وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR UNIVERSITY – ANNABA

UNIVERSITE BADJI MOKHTAR – ANNABA

جامعة باجي مختار – عنابة

Année : 2016

Faculté des sciences de l'ingéniorat
Département d'informatique

T H È S E

Pour obtenir le diplôme de
Docteur 3éme cycle

Une Approche dirigée par les Modèles pour les Architectures Logicielles

Filière : Informatique
Spécialité : Ingénierie des Logiciels Complexes

Préparée par

Sohaib HAMIOUD

Jury :

Président	Djamel MESLATI	Professeur	Université Badji Mokhtar - Annaba
Directrice	Fadila ATIL	Professeur	Université Badji Mokhtar - Annaba
Examinateurs	Abdelkrim AMIRAT	Professeur	Université Mohamed Chérif Messaidia - Souk Ahras
	Nora BOUNOUR	MCA	Université Badji Mokhtar - Annaba

ملخـــــص

أدى ارتفاع مستوى تعقيد الأنظمة البرمجية إلى ظهور ميدان البنى البرمجية الذي يهدف إلى تبسيط تطويرها ووصفها بدرجة عالية من الصورية. اهتم مؤخرا مجتمع هندسة البرمجيات بهذه المقاربة الجديدة مما أدى إلى ظهور الكثير من اللغات التي تسمح بالوصف الدقيق للبنيان البرمجي إضافة إلى اقتراح الطرق والأدوات المرفقة معها من اجل تحليل، تصميم وبناء أنظمة برمجية معقدة بطريقة آلية انطلاقا من وصفها البنيوي. بتحليل لغات وصف البنى البرمجية ADLs (Architecture Description Languages) الموجودة حاليا وأدواتها الداعمة، نستطيع أن نلاحظ وجود فجوة بين ما يصبوا إليه مجتمع الباحثين وما يوجد حقيقة في الواقع. أغلبية الأدوات التي تدعم لغات وصف البنى لا تلبي رغبات المهندسين والمطورين. للمساعدة على تجاوز هذه النقائص, نقترح مقاربة مبنية على النماذج والتي تعتمد على الاستغلال الأقصى لتقنيات النمذجة والتطوير المبني على النماذج لتسهيل بناء أدوات مناسبة، قابلة للتوسع و للتعديل حسب الطلب، وتسهيل إدماجها مع أدوات أخرى للنمذجة. مقاربتنا تستفيد من بيئات تطوير اللغات المختصة بالمجال DSLs (Domain Specific Languages) التي توفر آليات فعالة في التوليد الاتوماتيكي للأدوات المرافقة. لإقرار صلاحية المقاربة المقترحة، نقدم أداة تسمى ArchJaMoPP (ArchJava Model Parser and Printer) لدعم لغة وصف البنى ArchJava بمجموعة من الوظائف التي تمكن من تحليل، معالجة وتوليد النماذج البنيوية الموصوفة بهذه اللغة.

كلمات دالة : بنيان برمجي، هندسة مبنية على النماذج، لغة وصف البنى، أداة الدعم

Résumé

L'accroissement de la complexité des systèmes logiciels a donné naissance au domaine des architectures logicielles qui vise la simplification de leur développement et leur spécification à un haut niveau d'abstraction. Dernièrement, la communauté du génie logiciel s'est intéressée à l'exploitation de ce nouveau paradigme, ce qui a contribué à la définition de plusieurs langages permettant la description rigoureuse d'architectures logicielles ainsi que la proposition des méthodes et d'outils associés à ces langages afin d'automatiser l'analyse, la conception et la construction de systèmes complexes à partir de leur description architecturale. En analysant les langages de description d'architecture (*Architecture Description Languages* ou *ADLs*) existants et leurs outils d'assistance, on peut constater qu'il existe un fossé entre ce que la communauté de recherche identifie comme désirable et ce qui existe en réalité. La grande majorité des outils supportant les ADLs ne répondent pas aux besoins des architectes, ni aux besoins des développeurs. Pour aider à palier ces limites, nous proposons une approche dirigée par les modèles qui consiste à exploiter, au maximum, les techniques de modélisation et du développement dirigé par les modèles pour faciliter la construction d'outils adéquats extensibles et personnalisables, et pour assister leur intégration à d'autres outils de modélisation. Notre approche tire profit d'environnements de développement des langages dédiés qui offrent des mécanismes efficaces à la génération automatique d'outils d'accompagnement. Pour valider l'approche proposée, nous introduisons un outil nommé ArchJaMoPP (ArchJava Model Parser and Printer) qui assiste l'ADL ArchJava par un ensemble de fonctionnalités permettant l'analyse, la manipulation et la génération de modèles architecturaux décrits en ArchJava.

Mots-clés : Architecture logicielle, Ingénierie dirigée par les modèles, Langage de description d'architecture, Outil support.

Abstract

The increasing complexity of software systems has given rise to the discipline of software architectures. This new paradigm has attracted a large amount of interest lately within the software engineering community and has sparked the initiation of many languages which allow a precise description of software architectures, and the proposition of methods and tool support for these languages, to analyze, design and construct complex systems from their architectural descriptions. Inspecting the current state of ADLs (Architecture Description Languages) and their accompanying tools, allows to realize that there is a gap between what the research community has identified as desirable and what exists in reality. The majority of the existing tools that support ADLs do not fulfil the needs of neither architects nor developers. To tackle these limitations, we propose a model-driven approach that exploits modelling techniques and model-driven development to their full extent, to facilitate the construction of adequate, extensible and customizable tools, and to assist their integration with other modelling tools. Our approach benefits from language workbenches which provide efficient mechanisms for the automatic generation of accompanying tools. To validate the proposed approach, we introduce a tool called ArchJaMoPP (ArchJava Model Parser and Printer), which supports the ADL ArchJava with a set of functionalities for analyzing, manipulating and generating ArchJava architectural models.

Keywords : Software Architecture, Model driven engineering, Architecture description language, Tool support.

Ce travail est dédié à ma grand-mère

Remerciements

Je tiens à remercier tous les membres du jury. Un très grand merci à Pr. Fadila Atil d'avoir accepté de rapporter ce travail, pour la confiance qu'elle m'a accordée et pour ses remarques qui ont permis d'améliorer la qualité de ce document. Merci à Pr. Abdelkrim Amirat et à Dr. Nora Bounour d'avoir accepté d'examiner et d'évaluer mon travail. Enfin, je remercie Pr. Djamel Meslati de m'avoir accordé l'honneur d'être le président de mon jury.

Mes remerciements vont également aux enseignants de la spécialité ILC et aux membres du laboratoire LISCO pour leurs encouragements. J'ai particulièrement apprécié les conseils, les remarques et le soutien de Pr. L. Souici-Meslati et Dr. N. Bounour qui m'ont motivé à travailler encore plus dur.

