afin de décrire des mapping synchronisés, XMF fournit XSync. Nous prenons un simple exemple dans [CESW04] pour illustrer la nature d'un mapping synchronisé et la syntaxe de XSync.
  Un mapping synchronisé consiste en une portée (*scope*) et une collection de règles de synchronisation (*rule*). La portée du mapping est une collection d'éléments sur lesquels le mapping est appliqué. Dans cet exemple, ce sont les instances *c1*, *c2*. Une règle décrit la condition sous laquelle une action est exécutée. Des actions sont des synchronisations qui peuvent être effectuées des deux côtés. Une règle à la forme : un pattern, une condition booléenne *when* et une action *do*. Cet exemple est un mapping synchronisé qui prend deux classes quelconques et synchronise le nom de la première classe avec celui de la deuxième.
  La première remarque sur l'exemple ci-dessus est que XSync permet de définir des opérations pour des mappings. Cela veut dire que dans le cas de XMF, les modèles sont composés structurellement et comportementallement.

18. Atlas Transformation Language.

La deuxième remarque est que le mapping peut ne pas être la relation entre deux instances mais être un ensemble d'instances participant au même mapping. C'est-à-dire, la valeur du *scope* n'est pas une paire mais un tuple.

En résumé, dans XMF, les mappings sont les moyens de composition de modèles (**éléments de composition**). XMap et XSync sont les langage de composition (**formalisme**). La sémantique des mappings est fournie par l'utilisateur qui les écrit (**extensibilité du langage**). Ces mappings sont exécutables.

```
context Root
  @Operation sameName(c1,c2)
    @XSync
    @Scope
      Setc1,c2
    end
    @Rule r1 1
     x1 = Class[name = n1] when x1 = c1;
     x2 = Class[name = n2] when x2 = c2 and n1 <> n2
     do x1.name := x2.name
    end
   end
end
```

FIGURE 4.5 – Une mapping de synchronisation [CESW04]

C. Mécanisme de composition : l'établissement des mappings et l'exécution des mappings

Il faut distinguer deux formes de composition dans XMF : la composition de langages et la composition de modèles.

- **Composition de langages** : ceci concerne l'établissement des mappings entre les langages (i.e. écrire les codes de mappings comme l'exemple dans la figure 4.5).
- **Composition de modèles** : ceci concerne l'exécution des mappings pour la synchronisation d'état de deux modèles. L'exécution des mappings est réalisée par le moteur d'exécution de XMF.

A part des mécanismes de composition de langages et de modèles, XMF propose aussi les mécanismes d'extension de langages tels que :

- utiliser des mécanismes de réutilisation classiques de la programmation orientée objet tels que la spécification de classes.
- utiliser des stéréotypes, tags, et profils : définir le nouveau langage comme un profil basé sur des concepts d'un langage développé.
- utiliser le mécanisme de spécification de package et méta package : le package du nouveau langage est défini comme spécifiant le package d'un langage développé.
- traduire de nouveaux concepts vers un langage prédéfini.
  Ceux ci ont pour objectif d'économiser l'énergie de développement de nouveaux langages en réutilisant les langages existants. Cependant, ces mécanismes ne sont pas liés à l'idée de composition de modèles. Un nouveau langage créé par la spécification, en général, ne va pas pouvoir réutiliser des modèles créés avec le langage qu'il spécifie car, comme dans la programmation objet orienté, l'objet parent n'est pas capable d'être "casté" vers l'objet fils. Ce nouveau langage doit créer à nouveau tous ses modèles. De ce point de vue, cette technique de XMF ne permet pas de résoudre le problème de composition de modèles.

4.4 Bilan

Dans ce chapitre, nous avons fait une classification des approches de composition de modèles et étudié six approches de ce sujet. Le tableau 4.2 synthétise les caractéristiques de ces approches.

	Type de composition	**Modèle**	**Element(s) de composition**	**Formalisme de composition**	**Mécanisme de composition**
AMW	par opérateurs (définis par l'utilisateur)	en Ecore	modèles de tissages et transformations HOT.	langage de tissage AMW	tissage et transformation
EMF Editeur	par relations	en Ecore	références	*EReference* de du langage Ecore	établissement des références
EML	par opérateurs (fusion)	en EOL	spécification des règles de composition	langage de fusion EML	fusion
GME	par relations	en FCOs	références	*Références* de FCOs	établissement des références
Kompose	par opérateurs (fusion)	en Ecore	spécification des directives de composition	langage de fusion Kompose	fusion
XMF-Mosaic	par relations	en XCore, XOCL, XEBNF	mappings	XSync	établissement des mappings et l'exécution des mappings

TABLE 4.2 – Bilan des approches de composition de modèles

Discussions Dans notre travail, nous nous intéressons particulièrement à la *réutilisation des modèles* crées par *différents métamodèles* en préservant la structure modèles composés et assurant l'exécutabilité du modèle composite. Donc, dans ce paragraphe, les deux points importants que nous voulons discuter sont 1) la relation entre les métamodèles sources et leur rapport avec le métamodèle cible et 2) la réutilisation de modèles (sous la perspective de préservation de la structure et d'exécutabilité) dans les approches présentées.

Relation entre les métamodèles sources et cible Supposons que nous composons le modèle *m1* du métamodèle M1 avec le modèle *m2* du métamodèle M2 pour produire le modèle *m3* du métamodèle M3. En observant les approches présentées, nous trouvons que la relation entre *M1*, *M2* et *M3* peut être une des possibilités suivantes :

- **Possibilité 1** : $M1 \equiv M2 \equiv M3$: les modèles *m1*, *m2* et *m3* ont le même métamodèle.
- **Possibilité 2** : $M1 \neq M2$ et $M3 = M1 \cup M2$: les modèles *m1* et *m2* sont de différents métamodèles et le métamodèle de m3 est une union de M1 et M2.
- **Possibilité 3** : $M1 \neq M2 \neq M3$: les modèles *m1*, *m2* et *m3* sont crées par trois métamodèles différents.

La composition des modèles de même métamodèle (la possibilité 1) est dite endogène, et l'inverse (les possibilités 2 et 3) est dite exogène.

En général, les approches qui permettent la composition exogène est capable également de faire la composition endogène. Dans les approches présentées, toutes sont exogènes sauf GME.

GME permet de composer les modèles du même paradigme mais incapable de composer les modèles crées différents paradigmes. Il ne répond donc pas nos besoins.

Les possibilités 1 et 2 impliquent que la composition de modèles est une union des éléments des modèles *m1* et *m2*. Ceci est le cas de Kompose dans lequel les éléments en correspondances sont fusionnés en un élément unique et les éléments non correspondants seront copiés vers le modèle *m3*. Ceci est aussi les cas de EMF et XMF dans lesquels le modèle *m3* est l'union des *m1* et *m2* plus les liens entre eux ($m3 = m1 \cup m2 \cup relations$). Dans le cas EMF, les liens sont des *références*, dans le cas XMF, ce sont des *mappings*.

Le cas le plus complexe est la possibilité 3. C'est le cas dans EML, une spécification EML décrit non seulement les règles de correspondance et celles de fusion pour fusionner les éléments correspondants mais aussi les règles de transformation pour *transformer* les éléments non correspondants vers le modèle cible. Le fait qu'il y a une transformation dans le processus de composition de EML implique que cette approche suppose le cas où M1, M2, M3 sont différents. Ceci est un point fort intéressant de cette approche.

Il n'est pas facile de classer AWM entre les possibilités 2 et 3 parce que tout dépend de la sémantique des transformations générées à partir des modèles de tissage. La sémantique de ces transformations est fournie par l'utilisateur. Il n'est donc pas possible de savoir si l'utilisateur prend en compte ou pas le problème de l'hétérogénéité des métamodèles sources et cible dans sa transformation.

Une remarque importante est qu'il faut distinguer les transformations HOT dans AMW et les règles de transformation dans EML parce qu'elles ne sont pas au même niveau d'abstraction. Une transformation HOT ne produit pas les éléments du modèle final (comme les règles de transformation EML qui l'ont fait). En revanche, elle produit une autre transformation. C'est bien cette dernière qui est au niveau d'abstraction des transformations EML et qui va produire les éléments du modèle final.

Jusqu'ici, ces approches exogènes semblent répondre bien nos besoins. Cependant, il y a quelques autres contraints dans notre approche que ceux ci ne peuvent pas résoudre. Nous allons le discuter dans le paragraphe suivant.

Réutilisation de modèles La réutilisation de modèles doit assurer deux critères : la structure des modèles composés préservées et le modèle composite continue à être exécutable.

GME ne permet pas de réutiliser les modèles de différents métamodèles. C'est la raison pour laquelle nous ne pouvons pas adopter sa solution.

EMF et XMF sont les approches de composition par les relations, leur différence est que XMF permet de réutiliser les modèles en préservant leur structure tant dis que dans EMF, les modèles peuvent être réutilisés et composés mais leur structure est transformé. De plus, la limite de EMF est qu'il ne supporte pas les mécanismes d'exécution, cette approche est donc insuffisant pour notre travail. Les modèles dans XMF sont par contre exécutables, ceci est intéressant mais il y a deux points inadéquats de cette approche pour nous : d'abord le langage XCore est spécifique et moins populaire que les autres langages, par exemple Ecore, il faudrait donc penser au problème de l'interopérabilité lors de l'utilisation de XCore ; ensuite, XMF Mosaic n'est pas *open-source* et pas gratuit.

Kompose et EML permettent aussi la composition exogène, surtout EML a résolu très bien le problème de l'hétérogénéité des modèles sources et cible. Cependant, dans notre travail, les approches basées sur la fusion de modèles transforment la structure des modèles composés, de ce fait, nous ne pouvons pas les adopter.

AMW permet de réutiliser les modèles existants et semi-manuellement de créer les correspondances entre eux. Ces correspondances peuvent être utilisées par la suite pour les divers objectifs (ex. transformation, génération de code, composition, traçage etc.) Jusqu'ici, les structures des

modèles sont préservées. Cependant, dans le but de composition, cette approche propose d'utiliser des transformations pour transformer les modèles sources vers le modèle composite. De notre point de vue de préservation de la structure des modèles composés, nous ne sommes pas favorable la proposition de composition par fusion. Par contre, si on considère les transformations comme des calculs exécutés sur les modèles composés et leurs correspondances, ceci est beaucoup plus proche de notre idée de l'exécution de modèles et nous nous intéressons plutôt les transformations dans ce sens. A ce point de vue, la solution de AMW est intéressant pour nous.

Le tableau 4.3 résume la capacité des approches selon nos besoins abordés dans la disscussion ci-dessus.

Les approches	**Besoins**		
	Hétérogénéité de métamodèles des modèles composés	**Préservation de structure de modèles composés**	**Exécutabilité de modèle composite**
AMW	oui	oui	non
EMF Editeur	oui	non	non
EML	oui	non	non
GME	non	-	non
Kompose	oui	non	non
XMF-Mosaic	oui	oui	oui (mais le langage spécifique et l'outil payé)

TABLE 4.3 – Les défauts des approches de composition de modèle actuellees

Dans le chapitre suivant, nous allons présenter notre architecture Mélusine qui est le contexte de ce travail. En fait, Mélusine est une démarche et un environnement permettant la construction des systèmes complexes. Il est réalisé depuis longtemps au travers plusieurs traveaux de thèse dans notre équipe. Cette thèse repose sur ces traveaux. L'objectif du chapitre suivant est donc de les résumer, de donner une vue globale de notre démarche, et à partir de là, d'identifier notre travail dans ce contexte.

Chapitre 5

Architecture Mélusine : Contexte et Objectifs

5.1 Introduction

Le travail de cette thèse consiste à proposer une approche de composition de modèles pour la construction des systèmes à grande échelle. Deux raisons principales nous poussent à investiguer cette idée :

La première raison provient du fait que nous avons été confrontés à ce problème. Depuis longtemps, notre équipe développe une approche du Génie Logiciel par les modèles. L'idée principale est d'utiliser le modèle pour séparer la partie conceptuelle de l'implémentation d'une application. Une machine virtuelle exécute le modèle et de cette manière guide le fonctionnement d'un ensemble des composants logiciels constituant l'implémentation de l'application. Nous disons que l'exécution de l'application est dirigée par l'exécution de son modèle. L'outil Mélusine [ELV03, EVC01, LEV03, Le04], développé par l'équipe, et qui supporte cette approche, permet de développer des applications de manière efficace, avec une bonne vision de la conception car la construction d'applications par spécification de modèles est plus intuitive et facile qu'en écrivant du code. En utilisant Mélusine pour développer des applications à grande échelle, notre approche a évolué. Il faut permettre de construire des applications de grande taille complexes par extension d'une application existante ou par l'intégration d'applications de petite taille. Ce besoin exige de disposer d'un mécanisme de composition d'applications. Nos premières expérimentations sur ce sujet (même de façon très ad-hoc au début) nous ont montré que la composition d'applications requiert la composition de modèles conceptuels. C'est pour cette raison que le problème de la composition de modèles est pour nous un problème critique.

La deuxième raison provient de la réalité actuelle de l'IDM. Notre approche Mélusine demande un mécanisme de composition de modèles exécutables (c'est-à-dire, la composition requiert la composition des machines virtuelles associées). Malheureusement, l'IDM aujourd'hui, comme nous l'avons montré dans le chapitre 4, ne dispose pas de méthodes ou de technologie pour satisfaire nos besoins (bien que les approches existantes soient très intéressantes aussi). Dans ce contexte, il nous semble intéressant d'étudier ce sujet, d'une part pour résoudre nos problèmes dans le contexte Mélusine mais aussi pour apporter une contribution à l'IDM en général.

Les deux raisons présentées ci-dessus constituent la motivation de cette thèse et le contexte de ce travail. En ce qui concerne le contexte, nous pensons que nos discussions du chapitre 4 ont donné une vision assez détaillée de la réalité des techniques et des outils de composition de modèles disponibles actuellement (contexte général). Dans ce chapitre, nous voulons détailler le

contexte concret de ce travail : notre approche Mélusine à laquelle cette thèse contribue.

Ce chapitre est organisé de la manière suivante : Dans la section 5.2, nous présentons une vision globale de l'architecture Mélusine ; ses deux générations seront présentées rapidement dans les sous paragraphes 5.2.2 et 5.2.3. Le concept *Domaine* central de l'architecture qui est le grain de composition de notre travail sera particulièrement détaillé dans la section 5.3 ; la *Composition de Domaines* auquel est rattaché le sujet de cette thèse est présentée dans 5.4. Dans la context de l'architecture présenté, les objectifs de la thèse seront identifiés dans la section 5.5. La section 5.6 est une petite synthèse du chapitre.

5.2 Architecture Mélusine

5.2.1 Présentation

Notre approche Mélusine a été lancée au début des années 2000. Elle a été développée, validée et a évoluée au travers de plusieurs thèses réalisées au sein de notre équipe [Vil03, Le04, SJ05, Veg05].

Le terme Mélusine référence à la fois :

- Une méthodologie de développement d'applications à grande échelle facile à faire évoluer.
- Une architecture formalisant cette méthodologie.
- Et un environnement supportant la conception et l'exécution de cette architecture.

Du point de vue de la méthodologie, les deux premiers objectifs de Mélusine sont :

- Faciliter la construction d'applications à grande échelle.
- Faciliter l'évolution des applications déjà développées.

Pour réaliser ces objectifs, Mélusine a utilisé les solutions méthodologiques suivantes :

- Réutilisation.
- Remontée du niveau d'abstraction.
- Séparation des préoccupations.

Ces solutions sont des concepts fondamentaux du Génie Logiciel. La réutilisation permet de diminuer le temps de développement et d'évolution des applications. La séparation de préoccupation et la programmation à haut niveau d'abstraction permettent de gérer leur complexité.

Nous avons formalisé ces idées dans une architecture logicielle :

Mélusine est une architecture logicielle en trois couches permettant de développer des applications logicielles de grande taille par la réutilisation de composants logiciels hétérogènes existants et dirigé par les modèles. Les couches de Mélusine sont : la couche conceptuelle, la couche de médiation, et la couche outils.

En fait, l'architecture Mélusine a évoluée en deux générations. Dans la première génération, Mélusine se focalise sur le problème de l'*architecture d'applications* dont le concept central est celui de *Fédération*. Une Fédération est un ensemble des composants hétérogènes composés à travers un connecteur central appelé *Univers Commun*. Ceux-ci constituent une application. La deuxième génération de Mélusine s'oriente ensuite vers une approche de l'*architecture des familles d'applications* dans laquelle *Domaine* est le concept central. Un Domaine est un ensemble d'applications. Intuitivement, on peut imaginer Mélusine comme un ADL dont les deux générations définissent deux styles d'architecture différents.

L'architecture considérée dans cette thèse appartient à la deuxième génération. Toutefois, il y a un lien fort entre celle ci avec la précédente. Donc, nous allons présenter les deux.

5.2.2 La première génération : centrée architecture d'applications

Objectifs

Dans la première génération, l'objectif de Mélusine était de proposer une architecture d'application permettant l'intégration d'éléments logiciels hétérogènes existants et de diverses natures (e.g, des librairies, des logiciels patrimoniaux, des composants commerciaux sur étagère (COTS [1]) etc.) pour construire des applications complexes. Cet objectif fait partie des approches telles que BPM [2] et EAI [3] de cette époque favorisent l'intégration des composants logiciels hétérogènes dans les applications d'entreprise afin de fournir des fonctionnalités sophistiquées [Vil03, Le04].

Challenges

Les grands défis que l'architecture Mélusine a tenté de résoudre à cette époque sont :

- L'interopérabilité des éléments logiciels composés.
- L'hétérogénéité de ces éléments.

Solutions

Face à ces problèmes, l'architecture Mélusine a proposé les solutions suivantes :

- **À propos de l'interopérabilité des éléments logiciels** : L'architecture Mélusine a proposé une approche de *composition par coordination* dans laquelle les éléments logiciels sont guidés pour travailler ensemble.
  En effet, des éléments logiciels tels que les systèmes patrimoniaux, les composants commerciaux sur étagère etc. ont souvent été conçus indépendamment. Pour que ces éléments se comprennent et travaillent ensemble, il faut les connecter. Nous avons identifié deux options de composition que nous pouvons choisir : composition *directe* ou *indirecte*. La composition directe consiste à créer des connexions entre ces éléments logiciels. La composition indirecte consiste à les coordonner à l'aide d'un coordinateur. La composition directe fait un couplage fort entre les éléments composés ; par conséquent l'évolution de la composition devient difficile. La coordination au contraire garde un faible couplage et permet donc de préserver l'autonomie des participants et ainsi facilite l'évolution de la composition. Notre architecture a suivi la composition par coordination.
  La coordination est réalisée sous la direction d'un `Univers Commun`. Il s'agit d'un espace qui contient les concepts partagés (ex : `C1`, `C2` dans la figure 5.1), connus de tous les outils participant dans la composition. La construction d'une application consiste à instancier ces concepts. Ces instances maintiennent les états partagés qui coordonnent les états internes des instances des outils participants [4] de l'application.

1. Commercial Off-The-Shelf
2. Business Process Management.
3. Enterprise Application Integration.
4. Dans le reste de cette section, un outil participant à une composition est appelé succinctement un participant.

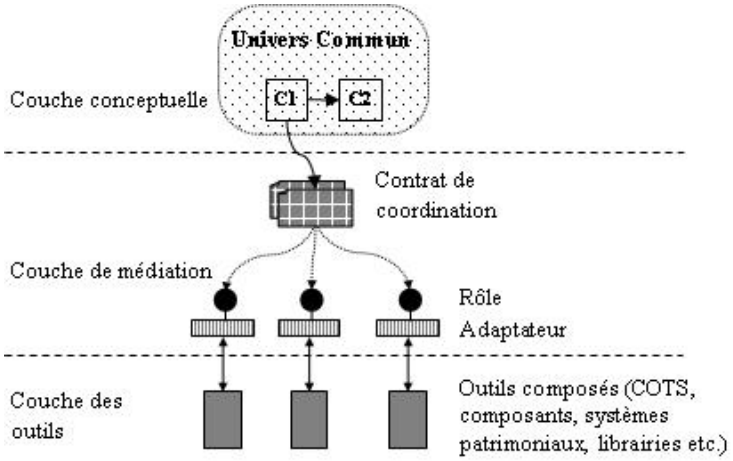


FIGURE 5.1 – Fédération

Lorsque l'état interne d'un participant change, il doit notifier l'univers commun pour que celui-ci modifie son propre état en conséquence, et ainsi permettre aux autres participants de se synchroniser. La communication entre l'univers commun et ses participants est faite par les contrats de coordination. L'univers commun propose ces contrats comme le moyen d'imposer explicitement ce que chaque participant doit faire. Si un outil veut participer dans la composition, il doit remplir et respecter les contrats de coordination proposés.

- **A propos de l'hétérogénéité des éléments logiciels participants** :
  Mélusine a proposé de réaliser des `adaptateurs`. Souvent les outils ont été conçus indépendamment. Ils sont souvent incompatibles. Chaque outil utilise un ensemble de concepts qui ne sont pas forcément ceux de l'univers commun. Un adaptateur permet alors de faire le lien entre les concepts de l'univers commun et les concepts utilisés par l'outil participant. De plus, pour diminuer le couplage entre l'outil participant et son univers commun, le concept de `rôle` a été introduit. Un rôle représente une abstraction des fonctionnalités du participant. Ce concept est matérialisé en terme d'une interface qui regroupe des méthodes fournies par le participant.

Fédération

La figure 5.1 montre l'architecture Mélusine de première génération :

- L'univers commun constitue la couche conceptuelle.
- Les contrats de coordination, les adaptateurs, et les rôles relient l'univers commun à ses participants, ils constituent la couche de médiation.
- La couche outils contient les participants.

Nous appelons une telle architecture une *Fédération* de logiciels. Le terme Fédération de logiciels est définie comme étant : "*une architecture logicielle qui matérialise la coordination, et qui permet de structurer les applications comme un ensemble de mondes qui coopèrent pour atteindre un but commun*" [Vil03].

Implémentation

L'architecture Mélusine propose non seulement les notions de briques architecturales (*Univers Commun*, *contrat de coordination*, *adaptateur*, *role*, *participant*), mais désigne encore comment

implémenter ces éléments pour obtenir une Fédération. Techniquement, les briques architecturales de Mélusine sont implémentés comme suit :

- L'univers commun est un petit programme implémentant la sémantique de l'application complète. Certaines méthodes de l'univers commun sont les points d'extension qui permettent aux participants d'étendre et d'enrichir la sémantique de l'univers commun (c'est-à-dire, qu'ils peuvent contribuer à remplir une tâche proposée dans la logique de coordination de l'univers commun).
- Les contrats de coordination sont implémentés par la technologie AOP. Une machine à objets étendus (MOE) permet d'intercepter les méthodes dans le programme de l'univers commun (i.e. des points d'extension) pour que les outils puissent venir compléter la sémantique de la coordination.
- Les rôles sont matérialisés par des interfaces qui définissent les méthodes fournies par les participants.
- Les adaptateurs sont du code supplémentaire ajouté pour adapter le code du programme de l'univers commun au code d'un participant.

Les travaux des thèses de [Vil03, Le04] ont donné naissance à la première version de la plateforme Mélusine. C'est l'environnement de conception et d'exécution des fédérations logicielles définies ci dessus.

5.2.3 La deuxième génération : centrée architecture de familles d'applications

L'architecture Mélusine de la première génération se base sur la vision d'*architecture des applications*. Ceci a changé dans sa deuxième génération dans laquelle elle évolue vers une approche centrée sur l'*architecture de familles d'applications*.

L'évolution la plus importante dans cette génération est la généralisation du concept de Fédération de logiciels par le concept de Domaine. Ce dernier vient des approches de l'Ingénierie de Domaines et les Lignes de Produits [5] [EV05, Thi98].

Conceptuellement, nous considérons qu'un domaine est un champ d'expertise métier dans lequel un ensemble de connaissances, de savoir-faire et d'outils utilisés dans ce métier sont structurés systématiquement dans l'objectif de favoriser leur réutilisation dans le développement des applications. "*A domain can be seen as a set of systems built from common assets*" [EVI05].

Du point de vue structurel, nous avons pensé que l'architecture en trois couches de Mélusine peut être utilisée pour structurer formellement un domaine. Cependant, certains éléments dans cette première génération doivent être reconsidérés.

Concrètement, en se basant sur une vision d'IDM, nous avons trouvé que la couche conceptuelle (univers commun) est à un niveau d'abstraction plus haut que la couche d'implémentation (les participants). En effet, l'univers commun ne contient que les concepts communs abstraits et laisse tous les détails d'implémentation à ses outils. Ceci correspond à la vision de séparation des niveaux d'abstraction entre le modèle et le code d'implémentation dans l'IDM. Nous avons alors redéfini l'univers commun en terme de *métamodèle*. L'application est définie comme des instances des concepts de l'univers commun et est représenté en terme des *modèles*. Évidemment, un modèle est conforme à son métamodèle.

En plus, comme nous l'avons dit, l'univers commun, lors de l'implémentation, est matérialisé comme un petit programme permettant d'instancier les concepts (i.e. faire des applications) et de synchroniser l'état de ses instances avec l'état des instances des participants (i.e. exécuter les applications). Du point de vue de l'IDM, ce programme est la *machine virtuelle* qui exécute les modèles conforme au métamodèle.

5. Product Lines.

Nous pensons que les nouvelles briques architecturales "à la IDM" *métamodèle*, *modèle* et *machine virtuelle* sont capables de formaliser les Domaines. En effet, le concept de Domaine est souvent lié au concept de langage spécifique de Domaine (le DSL [6]). Le métamodèle du Domaine est une représentation du langage du Domaine. De ce fait, les modèles définis dans ce métamodèle sont les applications crées avec ce DSL. La notion de la machine virtuelle introduite dans notre architecture permet d'assurer que les modèles fonctionnent également comme des applications exécutables.

La couche conceptuelle est re-conçue comme montrée dans la figure 5.2.

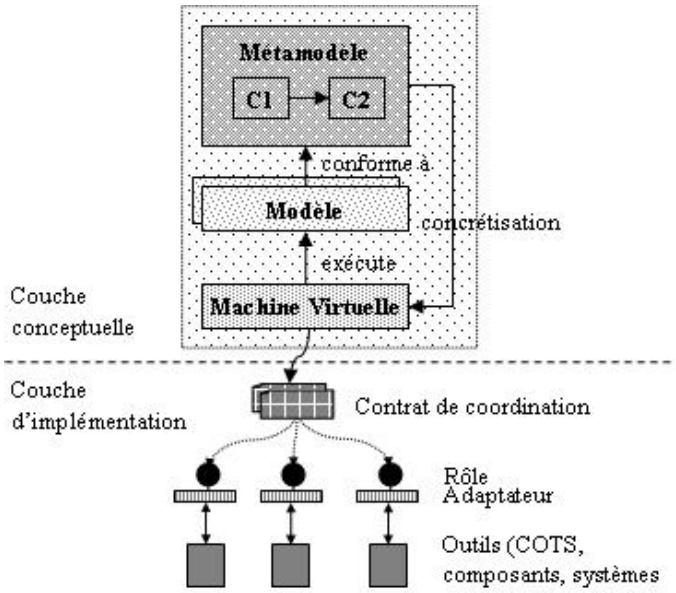


FIGURE 5.2 – Domaine

La machine virtuelle se charge maintenant de maintenir la liaison entre la couche conceptuelle, les couches de médiation et les outils. Dans cette génération, ces deux couches sont unifiées dans un terme unique : la couche d'implémentation (voir la figure 5.2).

Les autres concepts de Mélusine (*contrat de coordination*, *adaptateur*, *role*, *participant*) sont retenus et utilisés comme avant.

Notons que, du point de vue conceptuel, il y a une différence entre le concept de Fédération et celui de Domaine : le concept de Domaine implique l'idée de séparation des préoccupations ce qui n'est pas le cas pour le concept de Fédération. En effet, la couche conceptuelle de la fédération décrit des concepts communs aux éléments logiciels qui peuvent appartenir à divers champs métiers ; mais celle de domaine ne décrit que les concepts métiers relatifs à la logique d'un domaine.