Je profite également de cette occasion pour remercier chaleureusement mes collègues et mes amis pour leur aide et leurs encouragements.

Enfin, je remercie ma famille et mes proches, et en particulier mes parents qui m'ont soutenue avec patience durant mon parcoure académique et durant la réalisation de ce travail.

Je déclare que cette thèse, dans sa totalité, est un travail de recherche original et qu'elle est entièrement le fruit de mon travail personnel.

Sohaib HAMIOUD

Le mercredi 8 Avril 2016

Table des matières

1 INTRODUCTION 1

1.1 CONTEXTE ET PROBLÉMATIQUE 1

1.2 CONTRIBUTION 9

1.3 ORGANISATION DU DOCUMENT 11

2 L'INGÉNIERIE DES MODÈLES 13

2.1 ARCHITECTURE BASÉE MODÈLES 13

2.2 STANDARDS ET PROCESSUS DE MDA 15

2.2.1 Les Standards de L'OMG 16

2.2.2 Processus MDA 23

2.3 TRANSFORMATION DE MODÈLES 26

2.3.1 Plan de classification pour les problèmes de transformation de modèles 28

2.3.2 Outils et Langages de transformation de modèles 30

2.4 CONCLUSION 36

3 ARCHITECTURES LOGICIELLES 38

3.1 TERMINOLOGIES DES ARCHITECTURES LOGICIELLES 39

3.1.1 Les Composants 41

3.1.2 Les Connecteurs 54

3.1.3 Les Configurations 55

3.2 LES LANGAGES DE DESCRIPTION D'ARCHITECTURES 56

3.2.1 ACME 57

3.2.2 MetaH 61

3.2.3 AADL 64

3.2.4 Rapide 70

3.2.5 Darwin 75

3.2.6 Wright 78

3.2.7 ArchJava 81

3.3 DISCUSSION 83

3.4 CONCLUSION 85

4 APPROCHE DIRIGÉE PAR LES MODÈLES POUR LES ADLS 86

4.1 NOTRE APPROCHE 86

4.2 Cas d'Étude : ArchJava 87
4.3 Critères de Choix du Métamodèle Java 90
4.4 Étude comparative 91
4.4.1 Métamodèles Java 92
4.4.2 Comparaison des métamodèles Java 107
4.5 Modeles Java 107
4.6 Metamodele ArchJava 109
4.6.1 Composants 110
4.6.2 Ports 110
4.6.3 Méthodes de Port 111
4.6.4 Connexions 111
4.7 Conclusion 113

5 MISE EN ŒUVRE 114

5.1 Outils et technologies d'implémentation 114
5.2 Syntaxe Concrète d'ArchJava 115
5.3 Outillage de Support 118
5.3.1 ArchJaMoPP 120
5.3.2 Éditeur ArchJava 122
5.3.3 ArchJaMoPPC 124
5.4 Personnalisation 126
5.4.1 L'Assistant de Création de Projets et de Fichiers 126
5.4.2 Résolution des Références 130
5.4.3 Résolution des Ambiguïtés 134
5.5 Discussion 136
5.6 Conclusion 142

6 CONCLUSION ET PERSPECTIVES 143

6.1 Résumé des contributions 143
6.2 Perspectives 145

BIBLIOGRAPHIE 148

ANNEXE A : LISTE DE METAMODELES 158

A.1. Métamodèle ArchJava 158
A.2. Diagramme de classes UML 159

A.3. METAMODELE FAMIX 2.1 160
A.4. METAMODELE SPOON 161
A.5. METAMODELE JAMOPP 162
A.6. METAMODELE KDM 163
A.7. METAMODELE WRIGHT 164
A.8. METAMODELE RAPIDE 165
A.9. METAMODELE DARWIN/FSP 166

ANNEXE B : LISTE DES LISTINGS 167

B.1. ANALYSE D'UNE DESCRIPTION ACME EN JAVA 167
B.2. CRIBLE D'ÉRATOSTHENE EN ARCHJAVA 168
B.3. ARCHJAMOPPC.JAVA 169
B.4. ARCHJMINIMALMODELHELPER.JAVA 175
B.5. ARCHJNEWFILEWIZARDPAGE.JAVA 179
B.6. ARCHJNEWFILEWIZARD.JAVA 184
B.7. WORKER.ARCHJ.XMI 189
B.8. EXTRAIT DE ARCHJDEFAULTDELGATERESOLVER.JAVA 194
B.9. RESOLVE() DE CLASSIFIERREFERENCETARGETREFERENCERESOLVER.JAVA 198

Liste des tables

TABLE 3.1. PERSPECTIVES SUR LE CYCLE DE VIE DES COMPOSANTS. 45

TABLE 4.1. COMPARAISON DES METAMODELES JAVA. 107

Liste des figures

FIGURE 2.1. ARCHITECTURE DU MDA. 15
FIGURE 2.2. LES DIFFERENTS DIAGRAMMES UML. 17
FIGURE 2.3. L'ARCHITECTURE A QUATRE NIVEAUX DE MDA. 18
FIGURE 2.4. MOF (V1.4) SOUS FORME DE DIAGRAMME DE CLASSE [BLA 05]. 20
FIGURE 2.5. XMI ET LA STRUCTURATION DE BALISES XML [BLA 05]. 23
FIGURE 2.6. PROCESSUS DE DEVELOPPEMENT MDA [MES 06]. 25
FIGURE 2.7. LES ELEMENTS D'UNE TRANSFORMATION DE MODELES. 27
FIGURE 2.8. LES NIVEAUX META DE LA TRANSFORMATION DE MODELES. 28
FIGURE 2.9. ARCHITECTURE DE QVT. 32
FIGURE 2.10. METAMODELE ATL SIMPLIFIE. 33
FIGURE 3.1. METAMODELE ACME. 58
FIGURE 3.2. ACMESTUDIO. 60
FIGURE 3.3. METAMODELE METAH. 61
FIGURE 3.4. OUTILS METAH. 63
FIGURE 3.5. L'ENVIRONNEMENT OSATE. 69
FIGURE 3.6. METAMODELE DARWIN. 77
FIGURE 4.1. APPROCHE DE DEVELOPPEMENT. 87
FIGURE 4.2. COMPILATEUR ARCHJAVA. 88
FIGURE 4.3. REPERTOIRE SIEVE AVANT LA COMPILATION. 89
FIGURE 4.4. REPERTOIRE SIEVE APRES LA COMPILATION. 89
FIGURE 4.5. METAMODELE GRAMMYUML [GOR 03]. 94
FIGURE 4.6. ARCHITECTURE DE QUATRE NIVEAUX DE UNIQ-ART [ŞOR 13]. 95
FIGURE 4.7. MODELE DE HAUT NIVEAU DE DMM [LET 04]. 96
FIGURE 4.8. LA HIERARCHIE MODELOBJECT DE DMM [LET 04]. 97
FIGURE 4.9.LA HIERARCHIE SOURCEOBJECT DE DMM[LET 04]. 97
FIGURE 4.10. LA HIERARCHIE D'ASSOCIATION DE DMM [LET 04]. 98
FIGURE 4.11. EXTRAIT DU METAMODELE JAMOPP. 101
FIGURE 4.12. METAMODELE DE LA REPRESENTATION GRAPHE DE JAVA [HOF 08]. 103
FIGURE 4.13. CLASSE MOVIE SOUS FORME DE MODELE JAVA. 108

FIGURE 4.14. ARCHITECTURE A QUATRE NIVEAUX. 109
FIGURE 4.15. METACLASSE COMPONENTCLASS. 110
FIGURE 4.16. METACLASSE PORT. 111
FIGURE 4.17. METACLASSE PORTMETHOD. 112
FIGURE 4.18. METACLASSES DE CONNEXION. 112
FIGURE 5.1. ARCHITECTURE DU PLUGIN ARCHJAMOPP. 115
FIGURE 5.2. LE MODELE GENERATEUR ARCHJAVA.GENMODEL. 119
FIGURE 5.3. API JAVA DU METAMODELE ARCHJAVA. 120
FIGURE 5.4. PROJETS GENERES PAR EMFTEXT POUR REALISER L'OUTILLAGE. 121
FIGURE 5.5. EDITEUR ARCHJAVA. 123
FIGURE 5.6. EXECUTION D'ARCHJAMOPPC.JAR. 124
FIGURE 5.7. L'ASSISTANT DE CREATION DE FICHIERS ET DE PROJETS. 127
FIGURE 5.8. L'ASSISTANT PERSONNALISE DE CREATION DE FICHIERS ET DE PROJETS. 128
FIGURE 5.9. PROGRAMME MINIMAL D'ARCHJAVA. 129

Liste des listings

Listing 2.1. Declaration de l'entete de la transformation. 33
Listing 2.2. Déclaration d'une règle de transformation. 34
Listing 2.3. Pattern cible. 34
Listing 2.4. La forme d'un attachement. 34
Listing 2.5. Méthode en ATL. 35
Listing 2.6. Attribut en ATL. 35
Listing 5.1. La syntaxe concrète d'ArchJava en CS. 116
Listing 5.2. Modele simplifie de Worker en XMI. 126
Listing 5.3. Appel de la methode getMinimalModel(). 129
Listing 5.4. Initialisation de l'attribut namespaces de la classe composante. 129
Listing 5.5. Code ajoute a la classe ArchjNewFileWizard. 130
Listing 5.6. La méthode resolve() de ClassifierReferenceTargetReferenceResolver. 132
Listing 5.7. Appel de resolve() du resolveur importe. 133
Listing 5.8. Extrait de la methode parse_org_emftext_language_java_members_Member(). 135
Listing 5.9. parse_org_emftext_language_java_members_Member() non-ambiguë. 136
Listing 5.10. La fonction de transformation QVT archj2java(). 139
Listing 5.11. La fonction archj2java () en Java. 142

Liste des sigles et acronymes

ADL : Architecture Description Language

AADL : Architecture Analysis & Design Language

API : Application Programming Interface

ArchJaMoPP : ArchJava Model Parser and Printer

ASG : Abstract Syntax Graph

AST : Abstract Syntax Tree

ASTM : Abstract Syntax Tree Metamodel

ATL : Atlas Transformation Language

BCEL : Byte Code Engineering Library

BMP : Bean Managed Persistence

CBSE : Component Based Software Engineering

CCM : Corba Component Model

CEP : Complex Event Processing

CIDL : Component Implementation Definition Language

CIF : Component Implementation Framework

CIM : Computation Independent Model

CMOF : Complete Meta Object Facility

CMP : Container Managed Persistence

COTS : Component Off The Shelf

CS : Concrete Syntax

CSP : Communicating Sequential Processes

Ct : Compile time

CWM : Common Warehouse Metamodel

DMM : Dagstuhl Middle Metamodel

DTD : Document Type Definition

DSL : Domain Specific Language

EADL : Executable Architecture Description Language

EJB : Enterprise JavaBeans

EJC : Eclipse Compiler for Java

EMF : Eclipse Modeling Framework

EMOF : Essential Meta Object Facility

FSP : Finite State Process

GASTM : Generic Abstract Syntax Tree Metamodel

GCJ : GNU Compiler for Java

GMF : Graphical Modelinf Framework

GPL : General Purpose Language

GUI : Graphical User Interfaces

IDE: Integrated Development Environment

IDL : Interface Definition Language

JaMoPP : Java Model Parser and Printer

JDT : Java Development Tool

JLS : Java Language Specification

JMS : Java Message Service

KDM : Knowledge Discovery Metamodel

KM3 : Kernel Meta-MetaModel

LISCO : Laboratoire d'Ingénierie des Systèmes COmplexes

LTSA : Labelled Transition System Analyser

MDA : Model Driven Architecture

MDE : Model Driven Engineering

MDR : Meta Data Repository

MIA-T : Model-In-Action Transformation

MOF : Meta Object Facility

MOON : Minimal Object-Oriented Notation

MWB : Mobility WorkBench

OCL : Object Constraint Language

OMG : Object Management Group

OSATE : Open-Source AADL Tool Environment

POSET : Partially Ordered Event SET

PIM : Platform Independent Model

PSM : Platform Specific Model

QVT : Query View Transformation

SAE : Society of Automative Engineer

SAFA : Software Architecture and Formal Approaches

SASTM : Specialized Abstract Syntax Tree Metamodel

SDF : Syntax Definition Formalism

SDL : Specification and Description Language

TCS : Textual concrete Syntax

TOPCASED : Toolkit in Open Source for Critical Applications & Systems Development

UI : User Interface

UML : Unified Modeling Language

URI : Uniform Resource Identifier

XOCL : eXtensible Object Constraint Language

XMI : XML Metadata Interchange

XML : eXtensible Markup Language

Liste des abréviations

C.f. : Confer

E.g. : Exempli gratia

Etc. : Et cætera

I.e. : Id est

1 Introduction

1.1 Contexte et problématique

Le travail que nous présentons dans cette thèse a été élaboré au niveau de l'équipe SAFA du laboratoire d'ingénierie des systèmes complexes (LISCO) de l'université de Badji-Mokhtar d'Annaba. L'équipe SAFA (Software Architecture and Formal Approaches) s'active autour des thèmes liés à l'ingénierie du logiciel. Son activité de recherche se concentre actuellement sur les nouveaux paradigmes de génie logiciel, et plus particulièrement les paradigmes d'architectures logicielles et les approches formelles.