5.2.4 Synthèse

Dans cette section, nous avons présenté la vue globale de notre contexte de travail : l'architecture Mélusine. Cette architecture est structurée en trois couches. La couche la plus haute exprime les logiques métier de l'application (ou celles du domaine), la couche la plus basse rassemble les

6. Domain-Specific Language.

composants constituant l'application (ou constituant le domaine). Ces composants sont réutilisables et coordonnables sous la direction de la couche conceptuelle. La communication entre la couche conceptuelle et la couche outils est assurée par une couche de médiation. Cette architecture a évoluée en deux générations consécutives : la première se concentre sur l'architecture des applications, la seconde s'intéresse à l'architecture de familles d'applications.

Cette thèse travaille sur l'architecture Mélusine de deuxième génération. Cette architecture nous permet de *réutiliser* les connaissances, les savoir-faire, les outils, les environnements etc. d'un domaine particulier. Cependant, nous nous sommes rendu compte rapidement qu'il est possible de pousser la réutilisation à un niveau plus haut dans lequel les domaines eux-mêmes deviennent les *grains à composition*. Le problème de composition des domaines était posé et nous a particulèrement intéressés. Nos premières expérimentations sur ce sujet nous ont montrés que la composition de domaines peut être obtenue par la composition des couches conceptuelles. Ceci est suffisant pour diriger indirectement les couches "basses".

Alors, pour la composition des couches conceptuelles, nous devons prendre en compte tous les éléments de cette couche, i.e. son métamodèle, ses modèles et sa machine virtuelle. Notre travail de composition de modèles dans cette thèse se place dans ce contexte. Nous allons détailler la couche conceptuelle qui nous concerne directement dans la section suivante. Les points suivants sont abordés :

- La structure de la couche conceptuelle du domaine.
- Les hypothèses d'implémentation de la couche conceptuelle.

5.3 Domaines (couche conceptuelle)

La couche conceptuelle est fondamentale pour un domaine. D'une part, elle rassemble la logique métier du domaine, d'autre part, elle permet d'avoir une vision globale et de haut niveau d'abstraction du domaine. La couche conceptuelle est indépendante de sa réalisation (i.e. la couche d'implémentation). Elle comprend :

- le métamodèle de ce domaine
- les modèles spécifiant les applications du domaine
- la machine virtuelle qui exécute les modèles.

Notons que les termes métamodèles, modèles, et machines virtuelles utilisées dans cette section ont un sens défini ci-dessous.

5.3.1 Métamodèle

Nous définissons un métamodèle comme "*la définition d'un langage de modélisation spécialisé au domaine*" [Veg05]. Selon la philosophie de Mélusine, pour être efficace, ce langage doit être intuitif, expressif, et avoir une sémantique claire et suffisamment précise pour être capable de *représenter des modèles* et de *les rendre exécutable* [Veg05]. Pour Mélusine, le modèle du domaine est le métamodèle des applications du domaine [EVI05].

Revenons à la définition d'un langage que nous avons présenté dans la section 3.2.1 du chapitre 3. Un langage est constitué de trois facettes : sa syntaxe abstraite, sa syntaxe concrète, et sa sémantique. Chaque facette d'un langage peut être modélisée par un ou plusieurs métamodèles. Dans notre approche, la syntaxe abstraite et la sémantique d'un langage du domaine sont modélisées dans un seul métamodèle ; pour cette raison, par abus de langage, nous dirons "le metamodèle du domaine" pour parler de la définition de la syntaxe abstraite et de la sémantique du langage d'un domaine. La syntaxe concrète de ce langage est fournie par un éditeur de modèles (nous allons présenter les éditeurs dans la section suivante).

Concrètement, dans notre approche, le métamodèle du domaine définit :

- Un ensemble de concepts métier du domaine.
- Les relations entre ces concepts.
- Un ensemble de règles de validation déterminant quels sont les modèles corrects. Ces règles décrivent la sémantique axiomatique statique exprimée sur les concepts de la syntaxe abstraite.
- Un ensemble de règles d'exécution déterminant le comportement des modèles. Ces règles décrivent la sémantique opérationnelle dynamique du langage. Elles sont normalement définies par les opérations associées aux concepts.

Dans notre approche, les métamodèles sont formalisés dans le style orienté objet. Ils sont exprimés sous forme de diagrammes de classes UML. Leurs concepts sont représentés par des classes. Leurs règles de validation syntaxique sont exprimées en termes d'expressions OCL. Leurs règles d'exécution sont modélisées comme des méthodes opérationnelles dans les classes. Nous donnons ci-dessous l'exemple d'un métamodèle.

La figure 5.3 illustre le métamodèle du domaine de Produits. Ce domaine a un langage de gestion des versions des produits.

- La syntaxe abstraite de ce langage est modélisée par les concepts bleu foncés. Ces concepts nous permettent d'exprimer les modèles de données décrivant les types de produits (`ProductType`) qui définissent les caractéristiques (`Product Attribute`) d'un produit ; les méthodes associées décrivent une partie de la sémantique de ces concepts.
- L'autre partie de la sémantique du domaine est modélisée par les concepts vert foncés. Ces concepts n'appartiennent pas à la syntaxe abstraite mais contiennent les méthodes déterminant le comportement (i.e. les règles d'exécution) des concepts abstraits, ici le versionnement qui exprime comment créer une branche ou une révision d'un produit.
- Les expressions de validation syntaxique d'OCL (les notes en gris foncés attachés aux concepts) imposent les contraintes de cohérence sur les modèles de données. Par exemple, la contrainte `inv : isKey implies (shared and mandatory and readOnly and isList=false)` signifie qu'un attribut est défini comme *clé* pour un produit s'il est :
  - *partagé* (`shared`), i.e sa valeur est la même pour toutes les versions du produit ; et
  - *obligatoire* (`mandatory`) i.e sa valeur est obligatoirement initialisé au moment de création du produit ; et
  - *immuable* (`read-only`), i.e. si sa valeur a été affectée, on ne peut plus la changer ; et
  - *mono valuée* (`isList=false`).

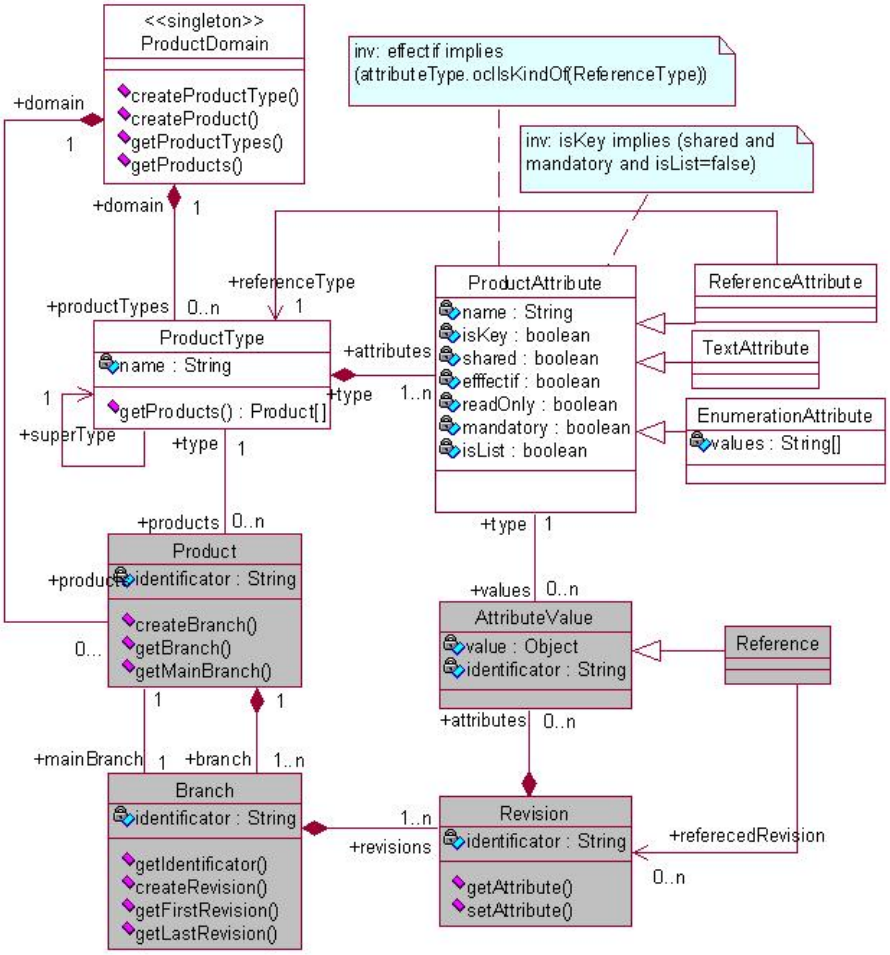


FIGURE 5.3 – Métamodèle du Domaine de Produit

Nous n'allons pas entrer dans les détails de ce métamodèle. Le but ici est simplement de montrer un exemple de la séparation de parties conceptuelles différentes (i.e. la syntaxe abstraite et la sémantique) constituant un métamodèle. Cet aspect est très important pour la réalisation de la composition de modèles. Nous allons le discuter en détail dans notre proposition (le chapitre 6) ; mais rapidement, nous pouvons dire que la composition de modèles requiert la composition de leurs métamodèles. Pour composer des métamodèles, il nous faudra composer leurs parties conceptuelles correspondantes : c'est-à-dire, composer la syntaxe abstraite de l'un avec celle de l'autre, la sémantique de l'un avec celle d'un l'autre. Cette composition ne doit pas mélanger ces deux parties. Nous considérons ceci comme étant un principe de base pour la composition de métamodèles valides.

Dans notre approche, le métamodèle est un artefact central de chaque domaine à partir duquel, peuvent être obtenus *l'éditeur des modèles* et *la machine virtuelle*.

L'éditeur de modèles fournit une notation (syntaxe concrète) pour la syntaxe abstraite définie par le métamodèle. Cette notation permet à l'utilisateur du domaine de rédiger le modèle de son application. La machine virtuelle implémente la sémantique opérationnelle modélisée dans le métamodèle. Elle va donc être capable d'exécuter tous les modèles élaborés à l'aide de l'éditeur.

Dans les deux sections suivantes, nous allons discuter ces deux artefacts en montrant la liaison entre ceux-ci et le métamodèle.

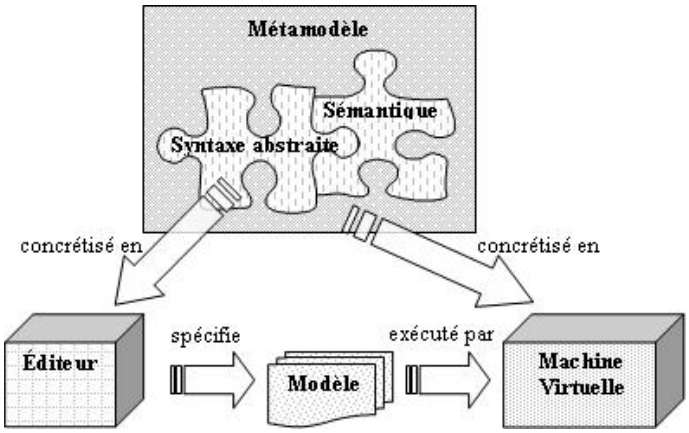


FIGURE 5.4 – Concrétisation des éditeurs et de la machine virtuelle

5.3.2 Modèles et éditeur de modèles

Modèle

Un modèle est une vision abstraite d'une application du domaine. Il est modélisé par la syntaxe abstraite de son langage (imposé par un éditeur via une notation concrète) et est exécuté par la sémantique opérationnelle du langage (implémentée par la machine virtuelle du domaine). Cette distinction amène deux conséquences :

- Le développement d'une application a besoin d'une phase d'élaboration des modèles avec l'aide d'un éditeur.
- Les modèles sont rendus exécutables par une machine virtuelle du domaine.

Le reste de cette section présente les éditeurs de modèles.

L'éditeur de modèles

L'éditeur de modèle ne s'intéresse qu'aux concepts de la syntaxe abstraite du métamodèle. Basé sur ces concepts, il va fournir les notations concrètes permettant de façon très simple de construire des modèles. L'éditeur est surtout concerné par les capacités de visualiser et de sauvegarder les modèles. Des technologies comme XML, et EMF/GMF peuvent être utilisés pour développer un éditeur. Dans notre approche, certains éditeurs sont développés en utilisant les technologies XML, EMF ; d'autres sont codés à la main, d'autres encore sont générés automatiquement. Il n'y a aucune contrainte sur la manière de développement un éditeur. Mais l'utilisation de technologie comme EMF apportent des avantages lorsque les éditeurs sont générés. Dans nos expérimentations, nous avons beaucoup utilisé ces technologies pour la génération d'éditeurs.

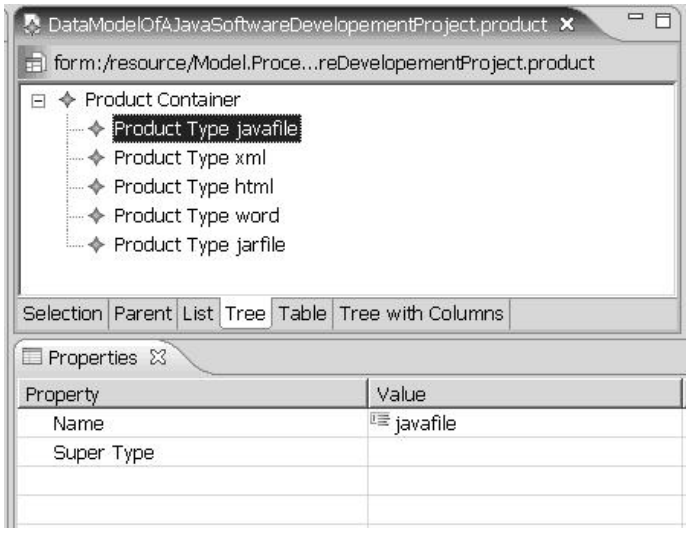


FIGURE 5.5 – Un exemple d'éditeur de modèles

La figure 5.5 présente un éditeur du Domaine de Produits. Cet éditeur est généré par la technologie EMF. Comme nous pouvons le voir, il spécifie un modèle de données d'une application dans ce domaine. Cet éditeur utilise les concepts de la syntaxe abstraite du métamodèle de Produits (ex : *ProductType* dans l'image) et fournit une visualisation d'arbre (l'onglet Tree) qui est un des modes de visualisation par défaut des éditeurs EMF générés. Avec une même syntaxe abstraite, on peut avoir plusieurs visions concrètes différentes, par exemple, dans cette image, les onglets Arbre (*Tree*), Table (*Table*), Arbre Avec Colonne (*Tree With Columns*) peuvent afficher un modèle selon différentes vues.

5.3.3 Machine virtuelle

La machine virtuelle est une concrétisation du métamodèle. Son objectif est d'exécuter les modèles. Pour cela, elle doit satisfaire à deux conditions :

- Elle doit connaître *la structure du modèle* et *la manière de l'exécuter*. Ces informations sont modélisées dans le métamodèle. La syntaxe abstraite détermine la structure du modèle et la sémantique opérationnelle détermine les comportements à l'exécution. La machine virtuelle peut les obtenir en consultant le métamodèle.
- Elle doit être implémenté dans un langage d'exécution. Dans notre approche, nous avons choisi de développer nos machines virtuelles en Java. Mais on peut implémenter les machines virtuelles en n'importe quel langage d'exécution comme C ou Kermeta. Le choix de Java résulte d'un compromis entre expressivité et disponibilité d'une technologie efficace, éprouvée et bien instrumenté.

Dans notre approche, une machine virtuelle du domaine peut être obtenue de façon directe à partir de son métamodèle. Chaque concept du métamodèle est matérialisé par une classe Java dans la machine virtuelle. Cette hypothèse de matérialisation est très importante car :

D'un part, elle permet de garder une relation directe entre le métamodèle et sa machine virtuelle, ce qui donne des règles de transformation entre le métamodèle et sa machine virtuelle.

D'autre part, cette hypothèse facilite la maintenance du lien entre le métamodèle et sa machine virtuelle. Lorsqu'il y a un changement dans métamodèle, la mise à jour dans la machine virtuelle peut être effectuée aisément.

Finalement, et surtout, cette hypothèse est à la base de nos techniques de composition de domaines. En effet, la composition de domaines requiert la composition de leurs métamodèles respectifs et donc la composition de leurs machines virtuelles. Ainsi composer la sémantique d'un concept A d'un domaine avec un concept B d'un autre domaine revient à composer les classes A et B correspondantes de leurs interpréteurs.

5.4 Composition de domaines

La composition de domaines est une technique qui nous permet d'atteindre à un niveau de réutilisation élevé dans lequel ce ne sont pas les applications qui sont réutilisées, mais des familles d'applications.

Ce sujet a le thème des travaux de thèse précédents dans notre équipe [Le04, Veg05]. Cette thèse est une continuation des expérimentations existantes. Nous allons résumer les résultats conceptuels et pratiques obtenus jusqu'à présent pour identifier ce qui a été faits, ce qu'il faut améliorer, les limites des expérimentations précédentes et à partir de là, identifier les objectifs de cette thèse.

5.4.1 Notre démarche

Le schéma 5.6 donne une vue globale de notre approche de composition de domaines. D'abord, les liens sémantiques entre les métamodèles sont établis. Basé sur ceux ci, les liens correspondant au niveau de modèles et des machines virtuelles seront établis. Les liens au niveau de la couche d'implémentation ne sont pas nécessaires.

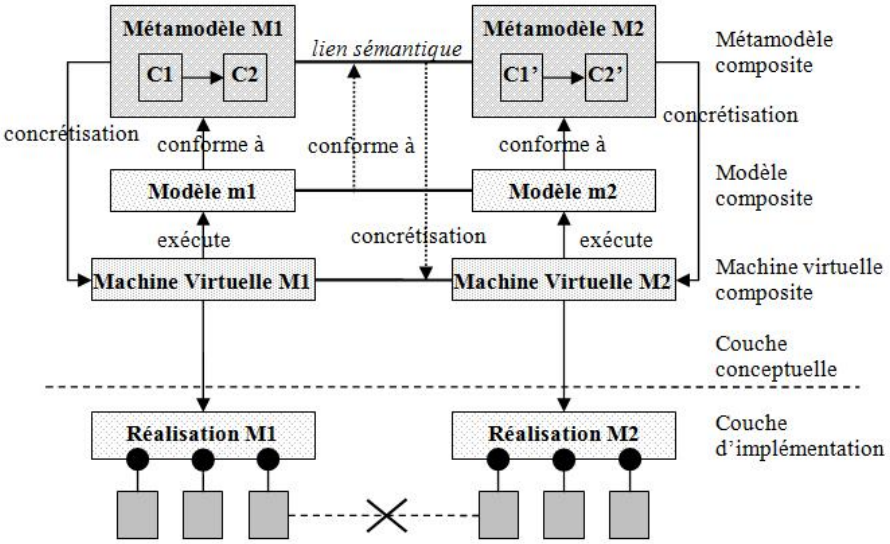


FIGURE 5.6 – La composition de domaines

Notons que les liens au niveau modèles sont conformes à ceux au niveau métamodèle ; les liens au niveau machine virtuelle sont les concrétisations des liens au niveau métamodèle qui sont plus abstraits.

Les sections suivantes présentent la composition de métamodèle, de modèle et de machine virtuelle. Le résultat d'une composition de domaine est un domaine composite dont la structure est celle d'un domaine normal. Notre approche de composition de domaines peut donc être appliquée à un domaine composite pour faire de la composition hiérarchique.

5.4.2 Composition de métamodèles

La composition de métamodèle consiste à établir des relations R entre des concepts des métamodèles (voir la figure 5.7). Ces relations expriment le lien sémantique entre les métamodèles dans le sens où la sémantique de l'un est étendue/ou complétée par l'autre et vice versa.

Un principe de base pour réaliser une composition de métamodèles est qu'il faut bien distinguer les concepts de la syntaxe abstraite de ceux de la sémantique du métamodèle et établir les bonnes relations entre les parties correspondantes.

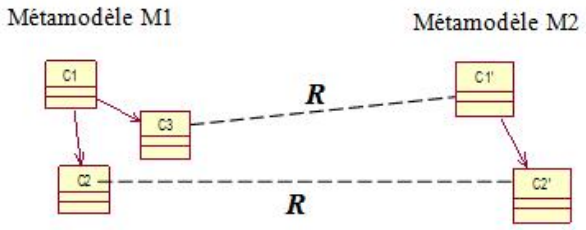


FIGURE 5.7 – Composition de modèles par relations

Dans notre approche, nous identifions qu'une relation R entre les concepts peut être matérialisée de plusieurs façons. Par exemple, les trois patrons de la figure 5.8 modélisent les différents possibilités pour relier deux concepts dans le style orienté objet :

- deux concepts reliés par une relation.
- deux concepts reliés par un (ou plus) concept(s) tiers.
- deux concepts reliés par une relation et sa (ses) classe(s) d'association.

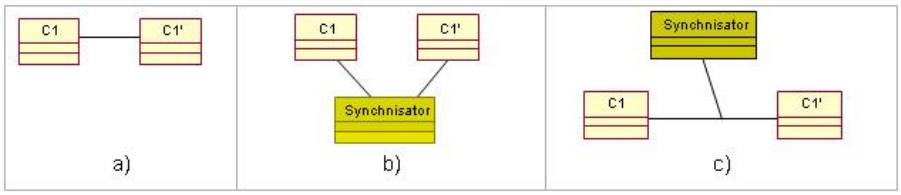


FIGURE 5.8 – Patrons simples de composition de modèles par relations

Conceptuellement, tous ces patrons pourraient être utilisés pour composer des métamodèles. Cependant, dans la plupart des expérimentations de composition que nous avons réalisé, le patron c) est le plus fréquent. Ce patron signifie qu'une relation de composition peut exprimer la sémantique de composition au travers de sa classe d'association.

Il y a d'autres patrons de composition de métamodèles proposés dans plusieurs travaux similaires par exemple [Cla02, KML+04]. Ceux ci proposent des mécanismes de composition sophistiqués comme la fusion des classes, l'héritage des classes, l'héritage de paquetages, ou la définition des paquetages paramétrables instanciés lors de l'utilisation etc. Cependant, nous n'utilisons pas

ces patrons car, d'un part, ils ne répondent pas notre besoin de pouvoir réutiliser *sans modification* les modèles existants et les environnements de modélisation de sous domaines ; d'autre part la simplicité des mécanismes que nous utilisons n'est pas une limitation de notre approche mais plutôt un compromis. Etant donné que nous privilégions la réutilisation des artefacts des sous domaines sans modification, nous nous restreindrons à utiliser des mécanismes simples tels que le patron c) ci-dessus.

Caractéristiques d'une relation

- **Dégré de la relation** : Une relation est *binaire* (i.e. relation entre deux concepts). Dans notre approche, une relation n-aire (i.e. relation entre plusieurs concepts) peut être transformé en plusieurs relations binaires. Par exemple, imaginons que nous voulions définir des stratégies de versionnement pour un document dans un espace de travail. Les stratégies de versionnement sont liées aux trois concepts *Activity*, *Document* et *Workspace* provenant respectivement du domaine de procédé, de document et d'espace de travail. Pour cela, au lieu de définir une relation ternaire, nous avons proposé d'utiliser un quatrième concept dit émergent de *Version* et d'établir des relations binaires entre ce nouveau concept et les concepts composés pour exprimer les stratégies de versionnement (Figure [?])

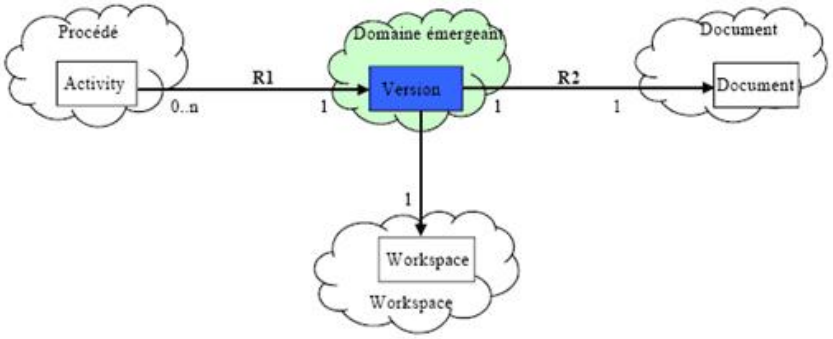


FIGURE 5.9 – Transformation d'une relation n-aire en relations binaires [Le04]

De la même manière, nous pouvons utiliser un concept émergeant pour transformer une relation n-aire en des relations binaires. Nous introduisons un concept émergeant pour exprimer la sémantique de la relation entre les concepts concernés. Puis, pour chacun de ces concepts, nous établissons une relation binaire avec le concept émergeant.
Puisqu'une relation n-aire peut être représenté en terme de plusieurs relations binaires, nous ne nous intéressons donc qu'à la réalisation d'une relation binaire.

- **Orientation d'une relation** : Une relation peut être orientée ou non. Dans le cas orienté, nous disons que cette relation désigne une liaison entre un domaine actif et un domaine passif. Dans le cas non orienté, elle est considérée comme la liaison entre deux domaines actifs. Un domaine est dit actif par rapport à une relation s'il est capable d'agir sur les instances des concepts qu'il propose dans cette relation sans y être invité par le domaine associé. En revanche, un domaine est dit passif si son concept participant dans la relation n'est modifié que comme conséquence d'une action sur le concept associé dans l'autre domaine. Dans notre approche, une relation non orienté est considérée comme deux relations orientées inverses.

- **Multiplicité de la relation** : Une relation peut avoir plusieurs instances. Une instance de la relation est une correspondance qui relie une paire d'instances dont chacune est l'instance d'une classe concernée par cette relation. Nous proposons d'utiliser la cardinalité d'une relation pour désigner le nombre de correspondances auxquelles une instance d'un concept peut participer.
- **Types de relation** : Nous n'utilisons pas l'héritage donc les relations crées selon notre approche sont considérées comme des association simples qui expriment une dépendance sémantique entre deux concepts.
  Nous pouvons toutefois raffiner une association en des cas particulier par exemple l'*association identique* qui indique que les concepts reliés ont le même sens, même s'ils ont des noms différents. Par exemple, l'association entre le concept *Data* qui représente une donnée circulant sur les procédés du Domaine de Procédés et le concept *Product* du Domaine de Produits est une association identique.
- **Sémantique de relation** : La sémantique d'une relation permet de spécifier la manière de gérer cette relation. Du point de vue conceptuel, la sémantique de la relation est la coopération de deux domaines pour synchroniser et garantir la cohérence de fonctionnement entre les modèles de ces domaines.

Dans la section 5.4.4 ci-dessous, nous allons discuter l'implantation des relations dans la couche des machines virtuelles.

5.4.3 Composition de modèles

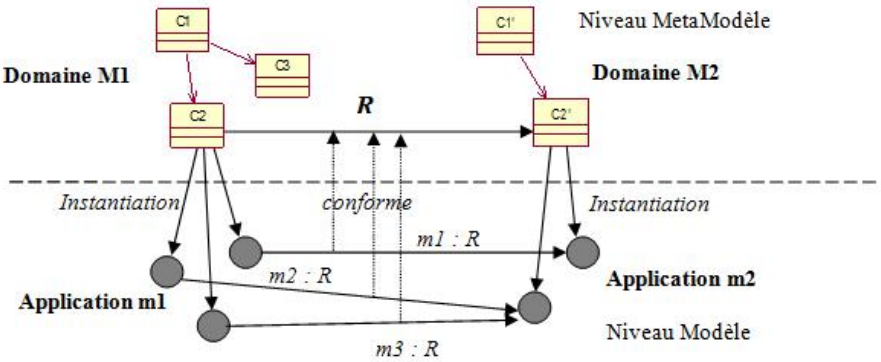


FIGURE 5.10 – Les instances d'une relation

Lorsque la définition des relations au niveau des métamodèles est effectuée, on peut les utiliser pour établir les liens au niveau modèle.

La composition au niveau modèle est similaire de celle au niveau des métamodèles. Elle consiste à établir des liens m entre les éléments des modèles. Ces liens sont conformes aux relations R définies au niveau métamodèle (figure 5.10).

Nous appelons un lien m une *correspondance (mapping)*. Puisqu'une *correspondance* est toujours liée à une relation R, il y a une contrainte de cycle de vie entre la correspondance et sa relation. Une correspondance n'est définie si et seulement si la relation entre les concepts des éléments de cette correspondance existe. Lorsque la relation est enlevée, toutes les correspondances de cette relation deviennent invalides.