Cette thèse s'inscrit à la croisée de deux disciplines principales : celle de l'architecture logicielle à base de composants et celle de l'ingénierie des modèles. Ces deux disciplines s'accordent sur la nécessité de consacrer et d'exploiter pleinement les techniques d'abstraction des systèmes logiciels. L'abstraction permet de gérer la complexité croissante des systèmes logiciels en réduisant la quantité d'information à traiter, tout en gardant ce qui est important et essentiel. Les nouveaux paradigmes du génie logiciel ont prouvé leurs efficacités dans l'industrie des systèmes informatiques. Des paradigmes comme la programmation orienté aspect, orienté composant et l'ingénierie dirigée par les modèles rendent le développement et l'évolution des logiciels plus faciles. Ces deux dernières approches sont en plein expansion dans le monde académique et sont présentes, de plus en plus, dans le monde industriel. Nous visons dans ce travail l'utilisation des techniques et principes de l'ingénierie des modèles pour faciliter le développement et l'évolution des outils supports manipulant les architectures logicielles à base de composants.

L'ingénierie dirigée par les modèles (MDE pour Model-Driven Engineering) préconise l'utilisation de modèles, au cours du développement logiciel, pour capturer la structure des différents domaines d'une application, à différents niveaux d'abstraction, qu'ils soient des domaines métiers ou techniques [Veg 05]. Le modèle représente un concept clé dans MDE et son importance peut être comparée avec celle du concept d'objet dans la programmation orientée objet [Béz 04]. MDE s'appuie principalement sur l'initiative MDA (Model-Driven Architecture) publiée en 2000 par l'OMG (Object Management Group) [Omg 16]. Cette

approche a engendrée des modifications radicales dans le cycle de développement des logiciels. Elle consiste à séparer les spécifications fonctionnelles d'un système des détails de son implémentation sur une plateforme donnée. Pour cela, MDA définit une architecture de spécification structurée en plusieurs types de modèles. Chaque type correspond à une pré-occupation particulière dans le système. MDA propose la projection de modèle abstrait lié à l'activité d'un système logiciel sur les plateformes, donc sur différentes technologies d'implantation. Le résultat de cette projection est un modèle spécifique à une plateforme qui appartient à un niveau d'abstraction plus concret. À partir de ce dernier, le code exécutable du système peut être généré. En d'autre terme, MDA décrit le processus de développement comme des modèles et des transformations de modèles. Les modèles sont transformés d'un niveau d'abstraction à un autre jusqu'à l'obtention du code source.

L'architecture logicielle comme nouvelle discipline du génie logicielle, décrit de manière abstraite les systèmes à l'aide de composants logiciels, les relations entre ces composants, leurs propriétés fonctionnelles et non fonctionnelles ainsi que les principes qui régissent leur conception et leur évolution [Med 00]. Les langages de description d'architecture (*Architecture Description Languages* ou *ADLs*) ont vu le jour pour fournir les abstractions selon lesquelles les architectures vont être spécifiées. Ce sont des formalismes qui offrent de larges opportunités de modélisation et de développement, axés essentiellement sur l'utilisation de modèles, qui explicitent les constituants d'un système indépendamment des plateformes. En d'autres termes, ils mettent l'accent sur la phase de conception plutôt que sur la phase d'implémentation. Dans la suite de cette thèse, nous allons adopter l'abréviation anglaise ADL, fréquemment utilisée dans la littérature, pour désigner ces langages.

Actuellement, on assiste à une prolifération d'ADLs et de modèles de composant que ce soit dans les milieux académiques ou dans l'industrie [Med 00]. Tous ces langages visent la description d'architecture logicielle mais possèdent des capacités différentes, et sont, dans la plupart des cas, dédiés aux domaines spécifiques. Malheureusement, plusieurs problèmes vont à l'encontre de l'utilisation, la définition et la généralisation des ADLs. Medvidovic et Rosenblum abordent sept types de problèmes (ou domaines architecturaux) qu'un ADL doit contourner [Med 97] :

- La représentation. Il est très important qu'une description d'architecture soit simple, compréhensible et si possible graphique. Une représentation explicite d'une architecture, étant un modèle qui présente une vue particulière du système, aide à la compréhension et la communication entre les différents intervenants (*stakeholders*), tels que les architectes, les développeurs, les managers et les clients. Typiquement, différents intervenants nécessitent différentes vues d'architecture. À titre d'exemple, le client peut demander un modèle à boites/flèches de haut niveau, les développeurs peuvent préférer un modèle détaillé de composants et de connecteurs, alors que les managers peuvent requérir une vue du processus de développement;
- Processus de conception. Les architectes décomposent les systèmes répartis et hétérogènes de grandes tailles en des blocs plus petits en prenant plusieurs décisions, en utilisant plusieurs outils et techniques de conception et de modélisation, et en mettant en évidence plusieurs problèmes. L'assistance durant le processus de conception ne se limite pas seulement à la modélisation d'architectures selon de multiples perspectives mais peut inclure l'aide à la décision, la documentation des décisions architecturales et des raisons de décisions (*Design Rationale*), et la revisite des étapes de conception;
- L'analyse. Parmi les arguments les plus souvent évoqués pour utiliser les ADLs est la fourniture d'une représentation formelle qui permet l'analyse et l'évaluation des propriétés des systèmes logiciels distribués de grandes tailles, ce qui diminue substantiellement le nombre d'erreurs très tôt dans le processus de développement. Ainsi, l'abstraction des détails d'implémentation au niveau d'architectures rend la tâche d'analyse beaucoup plus facile à effectuer et moins coûteuse que l'analyse du code source. L'analyse d'architectures peut être statique, dynamique ou les deux. L'analyse statique s'opère avant l'exécution et permet, par exemple, de vérifier la consistance interne d'architecture, la performance, la sécurité, la fiabilité, la présence d'interblocage (*deadlock*) et de famine (*starvation*), etc. Alors que l'analyse dynamique s'effectue à l'exécution d'architecture. Une architecture en cours d'exécution peut signifier que le système qui en découle est en cours d'exécution ou le comportement de l'architecture elle-même est simulé. L'analyse dynamique comprend le débogage, la vérification précise et l'évaluation dynamique des

contraintes structurelles et temporelles, de la performance, de la fiabilité, de la sécurité, etc.;