5.4.4 Composition de machines virtuelles

Une fois que la composition de métamodèles et de modèles est accomplie, la question de l'exécution des modèles du domaine composite s'impose. Très vite, on peut se rendre compte que les nouveaux liens qui viennent d'être établis entre les métamodèles et entre les modèles n'appartiennent à aucun des domaines composés. Par conséquent, les machines virtuelles de ces domaines ne sont pas capables de les interpréter, et par conséquent les nouveaux modèles composites ne peuvent pas être interprétés. Pour les rendre exécutables, nous avons besoin d'une machine virtuelle du domaine composite permettant d'exécuter ce modèle.

Afin de construire cette machine virtuelle, notre proposition est de s'appuyer sur les machines virtuelles existantes et de les faire coopérer. Ces machines virtuelles existent et sont supposé être bien testées ; réutiliser celles-ci est un gain de temps considérable ; le seul problème pour les réutiliser est qu'il faut développer un code supplémentaire qui se charge de les "coller".

Similairement à la façon de matérialiser les concepts du métamodèle en des classes de sa machine virtuelle, nous proposons de composer les machines virtuelles en matérialisant les relations de composition entre les métamodèles en code supplémentaire.

Le problème lors de la matérialisation des relations est qu'il faut garantir de les composer sans modification. En fait, "sans modification" doit être compris dans le sens où on peut ajouter mais pas enlever ni modifier un concept. Donc, pour résoudre le problème de la matérialisation des relations de composition, la solution utilisée est d'instrumenter le code des concepts existants par la technique AOP. Une relation est ajouté comme un attribut du concept à partir duquel il référence un autre concept, nous l'appelons concept source v.s. concepts cible (notons que les relations dans notre approche sont orientées, celles non orientées sont considérées comme deux relations orientées inverses). Cet attribut a pour objectif de garder le lien de synchronisation entre deux concepts de deux domaines.

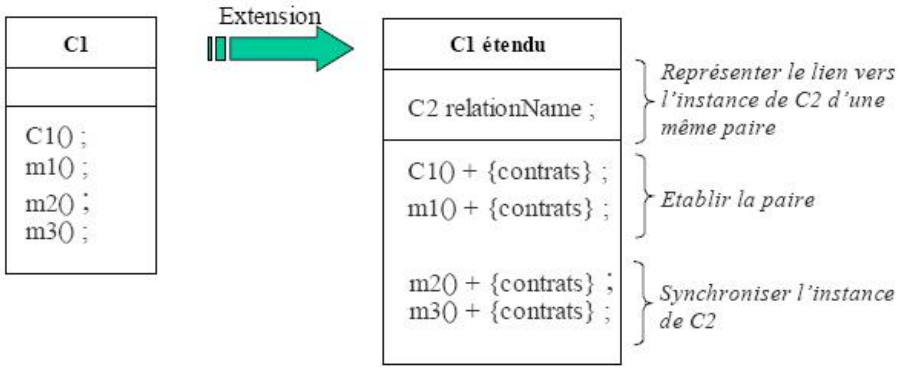


FIGURE 5.11 – L'extension par l'instrumentation [Le04]

A l'exécution, la première tâche est d'établir les liens entre les instances des concepts. L'établissement des liens est le plus souvent réalisé au moment de la création de l'objet du concept source. Nous interceptons ses constructeurs et/ou les méthodes ayant pour objectif de créer une instance, et nous y ajoutons une sémantique additionnelle visant à localiser l'instance du concept cible et à établir un lien physique (un pointeur Java) vers cette instance.

La synchronisation est réalisée de manière similaire, nous interceptons les méthodes du concept source et ajoutons la sémantique additionnelle qui peut par exemple invoquer une chaîne d'appels de méthodes sur l'objet du concept cible.

5.4.5 La réalisation de composition de domaines

La composition de domaines présentée dans la section précédente est du point de vue conceptuel. Dans la réalité, sa réalisation est un peu différente. Dans cette section, nous allons résumer nos expérimentations de composition de domaines pour, dans la section suivante, identifier les objectifs de la thèse.

Jusqu'à présent, nous avons réalisé les composition de domaines de manière ad-hoc.

- Pas de composition de métamodèles (explicite).
- Pas de composition de modèles (explicite).
- Composition de machine virtuelle ad-hoc.

Idéalement, le processus de composition doit exposer explicitement une phase d'établissement des liens au niveau métamodèle, puis ceux au niveau modèle, et finalement le développement de la machine virtuelle composite. Cependant, l'implantation actuelle réalise tout de suite au niveau des machines virtuelles sans composition de métamodèles ni composition de modèles. Les liens entre les métamodèles sont matérialisés directement en code faisant collaborer les machines virtuelles. Les liens entre les éléments de modèles sont établis automatiquement en se basant sur une convention de même nom. Cette convention posez des problèmes car les modèles réutilisés ne sont pas conçus en prévoyant l'existence d'autres modèles avec lesquels ils pourraient collaborer.

Malgré ces inconvénients, nos expérimentations ont obtenues des résultats successifs dans la construction d'applications industrielles en se basant sur le mécanisme de composition de domaines. A titre d'exemple, nous pouvons citer le système APEL pour la société Actoll **??**, la Démo Métier du projet Centr'Actoll **??**, un système d'alarme pour la société Thales **??**. Ceci nous fait penser que la composition de domaines est une approche prometteuse, mais perfectible !

Pour cette raison, dans la deuxième génération de l'approche, nous avons formalisé le processus de composition dans une vision plus proche de la vision conceptuelle proposées, afin d'éliminer les défauts signalés ci-dessus. Nous avons ensuite développé l'assistant du processus de composition de manière simple mais efficace.

5.5 Objectifs de la thèse

Le tableau 5.1 résume les défauts actuels et les besoins de notre démarche de composition de domaines.

Défauts	**Besoins**
Pas de composition de métamodèles	Composition de métamodèles et les outils
Pas de composition de modèles	Composition de modèles et les outils
Composition de machines virtuelles ad-hoc	Génération du code de composition semi automatique.

TABLE 5.1 – Défauts et besoins de la composition de domaines

Dans cette thèse nous cherchons à répondre à ces besoins. Le but final est de faciliter le processus de construction d'applications à grande échelle à partir des domaines. Pour cela, nous proposons un environnement de composition de modèles dont l'idée basique est qu'il faut composer leurs métamodèles. Ce principe est toujours valide pour toutes les approches de composition de modèles. Cette idée a été utilisée pour la composition de domaines de Mélusine. L'environnement de composition de modèles réalisé dans cette thèse a donc été développé en se basant fortement sur les expériences de la composition de domaines ci-dessus.

5.6 Synthèse

Dans ce chapitre, nous avons présenté de notre contexte de travail : l'architecture Mélusine.

Dans la première génération de Mélusine, par instanciation d'un ensemble (souvent très petit) de concepts (de l'univers commun), on peut construire des applications complexes en profitant des fonctionnalités implémentées dans les outils existants. De plus, le faible couplage entre l'univers commun et ses participants permet de substituer aisément des participants. Ceci permet au concepteur d'application de changer librement les outils ou les composants avec lesquels il est habitué à travailler. Une application développée donc peut avoir différentes versions d'implémentation.

Dans la deuxième génération, par composition d'un ensemble de domaines souvent étroits et bien cernés représentant une préoccupation du système, on peut construire des applications complexes multi-domaines.

La composition de domaines est liée à la composition de modèles. Cependant, elle est réalisée de manière ad hoc dans Mélusine. Il existe également beaucoup des limitations et de défauts. Nous proposons donc une approche de composition de modèles pour résoudre ces problèmes.

Chapitre 6

Un langage de composition de modèles

6.1 Introduction

Le chapitre précédent a présenté une approche conceptuelle de composition de modèles exécutables (obtenus par la composition de domaines). Cependant, sa réalisation reste ad-hoc, et assez éloigné des concepts proposés. Nous allons la formaliser afin d'obtenir un support pour la composition de modèles exécutables dans une vision plus proche des concepts proposés.

Cette formalisation permet d'avoir un environnement qui *automatise et systématise* la composition. L'automatisation consiste à outiller le processus de composition. Systématiser signifie que cet outillage doit être basé sur les étapes de composition de la proposition conceptuelle.

Cette formalisation présente deux difficultés :

- le manque de *formalisme* pour exprimer les compositions au niveau métamodèle et au niveau modèle.
- le manque d'*outils* permettant de décrire les compositions aux différents niveaux et permettant l'exécution des modèles composites.

De plus, nous souhaitons que l'environnement résultant de cette formalisation soit suffisamment générique pour qu'il puisse être utilisé dans divers scénarios de composition de modèles. Pour cela, le langage de composition et les outils associés doivent être :

- indépendants des domaines qu'ils composent et
- indépendant des applications qu'ils composent.

Dans ce chapitre ainsi que dans le suivant, nous présentons notre formalisation de la composition de modèles et nos efforts pour obtenir un environnement de composition. Cette formalisation se traduit par la définition d'un langage de composition de modèles autour duquel les outils supportant l'automatisation du processus de composition de modèles vont venir s'intégrer pour former cet environnement. Notre approche que nous nomerons Codèle fait référence à :

- Un langage de composition de modèles exécutables.
- Un environnement qui
  - reconnaît ce langage (et fournit un éditeur), qui
  - permet de réaliser des compositions de modèles et qui
  - permet de les exécuter.

Ce chapitre présente le langage de composition et le chapitre suivant présentera l'environnement.

Nous organisons ce chapitre de la manière suivante : la section 6.2 présente notre approche de

composition de modèles. Nous identifierons les éléments de composition participant à notre processus de composition de modèles. Les formalismes nécessaires pour élaborer ces artefacts seront aussi présentés. Un exemple de composition de modèles, utilisé pour illustrer notre travail, sera présenté dans la section 6.3. La section 6.4 donnera une vue globale de notre langage de composition de modèles. Nous nous attarderons ensuite sur les deux concepts principaux constituant ce langage, *Relation Horizontale* et *Correspondance*, dans les sections 6.5 et 6.6. Ils correspondent respectivement à la composition de métamodèles et à la composition de modèles. La section 6.7 conclut le chapitre.

6.2 Notre approche

Dans le chapitre 4, nous avons comparé les approches de composition de modèles en se basant sur deux critères : les *élements de composition* et le *mécanisme de composition*. Nous allons comparer notre approche avec les mêmes critères.

6.2.1 Élements de composition

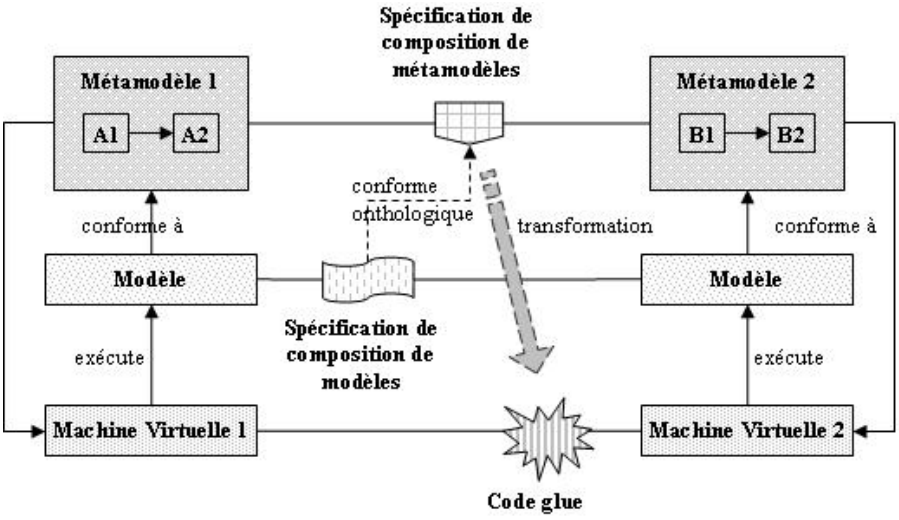


FIGURE 6.1 – Éléments de composition.

Notre approche se base sur un processus de composition à trois niveaux : métamodèle, modèle, et machine virtuelle (chapitre 5). La démarche proposée est d'établir des liens à chaque niveau.

En se basant sur ces niveaux, nous proposons ainsi trois éléments de composition. Chacun concrétise les liens (*conceptuels*) d'un niveau. Les éléments proposés comprennent :

- La spécification de composition de métamodèles.
- La spécification de composition de modèles.
- Le code composant les machines virtuelles (*glue code*).

La figure 6.1 les résume rapidement. Ces éléments ont besoin de formalismes pour les exprimer et d'outils pour les élaborer.

A. Spécification de composition de métamodèles

Cette spécification définit les relations entre les métamodèles. Les caracréristiques d'une telle relation a été présentée dans 5.4.2 du chapitre 5.

Un langage sera utilisé pour décrire ces relations. La première contrainte pour ce langage est qu'il doit prendre en compte tous les caractérisques de la relation. Dans notre cas, les relations sont binaires. Le langage doit être capable d'exprimer ce genre de relation. La figure 6.2 exprime le rapport entre la spécification de composition de métamodèles et son langage.

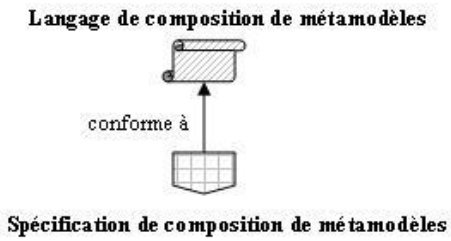


FIGURE 6.2 – Formalisme de composition de métamodèles.

La spécification de composition de métamodèles a deux usages :

- Premièrement, elle est utilisée pour *guider la composition de modèles*. En effet, les liens entre les modèles (définis par la spécification de composition de modèles) doivent être conformes aux relations définies entre leurs métamodèles (définis par la spécification de composition de métamodèles). Pour garantir cette conformité, la spécification d'un lien au niveau modèle doit référencer la relation au niveau métamodèle. Une vérification à la construction du lien est sufisante. Du point de vue ontologique, la dernière est le métamodèle de l'autre (voir la figure 6.1).
- Deuxièmement, cette spécification va être utilisée pour *générer le code gluant*. Puisque la spécification de composition de métamodèles est une abstraction de la collaboration entre les machines virtuelles [1], cette abstraction devrait être concrétisée. Elle est transformée en une partie du glue code (voir la figure 6.1).

B. Spécification de composition au niveau modèle

Cette spécification a pour objectif de modéliser les relations au niveau modèle. Nous avons besoin d'un langage pour la décrire (figure 6.3).

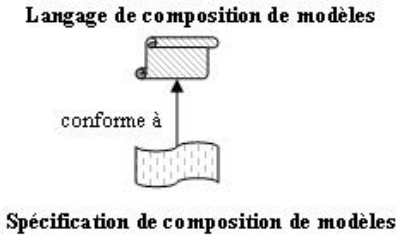


FIGURE 6.3 – Formalisme de composition de modèles

1. comme le métamodèle est l'abstraction de la machine virtuelle

Relation entre les formalismes de composition En principe, rien ne nous empêche d'avoir un langage pour chaque spécification de composition (au niveau modèle et métamodèle). Cependant, on se rend compte que ces deux spécifications ne sont pas indépendantes. Il y a une conformité entre les liens décrits au niveau modèles et ceux décrits au niveau métamodèle. Les langages de modélisation correspondants sont donc eux aussi liés. Le langage de composition de modèles demande l'existence préalable de la composition des métamodèles correspondants. Nous proposons donc un langage de composition constitué de deux parties : l'une permetant de définir la composition de métamodèles, l'autre permetant de définir la composition de tous modèles conformes à ces métamodèles. Entre ces deux parties il existe des relations qui permettent de spécifier la conformité entre les liens définis dans la composition de modèles et ceux définis dans la composition de métamodèles. Autrement dit, dans notre approche, nous n'avons qu'un seul langage commun pour modéliser les compositions de métamodèles et de modèles. Cette idée est illustrée dans la figure 6.4.

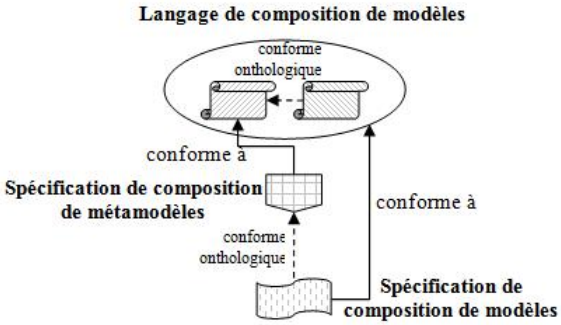


FIGURE 6.4 – Langage de composition de modèles

C. Glue code : Synchronisation de machines virtuelles

Une machine virtuelle est une concrétisation d'un métamodèle. Cette vision est exposée dans notre approche. Ceci nous différencie des approches tel que XMF ou Kermeta dans lesquels le métamodèle et sa machine virtuelle ne sont pas distingués. Cependant, pour nous, cette nuance n'est pas une différence conceptuelle considérable, au contraire, leurs diféranciation découlent des décisions d'implémentation. Pour des raisons historisques, nous avons implanté nos métamodèles en Java. Cependant, du point de vue conceptuel, les métamodèles exécutables et notre pair {métamodèle, machines virtuelle} représentent la même idée.

Cette décision d'implémentation a entrainé le besoin d'une phase de composition de machines virtuelles dans notre approche tandis que dans Kermeta ou XMF, la composition de métamodèles est suffisante pour la composition de modèles.

Pour coordonner les machines virtuelles, un glue code est nécessaire. Puisque le rapport métamodèle/machine virtuelle correspond à abstraction/concrétisation, ce glue code peut être obtenu directement de la spécification de composition de métamodèles via une transformation. La complexité du code généré dépend de la complétude des sémantiques fournies dans la spécification de composition au niveau métamodèle.

Nous avons adopté une démarche pragmatique de type bottom-up dans laquelle nous avons examiné le code écrit dans un ensemble d'expérimentations de composition puis identifié les principales fonctions réalisées par ce code. A partir de ces observations, nous avons identifié certains patrons de code récurent. De cette analyse, nous avons définis comment le code réalisant la composition peut être généré.

Les patrons de code trouvés dans les compositions concernent les activités de *création* et de *persistance* des liens de composition. Le formalisme de composition permet de décrire deux

abstractions pour ces patrons.

Outre ces patrons spécifiques, il est presque impossible de généraliser toute la sémantique spécifique des diverses compositions par un DSL, nous proposons donc une abstraction générique représentant la *synchronisation* entre les machines virtuelles.

En résumé, le glue code généré comprend :

- Une partie chargée de *créer les liens entre les modèles.*
- Une partie chargée de *persister les liens entre les modèles*, i.e. sauvegarder les liens créés dynamiquement à l'exécution pour sauvegarder l'état de l'application, ce qui permet par la suite de rertouver cet état au redémarrage.
- Une partie chargée de *synchroniser les machines virtuelles.* La synchronisation correspond à relier les comportements des machines virtuelles.

6.2.2 Mécanisme de composition : établissement des liens et transformation

Basé sur la classification réalisée dans le chapitre 4, notre approche rentre dans la catégorie des compositions par relation. Le mécanisme de composition utilisé est l'établissement de liens.

Le processus de création de modèle composite (Figure 6.5) consiste à :

1. définir les liens au niveau métamodèle.

2. définir les liens au niveau modèle en conformité avec les liens au niveau métamodèle.

3. générer le code à partir des liens au niveau métamodèle (i.e. transformation).

Les deux premiers points sont faits à travers deux éditeurs : l'un pour décrire la spécification de composition de métamodèles, l'autre pour définir la spécification de composition de modèles. La génération du code est réalisée par un générateur.

Le résultat de ce processus est le modèle composite qui comprend :

- Les modèles composés.
- Les liens entre les modèles. (ces deux premiers points constituent sa structure.)
- Le comportement défini par les machines virtuelles composées et le code généré.

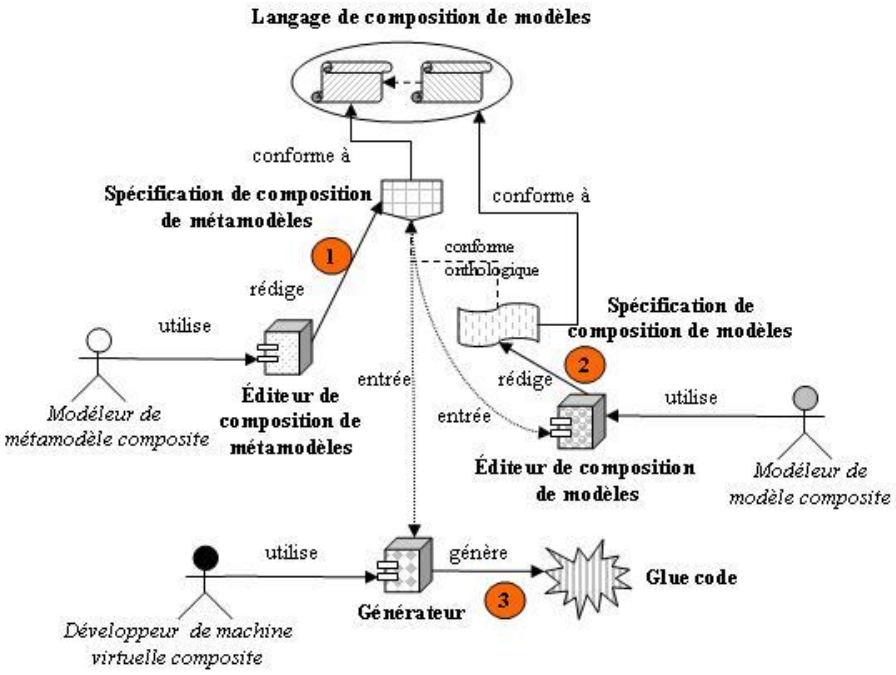


FIGURE 6.5 – Processus de création de modèle composite.

6.2.3 Une classification rapide

Le schéma 6.6 donne une vision globale de notre approche dans laquelle apparaissent les métamodèles et les modèles composés, les éléments de composition sur trois niveaux, le langage et les outils de composition.

Dans le tableau 6.1 [2], nous établissons une classification rapide de notre approche et des autres travaux présentés dans le chapitre 4.

6.2.4 Une approche générique

Notre objectif est d'avoir une approche suffisamment générique capable d'être appliquée dans "tous" les scénarios de composition de domaines et non pour une composition particulière.

Les leçons des approches de composition de modèles actuelles sont intéressantes à reprendre.

- De l'étude des approches de métamodélisation, nous retenons qu'en s'appuyant sur le niveau méta méta modèle lors de l'élaboration des spécifications de composition, notre langage de composition sera suffisamment générique pour exprimer différentes compositions indépendantes des métamodèles et des modèles qu'il compose.
- De l'analyse des approches de composition par fusion, nous retenons que le métamodèle composite est différent des métamodèles en entrée. Dans ce cas, il nous faut construire un nouvel éditeur, une nouvelle machine virtuelle pour chaque composition sans pouvoir à priori réutiliser les éditeurs et les machines virtuelles existantes des métamodèles composés. Il y n'a plus de réutilisation dans ce cas.

2. Ce tableau est un rafinement du tableau 4.2

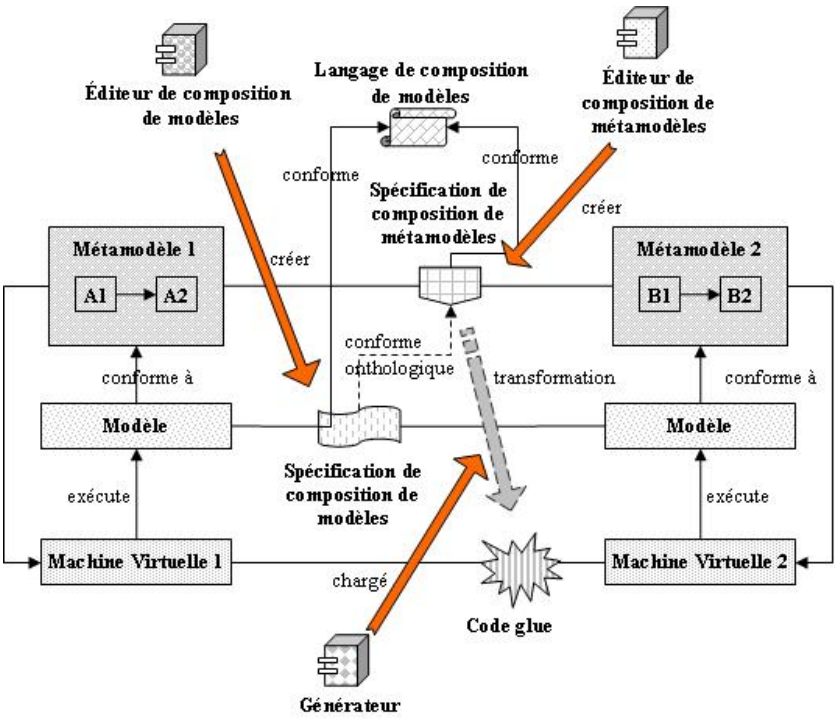


FIGURE 6.6 – Vision globale de notre approche.

- Dans les approches de composition par relation, le problème de réutilisation des outils peut être résolue. En effet, il devient plus simple de mettre des liens entre les métamodèles et les modèles existants, ce qui permet de réutiliser les outils spécifiques existants (i.e. les éditeurs et les machines virtuelles).

Reprenons la discussion de la fin du chapitre 4, nous avons identifié 4 types de composition exogènes et endogènes que nous rappelons ici :

- **Possibilité 1** : $M1 \equiv M2 \equiv M3$: les modèles *m1*, *m2* et *m3* ont le même métamodèle.
- **Possibilité 2** : $M1 \neq M2$ et $M3 = M1 \cup M2$: les modèles *m1* et *m2* ont des métamodèles différents et le métamodèle de *m3* est une union de M1 et M2.
- **Possibilité 3** : $M1 \neq M2 \neq M3$: les modèles *m1*, *m2* et *m3* sont de trois métamodèles différents.
- **Possibilité 4** : $M1 \neq M2$ et $M3 = M1 \cup M2 \cup C_E$: les modèles *m1* et *m2* sont de différents métamodèles. Le métamodèle de *m3* est une union de M1 et M2 plus ses propres concepts C_E.

Notre approche, comme on peut se rendre compte facilement, rentre dans la catégorie des compositions exogènes 2 (mais pas 3). Le cas 4 est une variante du cas 2 dans lequel le métamodèle composite est une union des métamodèles en entrée plus des concepts émergents de ce métamodèle. Pour ce cas, afin de garder la généricité de l'approche, nous proposons de bien séparer deux problèmes différents :

- la spécification des concepts émergents du métamodèle composite.
- la spécification des liens de composition.

Nous proposons de construire un éditeur à part pour les concepts émergents afin de bien

séparer ce dernier de l'éditeur des liens de composition. De cette manière, on peut rendre générique l'éditeur de composition (comme l'éditeur de tissage de AMW).

En résumé, nous proposons deux contraintes pour rendre notre approche générique :

- Tout d'abord, notre langage de composition ne doit pas dépendre d'un métamodèle en particulier mais des (meta) concepts, définis au niveau méta méta modèle.
- Les concepts émergents doivent être séparés des liens de composition. En d'autres mots, ceux ci doivent être modélisés dans des éditeurs différents. L'éditeur pour les concepts émergents peut être spécifique à leur métamodèle. Mais celui pour les liens de composition doit rester générique.

Le reste du chapitre présente notre langage de composition. Avant d'entrer plus dans les détails, nous présentons dans la prochaine section un scénario de composition auquel nous ferons référence par la suite.

	Type de composition	**Modèle**	**Élément(s) de composition**	**Formalisme de composition**	**Mécanisme de composition**
AMW	par opérateurs (définis par l'utilisateur)	en Ecore	modèles de tissages et transformations HOT.	langage de tissage AMW	tissage et transformation
EMF Editeur	par relations	en Ecore	références	*EReference* de du langage Ecore	établissement des références
EML	par opérateurs (fusion)	en EOL	spécification des règles de composition	langage de fusion EML	fusion
GME	par relations	en FCOs	références	*Références* de FCOs	établissement des références
Kompose	par opérateurs (fusion)	en Ecore	spécification des directives de composition	langage de fusion Kompose	fusion
XMF-Mosaic	par relations	en XCore, XOCL, XEBNF	mappings	XSync	établissement des mappings et l'exécution des mappings
Notre approche (Codèle)	par relations	en Ecore	spécification de composition de métamodèles, spécification de composition de modèles, glue code	langage de composition de Codèle	établissement des liens et transformation

TABLE 6.1 – Classification rapide de notre approche.