- L'évolution. Les architectures logicielles évoluent pour refléter les changements qui touchent un système logiciel ou pour définir des familles de systèmes. Clairement, les composants et les connecteurs peuvent aussi évoluer individuellement. L'évolution peut être réalisée à l'étape de spécification (i.e. évolution statique) ou à l'exécution (i.e. évolution dynamique). Généralement, l'évolution statique d'une architecture concerne le développement incrémental d'une famille de systèmes de telle sorte qu'elle s'accommode effectivement aux changements, tels que l'ajout de nouveaux composants. Les architectures exhibent la dynamique en permettant la réplication, l'insertion, la suppression et la reconnexion des éléments d'architecture pendant l'exécution. Ces modifications peuvent être planifiées lors de la spécification d'architecture ou imprévues;
- Le raffinement. Une architecture logicielle peut être définie à différents niveaux d'abstraction pour différentes raisons. Par exemple, une spécification d'architecture de haut niveau peut être utilisée comme un moyen de communication et de compréhension; une spécification de niveau inférieur peut être analysée afin de vérifier la consistance des interconnexions; une spécification plus fine peut être utilisée dans une simulation ou pour générer le code source du système. Par conséquent, le raffinement correct et consistant d'architecture, en descendant d'un niveau d'abstraction à un autre, est souhaitable et impératif;
- La traçabilité. Alors que le problème de raffinement se focalise essentiellement sur les niveaux d'abstractions d'architecture, la traçabilité peut avoir besoin de couvrir, en plus de cette direction, les différentes vues architecturales (e.g. graphique, textuelle, flux de données, flux de contrôle, processus, et implémentation). Les relations entre ces vues ne sont pas toujours bien déterminées et identifiées. Par exemple, les ADLs, en général, fournissent des mécanismes de traçabilité entre les vues textuelle et graphique, de telle sorte qu'un changement qui touche l'une sera automatiquement reflété dans l'autre. Ce n'est pas le cas de la vue du flux de données et la vue du processus, où les traces de changement et la manière dont les deux s'affectent l'une l'autre ne sont pas assez claires. Garantir la traçabilité entre les vues architecturales et les niveaux d'abstraction

simultanément par les ADLs est encore plus difficile. L'ultime problème de traçabilité dans ce contexte concerne les relations entre l'architecture et son implémentation. Les ADLs ne fournissent aucun moyen d'assurer que le code source qui implémente les composants d'architecture produit vraiment le comportement voulu. Même avec une implémentation idéale, les changements futurs qui seront apportés au code source devront être retracés à l'architecture et vice versa;

- La simulation/l'exécutabilité. La simulation d'une architecture logicielle dépend directement de la capacité de l'ADL à modéliser son comportement dynamique. Elle illustre partiellement le comportement dynamique final du système. La simulation d'architecture se révèle particulièrement bénéfique si les différents intervenants peuvent obtenir de l'information nécessaire sur le système et son comportement très tôt dans le processus de développement, de manière rapide, flexible et moins couteuse. L'ultime objectif d'une tentative de conception et de modélisation logicielle est la production d'un système complet et exécutable. D'ailleurs, un modèle d'architecture a peu de valeur, sauf s'il peut être converti en une application exécutable. Il est à noter qu'une transformation manuelle d'architecture en une application exécutable augmente le potentiel de problèmes de consistance et de traçabilité entre l'architecture et son implémentation;

Medvidovic et Rosenblum introduisent une évaluation de neuf ADLs selon les différents domaines architecturaux. Ils constatent que la pluparts des ADLs ne supportent qu'un sous ensemble de ces domaines [Med 97]. Dans la même optique, Medvidovic et Taylor [Med 00] font référence à l'importance des outils supports dans la valorisation des ADLs. Les résultats de leur étude montrent l'existence d'une forte relation entre les domaines architecturaux et l'outillage, dans le sens où la résolution des problèmes soulevés ne peut être réalisée qu'à travers des outils puissants. En jetant un coup d'œil rapide sur les fonctionnalités désirables des outils [Med 00], on apercevra facilement cette relation :

- La spécification active. Actuellement, peu d'ADLs fournissent des outils qui supportent la spécification active d'architectures. En général, ce type d'outils peut être réactif ou proactif. Alors que les outils proactifs (e.g. l'éditeur graphique de l'ADL UniCon [Sha 95]) préviennent les erreurs durant la conception, les outils réactifs détectent les erreurs de conception commises puis, soit ils informent seulement l'architecte (e.g. l'environnement

Argo de l'ADL C2 [Med 96]), soit ils l'obligent à corriger ces erreurs avant de poursuivre (e.g. l'éditeur graphique de l'ADL MetaH [Ves 96]). Les outils qui permettent la spécification active d'architecture contribuent à l'assistance de l'architecte tout au long du processus de conception;

- Vues multiples. Plusieurs ADLs (e.g. Rapide [Luc 95], UniCone, MetaH, Darwin [Mag 96], et C2) fournissent les deux vues de base d'une architecture : textuelle et graphique. De plus, ils permettent d'avoir une vue d'ensemble de haut niveau ou détaillée des éléments composites. En revanche, les autres vues sont souvent négligées. C2 offre une vue de processus de développement qui correspond à l'architecture. Rapide et C2 permettent la visualisation du comportement d'architecture par des outils de visualisation et de filtration d'évènements générés lors de la simulation d'architecture. Clairement, les outils sont indispensables pour résoudre le problème de représentation que nous avons évoqué et, voyant l'état actuel des ADLs, beaucoup de travail reste à faire;
- L'analyse. Les ADLs sont généralement dotés par un parseur, qui est, en fait, un outil d'analyse qui permet de vérifier la validité syntaxique des spécifications architecturales. Certains ADLs (e.g. Rapide, UniCon et MetaH) utilisent un compilateur pour vérifier aussi la sémantique d'architecture. Les capacités d'analyses des ADLs se différencient selon le modèle sémantique qui les caractérise. Par exemple, Wright [All 98] supporte la détection des interblocages au niveau des connecteurs, SADL [Mor 95] permet de vérifier si le raffinement d'une architecture vers une autre est correct, Rapide et C2 fournissent des outils de traitement et de filtrage d'événements, C2 permet de vérifier l'adhérence aux directives et règles gouvernant la conception d'architecture, Rapide est doté par un outil de vérification et de validation de contraintes, etc. Il est évident que l'analyse ne peut s'effectuer correctement et efficacement qu'avec des outils logiciels adéquats;
- Le raffinement. Parmi les ADLs présents dans la littérature, seuls Rapide et SADL offrent un support automatique pour le raffinement d'architecture. C'est une tâche complexe dont la réalisation requiert des outils d'assistance assurant la préservation du comportement;
- Génération de code. La production automatique d'un système logiciel à partir d'une description d'architecture atténue les problèmes de traçabilité et de consistance. Plusieurs

ADLs possèdent des outils de génération de code, tels que MetaH (génère un code Ada), UniCon (code C), C2 (Java, C++ et Ada), Rapide (C, C++, Ada), etc. D'autres ADLs tels que ACME, Wright et SADL sont uniquement utilisés comme des notations de modélisation et ne fournissent aucun support de génération de code. De tels outils permettent aux ADLs de surmonter les difficultés de simulation/exécutabilité et de raffinement;

- La dynamique. Il existe très peu d'ADLs qui supportent, par des outils, la modélisation et la spécification de la dynamique de l'architecture. Rapide, Darwin et C2 peuvent modéliser les modifications planifiées qui seront opérées sur l'architecture au cours de l'exécution. Wright permet de modéliser de manière abstraite et formelle l'aspect dynamique de l'architecture (i.e. le comportement des composants, des connecteurs et de l'architecture) mais n'offre pas d'outils associés (environnement d'utilisation ou d'exécution).