6.3 Un scénario de composition

Le scénario utilisé est une composition entre le domaine de la *Gestion de Produits*[3] (illustré dans la figure 5.3, section 5.3.1 du chapitre 5) avec le domaine de la *Gestion de Procédés*[4]. La fonctionnalité envisagée est une gestion de l'évolution des données dans un procédé. En effet, dans ce scénario, le domaine de *Procédés* ne propose que des fonctionnalités de création, de manipulation, et de transformation des données circulant dans un procédé mais il ne fournit pas les fonctionnalités de gestion de données telles que le versionnement ou la persistance. Par la collaboration avec le domaine de *Produits*, le domaine de *Procédés* peut obtenir un contrôle complexe de l'évolution des données.

6.3.1 Domaine de Gestion de Procédés

Le domaine de *Procédés* permet de définir des procédés qui sont une suite d'étapes réalisées dans un but donné. Ce domaine a un langage nommé APEL (*Abstract Process Engine Language*) [EDA98, Vil02]. Les procédés modélisés par APEL rentrent dans la catégorie des procédés exécutables comme EPOS SPELL, MARVEL, SPADE[5].

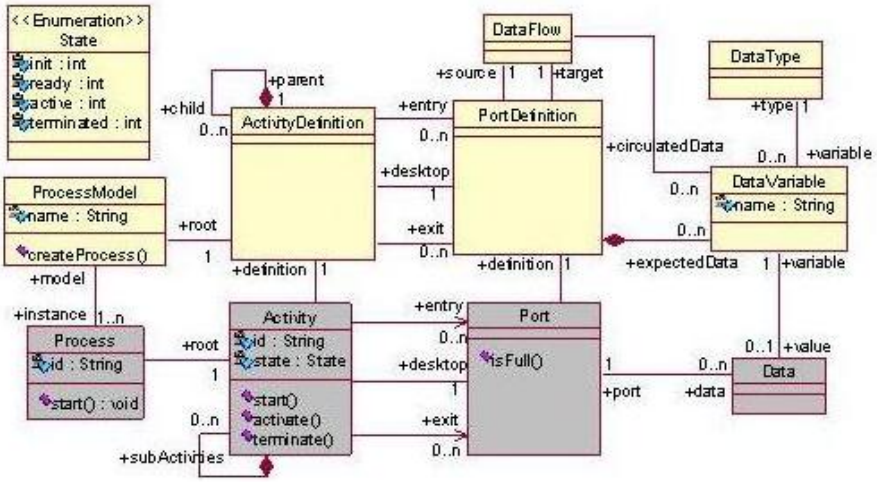


FIGURE 6.7 – Métamodèle de Gestion de Procédés

La figure 6.7 est le métamodèle de ce langage. Il contient le minimum de concepts permettant de modéliser et d'exécuter un procédé. Au moment de la modélisation, un modèle de procédé peut être défini par les concepts en jaunes. Pendant l'exécution, un procédé qui est une instance de ce modèle va être exécuté par les concepts en gris.

Un procédé (*Process*) est constitué d'un ensemble d'*activités* (*Activity*) qui sont reliées par des *flots de données* (*DataFlow*) à travers des *ports* (*Port*).

Une activité est une étape du procédé au cours de laquelle une action est exécutée. L'activité produit, consomme, et transforme les données (*Data*). Une donnée circulant dans un procédé a un type (*DataType*). Au niveau modèle, la donnée est représentée par une variable (*DataVariable*). La valeur de cette variable (i.e. la donnée réelle) n'est disponible qu'au moment de l'exécution du procédé. Par contre, les autres éléments structurels du procédé tels que ses activités, ses ports, ses

3. Dans le reste du chapitre, ce domaine est appelé "le domaine de *Produits*"
4. Dans le reste du chapitre, ce domaine est appelé "le domaine de *Procédés*"
5. Les références citées par [SJ05]

flots de données, etc. peuvent être définis avant l'exécution (*ActivityDefinition*, *PortDefinition*, *DataFlow*). Ces concepts modélisent la structure du procédé.

Les flots de données décrivent la façon dont les activités s'enchaînent et s'échangent des données.

Les ports sont les endroits où une activité reçoit ou transfère un produit. Il y a trois types de ports : entrée (*entry*), sortie (*exit*) et desktop (*desktop*). Le Desktop (poste de travail) représente les données en cours de modification ou transformation par l'activité. Un Desktop ne peut être relié par un flot de données qu'aux ports d'entrée ou de sortie de la même activité. Une activité peut comporter plusieurs ports d'entrée et de sortie. Lorsque toutes les données attendues dans un port d'entrée sont arrivées, l'activité est démarrée. De manière similaire, lorsque le port de sortie de l'activité contient toutes les données attendues, l'activité peut être terminée. Le démarrage et la terminaison d'une activité peuvent être manuel, à la charge de l'utilisateur ou automatique. Dans le cas où le port est déclaré *automatique*, le démarrage et la terminaison seront faits automatiquement. Si l'activité a plusieurs ports de sortie, il faut choisir un port spécifique pour la terminer.

Une application du domaine de Gestion de Procédés : un Procédé de Développement Logiciel simple.

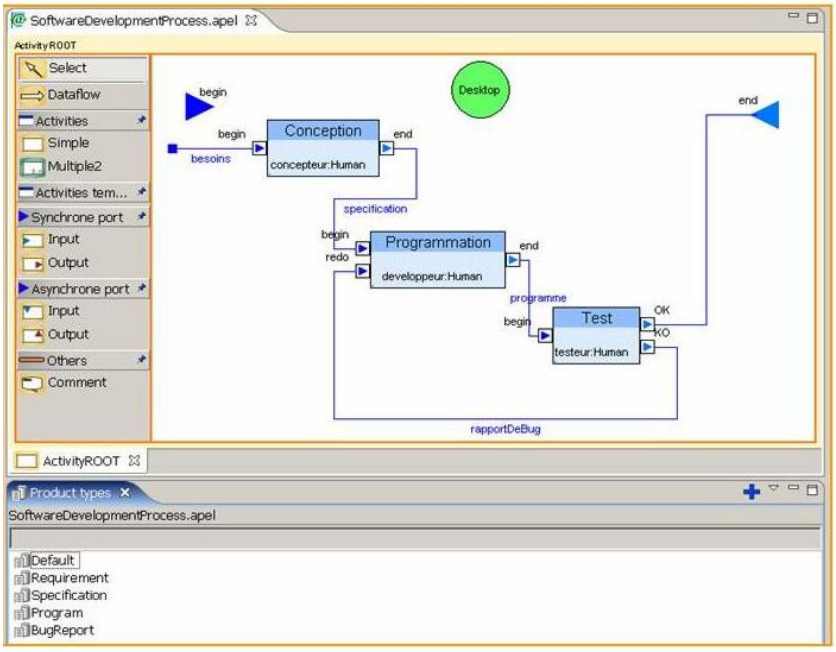


FIGURE 6.8 – Un procédé de Développement Logiciel simple.

Nous décrivons dans la figure 6.8 un modèle de procédé conforme au métamodèle ci-dessus. Il s'agit d'une application de ce domaine décrivant un procédé de Développement Logiciel simple qui ne comporte que trois activités : *Conception*, *Programmation* et *Test*. L'activité *Conception* reçoit les besoins (*besoins*) et produit une spécification (*specification*). Ensuite, cette spécification est transférée vers le port *begin* de l'activité *Programmation*. Cette activité utilise cette donnée et produit un programme (*programme*) qui va être testé par l'activité *Test*. Si le résultat de test est bon, le procédé va terminer par le port *OK*, si non, un raport listant les problèmes (*rapportDeBug*) sera renvoyé au port *redo* de l'activité *Programmation* pour le deboggage.

Ce modèle est défini à l'aide d'un éditeur graphique développé pour ce domaine (la figure 6.8). Le panel en bas (*Product types*) définit les types de données qui sont instances du concept *DataType*. Dans cet exemple, il y a 4 types de données : *Requirement*, *Specification*, *Program*, et *BugReport*. *Defaut* est un type générique qu'APEL introduit pour assurer que toutes les données circulant dans le procédé sont typées. Les données (représentées par des annotations sur les flots de données) font référence à ces types. A titre d'exemple, la donnée nommée *besoins* (annotée sur le flot de données d'entrée de l'activité *Conception*) est de type *Requirement*.

6.3.2 Composition des Domaines de Procédés et de Produits

Puisque le domaine de *Procédés* ne se concentre que sur les aspects liés aux activités et leurs interactions, son concept de "donnée" (*Data*) est très abstrait. Il représente uniquement un jeton qui circule entre les activités. Afin d'avoir une définition et une gestion précise des "données", ce concept abstrait doit être concrétisé. C'est pourquoi nous l'associant au concept de "révision" (*Revision*) du domaine de *Produit*. Ce cas est typique de la séparation des préoccupations où le concepteur d'un domaine ne se concentre que sur une facette de l'application globale, et fait abstraction de tous les détails qu'il considère NE pas être centraux à cette préoccupation. Cependant, lors de l'intégration des différentes préoccupations dans une composition, ces concepts seront concrétisés pour pouvoir exposer les fonctionnalités complètes.

Dans notre exemple, le concept abstrait *Data* du domaine de *Procédés* est associé au concept de *Revision* du domaine de *Produits* (voir la figure 6.9). Le cas d'utilisation typique de cette association est la création d'un historique de versions d'une donnée modifiée dans un procédé. Dans le domaine de *Procédés*, les activités font évoluer la donnée alors que dans le domaine de *Produits*, les révisions retracent toutes les évolutions sous la forme de versions consécutives correspondant aux différent états de la donnée dans le temps (entre les manipulations effectuées sur celle-ci). Ceci est un exemple typique des interactions inter domaines exploitées par la relation établie entre deux concepts.

La figure 6.9 exprime la composition entre le domaine de *Procédés* et le domaine de *Produits*. Puisque les données et les révisions sont typées, leurs types doivent être eux aussi composés. La relation entre *DataType* et *ProductType* signifie que les données doivent être associées à une révision si et seulement si il existe la relation correspondante au niveau de leurs types. Si les types ne sont pas composés, aucune relation entre leurs données et leurs révisions ne sera créée.

Nous allons donner ci dessous deux applications de ce domaine composite. L'une est un procédé de Développement Logiciel en Java et l'autre est celui en PHP. Chaque application a son propre modèle composé de deux sous modèles : l'un appartenant au domaine de *Procédés* et l'autre du celui de *Produits*. Le modèle de procédé utilisé dans la composition pour les deux applications est le même ; c'est celui qui a été présenté dans la section 6.3.1.

Comme on peut le remarquer, ce modèle est générique. Ses types de données (*Requirement*, *Specification*, *Program*, *BugReport*) ont un sens général. Ils ne dépendent d'aucune plateforme et technologie de développement spécifique. Ceci est un avantage car ce modèle peut être utilisé dans divers scénarios. Mais il représente également un inconvénient car en pratique, un modèle trop générique n'est pas directement utilisable (exécutable). Le besoin de spécialiser ce procédé pour le rendre utilisable dans un contexte spécifique (par exemple avec les technologies Java ou PHP) est nécessaire.

Cette spécialisation signifie que les types de données du procédé doivent être les types spécifiques de ce contexte. Par exemple, *Program* devient *JavaFile* ou *PHPFile*. Ceci peut être obtenu par une composition avec un modèle de *Produits* modélisant ces types spécifiques. Dans les deux applications que nous avons vu, le modèle de procédé est composé alternativement avec deux modèles de produits différents, contenant les types de données spécifiques, destinés à un Projet de Développement Logiciel en Java et à celui en PHP. Nous retrouvons l'idée de concrétisation des préoccupations mentionnée au début de cette section.

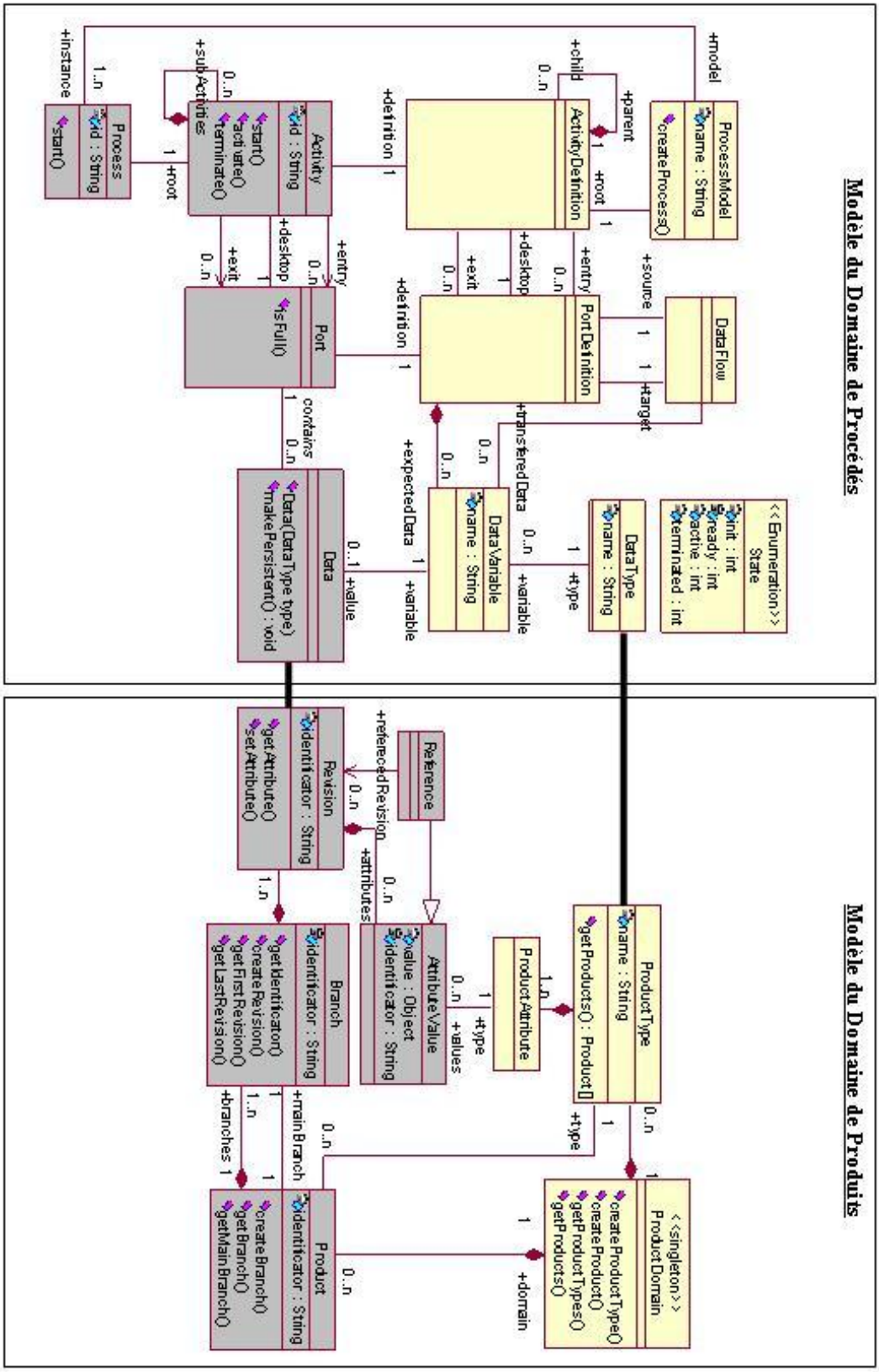


FIGURE 6.9 – Composition des Domaines de Procédés et de Produits.

Application 1 : Procédé de Développement Logiciel en Java

Le procédé de Développement Logiciel en Java reconnaît les types de données spécifiques à ce langage tel que les fichiers .java, .jar, etc. Dans cette application, un *Requirement* est un *Use Case Document*, la spécification du logiciel est écrite en langage JML [6]. Le programme est constitué de fichiers Java, et les bugs sont gérés par Bugzilla [7]. Le tableau 6.2 résume la composition de cette application.

Domaine	**Gestion de Procédés**	**Gestion de Produits**
Concept	`DataType`	`ProductType`
Valeur	*Requirement* *Specification* *Program* *BugReport*	*Use Case Document* *JML Specification* *Java File* *URL Bugzilla* *Jar File*

TABLE 6.2 – Procédé de Développement Logiciel en Java.

Application 2 : Procédé de Développement de Logiciel en PHP

Dans le second scénario, le procédé de Développement Logiciel en PHP concerne d'autres types de données. Par exemple, une spécification n'est pas écrite en JML mais en UML. Le programme n'est pas un ensemble de fichiers Java mais PHP. Le système de gestion des problèmes dans ce cas est Mantis Bug Tracker [8] .

On observe dans cet exemple que le même modèle de procédé a été utilisé dans deux contextes différents. Bien sur, on peut définir un modèle de procédé spécifique en créant directement les types de données spécifiques souhaités. Mais ceci implique qu'on aura des modèles de procédé différents pour chaque contexte, ce qui d'un point de vue pratique est une perte de temps, et d'un point de vue conceptuel empêche la réutilisation et est contradictoire avec la vision de séparation des préoccupations. La solution consistant à définir de petits modèles génériques et de les réutiliser en les composant est plus intéressante que de définir pour chaque nouvelle application un unique modèle spécifique.

Domaine	**Gestion de Procédés**	**Gestion de Produits**
Concept	`DataType`	`ProductType`
Valeur	*Requirement* *Specification* *Program* *BugReport*	*Use Case Document* *UML Specification* *PHP File* *URL Mantis Bug Tracker*

TABLE 6.3 – Procédé de Développement Logiciel en PHP.

Le reste de ce chapitre décrit notre langage de composition de modèles.

6. Java Modeling Language http://www.cs.ucf.edu/ leavens/JML/.
7. http://www.bugzilla.org/.
8. http://www.mantisbt.org/.

6.4 Un langage de composition de modèles : vue globale

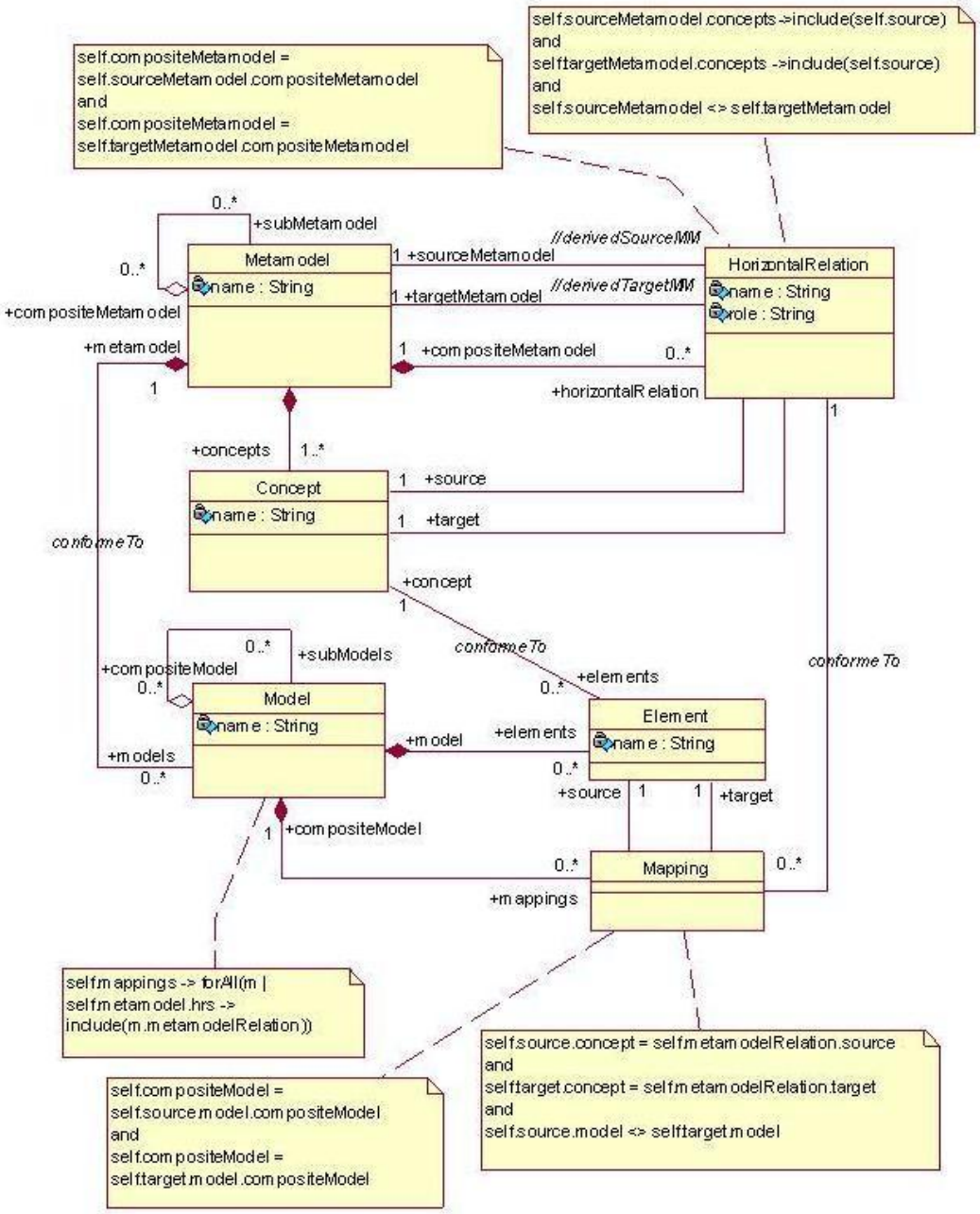


FIGURE 6.10 – Vision globale du Métamodèle de Codèle

Notre langage permet d'exprimer les informations liées à la composition de modèles. Pour structurer ces informations, nous définissons un métamodèle (Figure 6.10). Il est décomposé en deux parties distinctes, liées aux concepts de composition de métamodèles et de composition de modèles. Par rapport au scénario ci-dessus, ces parties correspondent aux liens entre les concepts *DataType* et *ProductType* (Figure 6.9) puis entre leurs instances *Specification* et *UMLSpecification* (Tableau 6.3).

La difficulté du travail est de s'abstraire des concepts des domaines composés et de s'appuyer uniquement sur le méta méta modèle dans lequel les concepts de composition sont définis.

Dans la figure 6.10, un métamodèle est représenté par le concept `Metamodel`. Il a un nom (`name`) et contient un ensemble des concepts (`Concept`). Il peut être composite (`compositeMetamodel`) s'il est construit à partir de sous métamodèles (`subMetamodels`). Il peut avoir plusieurs sous métamodèles et peut appartenir à plusieurs métamodèles composites différents.

A titre d'exemple, nous prenons le métamodèle de *Produits*. Il porte le nom "*Gestion de Produits*". Ce métamodèle a un ensemble de concepts tels que *Product*, *Branch*, *Revision* etc. et il est sous métamodèle d'un métamodèle composite nommé *Procédé_Produit*. Dans ce scénario, *Procédé_Produit* a deux sous métamodèles : *Procédé* et *Produit*. Cependant, *Produit* peut participer à plusieurs autres métamodèles composites, par exemple, avec le métamodèle de *Gestion de l'Espace de Travail*. Dans cette composition, les artefacts créés dans un espace de travail modélisé par le métamodèle de *Gestion des l'Espace de Travail* seront versionnés par le domaine des *Produits*.

De façon similaire, au niveau modèle, un modèle est représenté par le concept `Model`. Il est identifié par son nom (`name`). Il peut être composé de nombreux modèles (`compositeModel`) et/ou participe, comme sous modèle, à de nombreux modèles composites différents (`subModel`). Les éléments du modèle sont modélisés par le concept `Element`. Un élément est une instance d'un concept (`instanceOf`) et possède un identificateur. Son modèle est conforme au métamodèle (`conformeTo`).

A titre d'exemple, nous prenons le procédé de développement logiciel présenté dans 6.3.1 comme illustration. Ce modèle est conforme à son métamodèle, i.e. le métamodèle de *Procédés*. Ses éléments tels que *Conception*, *Programmation*, *Test* sont des instances du concept *Activity-Definition*. Ce modèle participe, comme sous modèle, à deux modèles composites différents, i.e. Application 1 (Tableau 6.2) et Application 2 (Tableau 6.3).

Notre approche s'appuie sur le principe de composition par relation ; deux concepts *Relation Horizontale* (`HorizontalRelation`) et *Correspondance* (`Mapping`) sont introduits comme les principaux moyens pour modéliser les liens.

Les relations horizontales modélisent les liens au niveau métamodèle. Alors que les correspondances expriment les liens au niveau modèle. Le nom "*Relation Horizontale*" provient de notre propre histoire de l'architecture Mélusine (chapitre 5). Dans cette architecture, la composition d'outils hétérogènes dans une fédération logicielle est appelée *composition verticale* car la couche conceptuelle et la couche d'implémentation (outils) sont à des niveaux d'abstraction différents. Par contre, la composition entre domaines réalisée entre leurs couches conceptuelles, qui sont au même niveau d'abstraction, est appelée *composition horizontale*. Nous allons détailler ces concepts dans les sections 6.5 et 6.6.

6.5 Composition de métamodèles : Relations horizontales

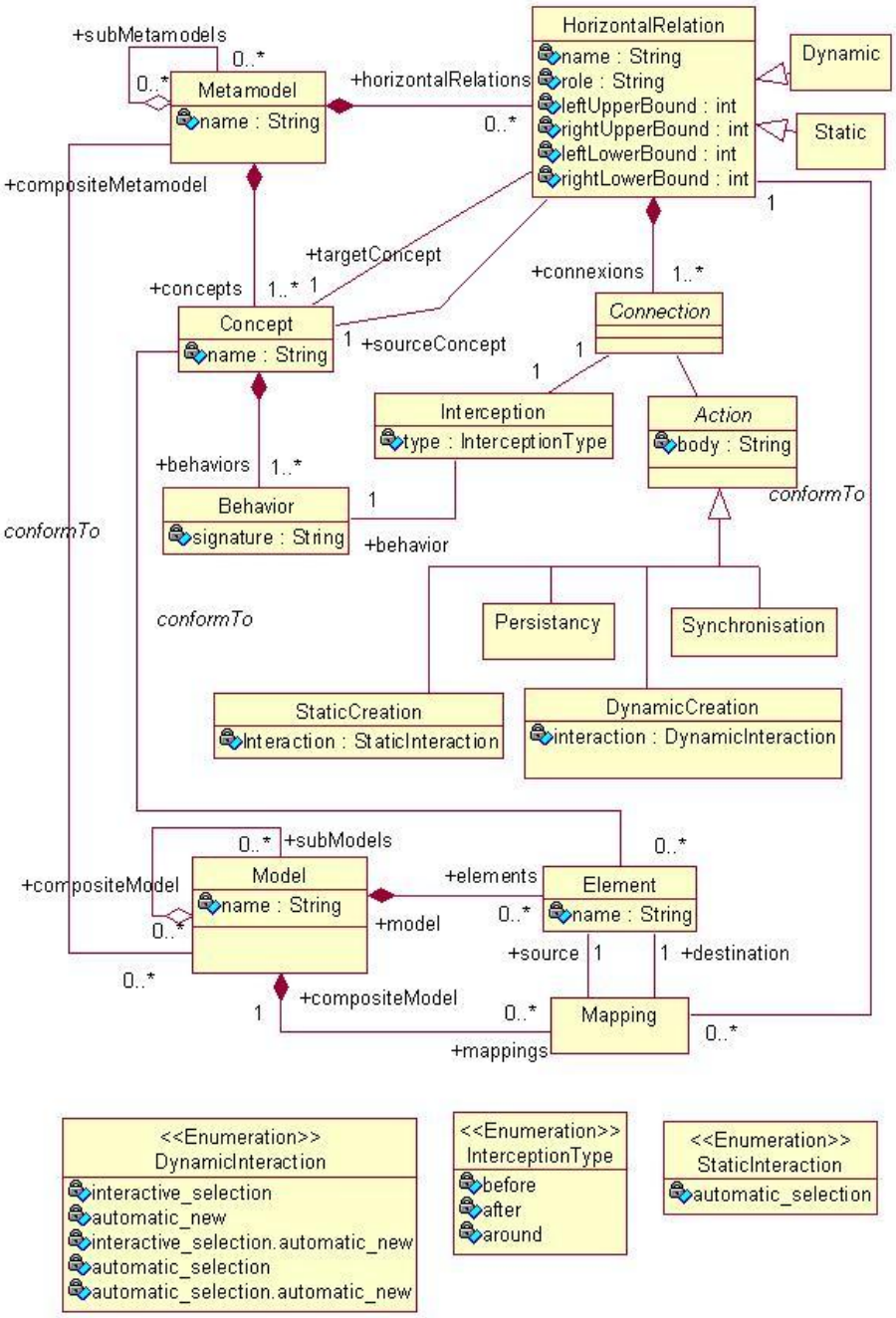


FIGURE 6.11 – Métamodèle de Codèle.

La composition de métamodèles consiste à :

- Définir la structure du modèle composite, et
- Définir la manière de l'exécuter.

Le premier point concerne *la composition structurelle* (des modèles) et le deuxième concerne la *composition comportementale* (des modèles).

Dans notre approche, les relations expriment non seulement la structure du modèle composite (i.e. composition structurelle) mais aussi la coopération sémantique entre les métamodèles à l'exécution (i.e. composition comportementale). Ces deux problèmes sont connus comme une des difficultés du concept d'`Association` dans la communauté UML : "*One of the biggest difficulties in modeling with UML stems from the attempt to abstract with one construct, the association, both the static structure of the system and the structure of interactions between objects*" [GLF04].

En effet, il est clair qu'il existe plusieurs types d'interaction dynamique entre objets tels que les appels de méthode que les associations ne sont pas capables d'exprimer. Dans notre approche, la coopération sémantique entre les métamodèles est réalisée par le mécanisme AOP (chapitre 5). Il s'agit d'un type d'interaction qui n'est pas supporté par le concept d'association UML traditionnel. Nous avons donc besoin d'un concept de "relation" plus fort capable d'exprimer non seulement la structure du modèle composite mais aussi les captures AOP interceptant les méthodes définies dans les métamodèles. Le concept de *Relation Horizontale* (voir la Figure 6.11) est introduit pour cette raison.

La *Relation Horizontale* est un concept permettant de modéliser les liens à établir entre les concepts de deux métamodèles. Ces relations sont chargées de réaliser la composition structurelle et comportementale (pour les modèles). Dans [Ste02], Stevens a proposé de sous typer les associations UML en *association statique* et *association dynamique* ; dans [GLF04], Genova propose de spécialiser en *association structurelle* et *association de vue contextuelle* (*contextual view association*). Cependant, nous pensons que ces distinctions ne sont pas adéquates lorsque chaque relation contient les deux aspects. Notre distinction : *composition structurelle* (paragraphe 6.5.1) et *composition comportementale* (paragraphe 6.5.2) exprime simplement qu'une relation peut être vu sous plusieurs facettes ; les deux aspects : *structurel* et *comportemental* sont décrits par un seul concept de *Relation Horizontale*.

6.5.1 Composition structurelle (pour les modèles)

Du point de vue structurel, une relation horizontale équivaut à une association UML. Il s'agit de relations binaires, qui relient deux concepts de deux métamodèles. Dans notre scénario, une relation est établie entre les concepts de *Product* et de *Data*.

Une relation horizontale a des cardinalités (`leftUpperBound`, `leftLowerBound`, `rightUpperBound`, `rightLowerBound`) qui spécifient le nombre minimum et maximum d'éléments source et cible pouvant participer à un lien d'instance de cette relation horizontale. Par exemple, les cardinalités de la relation *Product_Data* sont [1,1]..[1,1]., c'est-à-dire, un élément du concept *Product* est associé à un seul et un seul élément du concept *Data*.

Dans une composition, plusieurs relations horizontales peuvent être établies. Dans le scénario de la section 6.3, nous avons deux relations : *ProductType_DataType* et *Product_Data*.

Une relation horizontale est orientée. L'orientation se retrouve dans le nom du rôle des concepts qu'elle compose (`sourceConcept` v.s. `targetConcept`). Dans la relation *ProductType_DataType*, le concept source est *ProductType* et le concept cible est *DataType*. La flèche pointe du concept source au concept cible.

Dans notre approche, certains éléments du modèle peuvent être définis avant l'exécution, mais d'autres n'aparaissent qu'à l'exécution. On peut définir par exemple les types de données (instances du concept *DataType*) d'un modèle de procédé avant l'exécution mais les données réelles (instances du concept *Data*) ne sont connues qu'au moment de l'exécution. Basé sur

ces caractéristiques, nous proposons deux types de relations horizontales : *statique* (`Static`) et *dynamique* (`Dynamic`) (voir la Figure 6.11).

- `Static` : Toutes les instances de la relation sont connues avant l'exécution.
- `Dynamic` : Les instances de la relation ne peuvent définies que pendant l'exécution.

6.5.2 Composition comportementale (pour les modèles)

La composition comportementale consiste à définir les interactions permettant à deux méta-modèles de collaborer afin de réaliser une fonctionnalité complexe inter domaines. Conceptuellement, nous considérons chaque interaction comme une connexion sémantique entre les domaines. Dans notre scénario *Procédé_Produit*, on définit la connexion sémantique suivante : lorsque la méthode *MakePersitent()* du concept *Data* est appelée, il faut créer une révision dans le domaine de *Produits*. Cette connexion est réalisée, dans notre approche, par la technique AOP. En effet, une capture AOP va intercepter la méthode *MakePersitent()* et va créer la révision correspondante permettant de rendre persistante cette donnée. Afin de représenter ces connexions sémantiques, nous introduisons le concept de *Connexion* (`Connection`). Une relation horizontale définit un ensemble de connexions.

Connexion

Une connexion établit une ligne de communication entre deux instances de deux concepts. Elle est définie par une tuple `Interception` et `Action`

$$\texttt{Connexion} == <\texttt{Interception}, \texttt{Action}>.$$

L'`Interception` est chargée d'établir une connexion lorsque le comportement (`Behavior`) est invoqué sur une instance du concept (`Concept`). Une `Interception` a un `InterceptionType` (`before`, `after`, `around`) définissant la position relative à la méthode capturée où l'`Action` sera insérée.

$$\texttt{Interception} == <\texttt{Concept}, \texttt{Behavior}>$$

`Concept` est associé à une classe Java représentant ce concept, et `Behavior` est associé à une méthode de cette classe. Par exemple, on peut définir une interception capturant le méthode *MakePersistance()* du concept *Data*.

`Action` est le code qui synchronise les machines virtuelles de deux domaines. Ce code, comme explicité dans la paragraphe 6.2.1, réalise 3 opérations : création des liens, synchronisation et persistance. Pour la création de liens, nous distingeons deux cas :

- Réification des liens déjà établis.
- Etablissement des liens pendant l'exécution.

La réification des liens déjà établis est appelée la *création statique* (`StaticCreation`), alors que l'établissement des liens pendant l'exécution est appelé la *création dynamique* (`DynamicCreation`).

Les deux premières opérations (création des liens et synchronisation) concernent l'aspect fonctionnel du système, alors que le dernier (persistance) concerne un aspect non fonctionnel. De ce point de vue, on peut dire que la composition comportementale de notre approche permet de faire la composition fonctionnelle et non fonctionnelle.

Les connexions peuvent être catégorisées selon les 3 types d'opérations ci-dessus. Nous avons alors les types de connexions suivants : *connexion de création statique* (`StaticCreation`), *connexion de création dynamique* (`DynamicCreation`), *connexion de synchronisation* (`Synchronisation`), et *connexion de persistance* (`Persistancy`).

A. Connexion de création Une connexion de création établit la correspondance entre deux éléments du modèle. Établir les correspondances consiste à définir :

- Le moment où il faut établir les correspondances.
- La façon d'établir les correspondances.

(a) Le moment d'établir les correspondances Le plus souvent, une correspondance est établie lors de la création de l'élément source. Dans notre scénario, lors de la création d'une donnée, une révision sera créée dans la base. Cependant, le moment d'établir les correspondances n'est pas forcément lors de la création de l'élément source. Ce peut être un moment quelconque qui dépend de la sémantique des interactions de chaque composition. Par exemple, si on ne crée pas le lien entre une donnée et sa révision au moment de la création de cette donnée, on peut le créer au moment de la rendre persistante lors de l'appel du méthode *MakePersitent()*.

L'information spécifiant le moment où sont établies les correspondances est définie dans `Interception`. C'est la méthode à capturer définie dans l'interception qui spécifie ce moment (`Interception.behavior`). Lorsque cette méthode est appelée, elle est interceptée et la correspondance est créée. Dans notre scénario, si on veut établir le lien lors de la création d'une donnée, il faut définir, dans l'interception, que la méthode à capturer correspond au constructeur *Data(DataType type)* du concept *Data*.

(b) La façon d'établir les correspondances Lors de la création d'un lien, il faut considérer deux problèmes :

- L'existence de l'élément destination, et
- La façon de retrouver cet élément.

L'existence de l'élément destination i.e. l'élément destination existe ou pas au moment de l'interception. Par exemple, le lien entre le type de donnée *Program* et le type de produit *Java File* 6.2 est créé lorsque l'élément destination *Java File* existe. Par contre, la révision n'existe pas lors de la création d'un lien entre une donnée et sa révision ; on doit créer la révision au moment de la création du lien.

Notre stratégie est donc simple : si l'élément destination existe, on le selectionne ; s'il n'existe pas, on le crée. Nous identifions ces deux cas comme `new` et `selection` :

- `new` : L'élément destination de la correspondance n'existe pas encore. La correspondance est établie après une phase de création de l'élément destination.
- `selection` : L'élément destination de la correspondance existe, la correspondance le sélectionne et le rattache à la correspondance.

La façon de retrouver l'élément destination La manière de retrouver l'élément destination doit également être spécifiée. Par expérience, nous identifions deux façons distinctes pour retrouver l'élément destination) : *automatique* (`automatic`), ou *manuel* (`interactive`).

- `automatic` : l'élément destination est retrouvé automatiquement.
- `interactive` : l'élément destination est retrouvé par le biais d'une intervention humaine.

La combinaison des options ci-dessus donne *des façons d'établir les correspondances* différentes. Nous avons donc les possibilités valides suivantes :

- `automatique_new` : La destination est créée automatiquement, et la correspondance est établie automatiquement après la création.
- `automatic_selection` : La destination est sélectionnée automatiquement parmi les éléments existants, et la correspondance est établie automatiquement après la sélection.

- `automatic_selection.automatic_new` : La destination est d'abord sélectionnée automatiquement, si la sélection échoue, un nouvel élément va être créé.
- `interactive_selection` : La destination est sélectionnée par une intervention humaine. La correspondance est établie après la sélection.
- `interactive_selection.automatic_new` : La destination est sélectionnée par un humain ; mais ce dernier peut décider de demander la création d'un nouvel élément.

Interactions : `DynamicInteraction` v.s. `StaticInteraction` (i.e. les façons d'établir les correspondances selon le type de relation horizontale)

Nous appelons *Interaction* la façon d'établir les correspondances. Dépendant du type de relation horizontale (*dynamique* ou *statique*), nous spécialisons *Interaction* en `DynamicInteraction` et `StaticInteraction`.

Une relation horizontale peut être statique ou dynamique. Si une relation horizontale est statique, toutes les correspondances sont établies avant l'exécution. Si elle est dynamique, toutes les correspondances sont établies pendant l'exécution. Autrement dit, nous faisons une distinction selon que les liens soient établis au moment de la définition du modèle composite, ou qu'ils soient établis à l'exécution.

La distinction entre le moment de modélisation et le moment d'exécution (d'un modèle composite) implique qu'à l'exécution de ce modèle, il y a deux façons différentes d'établir la correspondance :

- La réification d'une correspondance déjà définie (au moment de la modélisation)
- La création d'une nouvelle correspondance.

En se basant sur cette distinction, nous définisons les concepts : *connexion de création statique* (`StaticCreation`) et *connexion de création dynamique* (`DynamicCreation`).

- `DynamicInteraction` : La correspondance n'existe pas avant l'exécution, elle est créée au moment d'exécuter le modèle. La façon de retrouver la destination pour cette correspondance est définie par le choix d'une des 5 options listées ci-dessus.
- `StaticInteraction` : La correspondance est définie avant l'exécution, entre les deux modèles composés. On a besoin ni de créer la destination, ni de demander une intervention humaine pour la trouver. La réification de cette correspondance consiste simplement à sélectionner automatiquement la destination prédéfinie. Une seule option est donc valide dans ce cas : `automatic_selection`.

B. Connexion de Synchronisation La connexion de synchronisation (`Synchronisation`) est un type de connexion générique, dans laquelle on effectue la synchronisation entre deux modèles. La sémantique de la synchronisation est arbitraire, elle dépend de chaque composition ; on ne peut donc pas les définir, de façon déclarative.

C. Connexion de Persistance La connexion de persistance (`Persistance`) est chargée de faire persister l'état du modèle composite ; plus, précisément, elle est chargée de faire persister les liens créés pendant l'exécution. Ces liens sont crées dynamiquement et font évoluer l'état du modèle composite. Comme ces liens sont considérés comme des éléments de ce modèle, il faut les persister comme les autres éléments.

Bilan sur les Connexions

Le concept de *Connexion* dans notre langage exprime l'aspect comportemental de la relation horizontale. Si on ne prend en compte que l'aspect structurel, le concept "association" d'UML

est suffisant pour composer ; il est par contre incapable d'exprimer le comportement associé. Notre concept de *Connexion* modélise une capture AOP dans l'implémentation. Le thème de recherche de la modélisation orienté aspect s'attaque à ce problème. Dans [AEB03], Aldawud et al. a proposé un profil UML pour la modélisation orienté aspect. Ils utilisent des stéréotypes permettant de représenter les concepts de l'AOP. Par exemple, le stéréotype "aspect" sur une classe représente un aspect AOP. Des travaux similaires sont décris dans [KTG+06][MV06]. Nous pensons que l'objectif de ces travaux est similaire au notre : abstraire les captures AOP.

6.6 Composition de modèles : Les Correspondances

Nous introduisons le concept de Coresspondance (`Mapping`) pour la modélisation des liens entre les modèles.

Conceptuellement, une correspondance est une instance d'une relation horizontale. Ses caractéristiques (structurelles et conportementales) sont définies par la relation horizontale pour laquelle elle est conforme (conformeTo) (figure 6.12). Une correspondance ne contient qu'une paire d'éléments qu'elle relie. Elle ne décrit pas la sémantique. Nos correspondances sont similaires aux correspondances du modèle de tissage de AMW.

À titre d'exemple, nous pouvons prendre la correspondance *Specification_UML Specification* (tableau 6.3) comme illustration. L'élément source est *Specification* et l'élément cible est *UML_Specification*.

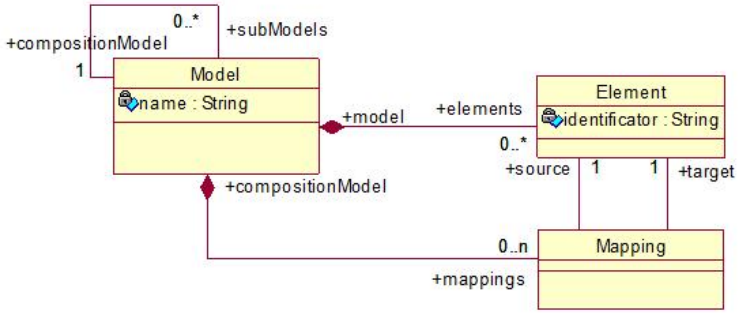


FIGURE 6.12 – Le concept de Correspondance.

6.7 Synthèse

Dans ce chapitre, nous avons présenté notre approche de composition de modèles. Les éléments nécessaires à la composition sont identifiés. Nous avons également proposé un langage pour les décrire de façon déclarative. Ce langage est formalisé dans un métamodèle constitué de deux parties : l'une est liée à la composition de métamodèles et l'autre est liée à la composition de modèles. Ces deux parties ne sont pas disjointes, en revanche, les concepts de la composition de modèles sont liés aux concepts de la composition de métamodèles par une relation de conformité ontologique. Les deux concepts principaux dans ce langage sont *Relation Horizontale* et *Correspondance* qui caractérisent notre approche dans la catégorie des approches de composition par des relations. Ce type de composition nous permet de réutiliser les métamodèles/modèles sans modification (fusion des classes, héritage des classes/enlèvement des éléments, reconsidération des éléments etc.)

La *Relation Horizontale* est le concept représentant les liens entre les métamodèles. Nous en avons distingué deux types : statique et dynamique. Ces relations expriment la structure des modèles ainsi que les interactions entre eux. Trois types d'interactions ont été identifiés :

- Création des liens entre les modèles.
- Persistance des liens entre les modèles.
- Synchronisation des machines virtuelles.

En ce qui concerne la création des liens, nous avons identifié plusieurs façons de créer par choix des options `new` v.s. `selection` et `automatic` v.s. `interactive`.

La *Correspondance* est le concept modélisant le lien au niveau modèle. Ce concept relie une paire d'éléments de modèle. Une correspondance est créée, manipulée et persistée sous la direction des politiques de création, de synchronisation et de persistance définies par sa relation horizontale.

Les concepts dans notre langage sont suffisamment génériques pour permettre de faire différentes compositions de manière indépendante des métamodèles et des modèles qu'ils composent.

Dans le chapitre suivant, nous allons présenter la mise en oeuvre de ce langage. Elle comporte des outils permettant de définir les spécifications de composition et de produire le code exécutant les modèles.

Chapitre 7

Codèle : Un outil de composition de modèles exécutables

Dans ce chapitre, nous présentons notre outil de composition de modèles qui reconnaît et implémente le langage proposé. Notre outil a trois modules. Chaque module automatise une étape du processus de composition.

- **Composeur de métamodèles** : réalise la composition de métamodèles. Ce module permet de spécifier, de manière descriptive, les relations horizontales entres les métamodèles composés.
- **Composeur de modèles** : réalise la composition de modèles. Il permet de créer les correspondances entre les modèles.
- **Générateur** : réalise (partiellement) la composition de machines virtuelles. Il transforme les relations horizontales en code "collant" deux machines virtuelles. Dans notre approche, les machines virtuelles sont réalisées en Java. Les codes générés sont donc en langage Java.

Ces trois modules constituent l'outil Codèle. Nous allons le présenter en se concentrant sur les trois sujets suivants :

- La description des fonctionalités de l'outil.
- La mise en oeuvre et les expérimentations de l'implémentation de Codèle.
- La validation et l'utilisation de Codèle en pratique.

Ce chapitre est organisé comme suit : la section 7.1 présente les modules de Codèle et leurs fonctionnalités. Sa mise en oeuvre est décrite dans la section 7.2. La section 7.3 discute de la validation et de l'usage de Codèle. La section 7.4 conclut le chapitre.

7.1 Les modules de Codèle

7.1.1 Composeur de métamodèles

La figure 7.1 exprime l'architecture générale du composeur de métamodèles.

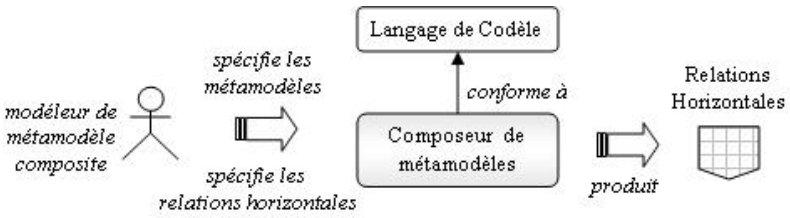


FIGURE 7.1 – Architecture générale du composeur de métamodèles

Ce module implémente le langage Codèle 6. Il reçoit les métamodèles sources (*réifiés* dans les concepts du langage Codèle[1]). En se basant sur ceux-ci, il permet au modéleur du métamodèle composite de *spécifier* ses relations horizontales.

Le processus de composition de métamodèles réalisé par ce module comporte deux étapes :

1. D'abord, il faut spécifier les métamodèles sources en langage Codèle.
2. Ensuite, il faut spécifier les relations horizontales.

A. Spécification de métamodèles

La figure 7.3 est l'aperçu du composeur de métamodèles. Il s'agit d'un éditeur dans lequel le modéleur du métamodèle composite peut utiliser les concepts du langage Codèle tels que `Domain`, `Class`, `Method` pour décrire ses métamodèles sources.

Dans cette figure, le composeur de métamodèle illustre l'exemple de nos domaines de *Procédé* et de *Produit*. La structure des métamodèles de *Procédé* et de *Produit* sont spécifiés en terme des instances des concepts `Domain`[2], `Class`, `Method` du langage Codèle.

Le métamodèle de *Procédé*, comme montré dans la figure, est représenté par l'élément *Domain Process*. Ses concepts, décrits par les éléments de `Class`, sont par exemple *Class Data*, *Class Activity*. *Class Data* définit deux éléments de type `Method` *public Data(DataType type)* et *public void makePersistent()*.

L'interopérabilité Le processus de spécification des métamodèles est fait manuellement dans cet éditeur. Le modéleur est donc chargé de décrire les métamodèles en langage Codèle. En principe, ce procesus peut être automatisé. Si les métamodèles sont modélisés dans un langage de méta modélisation orienté objet comme MOF/UML ou Ecore, on peut les réutiliser dans cet éditeur après une phase de transformation de ces métamodèles en langage Codèle sans avoir besoin de les spécifier directement. Une transformation depuis un langage comme MOF/UML ou Ecore vers Codèle est possible, car la plupart des concepts décrivant la structure du modèle des langages de modélisation orientée objet sont similaires. On peut "mapper" les concepts de MOF/UML vers ceux de Codèle sans difficulté, par exemple `Package` vers `Domain` ; `Class` vers `Class` ; `Operation` vers `Method`. Ces transformations peuvent éventuellement être utilisées pour construire les chargeurs de réification automatique de métamodèles (voir la figure 7.2). Ceux ci ont la responsabilité de lire et de transformer les métamodèles sauvegardés dans différent méta langages vers le langage Codèle.

1. C'est-à-dire, les métamodèles sont représentés en terme des instances des concepts `Métamodel`, `Concept` et `Behavior` du langage Codèle.

2. Pour des raisons historiques, nous utilisons le concept de `Domain` au lieu du concept de `Metamodel`. De manière similaire, le concept `Class` remplace le concept `Concept` et `Method` remplace `Behavior`. Nous l'appelons le langage d'implémentation de Codèle. Mais ceci n'est qu'un problème de terminologie, il ne change pas la philosophie du langage.

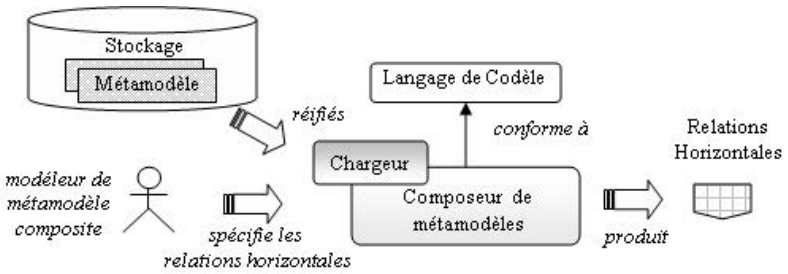


FIGURE 7.2 – Réification automatique des métamodèles par les chargeurs.

Cependant, dans nos expérimentations, nous nous limitons à la spécification des métamodèles directement en Codèle car nous n'avons pas encore développé le support de réification automatique des métamodèles composés. Ce point est considéré comme un aspect à améliorer dans les prochaines versions de Codèle mais ne correspond pas à un aspect critique.

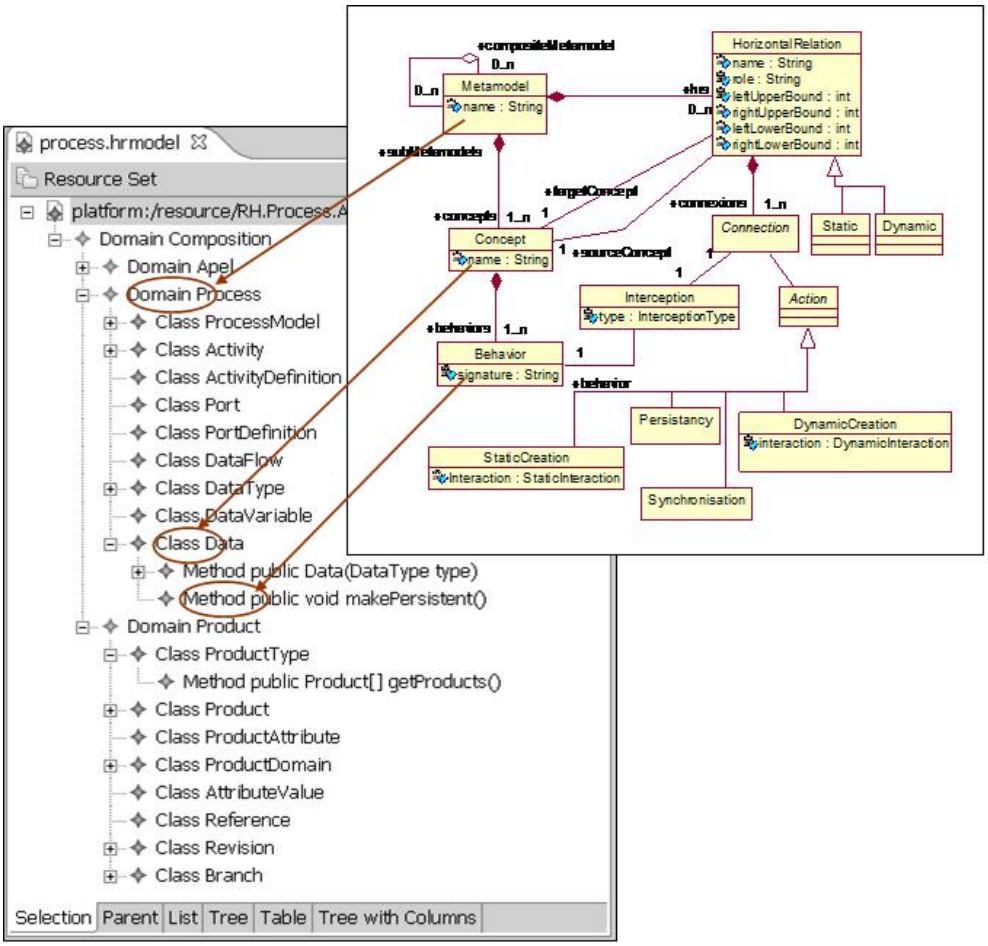


FIGURE 7.3 – Les métamodèles de Procédé et de Produit en langage Codèle.

B. Spécification des relations horizontales

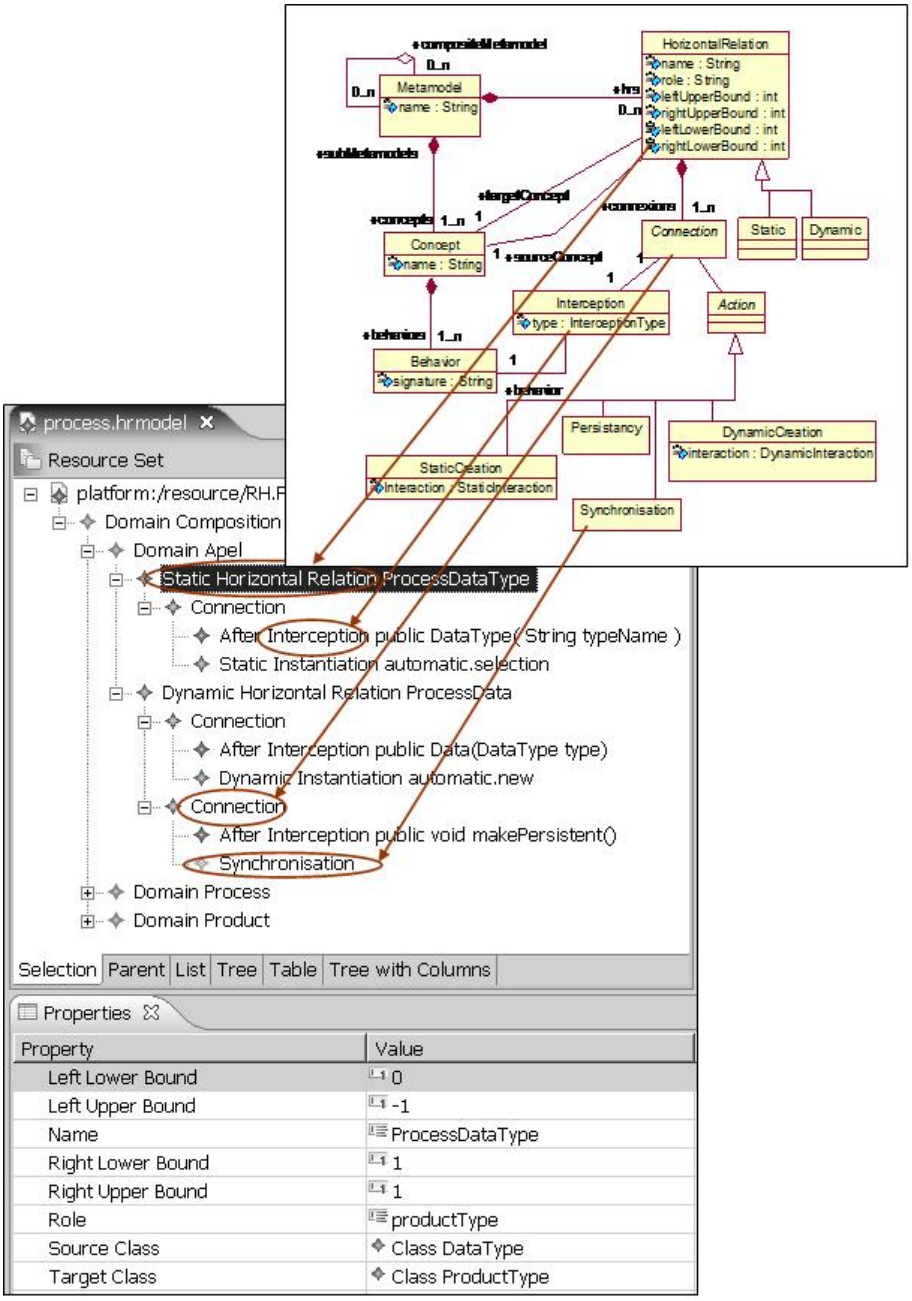


FIGURE 7.4 – Spécification des relations horizontales.