En gros, une grande partie des problèmes soulevés sont liés aux outils accompagnant les ADLs. La disponibilité, l'efficacité et la maturité des outils supports de la modélisation, l'analyse, l'évaluation, la vérification et la génération automatique du code implémentant le système, ont un impact indéniable sur l'usage des ADLs. En analysant les ADLs existants et leurs outils d'assistance, on peut constater qu'il existe un fossé entre ce que la communauté de recherche identifie comme désirable et ce qui existe en réalité. Actuellement, la grande majorité de ces outils ne répondent pas aux besoins des architectes, ni aux besoins des développeurs. Pire encore, nous avons constaté que certains d'entre eux sont des prototypes abandonnés par leurs auteurs et ne sont plus supportés.

D'autre part, les ADLs tendent à se concentrer sur un seul domaine d'intérêt. Par exemple, l'ADL Wright se focalise sur l'analyse, SADL sur le raffinement, Darwin sur le dynamisme, etc. Pour tirer profit des fonctionnalités offertes par les divers outils, une des solutions proposées suggère l'utilisation d'un format d'échange entre eux (e.g. ACME [Gar 00]). Cependant, cette solution n'est pas suffisante puisqu'elle nécessite, à son tour, des mécanismes automatiques qui transforment une architecture d'un ADL à un autre. L'implémentation cohérente des architectures est une autre difficulté à surmonter. Le passage de l'architecture au code engendre, dans la plupart des cas, la violation des contraintes architect-

urales et la détérioration des concepts qui étaient explicites au niveau de l'architecture (e.g. les connecteurs).

Conscients de ces lacunes, nous avons étudiés un ensemble d'ADLs qui nous semblent parmi les plus représentatifs afin de déterminer les possibilités d'amélioration en utilisant les principes de l'ingénierie des modèles. Dans cette quête, nous avons constaté le fait que les techniques de modélisation, de métamodélisation, de transformation de modèles, et de génération de code, étant proposées pour accélérer et faciliter la construction des systèmes logiciels, sont inexploitées dans le contexte des ADLs. Pourtant l'architecture elle-même est un modèle abstrait qui représente un système et qui s'intègre naturellement dans le processus de développement dirigé par les modèles.

Nous proposons dans cette thèse une approche dirigée par les modèles pour les architectures logicielles, dont l'objectif est l'amélioration de l'état actuel du support automatique des ADLs. Nous pensons que l'infrastructure AcmeLib[1] d'Acme est un exemple à suivre. Il s'agit d'une bibliothèque conçue pour faciliter la construction de nouveaux outils manipulant Acme. Le listing en annexe B.1 donne un exemple de code Java qui permet l'analyse d'une description d'architecture Acme (un fichier qui porte l'extension acme), de charger le modèle construit (un arbre syntaxique abstrait) dans une ressource Acme (*IAcmeResource*), et de le récupérer sous forme d'un modèle Acme (*AcmeModel* ou *IAcmeModel*). Cette bibliothèque est l'une des raisons pour lesquelles Acme est accepté comme un médiateur entre ADLs puisqu'elle permet de manipuler les descriptions Acme par programmation, et donc facilite leur manipulation par des outils externes.

Vu qu'un ADL n'est qu'un DSL (*Domain Specific Language*), nous préconisons l'utilisation d'un *language workbenche*, environnement de développement de langages dédiés. Selon notre approche, le métamodèle d'un ADL est considéré comme un modèle clé qui sera utilisé pour créer les outils d'assistance. Le processus de création inclut la métamodélisation, la transformation de modèles, la génération de code exécutable et la modification de ce dernier

[1] http://www.cs.cmu.edu/~./acme/

au gré des besoins. Le résultat de ce processus est une infrastructure de manipulation riche formant un support de construction d'outils. L'ADL ArchJava est choisi comme un cas d'étude pour valider l'approche par la création d'un ensemble d'outils d'accompagnement.

1.2 Contribution

La démarche suivie consiste à exploiter, au maximum, les techniques de modélisation et du développement dirigé par les modèles pour répondre aux problématiques levées précédemment, à savoir, surmonter le manque d'outils adéquats, faciliter leur intégration à d'autres outils de modélisation, créer des outils extensibles et personnalisables, et supporter le passage de l'architecture au code. Un cas d'étude est choisi pour valider l'approche proposée. Il s'agit de l'ADL ArchJava. Ce dernier est une extension du langage Java, qui intègre directement un langage de description d'architecture (i.e. un DSL) dans un langage de programmation générique (i.e. *GPL* ou *General Purpose Language*). Ce choix est basé sur deux motifs : le fait que ArchJava représente un cas d'étude extrême vu qu'il est situé réellement entre un ADL et un GPL, ce qui augmente la difficulté de la création d'outils d'assistance, et le fait qu'il y a un manque de logiciels qui facilitent sa manipulation par les outils de modélisation existants.

Cette thèse a donné lieu aux contributions suivantes :

- La proposition d'une approche dirigée par les modèles pour améliorer l'état actuel des ADLs, particulièrement en ce qui concerne l'outillage. Les modèles sont centrales, à la fois pour guider le processus de développement et faciliter l'intégration du langage de description au sein des outils de modélisation. Le modèle d'entrée est le métamodèle définissant l'ADL, qui sera utilisé pour générer une partie du code d'implémentation. Il joue donc un rôle prépondérant dans les transformations de modèles que ce soit durant le développement ou durant la manipulation des descriptions d'architectures;
- La création d'un métamodèle qui décrit l'ADL ArchJava. Ce métamodèle joue un rôle clé dans l'approche car il définit une base formelle sur laquelle les outils seront construits. Ainsi, il facilite la manipulation des architectures décrites en ArchJava par les environnements de modélisation. Le métamodèle se base sur JaMoPP [Hei 10] qui est un

métamodèle Java, et qui a été sélectionné après une étude comparative extensive des métamodèles Java qui existent dans la littérature;

- La définition d'une syntaxe concrète d'ArchJava en utilisant un langage appelé *CS* (*Concrete Syntax*). Ce langage est utilisé par l'environnement de développement de langages dédiés *EMFText* pour définir les syntaxes textuelles [Hei 09];
- Le développement et la personnalisation d'un ensemble d'outils qui sont à la fois extensibles et personnalisables. Ils sont en grande partie générés automatiquement par EMFText en se basant sur le métamodèle et la syntaxe ArchJava. L'ensemble inclut (1) un analyseur (parseur) qui lit une description ArchJava, puis construit un modèle qui est conforme au métamodèle et est décrit en utilisant un format standard de modélisation, (2) un générateur de code qui génère, à partir d'un modèle ArchJava, une description d'architecture en format textuel, et (3) un éditeur textuel doté de plusieurs fonctionnalités tels que l'autocomplétion du code, le correctif rapide, la refactorisation et le coloriage du code.

De manière semblable à AcmeLib, l'infrastructure d'ArchJava produite selon notre approche permet la création des outils logiciels en mode standalone ou sous forme de plugins Eclipse. Toutefois, son extension et sa modification s'avèrent plus facilement réalisables. D'une part, le fait qu'elle soit le résultat du passage automatique des modèles rigoureux et précis (i.e. métamodèle et modèle de syntaxe) vers le code signifie que tout changement qui touche ces modèles pour ajouter, supprimer ou encore modifier des éléments, sera automatiquement reflété dans le code grâce à l'application des mêmes règles de transformation utilisées lors du premier passage. D'autre part, l'environnement de développement de DSL qui implémente le processus de développement offre plusieurs techniques d'extension telles que la redéfinition des artéfacts générés et l'utilisation des points d'extension. AcmeLib étant une infrastructure encodée manuellement, elle n'offre pas les avantages qui découlent de l'utilisation des modèles et leurs transformations automatiques pour développer ce type d'infrastructure.

1.3 Organisation du document

Cette thèse se décompose en six chapitres au total dont un chapitre d'introduction, deux chapitres d'état de l'art, un chapitre de présentation de l'approche et du métamodèle, un chapitre dédié à la mise en œuvre de l'approche, et un dernier chapitre de conclusion.

Le premier chapitre (Introduction) sert à introduire le contexte du travail, les problématiques traitées dans ce document de thèse, et enfin nos contributions en réponse à ces problématiques.

L'état de l'art est partagé en deux chapitres suivant les deux paradigmes qui ont donné naissance à notre approche. Le premier (chapitre 2 : L'ingénierie des modèles) est consacré à l'approche MDA. Dans ce chapitre, nous nous intéressons, dans un premier temps, à la définition du concept de modèle et sa place dans les architectures basées modèles. En suite, nous introduisons les standards sur lesquels MDA est basée. Après, nous illustrons le processus de développement basé modèles. La dernière partie est consacrée à la transformation des modèles, ses principes, les différents types de transformation ainsi que les outils et langages qui existent actuellement pour la réaliser. Le chapitre 3 (Architectures logicielles) présente, dans un premier temps, les concepts fondamentaux de l'architecture logicielle, à savoir, les composants, les connecteurs et les configurations. Dans un second temps il aborde un ensemble d'ADLs selon deux perspectives : la métamodélisation et l'outillage de support.

C'est dans le chapitre 4 (Approche dirigée par les modèles pour les ADLs) que nous présentons notre approche et le cas d'étude (ArchJava). Nous introduisons ici une étude comparative de métamodèles Java puis nous détaillons les constructions ajoutées au métamodèle Java adopté pour définir le métamodèle ArchJava.

Dans le chapitre 5 (Mise en œuvre), nous spécifions les techniques d'implémentation utilisées pour valider notre approche en construisant des outils assistant l'ADL ArchJava. Nous définissons également la syntaxe concrète de l'ADL puis nous y montrons comment, à partir du métamodèle et la syntaxe concrète, les outils peuvent être créés.

Enfin, le chapitre 6 vient conclure cette thèse par la présentation des contributions et les perspectives de recherches associées.

2 L'ingénierie des Modèles

L'ingénierie des modèles et plus particulièrement l'initiative MDA (*Model Driven Architecture*) est une approche qui permet de décrire les logiciels sous forme de modèles et leur développement sous forme de transformations de modèles. Les bénéfices apportés par cette approche qui considère le modèle comme un point de référence tout au long du cycle de développement, ont attiré une attention considérable au cours de la dernière décennie. À travers ce chapitre, nous présentons l'approche MDA et le concept de modèle. Dans un second temps nous décrivons les standards sur lesquels l'architecture de MDA est basée ainsi que le processus de développement basé modèles. Par la suite, nous nous focalisons sur la transformation de modèles, son principe, les langages et les outils développés pour la réaliser.

2.1 Architecture Basée Modèles

Au fur et à mesure de l'apparition de nouvelles machines beaucoup plus puissantes, les applications logicielles deviennent de plus en plus complexes et de grosses tailles. Cette évolution engendre une difficulté lors de la tâche de développement et de maintenance et augmente significativement la complexité des logiciels (plus de dépendances entre les éléments d'un système logiciel) ainsi que la complexité de leurs développements. De par sa nature, la maintenance ou l'évolution des produits logiciels est difficile à effectuer. Cette difficulté est encore pire lorsque le produit à modifier, soit pour le maintenir ou l'évoluer, atteint un degré de complexité élevé. Ces problèmes nécessitent une approche interdisciplinaire qui permet de développer les systèmes logiciels de telle sorte qu'ils soient plus facilement et fiablement changés dans les phases qui suivent le développement [Mes 06].

L'ingénierie dirigée par les modèles (*MDE* pour *Model-Driven Engineering*) est l'une des nouvelles approches proposées pour résoudre les problèmes de développement, de maintenance et d'évolution. Elle fait partie des approches dites basées processus du fait qu'elles visent directement le processus de développement et de maintenance. L'ingénierie des modèles engendre une modification profonde dans le processus de production et de maintenance [Mes 06].

Cette approche fait suite à la proposition nommée MDA (pour *Model Driven Architecture*) faite par l'OMG (*Object Management Group*) en novembre 2000 [Omg 16]. L'objectif principal de cette approche est de séparer la logique métier (partie fonctionnelle) de la partie implémentation. De par le fait que la logique métier est stable, indépendante des plateformes d'exécution, et ne subit que peut de modifications à travers le temps, il est donc intéressant de séparer les deux parties parce que cette séparation va conduire à une meilleure portabilité, interopérabilité, réutilisation et productivité [Bla 05]. Ainsi, tout changement effectué sur une des préoccupations (i.e. fonctionnalités métiers, règles de gestions et plateforme d'exécution) n'aura aucun effet sur les autres.

Le point clés de l'approche MDA est l'utilisation massive des modèles. Un modèle peut être défini comme une description conceptuelle d'un système et de son environnement [Mes 06]. MDA consiste à décrire les systèmes en utilisant des modèles. Ces derniers peuvent être de différents types : Les modèles décrivant la logique métier qui visent à représenter la partie fonctionnelle des systèmes sans entrer dans les détails des plateformes d'exécution (e.g. J2EE, .Net, PHP, etc.) et les modèles incluant les détails techniques des plateformes.