Après avoir spécifié la structure des métamodèles en Codèle, les relations horizontales peuvent être spécifiées en utilisant ces informations. La figure 7.4 montre deux relations horizontales créées dans notre exemple : une relation statique entre *DataType* et *ProductType* ; et une relation dynamique entre *Data* et *Product*. Elle montre une relation statique qui définit une interception de

la méthode *public DataType (String typeName)* et une action de création statique avec l'option d'interaction `automatic.selection`. Le panel *Properties* en bas exprime les autres caractéristiques de cette relation. Dans l'exemple, c'est une relation entre *DataType* et *ProductType*, sa classe source (`SourceClass`) est *DataType* et sa classe cible est *ProductType*. Nous affichons un fragment de métamodèle Codèle en haut pour montrer les concepts de Codèle utilisés dans cet éditeur.

7.1.2 Composeur de modèles

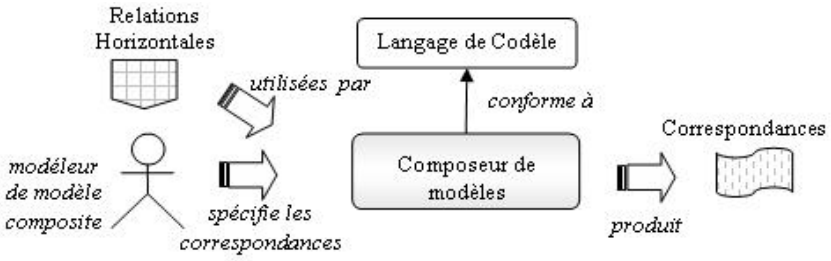


FIGURE 7.5 – Architecture générale du composeur de modèles.

Le composeur de modèles permet au modéleur de modèle composite d'établir les correspondances entre les éléments de ses modèles. Les correspondances doivent être conformes à leurs relations horizontales au niveau métamodèles. Ce composeur doit donc connaître les spécifications des relations horizontales pour établir les correspondances.

Nous avons développé un éditeur de composition de modèles dont l'interface est montrée dans la figure 7.6.

Comme on peut l'observer, notre composeur de modèles est constitué de 4 panels. Les deux panels à gauche liste les relations récupérées du fichier de spécification produit par le composeur de métamodèles ainsi que leurs détails. En fonction de la sélection d'une relation horizontale du modéleur, l'outil va afficher, dans les deux petits panels en haut à droite, les éléments des sous modèles que le modéleur veut composer. Ces derniers sont du type des concepts définis dans la relation sélectionnée.

Dans cette figure, le petit panel à gauche du bouton *Map* affiche les éléments du modèle source ; celui à droite du boutton *Map* affiche ceux du modèle cible. Dans cet exemple, on peut voir que le modèle source a deux éléments : *Default*, *Requirement* alors que le modèle cible a cinq éléments *Use Case Document*, *JML Specification*, *JavaFile*, *URL Bugzilla*, *Jar File*. Le bouton *Map* sert à créer une correspondance entre les éléments sources et cibles sélectionnés dans ces petits panels.

Enfin, le dernier panel en bas à droite est celui qui affiche toutes les correspondances établies. Dans l'image ci-dessus, une correspondance entre *Program* et *JavaFile* a été créée et ajoutée dans le tableau *Mappings*.

Application des règles de validation (well-formedness rules) Cet éditeur permet non seulement de créer les correspondances mais il réalise aussi quelques vérifications automatiquement. Par exemple, lors de la création d'une correspondance, notre outil va vérifier automatiquement la contrainte de cardinalité de la relation. Si le nombre de correspondances a atteint le seuil supérieur, les éléments restants vont être retirés de la liste des éléments du modèle. Ceci va empêcher la création de correspondances qui violeraient les contraintes de cardinalité.

Codèle vérifie également que le nombre de correspondances atteint le seuil inférieur, dans le cas contraire, le modèle composite est invalide. L'outil affiche un message d'erreur demandant de créer les correspondances nécessaires pour rendre valide le modèle composite. Ces messages sont affichés dans une vue nommée *Codele Log* que nous pouvons voir en bas de la figure 7.6 ci-dessus.

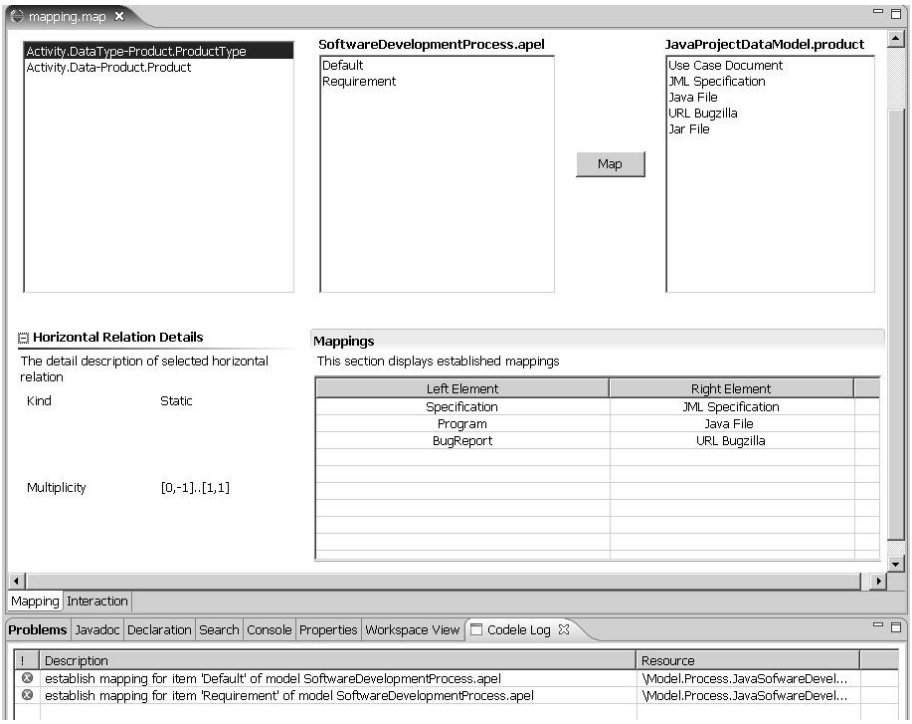


FIGURE 7.6 – Composeur de modèles.

7.1.3 Générateur

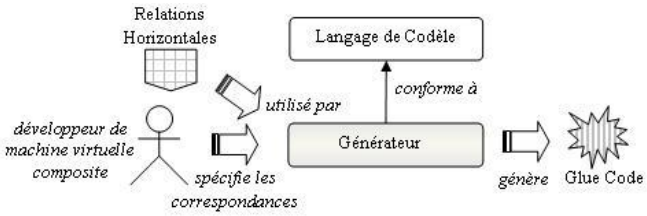


FIGURE 7.7 – Architecture générale du générateur.

Le générateur reçoit la spécification des relations horizontales et, à partir de là, génère le code Java correspondant. Le générateur va tout d'abord générer un projet Java représentant le métamodèle composite. Par exemple, pour notre domaine *Procédé_Produit*, un projet Java sera généré pour lui. Ce projet contiendra le codes qui composent les machines virtuelles de *Procédé* et de *Produit*. D'autre part, dans ce projet, le générateur va générer le code matérialisant les relations horizontales. Concrètement, pour chaque relation, nous générons :

- Un fichier AspectJ représentant la relation. Ce fichier contient le code pour toutes les connexions de cette relation. Une connexion est matérialisée par une capture AspectJ. Le corps de ces captures sont cependant délégués à des fichiers Java séparés selon le type de capture. Concrètement, ils sont regroupés dans trois fichiers générés pour la relation :
- Un fichier Java pour les connexions du type `Creation` de la relation,
- Un fichier Java pour les connexions du type `Synchronisation` de la relation, et
- Un fichier Java pour les connexions du type `Persistence` de la relation.

Les captures générées dans le fichier AspectJ ne contiennent que des appels de méthodes contenues dans les fichiers Java. Nous allons détailler ces quatre fichiers dans les sections suivantes.

A. Génération des captures : Fichier AspectJ

Le contenu de ce fichier comprend :

- Une déclaration pour la destination de la relation.
- Une liste de captures (une capture par connexion).
- Les méthodes d'accès *get/set* pour la destination déclarée.

Ces contenus sont obligatoires, générés, et complets (prêts pour l'exécution). L'utilisateur n'a pas besoin de le modifier. Le patron de ce fichier est décris dans la figure 7.8.

- **Ligne 1** : la déclaration de l'aspect. *RelationName* correspond au nom de la relation. Dans notre exemple, c'est *Procédé_Produit*. Le nom de l'aspect est dérivé de ce nom.
- **Ligne 3** : la déclaration de la destination de la relation. Dans notre exemple, elle est du type de produit *productType*.
- **Lignes 6-15** : les captures pour les connexions de création. Les lignes 13 et 14 sont le corps de la capture. Ici, nous gérérons simplement un appel de méthode vers le code chargé de créer le lien. Ce code se trouve dans le fichier Java généré pour les connexions de création. A titre d'exemple, la capture du constructeur *Data (String idData, DataType type)* sera générée de la façon suivante :

```
after  (Data source, String idData, DataType type)
: target(source)
&& args(idData, type) && execution (public Data.new (String, DataType))
{
   Data_ProductLinkCreation.createLink_new(source, idData, type);
}
```

- **Lignes 18-27** : le patron pour les captures de synchronisation. Nous générons ces captures de la même manière que les captures de création. Leur corps est un appel de méthode vers une classe de synchronisation regroupant l'ensemble du code pour les connexions de type synchronisation.
- **Lignes 30-39** : le patron pour les captures de persistance. Nous générons de la même façon les captures de persistances. Une classe de persistance contient l'ensemble du code pour ce type de capture.
- **Lignes 42-48** : le patron pour les méthodes d'accès *get/set*.

```
01 public aspect <%=RelName%>{
02   // declaration for destination.
03   private <%=DestConcept%> <%=SrcConcept%>.<%=dest%> = null ;
04
05
06   // for each creation connexion, we generate this captures
07   after (<%=SrcConcept%> source, <%=CreationMethodParams%>)
08   : target(source)
09   && args(<%=NamesOfCreationMethodParams%>)
10   && execution(<%=ModifiersOfCreationMethod%>
11                <%=SrcConcept%>.<%=CreationMethodName%>(
12                <%=TypesOfCreationMethodParams%>)) {
13     <%=RelName%>LinkCreation.createLink_<%=CreationMethodName%>
14                (source, <%=NamesOfCreationMethodParams%>);
15   }
16
17
18   // for each synchronisation connexion, we generate this capture
19   <%=CaptureKind%> (<%=SrcConcept%> source, <%=SynchroMethodParams%>)
20   : target(source)
21   && args(<%=NamesOfSynchroMethodParams%>)
22   && execution(<%=MofidiersOfSynchroMethod%>
23       <%=SrcConcept%>.<%=SynchroMethodName%>(
24       <%=TypesOfSynchroMethodParams%>)){
25      <%=RelName%>LinkSynchronisation.<%=SynchroMethodName%>_synch
26                 (source, <%=NamesOfSynchroMethodParams%>);
27   }
28
29
30   // for each persistency connexion, we generate this capture
31   <%=CaptureKind%> (<%=SrcConcept%> source)
32   : target(source)
33   && args(<%=NamesOfPersistencyMethodParams%>)
34   && execution(<%=MofidiersOfPersistencyMethod%>
35                <%=SrcConcept%>.<%=PersistencyMethodName%>(
36                <%=TypesOfPersistencyMethodParameters%>)){
37      <%=RelName%>LinkPersistency.<%=PersistencyMethodName%>_persist
38 (source, <%=NamesOfPersistencyMethodParams%>);
39   }
40
41
42    // getter/setter
43    public <%=DestConcept%> <%=SrcConcept%>.get<%=dest%>(){
44     return this.<%=dest%>;
45    }
46    public void <%=SrcConcept%>.set<%=dest%>(<%=DestConcept%>  value){
47     this.<%=dest%>= value;
48    }
49}
```

FIGURE 7.8 – Patron du fichier AspectJ de la relation horizontale.

Dans la figure 7.9, nous présentons un exemple complet du fichier AspectJ pour la relation *Data_Product* généré à partir du patron de la figure 7.8.

```
public aspect Data_Product{
 // declaration for destination.
 private Product Data.product = null;

 after (Data source, String idData, DataType type)
 : target(source)
 && args(idData, type)
 && execution (public Data.new(String, DataType))
 {
    Data_ProductLinkCreation.createLink_new(source, idData, type);
 }

 after (Data source, String idData, DataType type)
 : target(source)
 && args(idData, type)
 && execution (public Data.makePersitent())
 {
    Data_ProductLinkPersistency.makePersitent_persist(source);
 }

 public Product getProduct(){
    return this.product;
 }

 public void Data.setProduct(Product value){
    this.product = value;
 }
}
```

FIGURE 7.9 – Exemple : Fichier Data_Product.aj

B. Génération de code de création : Fichier Java pour les connexions de création.

Dans 6.5.2 du chapitre 6, nous avons donné une définition de la connexion de création.

Le tableau 7.1 résume les diverses façons d'établir les correspondances qui correspondent à nos différentes connexions de création. Selon les options choisies, le code java généré pour ces connexions est différent.

Pour des raisons d'espace, nous ne pouvons pas expliquer toute la complexité du code généré. Nous choisissons de ne présenter seulement que le principe général de ce code. Le paragraphe suivant présente la sémantique de connexion de la relation statique avec l'option de création `Automatique_Selection`. Une description des connexions avec les autres options peut être consulté en annexe de ce document.

Type de la relation	**Statique**	**Dynamique**
Façons d'établir les correspondances	`Automatique_Selection`	`Automatic_New`
		`Automatique_Selection`
		`Automatique_Selection_New`
		`Interactif_Selection`
		`Interactif_Selection_New`

TABLE 7.1 – Façons d'établir les correspondances.

B.1. Génération de code de création pour la relation statique Ce code est chargé de réifier les correspondances statiques qui sont connues avant l'exécution. La figure 7.10 explique la procédure de réification d'une correspondance statique.

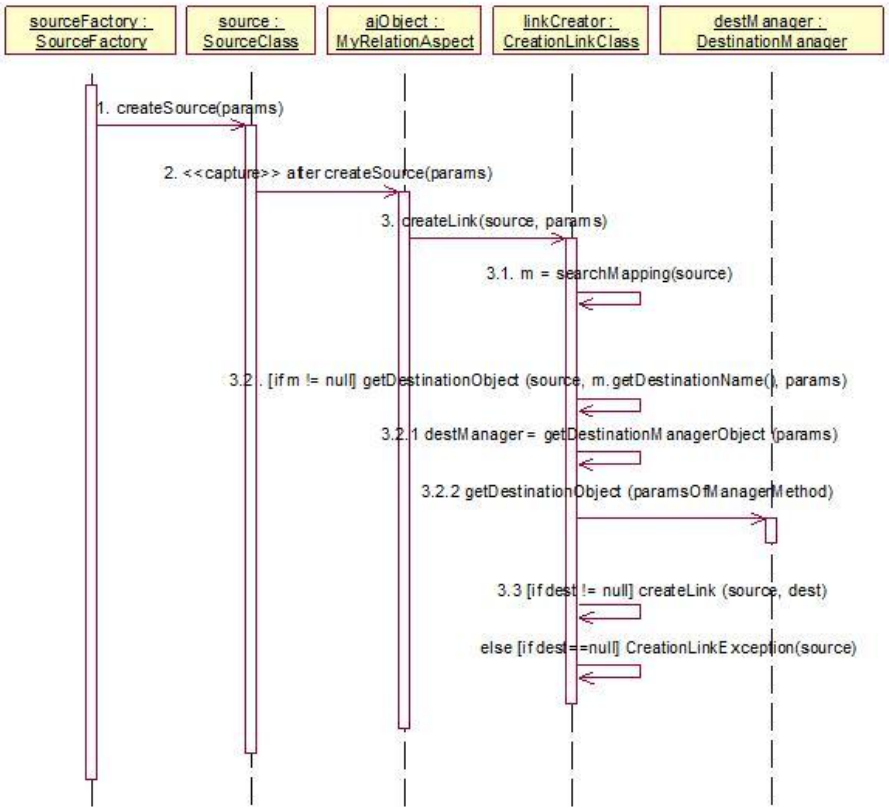


FIGURE 7.10 – Établissement d'une correspondance statique, option Automatique_Selection.

Nous pouvons résumer ses étapes en :

```
1. Demander la création de l'objet source.
2. Capturer la demande.
3. Demander de créer le lien.
   3.1 Lire le fichier de mapping et prendre la bonne correspondance
       de l'objet source.
   3.2 Chercher l'objet destination de la correspondance trouvée.
       3.2.1 Déterminer quel objet gère les objets destination.
       3.2.2 Déterminer quel est l'objet destination.
   3.3 Établir le lien.
```

Le code généré couvre environ 90% de la sémantique de l'opération ci-dessus ; le seul code qu'il est necessaire d'ajouter est celui pour la détermination de l'objet qui gère les destinations. Par exemple, dans le cas de création d'un lien entre une donnée avec un produit ; pour obtenir le produit, il faut interroger l'objet *productDomain* qui gère la liste des produits et qui fournit des méthodes pour rechercher un produit. Le code déterminant cet objet doit être ajouté à la main. D'après nos expérimentations, le volume du code ajouté reste cependant très petit.

B.2. Génération de code de création de la relation dynamique Cette génération varie selon les options de créations dynamiques choisies. Une description détaillée de cette génération se trouve en annexe.

C. Génération de code de persistance : Fichier Java pour les connexions de persistance.

Le code de persistance prend en charge la sauvegarde des correspondances dynamiques crées pendant l'exécution. La figure 7.11 montre le patron de ce code.

```
01 public class <%=RelationName%>LinkPersistency{
02
03   /**
04   * @generated
05   */
06 private static void registerMapping(String <%=sourceDomainName%>,
07 String <%=targetDomainName%>,String <%=sourceConceptName%>,
08 String <%=targetConceptName%>, String source, String dest){
09
10 //read mapping file
11 java.io.File directory = FindModel.find(<%=CompositionDomainName%>);
12 String mappingFilePath = directory.getAbsolutePath()
13                          + \\model\\mapping.map";
14
15    java.io.File mappingFile = new java.io.File(mappingFilePath);
16 MappingManager manager = new MappingManager(mappingFile);
17
18 //create mapping
19 manager.createMapping(<%=sourceDomainName%>, <%=targetDomainName%>,
20 <%=sourceConceptName%>, <%=targetConceptName%>, source, dest);
21
22 //save mapping
23 try {
24      manager.save();
25    }catch (IOException e) {
26     // TODO Auto-generated catch block
27      e.printStackTrace();
28    }
29 }
30}
```

FIGURE 7.11 – Patron du code de persistance.

Il possède une seule méthode *registerMapping* qui sauvegarde les correspondances. Le moment où la sauvegarde est effectuée n'est pas déterminé grâce à ce code. Cette méthode doit être appelée en dehors de cette classe. En général, cet appel est fait dans la classe chargée de la création de correspondances juste après la création d'une correspondance. Nous avons séparé la fonction de persistance dans le but qu'elle soient partagée par plusieurs codes de création. Le nom des domaines et des concepts source et cible sont passés en paramètres (lignes 6-8). Toutes les correspondances de toutes les relations sont sauvegardées au même endroit (ligne 11-13). La figure 7.12 ci-dessous est un exemple d'usage de cette classe dans la classe *Data_ ProductLinkCreation*. On sauvegarde la correspondance entre une donnée et un produit après sa création (ligne 12-13).

```
01 public class Data_ProductLinkCreation{
02
03 /**
04  * @generated
05  */
06  static void createLink_New(Data data, String idData, DataType type){
07   try{
08    Product product = createProduct_New(data, idData, type);
09    if(product != null){
10       data.setProduct (product);
11       // TODO : code inserted when the destination is found.
12       registerMapping("Process", "Product", "Data","Revision",
13                                                 data, product);
14   }else{
15      // TODO : code inserted when the destination is not found.
16      throw new CreateLinkException("Can not create link for " + data) ;
17   }catch(CreateLinkException e){
18      // TODO : code ajouté pour traiter l'exception.
19   }finally{
20      // TODO : code ajouté pour finir la création du lien.
21   }
22 }
```

FIGURE 7.12 – Persistance des correspondances de *Product_Procédé*.

D. Génération de code de synchronisation : Fichier Java pour les connexions de synchronisation

Le code de synchronisation est généré dans une classe Java qui contient une méthode par capture de synchronisation. Chaque méthode exprime le comportement d'une synchronisation. Ce comportement correspond à la partie sémantique que la spécification des relations horizontales ne fournit pas au générateur. Ce dernier est donc incapable de la générer, c'est au développeur de la machine virtuelle composite de le compléter.

Nous générons un squelette de cette classe avec des commentaires *TODO* indiquant les endroits où l'utilisateur doit ajouter du code. Le patron de cette classe est décris dans la figure 7.13

```
public class <%=RelationName%>LinkSynchronisation{

  // for each synchronisation connexion, we generate this method
  /**
   * @generated
   */
   static void <%=SynchroMethodName%>_synch(<%=SrcConcept%> source){
     // TODO: synchronasation code must be implemented here

   }
}
```

FIGURE 7.13 – Patron du code de synchronisation.

7.1.4 Exécution des correspondances

Cette section présente le résultat de l'exécution du fichier de correspondances défini dans la section 7.1.2 et du code généré dans la section 7.1.3. Tous les travaux réalisés dans les sections précédentes visent un seul but : l'exécution des correspondances, ou plutot, l'exécution du modèle composite.

La figure 7.14 est un aperçu de l'outil Agenda développé pour le *Domaine de Procédé* qui exécute les modèles de ce domaine. Nous retrouvons le modèle *SoftwareDevelopmentProcess* de notre exemple.

Dans cette démonstration, nous réalisons le scénario dans lequel une donnée de ce procédé est demandée, l'application va automatiquement chercher le *produit* correspondant dans le domaine de *Produit* et établir une lien entre eux. Le produit est l'état persité de cette donnée et le domaine de *Produit* joue le rôle d'une base de données dans cette composition. Si le produit correspondant est introuvable, l'application va le créer et le sauvegarder dans la base de donnée. Ce scénario montre une collaboration entre différents domaines.

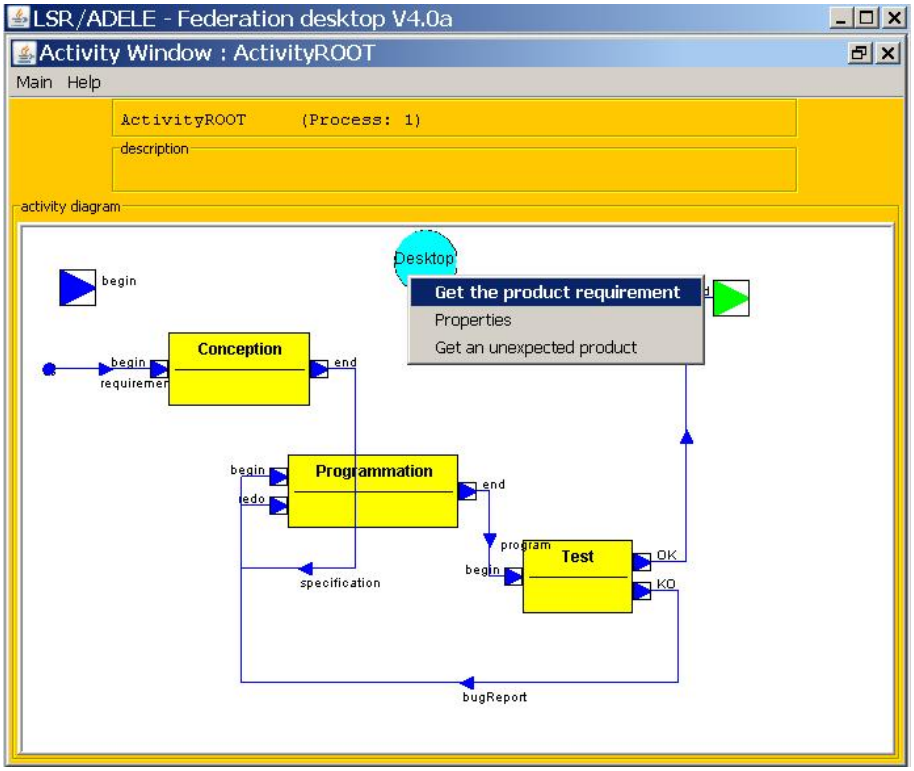


FIGURE 7.14 – Exécution du procédé SoftwareDevelopmentProcess - étape 1.

Nous observons dans la figure 7.14 qu'un menu demande la donnée *besoin* du port *desktop*. Après la sélection, trois activités vont être effectuées successivement :

- Du côté du domaine de *Procédé*, une donnée est créée. L'information de cette donnée est affichée dans un tableau en bas du procédé en cours d'exécution (voir la Figure 7.15). La première colonne du tableau correspond au numéro d'ordre de la donnée, la deuxième colonne affiche le nom de la donnée, la troisième colonne affiche l'identificateur, et la dernière colonne représente le type de la donnée. Dans cet exemple, le type de la donnée *besoin* est

Requirement.

- Du côté du domaine de *Produit*, un produit est créé automatiquement. Nous ne pouvons pas cependant montrer cette activité car elle est efféctuée à l'intérieur de la base de donnée.
- Du côté du domaine composite *Procédé_Produit*, le lien entre la donnée et le produit est établi. Cette troisième activité nous intéresse car l'exécution du modèle composite ne réussit que si les liens entre les sous modèles sont bien établis. Dans cet exemple, nous avons implémenté une petite fênêtre *Property* sur la donnée (voir la Figure 7.15) qui affiche les informations concernant la donnée dans le système (voir la Figure 7.16).

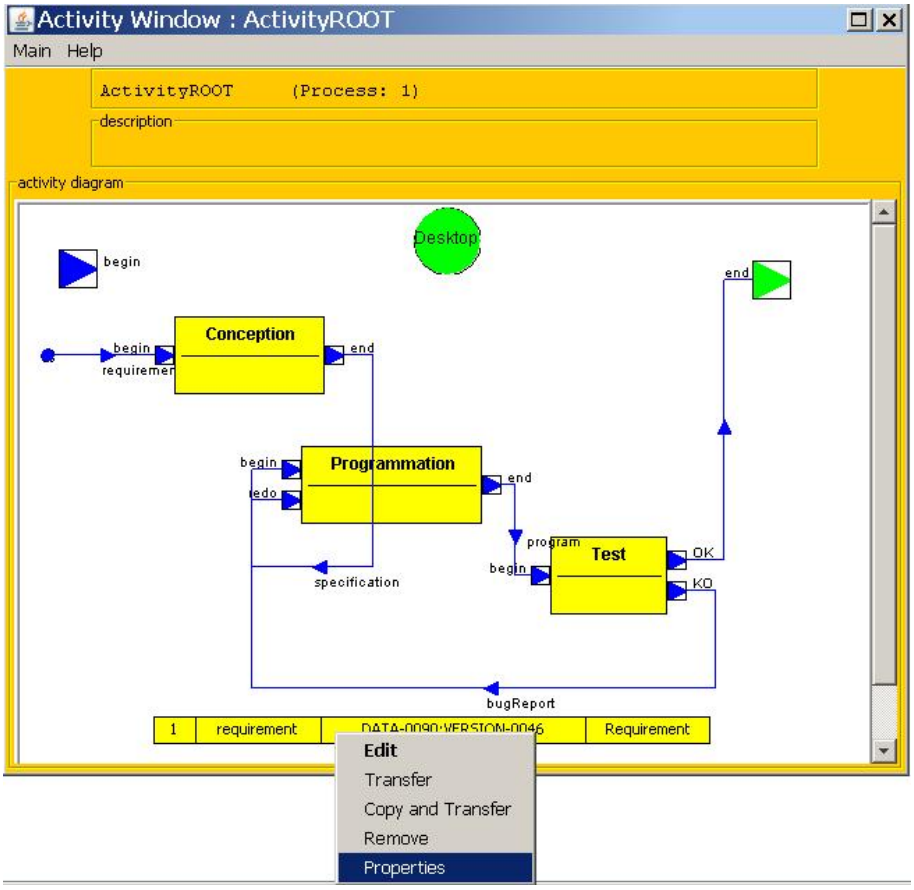


FIGURE 7.15 – Exécution du procédé SoftwareDevelopmentProcess - étape 2.

Dans la fenêtre *Notification* de la figure 7.16, nous pouvons voir qu'un produit a été créé dans le domaine de *Produit* avec le même id *DATA_0090* que la donnée. Le lien entre la donnée et le produit existe probablement.

De plus, le type du produit est *UseCaseDocument* alors que le type de la donnée est *Requirement*. Nous observons donc que les correspondances statiques entre les types ont été bien établies lors de l'exécution.

Lorsque les liens sont établis, les domaines semblent fonctionner indépendamment mais une collaboration est assurée par ces liens.

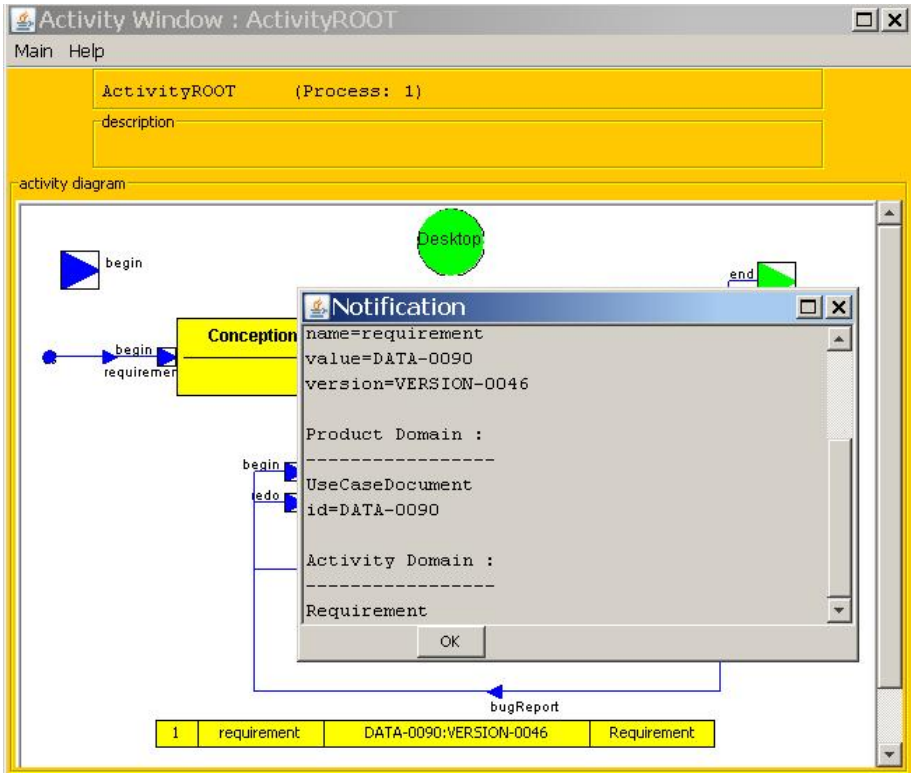


FIGURE 7.16 – Exécution du procédé SoftwareDevelopmentProcess - étape 3.

7.1.5 Synthèse sur l'outil

Dans la première partie du chapitre, nous avons présenté notre outil Codèle. Il comporte trois modules : un composeur de métamodèles, un composeur de modèles et un générateur. L'exécution du modèle composite a aussi été présentée. Nous avons montré comment les correspondances sont exécutées et assurent la collaboration entre les modèles lors de l'exécution.

7.2 Codèle : Mise en ouvre

Codèle est développé au dessus de la plateforme Eclipse. Chaque module de Codèle est un plugin.

7.2.1 Composeur de métamodèles

Le composeur de métamodèles a été généré grace à la technologie Ecore/EMF [3] qui fournit un éditeur dans lequel nous pouvons spécifier le modèle de notre composeur (i.e. le métamodèle de Codèle) en langage Ecore (un langage simplifié du MOF) (voir la figure 7.17). Ensuite, grâce au générateur de Ecore/EMF, nous pouvons générer l'éditeur de composition de modèles dont l'interface a été montrée dans la figure 7.3.

3. Il s'agit d'un framework de modélisation et de génération de code intégré à Eclipse permettant construire des applications à partir de modèles de données (*data model*).

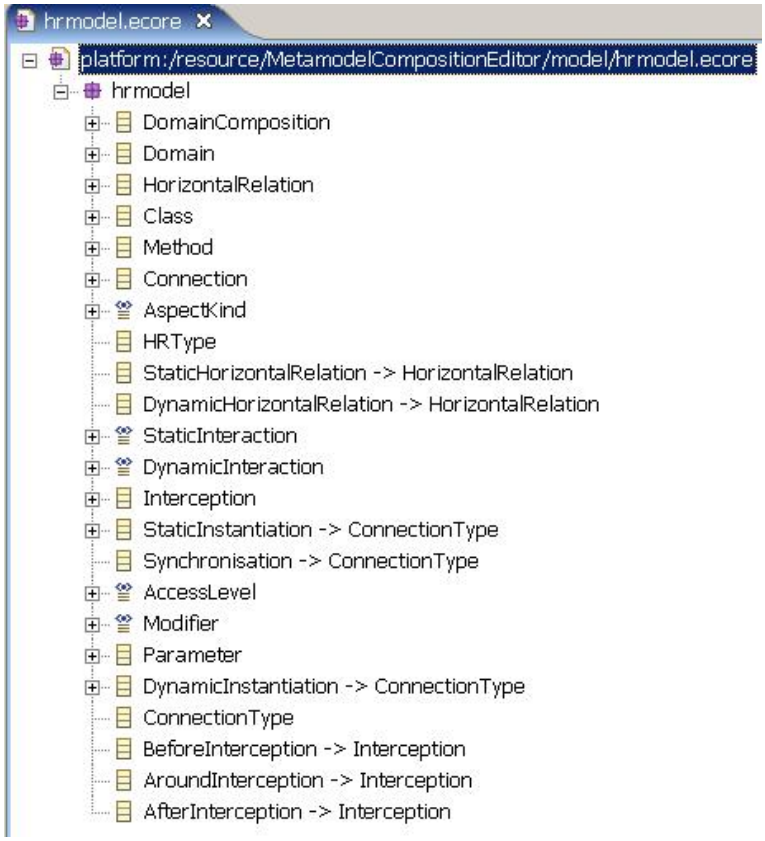


FIGURE 7.17 – Métamodèle de Codèle en Ecore.

La technologie Ecore/EMF nous a grandement facilité l'implémentation de ce module. Puisque cette technologie possède une approche générative très puissante, nous avons réduis de manière très significative notre temps de développement. La maintenance est elle aussi simplifiée car chaque modification dans le modèle peut mettre à jour immédiatement le code en effectuant une nouvelle génération.

Au sujet de la performance, l'éditeur généré n'est pas très sophistiqué mais il est quand même capable de remplacer l'éditeur construit par un développement classique. Dans notre approche, l'idée la plus appréciée des utilisateurs est de pouvoir bénéficier au maximum de l'outillage disponible.

7.2.2 Composeur de modèles

Le composeur de modèles, réalisé en Java, constitue environ 2000 lignes de code. Actuellement, sa version est 1.0. Son interface gaphique a été montrée dans la figure 7.6. Elle a été construite en utilisant la librairie SWT/JFace d'Eclipse.

Le composeur de modèles a besoin des relations horizontales et des deux modèles source. Dans cette version, pour que le module puisse les retrouver, nous avons utilisé certaines conventions sur les répertoires.

En ce qui concerne les relations horizontales, nous avons placé le fichier qui les contient dans un sous répertoire du plugin du composeur. Lorsque nous déployons ce plugin, il faut définir

ce fichier comme une ressource déployée du plugin. En d'autre termes, chaque composition de domaines a sa propre version du composeur.

Le deuxième point concerne la recherche des modèles source. Dans notre cas, nous supposons que chaque modèle a un projet dont le nom est nommé conventionellement et le composeur peut calculer ces noms et obtenir les fichiers dont il a besoin dans ces projets.

Les correspondances créées par le composeur sont sauvégardées dans un fichier .map placé dans le répertoire du modèle composite.

7.2.3 Générateur

Le générateur se charge de transformer les relations horizontales en code Java. Dans notre cas, le générateur est développé en utilisant la technologie JET [4].

Le moteur JET est un outil de génération de code puissant et efficace fourni par le framework EMF. Il peut être utilisé pour générer du code Java, XML, SQL ou tout autre format en se basant sur des patrons.

Les patrons JET sont écrits dans un langage dont la syntaxe est similaire à JSP [5]. Par défaut, un patron JET est un fichier texte dont le nom est terminé par le suffixe "jet". Le moteur JET prend ce fichier puis le transforme en une classe java [6]. Ensuite, le modèle de données source [7] doit être réifié également en terme d'objets Java. Ces objets sont passés comme paramètres à la classe d'implémentation du patron. Ces paramètres fournissent les informations necessaires pour remplacer les références dans le patron. Enfin, le compilateur Java va exécuter la méthode *generate()* de la classe d'implémentation pour générer le code cible. Ce processus est schématisé dans la figure 7.18.

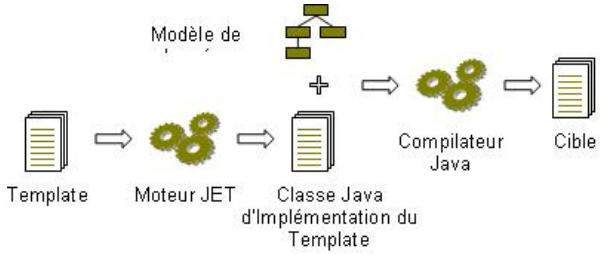


FIGURE 7.18 – Génération de code par les patrons JET.

Notre générateur contient un ensemble de patrons JET et les classes d'implémentation associées. Actuellement, nous utilisons 10 patrons différents. Le code de ce module avoisine les 4800 lignes dans lesquelles la plupart sont générées par le moteur JET, seul 500 lignes ont été écrites à la main.

7.3 Codèle : Validation

J'ai réalisé l'outil Codèle contenant les trois modules décris précédement. À ce jour, il est en version 1.0. Notre outil peut fonctionner sous deux optiques :

4. Java Emitter Templates
5. JavaServer Pages.
6. Cette classe est une représentation de ce patron en langage Java.
7. Le modèle de données (*data model*) source correspond aux paramètres d'entrées de la génération qui organise les données dans une structure spécifique qui doit être transformée.

- Dans un contexte MDE en général, indépendant du contexte de notre approche.
- En suivant notre approche, en supportant les environnements déjà développés dans l'équipe.

En ce qui concerne la première optique, le composeur de modèles de Codèle peut être utilisé comme un outil de tissage permettant d'établir des liens de manière similaire à Atlas Model Weaver[**?**]. Notre fichier de correspondances rentre dans la même catégorie que le modèle de tissage AMW. Il peut être aussi utilisé dans le but de traçer l'évolution, de comparer, de traduire ou encore de tranformer des modèles.

En ce qui concerne le deuxième optique, le composeur de modèles est aujourd'hui utilisé par l'environement FOCAS [PE08]. C'est une extension à l'application de domaine *Procédé_Produit* que nous avons pris comme exemple dans cette thèse. Cet environnement est en cours de developpement dans le cadre d'une autre thèse de notre équipe. Au moment de la rédaction de ce document, FOCAS a été utilisé pour le developpement d'un système d'alarme pour la société Thales [PE08].

Un autre usage de Codèle est l'intégration de ses modules dans l'environnement Mélusine présentée dans le chapitre 5 afin de faciliter la conception des architectures de famille d'applications basées sur des briques architecturales de métamodèles et de modèles. Dans cet environnement, les composeurs de métamodèles et le générateur de Codèle peuvent assister le développeur du domain pour construire un domaine composite. Cependant, au moment de la rédaction de cette thèse, ce travail n'a pas encore été réalisé.

Il reste encore des travaux à faire pour consolider Codèle. Les points suivants devraient être améliorés :

- Le composeur de métamodèles ne peut pas importer automatiquement les métamodèles sources. Ces derniers doivent être réifiés à la main par l'utilisateur. Il serait souhaitable d'avoir une solution plus automatisée.
- Le composeur de modèles n'importe pas explicitement le fichier des relations horizontales. Ce fichier est déployé implicitement dans le composeur, ce qui rend ce module trop spécifique. Une solution à base de paramètrage permetantt de configurer le module serait plus adéquate.
- Le générateur ne génère que le code pour des relations simples avec la cardinalité 1..1. En pratique, Il existe des cas plus complexes avec des cardinalités 1..*, *..1 ou *..* etc. qu'il faudrait étudier.
- Le langage Codèle peut aussi être amélioré sur certains aspects. Par exemple, les relations horizontales pourraient être associées aux expressions de correspondances qui exprime la condition d'établissement de la correspondance au lieu de se baser uniquement sur le critère de nom comme dans la version actuelle.

7.4 Conclusion

Dans ce chapitre, nous avons présenté nos expérimentations de l'implémentation du langage Codèle. Ces expérimentations ont montré que nous avons réussi à réaliser conceptuellement et pratiquement notre approche de composition de modèles exécutables. Nous disposons d'outils supportant ce processus et les applications réalisées constituent un résultat intéressant.

Notre outil est suffisament générique pour supporter de nombreux cas de compositions. Il est simple et assez efficace. Malgré les nombreuses améliorations qui restent à faire, ce prototype satisfait nos besoins.

Chapitre 8

Conclusion

8.1 Synthèse de notre approche

Cette thèse propose une approche de composition de modèles par relation. Au travers de ce travail, nous séparons la composition en général et la composition de modèles en particulier en deux types : dans le premier, les entités à composer sont modifiées, et dans l'autre non. La proposition dans cette thèse appartient au second type.

Dans notre approche on associe le modèle, son métamodèle et la machine virtuelle qui l'exécute. La composition de tels modèles est donc intimement liée à la composition de leurs métamodèles et à la composition de leurs machines virtuelles.

Du point de vue conceptuel, la composition de métamodèles concerne le problème de la composition des langages ce qui exige de prendre en compte les concepts établis des langages spécifiques des domaines (les DSLs) afin de choisir la solution de composition la plus appropriée. La composition des métamodèles est une étape clef, car c'est en basant sur le résultat de cette composition que la composition aux autres niveaux seront réalisées. Mais composer des métamodèles n'est pas toujours facile à faire ; cela requiert de bien comprendre les langages à composer aussi les exigences vis-à-vis du langage composite produit. Nous avons mentionné dans cette thèse certains mécanismes permettant de composer les métamodèles, par exemple la fusion de classes, l'héritage des classes, la collaboration des classes par associations etc. Dans notre approche, nous proposons un type de relation appelé Relation Horizontale. Nous avons montré que ce mécanisme permet de conserver les métamodèles tel quel sans aucun modification lors de composition. Ceci est un critère très important pour la réutilisation des modèles, car avec un changement minime des métamodèles composés, il y a un risque fort d'être obligé de modifier ou même reconstruire toutes les machines virtuelles à composer.

La composition de modèles consiste à établir des correspondances, de façon similaire à ce qui est fait dans la fusion de modèles. Établir les correspondances peut être réalisé de façon automatique, semi automatique ou manuelle. Pour le moment, ceci est fait manuellement dans notre approche.

La composition de machines virtuelles repose sur le mécanisme AOP. Cette composition est de type "boîte blanche". La sémantique du code nécessaire pour faire collaborer les machines virtuelles varie selon la composition. De ce fait, le code exprimant la sémantique de la composition doit être réalisé manuellement ; par contre, les expériences réalisées jusqu'à présent ont montré qu'une partie importante, sinon principale, du code, concernant divers aspects non fonctionnels, comme la persistance, et la gestion du cycle de vie des liens peut être généré.

8.2 Contributions et résultats obtenus

Les apports principaux de cette thèse peuvent être résumés dans les grands points suivants : Le premier point est une étude des approches de Génie Logiciel bati autour des concepts de composition. Dans cet étude, il ressort que nous pouvons distinguer deux types de composition : celles qui préservent et celles qui transforment les entités à composer.

Le deuxième point est une étude des approches de composition de modèles. Dans cet étude, les deux leçons importantes que nous avons retenues sont : 1) Les règles de composition doivent être exprimées au niveau langage et 2) une composition particulière doit respecter ces règles de composition. Ceci a fondé un des principes de base de notre approche : pour faire la composition de modèles, il faut réaliser d'abord la composition de leurs métamodèles. De ce fait, les outils que nous proposons pour supporter ce processus sont au niveau du méta métamodèle commun aux deux métamodèles à composer.

Le troisième apport de la thèse est un langage de composition de modèles. Ce langage a été formalisé dans un métamodèle qui se divise en deux parties :

- une partie est liée à la composition des métamodèles.
- une partie est liée à la composition des modèles.

Ce langage contient les concepts principaux pour la composition de modèles. Pour réaliser une composition, les métamodèles et les modèles à composer doivent d'abord être formalisés. Pour ce faire, les concepts suivants ont été proposés :

- `Metamodel` : un métamodèle est représenté par le concept de Metamodele. Chaque métamodèle a un nom. Il peut être composite c'est à dire comporter plusieurs sous métamodèles.
- `Modele` : Un modèle a un nom. Il est lié à un métamodèle par la relation de conformité. Un modèle peut être composite.
- `Concept` : ce concept modélise les concepts formalisés dans un métamodèle.
- `Behavior` : les concepts définissent des comportements.
- `Element` : les modèles contiennent les éléments qui sont ldes instances des concepts.

Les métamodèles et les modèles sont composés par des relations. Les deux concepts suivants ont été proposés :

- `HorizontalRelation` : représente les liens entre les concepts.
- `Mapping` : représente les liens entre des éléments des modèles.

Notre langage peut être considéré comme un langage de spécification des captures AOP lorsque nous utilisons ce mécanisme comme le moyen de composer les machines virtuelles exécutant les modèles. Les concepts suivants ont été proposés pour ce but :

- `Connection` : généralise une capture AOP.
- `Interception` : définit la méthode intercepté par la capture.
- `Action` : définit la sémantique d'interactions entre deux concepts composés.

Notre métamodèle de composition abstrait les concepts spécifiques liés à un domaine particulier. Il est indépendant des domaines et des applications qu'il compose, ce qui lui permet de réaliser un grand nombre de compositions différentes.

La dernière contribution de cette thèse est un outil supportant le processus de composition. Nous avons construit un outil implémentant le langage proposé. Cet outil est constitue de trois modules permettant de réaliser la composition de métamodèles, de modèles et de machines virtuelles. Il est utilisé actuellement dans l'environnement FOCAS[PE08] - un environnement de modélisation et d'exécution des procédés - qui est développé dans notre équipe.

8.3 Perspectives

Les perspectives de cette thèse peuvent être divisées en deux axes : 1) le langage et 2) l'outil. En ce qui concerne le langage, nous avons identifié :

- Le manque d'un langage pour exprimer les contraintes de cohérence : Les composeurs de métamodèles et de modèles peuvent être associés à un interpréteur OCL pour modéliser et interpréter les contraintes de cohérences. Ces contraintes seraient modélisés au niveau des métamodèles par le composeur de métamodèles et interprétés au niveau modèle par le composeur de modèles pour effectuer un filtrage des éléments valides sur lesquels il sera possible d'établir des liens. Dans la version actuelle, le composeur de modèles de Codèle n'effectue aucun filtrage. Ce besoin pourrait ainsi faire apparaitre éventuellement un nouveau concept `ExpressionOcl` dans notre langage.
- Le manque d'un langage pour exprimer les critères pour l'établissement des liens : si nous voulons exprimer la manière de déterminer des objets à relier de façon déclarative, il faut avoir un langage. Les expressions OCL semblent être capables de se charger de cette tâche. Le composeur de métamodèles peut décrire ces critères en OCL mais évidemment il faut avoir un interpréteur OCL dans le composeur de modèles pour qu'il puisse les exécuter. Ceci n'est pas le cas pour le moment de la version de Codèle actuelle. Un nouveau concept MatchExpression peut être ajouté au langage comme un sous type du concept `ExpressionOcl`.
- La structure d'une capture doit être spécifiée de façon plus fine. Par exemple il faut être capable de modéliser des concepts sophistiqués de AspectJ tels que `cflow`.
- - Les types d'interaction entre les concepts devraient être étudiés de façon plus approfondie. En ce moment nous avons identifié 3 types : *création*, *synchronisation*, et *persistance*. Mais il est certain qu'il existe bien d'autres types. Plus de types précis seront identifiés, plus nous serons capable de générer du code automatiquement. Pour le moment, la génération de code de Codèle est partielle car le code exprimant la sémantique de la composition est manuel, mais la plupart du temps, environ 80% du code de synchronisation est généré.

En ce qui concerne l'outil, les points suivants devraient être améliorés :

- Il faudrait fournir des chargeurs automatiques des métamodèles.
- Il serait intéressant de disposer de transformateurs automatiques pour des métamodèles exprimés dans différents langages par exemple MOF/UML en Codèle.
- Le composeur de modèles n'importe pas explicitement le fichier des relations horizontales. Ce fichier est déployé implicitement dans le composeur, ce qui rend ce module spécifique.
- Le générateur ne génère que du code pour des relations simples avec la cardinalité 1..1. Il y a certainement des cas plus complexes avec les cardinalités 1..*, *..1 ou *..* etc. que nous ne savons pas gérer actuellement (automatiquement)

8.4 Conclusion

La composition de modèle proposé dans cette thèse permet de construire des applications complexes en réutilisant des modèles existants.

Le principe du mécanisme de composition proposé ici n'est pas nouveau. Il a adopté le principe de composition de boîte noire très basique dans les approches comme CBSE ; mais l'a appliqué au niveau modèle. De ce point de vue, le langage proposé peut être comparé aux ADLs utilisés en CBSE, sauf que son grain de composition est des modèles pas des composants.

L'utilisation des modèles devient le paradigme dominant dans le développement de logiciel ; ainsi nous pensons que les mécanismes de composition de modèles en réutilisant les implémentations associées comme ce qui est proposé dans cette thèse est une contribution appréciable au domaine de l'IDM.

Bibliographie

[AEB03] O. Aldawud, T. Elrad, and A. Bader, *UML Profile for Aspect-Oriented Software Development.*, Proceedings of Third International Workshop on Aspect-Oriented Modeling, Mars 2003.

[AL08] Sven Apel and Christian Lengauer, *Superimposition : A Language-Independent Approach to Software Composition*, Software Composition, 2008, pp. 20–35.

[All97] Robert Allen, *A Formal Approach to Software Architecture*, Ph.D. thesis, Carnegie Mellon, School of Computer Science, January 1997, Issued as CMU Technical Report CMU-CS-97-144.

[AVP07] D. H. Akehurst, R. Vogel, and R. F. Paige (eds.), Lecture Notes in Computer Science, vol. 4530, Springer, 2007.

[Bar05] O. Barais, *Construire et Maîtriser l'Évolution d'une Architecture Logicielle à base de Composants*, Ph.D. thesis, Université de Lille, Novembre 2005.

[BB01] E. Breton and J. Bézivin, *Towards an Understanding of Model Executability*, Proceedings of the International Conference on Formal Ontology in Information Systems (Ogunquit, Maine, USA), vol. 2001, 2001, pp. 70–80.

[BBB+] J. Bézivin, M. Blay, M. Bouzeghoub, J. Estublier, and J.-M. Favre, *Rapport de Synthèse : Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles.*, Centre National de la Recherche Scientifique CNRS.

[BBF+06] J. Bézivin, S. Bouzitouna, M. Didonet Del Fabro, M. P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R.C. Paige, *A canonical scheme for model composition*, Proc. European Conference in Model Driven Architecture (EC-MDA) 2006 (Bilbao, Spain), Juillet 2006.

[BCE+06] G. Brunetal, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh, *A Manifesto for Model Merging*, Proceedings of the 2006 international workshop on Global integrated model management (2006).

[BCR05] A. Boronat, J. A. Carsi, and I. Ramos, *Automatic Support for Traceability in a Generic Model Management Framework*, European Conference on Model Driven Architecture - Foundations and Applications (Nuremberg (Germany)) (Springer LNCS, ed.), vol. 3748, November 2005, pp. 316–330.

[Ber03] P.A. Bernstein, *Applying Model Management to Classical Meta Data Problems*, In Proceedings of the Conference on Innovative Database Research (CIDR) (Janvier 2003).

[BFCD+06] Mireille Blay-Fornarino, Pierre Combes, Laurence Duchien, Tristan Glatard, Philippe Lahire, Stéphane Lavirotte, Clémentine Nemo, Audrey Occello, Renaud Pawlak, Anne-Marie Pinna-Dery, Lionel Seinturier, and Jean-Yves Tigli, *État de lŠart sur la contractualisation et la composition*, Tech. report, Août 2006.

[BG01] Jean Bézivin and Olivier Gerbé, *Towards a Precise Definition of the OMG/MDA Framework*, Proceedings of the 16th Conference on Automated Software Engineering (San Diego, USA), IEEE Computer Society Press, Novembre 2001, pp. 273–280.

[BG04] S. Bouzitouna and M. P. Gervais, *Composition rules for PIM Reuse*, Proceedings of the 2nd European Workshop on MDA with Emphasis on Methodologies and Transformations (EWMDA-MT'04) (Canterbury, UK), September 2004.

[BGMR03] J. Bézivin, S. Gérard, P.-A. Muller, and L. Rioux, *MDA Components : Challenges and Opportunities*, Proceedings of Metamodelling for MDA (York, England), 2003.

[BJPV04] J. Bézivin, F. Jouault, and P. P. Valduriez, *First Experiments with a ModelWeaver*, Workshop on Best Practices for Model Driven Software Development held in conjunction with the 19 th Conference on Object-Oriented Programming, Systems, Languages, and Applications (2004).

[BMH06] Bill Burke and Richard Monson-Haefel, *Enterprise JavaBeans 3.0 (5th Edition)*, O'Reilly Media, Inc., 2006.

[Boo87] Grady Booch, *Software Component with ADA*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1987.

[Boo93] Grady Booch, *Object-Oriented Analysis and Design with Applications, 2nd Edition*, Redwood City : Benjamin Cummings. ISBN 0-8053-5340-2., 1993.

[Box97] D. Box, *Essential COM*, Addison-Wesley, 1997.

[BSL01] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux, *Le point sur la Programmation par Aspects*, Technique et Sciences Informatiques **20** (2001), no. 4.

[BSM+03] F. Budinsky, D. Steingerg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework : A Developer's Guide*, Addison Wesley, 2003.

[Béz05] J. Bézivin, *On the Unification Power of Models*, on Software and Systems Modeling **4** (2005), no. 2, 171–188.

[Cer04] H. Cervantes, *Vers un Modèle à Composants Orienté Services pour Supporter la Disponibilité Dynamique*, Ph.D. thesis, Université Joseph Fourier, Mars 2004.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans, *Applied Metamodelling A Foundation for Language Driven Development*, Août 2004.

[Cla02] S. Clarke, *Extending Standard UML with Model Composition Semantics*, Science of Computer Programming 44 **44** (2002), no. 1, 71–100.

[CS06] Thierry Coupaye and Jean-Bernard Stefani, *Fractal Component-Based Software Engineering*, ECOOP Workshops, 2006, pp. 117–129.

[Dav] J. Davis, *GME : The Generic Modeling Environment*, Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '03) (Anaheim, CA, USA), pp. 82 – 83.

[DDFBJ+05] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas, *AMW : A Generic Model Weaver*, Proceedings of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles, IDM'05, 2005.

[DDFBV06] M. Didonet Del Fabro, J. Bézivin, and P. Valduriez, *Weaving Models with the Eclipse AMW plugin*, In : Eclipse Modeling Symposium, Eclipse Summit Europe 2006 (Esslingen, Germany), 2006.

[DDFJ05] M. Didonet Del Fabro and F. Jouault, *Model Transformation and Weaving in the AMMA Platform*, Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE) (Braga, Portugal), 2005, pp. 71–77.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt, *A Framework for the Detection and Resolution of Aspect Interactions*, Proceedings of the 1st ACM SIGPLAN/-SIGSOFT Conference on Generative Programming and Component Engineering (GPCE), Octobre 2002, pp. 173–188.

[DK76] F. DeRemer and H. Kron, *Programming-in-the-Large versus Programming-in-the-Small*, IEEE Transactions on Software Engineering **2** (Juin 1976), no. 2, 80–87.

[DLBS] Pierre-Charles David, Thomas Ledoux, and Noury M. N. Bouraqadi-Saâdani, *Two-Step Weaving with Reflection using AspectJ*, Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001), year = 2001.

[DSB+05] Pascal Durr, Tom Staijen, Lodewijk Bergmans, Aksit, and Mehmet, *Reasoning About Semantic Conflicts Between Aspects*, 2nd European Interactive Workshop on Aspects in Software (EIWAS) (Brussels, Belgium), Septembre 2005.

[EDA98] J. Estublier, S. Dami, and M. Amiour, *APEL : A Graphical yet Executable Formalism for Process Modeling*, Automated Software Engineering, ASE journal. **5** (1998), no. 1.

[ELV03] J. Estublier, A-T LE, and J. Villalobos, *Using Federations for Flexible SCM Systems*, SCM-11 (Mai 2003).

[EV05] Jacky Estublier and German Vega, *Reuse and Variability in Large Software Applications*, ESEC/FSE-13 : Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA), ACM, 2005, pp. 316–325.

[EVC01] J. Estublier, H. Verjus, and P-Y. Cunin, *Building Software Federations*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (2001).

[EVI05] J. Estublier, G. Vega, and A. Ionita, *Composing Domain-Specific Languages for Wide-scope Software Engineering Applications*, MoDELS/UML (2005).

[Fav04a] J.-M. Favre, *Foundations of Meta-Pyramids : Languages vs. Metamodels - Episode II : Story of Thotus the Baboon1*, Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development (Dagsthul, Germany), 2004, pp. 70 – 80.

[Fav04b] _____, *Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I, Stories of the Fidus Papyrus and of the Solarus*, Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development (Dagsthul, Germany), 2004.

[FBFG07] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh, *A Generic Approach For Automatic Model Composition*, Aspect Oriented Modeling (AOM) Workshop (Octobre 2007).

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, *Concepts de base de l'IDM : Modèle, métamodèle, transformation, mégamodèle*, Edition Hermes, 2006.

[FRF+] Franck Fleurey, Raghu Reddy, Robert France, Benoit Baudry, and Sudipto Ghosh, *Kompose : Generic Model Composition Tool.*

[GAO94] David Garlan, Robert Allen, and John Ockerbloom, *Exploiting style in architectural design environments*, Proceedings of SIGSOFT'94 : The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, Décembre 1994.

[Gar95] D. Garlan, *An introduction to the aesop system*, Tech. report, July 1995.

[GLF04] Gonzalo Génova, Juan Llorens, and José Miguel Fuentes, *UML Associations : A Structural and Contextual View*, Journal of Object Technology **3** (2004), no. 7, 83–100.

[GME] *GME 5 Usersś Manual*, document en ligne, disponible à http://www.isis.vanderbilt.edu/projects/gme/GMEUMan.pdf.

[GMW00] David Garlan, Robert T. Monroe, and David Wile, *Acme : Architectural Description of Component-Based Systems*, p. 47, Cambridge University Press, 2000.

[Gro06] Object Management Group, *Meta Object Facility (MOF) Core Specification. version 2.0.*, document en ligne, disponible à http://www.omg.org/spec/MOF/2.0/HTML/, Janvier 2006.

[HH85] C. A. R. Hoare and C. A. R. Hoare, *Communicating Sequential Processes*, Communications of the ACM **21** (1985), 666–677.

[JK05] F. Jouault and I. Kurtev, *Transforming Models with ATL*, Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005 (Montego Bay, Jamaica), Avril 2005.

[JZM08] Ke Jiang, Lei Zhang, and Shigeru Miyake, *Using OCL in Executable UML*, Electronic Communications of the EASST- ECEASST, vol. 9, 2008.

[KBA02] I. Kurtev, J. Bézivin, and M. Aksit, *Technological Spaces : An Initial Appraisal*, Proceedings of the Confederated International Conferences CoopIS, DOA, and ODBASE 2002, Industrial track (Irvine, CA, USA), 2002.

[KDDF06] I. Kurtev and M. Didonet Del Fabro, *A DSL for Definition of Model Composition Operators*, Models and Aspects Workshop at ECOOP (Nantes, France), Juillet 2006.

[Küh06] T. Kühne, *Matters of (Meta-) Modeling*, Software and Systems Modeling (SoSyM), Springer **5** (2006), 369–385.

[KLM+] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingier, and J. Irwin, *Aspect Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer Verlag LNCS 1241.

[KML+04] Gabor Karsai, Miklos Maroti, Akos Ledeczi, Jeff Gray, and Janos Sztipanovits, *Composition and Cloning in Modeling and Meta-Modeling*, IEEE Transactions on Control Systems Technology **12** (Mars 2004), no. 2, 263–278.

[KPP06a] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack, *Model Comparison : A Foundation for Model Composition and Model Transformation Testing*, GaMMa 2006, 1st International Workshop on Global Integrated Model Management (2006).

[KPP06b] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack., *Eclipse Development Tools for Epsilon*, Eclipse Summit Europe, Eclipse Modeling Symposium (Esslingen, Germany), October 2006.

[KPP06c] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack, *Merging Models with the Epsilon Merging Language (EML)*, Proceedings ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006) (Genova, Italy), vol. LNCS, October 2006.

[KTG+06] I. Krechetov, B. Tekinerdogan, A. Garcia, C. Chavez, and U. Kulesza, *Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design*, 8th International Workshop on Aspect-Oriented Modeling, AOSD 2006 (Bonn, Germany), 2006.

[Le04] A.T. Le, *Fédération : une Architecture Logicielle pour la Construction dŠApplications Dirigée par les Modèles*, Ph.D. thesis, Université Joseph Fourier, Janvier 2004.

[LEV03] A-T LE, J. Estublier, and J. Villalobos, *Multi-Level Composition for Software Federations*, SC'2003 (Avril 2003).

[LJW04] Bert Lagaisse, Wouter Joosen, and Bart De Win, *Managing Semantic Interference with Aspect Integration Contracts*, International Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT) (Lancaster, UK), Mars 2004.

[LMV01] A. Ledeczi, M. Maroti, and P. Volgyesi, *The Generic Modeling Environment*, Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISPŠ01), 2001.

[LNK+01] A Lédeczi, G Nordstrom, Gabor Karsai, P Völgyesi, and M Maroti, *On Metamodel Composition*, Proceedings of the 11th IEEE International Conference on Control Applications (CCA '01) (Mexico City, Mexico), 2001, pp. 756–760.

[LRdS04] Miguel Luz and Alberto Rodrigues da Silva, *Executing UML Models*, 3rd Workshop in Software Model Engineering (WiSME 2004) (Octobre 2004).

[Luc97] David C. Luckham, *Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*, POMIV'96 : Proceedings of the DIMACS Workshop on Partial Order Methods in Verification (New York, NY, USA), AMS Press, Inc., 1997, pp. 329–357.

[Mar04] Christophe Mareschal, *Adaptation d'un Langage de Description d'Architecture à l'Expression du Comportement*, Journées Formalisation des Activités Concurrentes (FAC'2004) (Mars 2004).

[MB02] Steve Mellor and Marc Balcer, *Executable UML : A Foundation for Model Driven Architecture*, Addison Wesley, 2002.

[MBL07] Z. Molnár, D. Balasubramanian, and A. Lédeczi, *An Introduction to the Generic Modeling Environment*, In Proceedings of the TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum (Zurich, Switzerland), June 2007.

[MFJ05] P.-A. Muller, F. Fleurey, and J-M Jézéquel, *Weaving Executability into Object-Oriented Meta-Languages*, Proceedings of MODELS/UML'2005 (Montego Bay, Jamaica) (S. Kent and L. Briand, eds.), vol. 3713, Springer, Octobre 2005, pp. 264–278.

[MK96] J. Magee and J. Kramer, *Dynamic Structure in Software Architectures*, Proceedings of the ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering (FSE4), vol. 21, Octobre 1996, pp. 3–14.

[MRB03] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein, *Rondo : A Programming Platform for Generic Model Management*, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 2003, pp. 193–204.

[MT00] N. Medvidovic and R.N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering **26** (Janvier 2000), no. 1, 70 – 93.

[MV06] F. Mostefaoui and J. Vachon, *Formalization of an Aspect-Oriented Modeling Approach*, Proceedings of Formal Methods 2006 (Hamilton, ON), 2006.

[NBA05] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit, *Composing Aspects at Shared Joinpoints*, Proceedings of International Conference NetObjectDays (NODe) (Erfurt, Germany), Septembre 2005, pp. 19–38.

[NSC+07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave, *Matching and Merging of Statecharts Specifications*, ICSE '07 : Proceedings of the 29th international conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2007, pp. 54–64.

[OMG06] OMG, *CORBA Component Model, v4.0*, Document en ligne, disponible à http://www.omg.org/technology/documents/formal/components.htm, Avril 2006.

[Par72] D.L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM **15** (Decembre 1972), no. 12.

[PDN86] R. Prieto-Diaz and J.M. Neighbors, *Module Interconnection Languages*, Journal of Systems and Software **6** (Novembre 1986), no. 4, 307–334.

[PE08] Gabriel Pedraza and Jacky Estublier, *An Extensible Services Orchestration Framework through Concern Composition*, International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NFPDSML 2008) (Toulouse, France), Septembre 2008.

[PS04] Risto Pitkanen and Petri Selonen, *A UML Profile for Executable and Incremental Specification-Level Modeling*, 7th International Conference of the Unified Modeling Language (Lisbon, Portugal), 2004.

[RBP+91] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe, *The Architecture of a UML Virtual Machine*, Proceedings of 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01), ACM Press, 2001, pp. 327–341.

[RGF+06] Y. Raghu Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg, *Directives for Composing Aspect-Oriented Design Class Models*, Transactions on Aspect-Oriented Software Development I, LNCS 3880 (Springer-Verlag, 2006), 75–105.

[RKRS05] T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger, *Model Integration Through Mega Operations*, Workshop on Model-driven Web Engineering (MDWE) (2005).

[RM89] Arnold Rochfeld and José Morejon, *La Méthode Merise - Tome 3 Gamme Opératoire*, Editions d'organisation (Paris), 1989.

[SDK+95a] Mary Shaw, Robert Deline, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik, *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transactions on Software Engineering **21** (1995), 314–335.

[SDK+95b] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik, *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transactions on Software Engineering **21** (1995), no. 4, 314–335.

[Sei03] E. Seidwitz, *What Models Mean*, IEEE Software (September 2003).

[SG94] Mary Shaw and David Garlan, *Characteristics of Higher-level Languages for Software Architecture*, Tech. Report CMU-CS-94-210, Carnegie Mellon University, School of Computer Science, December 1994.

[SJ05] S. Sanlaville (Jamal), *Environnement de Procédé Extensible pour l'Orchestration - Application aux Services Web*, Ph.D. thesis, Université Joseph Fourier, Décembre 2005.

[SM05] T. Schattkowsky and W. Müller, *A UML Virtual Machine for Embedded Systems*, In Proceedings of International Conference on Information Systems - New Generations (ISNG) (USA, 2005), Avril 2005.

[Ste02] Perdita Stevens, *On the Interpretation of Binary Associations in the Unified Modelling Language*, Software and System Modeling **1** (2002), no. 1, 68–79.

[Szy98] Clemens Szyperski, *Component Software : Beyond Object-Oriented Programming*, ACM Press and Addison-Wesley, New York, NY, 1998.

[Thi98] S. Thibault, *Langage Dédiés : Conception, Implémentation et Application*, Thèse de doctorat, Université de Rennes 1, France, October 1998.

[Veg05] G. Vega, *Développement d'Applications à Grande Echelle par Composition de Méta-Modèles*, Ph.D. thesis, Université Joseph Fourier, Decembre 2005.

[Vil02] J. Villalobos, *APEL : Spécification Formelle du Moteur. rapport technique Équipe adèle*, Mars 2002.

[Vil03] ———, *Fédération de Composants : une Architecture Logicielle pour la Composition par Coordination*, Ph.D. thesis, Université Joseph Fourier, Juillet 2003.

[Wam03] Dean Wampler, *The Role of Aspect-Oriented Programming in OMGŠs Model-Driven Architecture.*

[Wan00] Ju An Wang, *Towards Component-Based Software Engineering*, Journal of Computer Science in Colleges **16** (Novembre 2000), no. 1, 177–189.

[Wik] Wikipedia, *Aspect-Oriented Programming*, http://en.wikipedia.org/wiki/Aspect-oriented_programming.

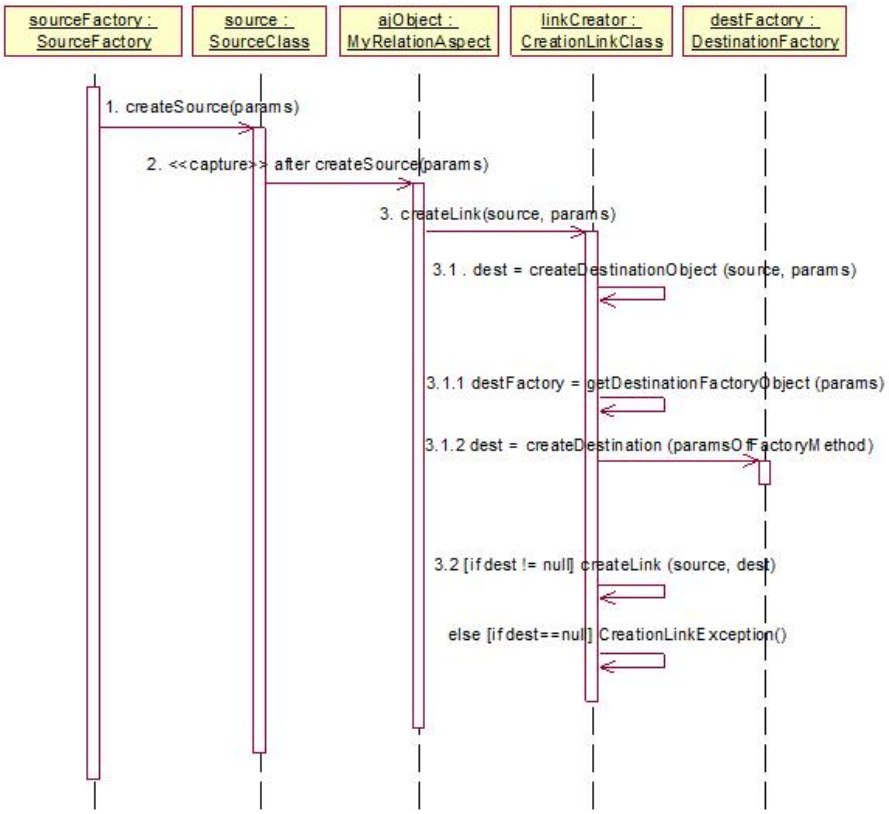
[ZCvdBG06] J. Zhang, T. Cottenier, A. van den Berg, and J. Gray, *Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver*, Proceedings of the 9th International Workshop on Aspect-Oriented Modeling at the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELSŠ06) (Milan, Italy), 2006.

[Zel96] Gregory Zelesnik, *Unicon Manual*, document en ligne, disponible à http://www.cs.cmu.edu/ UniCon/reference-manual/Reference_Manual_38.html, 1996.

Annexe

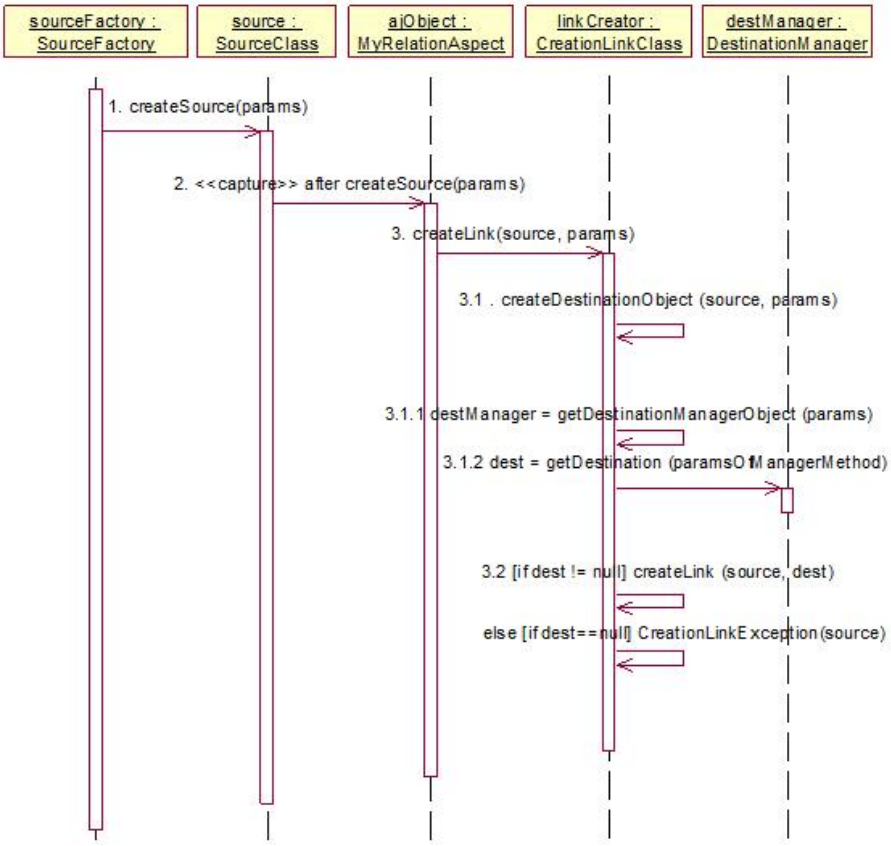
A.1 Génération de code pour les relations dynamiques

A.1.1 Automatic_New



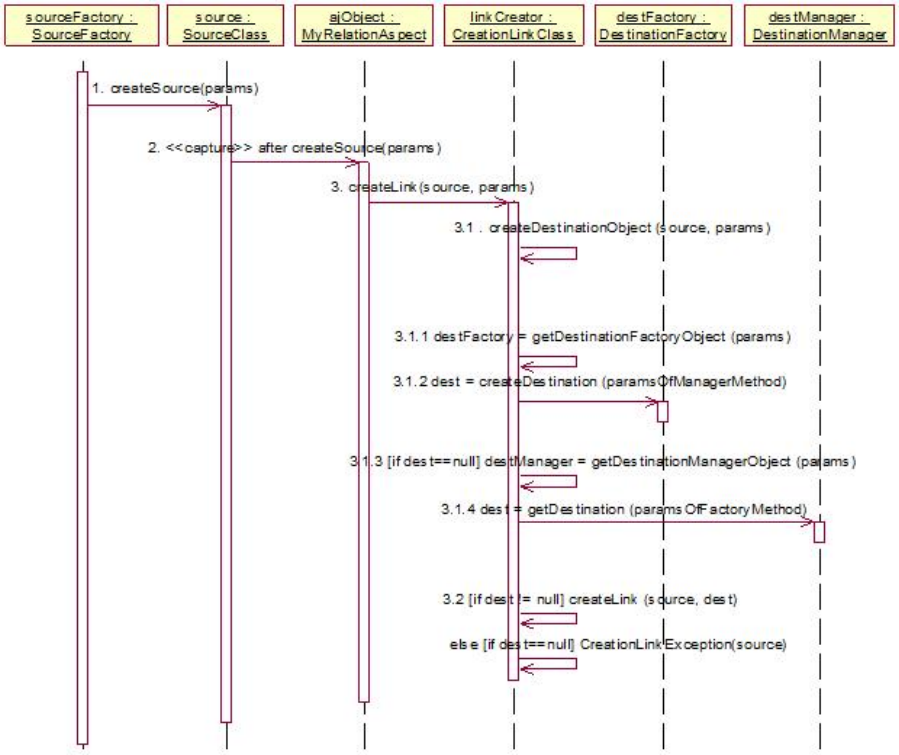
```
1. Demander la création de l'objet source.
2. Capturer la demande.
3. Demander de créer le lien.
   3.1 Créer l'objet destination.
        3.1.1 Chercher l'objet "factory" des objets destination.
        3.1.2 Créer l'objet de destination.
   3.2 Établir le lien.
```

A.1.2 Automatic_Selection



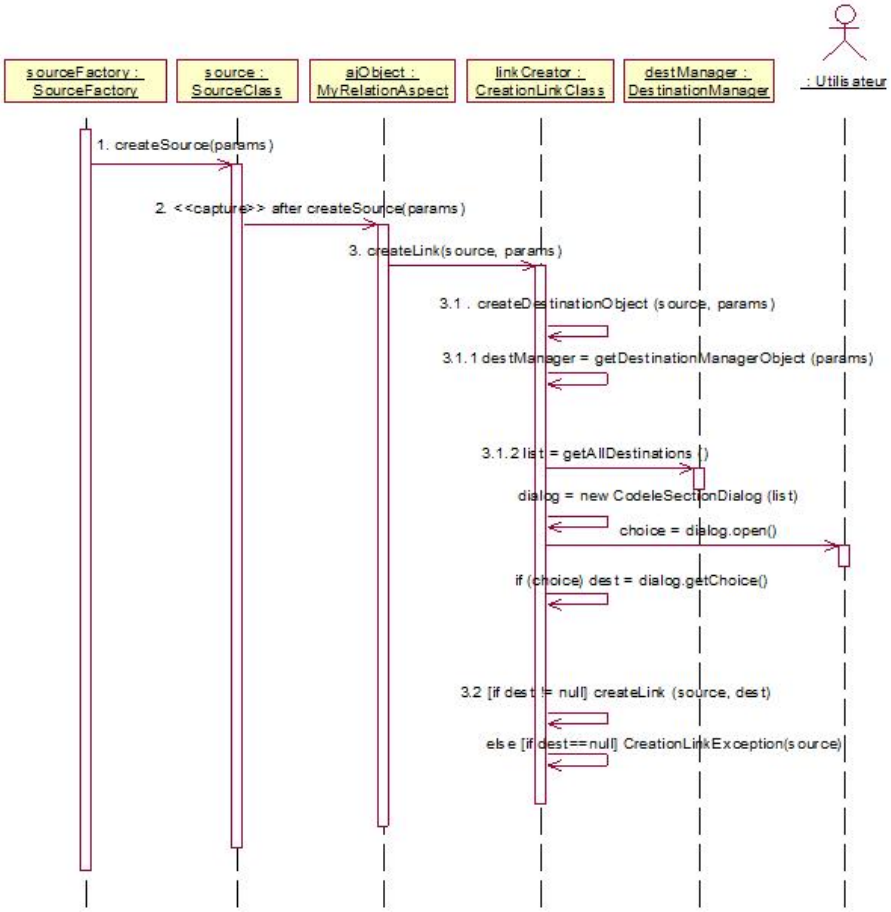

```
1. Demander la création de l'objet source.
2. Capturer la demande.
3. Demander de créer le lien.
 3.1 Chercher l'objet destination.
       3.1.1 Déterminer l'objet qui gère les objets destination.
       3.1.2 Déterminer l'objet destination.
   3.2 Établir le lien.
```

A.1.3 Automatic_Selection.Automatic_New



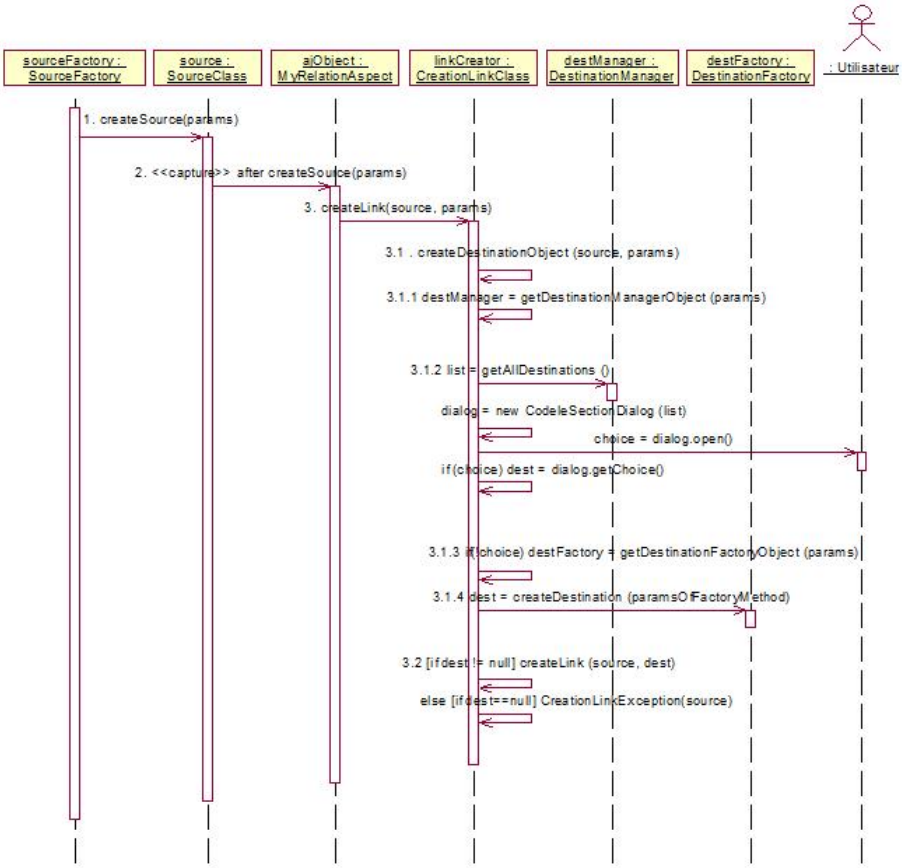

```
1. Demander la création de l'objet source.
2. Capturer la demande.
3. Demander de créer le lien.
   3.1 Chercher l'objet destination.
       3.1.1 Déterminer l'objet qui gère les objets destination.
       3.1.2 Déterminer l'objet destination (automatiquement).
       3.1.3 Si l'objet destination n'est pas trouvé, chercher
             l'objet "factory" de l'objet destination.
       3.1.4 Créer l'objet destination (automatiquement).
   3.2 Établir le lien.
```

A.1.4 Interactive_Selection




```
1. Demander la création de l'objet source.
2. Capturer la demande.
3. Demander de créer le lien.
   3.1 Chercher l'objet destination.
       3.1.1 Déterminer l'objet qui gère l'objet destination.
       3.1.2 Déterminer interactivement l'objet destination
             (par afficher une liste des destinations
              à l'utilisateur pour qu'il choisisse).
   3.2 Établir le lien.
```

A.1.5 Interactif_Selection.Automatic_New




```
1. Demander la création de l'objet source.
2. Capturer la demande.
3. Demander de créer le lien.
   3.1 Chercher l'objet destination.
       3.1.1 Déterminer l'objet qui gère l'objet destination.
       3.1.2 Déterminer interactivement l'objet destination
             (par afficher une liste des destinations
             à l'utilisateur pour qu'il choisisse).
       3.1.3 Si l'objet destination n'est pas trouvé,
             chercher l'objet "factory" des objets destination.
       3.1.4 Créer l'objet destination (automatiquement).
   3.2 Établir le lien.
```