L'utilisation des modèles présente plusieurs avantages [Bla 05]. Le modèle fournit plusieurs nivaux d'abstraction allant de modèle très abstrait, tel que la présentation de l'architecture générale d'un système logiciel, au modèle très concret, tel que la spécification d'un protocole de communication réseau. La possibilité de présenter les modèles sous format graphique est un avantage indéniable du fait qu'elle facilite la communication entre les acteurs des projets informatiques et augmente la compréhension et la métrise de la complexité. Ainsi, l'approche basée modèles emploie des outils et des langages standardisés pour créer les modèles (e.g. UML). Par conséquent, la communication et l'échange par les modèles seront grandement simplifiés, ce qui laisse un effet important sur les processus de développement et de maintenance.

Bien entendu, les modèles peuvent appartenir à différents niveaux d'abstraction. L'approche MDA fournit un point clés puissant qui permet de transformer un modèle d'un niveau d'abstraction à un autre, on parle de la transformation des modèles. Par exemple, un modèle concret (e.g. modèle de codage) peut être obtenu par la transformation d'un modèle abstrait (e.g. modèle de conception) en raffinant ce dernier par l'ajout de plus de détails (e.g.

détails concernant une plateforme technologique). Cette possibilité apporte des bénéfices remarquables car le fait qu'on soit capable d'automatiser le processus de développent et de maintenance par la transformation automatique des modèles, engendre un gain important en temps et en coût.

Ces avantages apportés par l'approche MDA ont dégagé un courant de généralisation qui considère qu'un système logiciel n'est qu'un ensemble de modèles donnant chacun une facette de ce dernier, et que les activités liées à l'ingénierie des logiciels et leur maintenance ne sont en fait que des transformations de modèles [Béz 04].

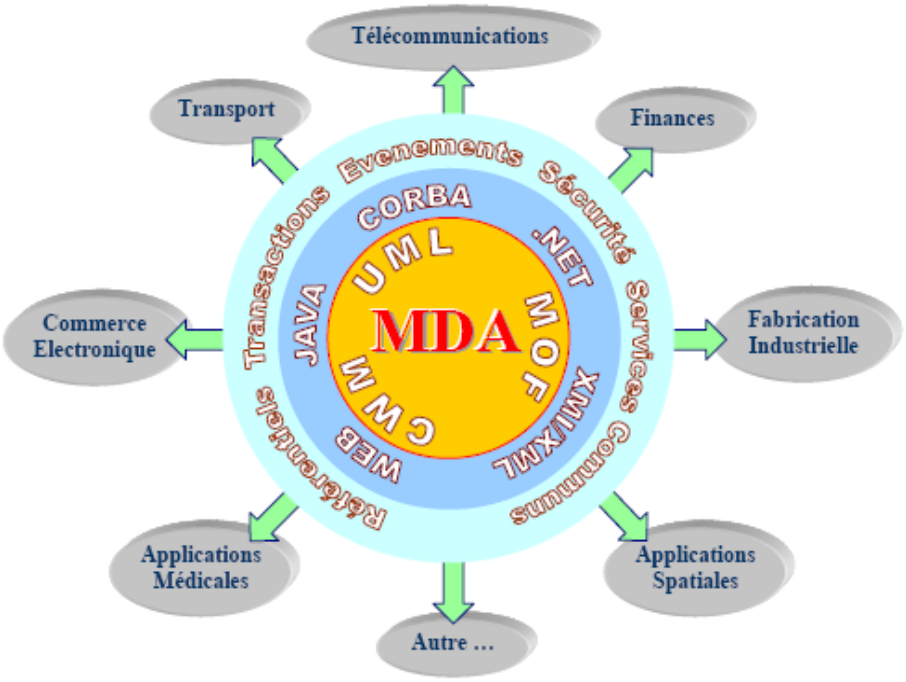


Figure 2.1. Architecture du MDA.

2.2 Standards et processus de MDA

L'approche MDA se base sur plusieurs standards universellement utilisés dans le domaine de génie logiciel, et vise un large éventail d'applications. La figure 2.1 présente une vision générale de MDA. Elle illustre que l'architecture du MDA se découpe en quatre couches. Au centre, on cite le standard UML (*Unified Modeling Language*), MOF (*Meta Object Facility)* et CWM (*Common Warehouse Metamodel*). La deuxième couche contient le

standard XML-XMI (*XML Metadata Interchange*) et un ensemble de middlewares (Java, .Net, CORBA et web services). La troisième couche propose des services permettant de gérer les transactions, la sécurité, les évènements, les référentiels et autres services. La dernière couche contient des frameworks spécifiques au domaine d'application (médecine, Transport, Finance, commerce électronique, Télécommunication, Espace, manufacture,…).

2.2.1 Les Standards de L'OMG

L'approche MDA se base sur un ensemble de standards adoptés par l'OMG, incluant UML, MOF, XMI/XML et CWM, qui permettent la création et la gestion des modèles (voir [Bla 05] pour plus de détail).

UML

UML (*Unified Modeling Language*) est un langage graphique et visuel pour la spécification, la construction et la documentation de tous les éléments (artefacts) d'un système logiciel [Boo 05]. UML peut aussi être considéré comme un langage de modélisation général qui peut être utilisé avec les méthodes basées objets et basées composants, appliqué dans tous les domaines d'application (e.g. santé, finance, télécommunication, etc.) et accepté par toutes les plateformes d'implémentation (e.g. J2EE, .NET).

UML fournie un ensemble de diagrammes (i.e. une représentation graphique) qui représente chacun un aspect particulier du système à modéliser [Boo 05]. La combinaison de différents diagrammes permet d'obtenir une vue complète de tous les aspects caractérisant un système informatique (aspect statique, aspect dynamique, aspect fonctionnel). Dans ses versions précédentes, UML n'a adopté que neuf diagrammes. Plus tard, UML dans sa version 2.0 est enrichi par quatre nouveaux diagrammes (voir figure 2.2).

Le langage UML occupe une place importante dans MDA. L'approche MDA préconise UML comme étant le langage utilisé pour créer les modèles indépendants des plateformes par le fait que UML est indépendant de toute plateforme d'implémentation, qu'elle soit J2EE, .Net, PHP, etc. En plus des modèles UML, MDA propose la définition d'une adaptation du langage UML à un domaine spécifique (e.g. domaine des EJB), on parle alors de la notion de profil UML. Un profil UML cible une plateforme d'exécution permettant, donc, d'adapter

UML à une plateforme d'exécution. Grâce aux profils UML, il est possible d'utiliser UML pour élaborer des modèles dépendants des plateformes d'implémentation.

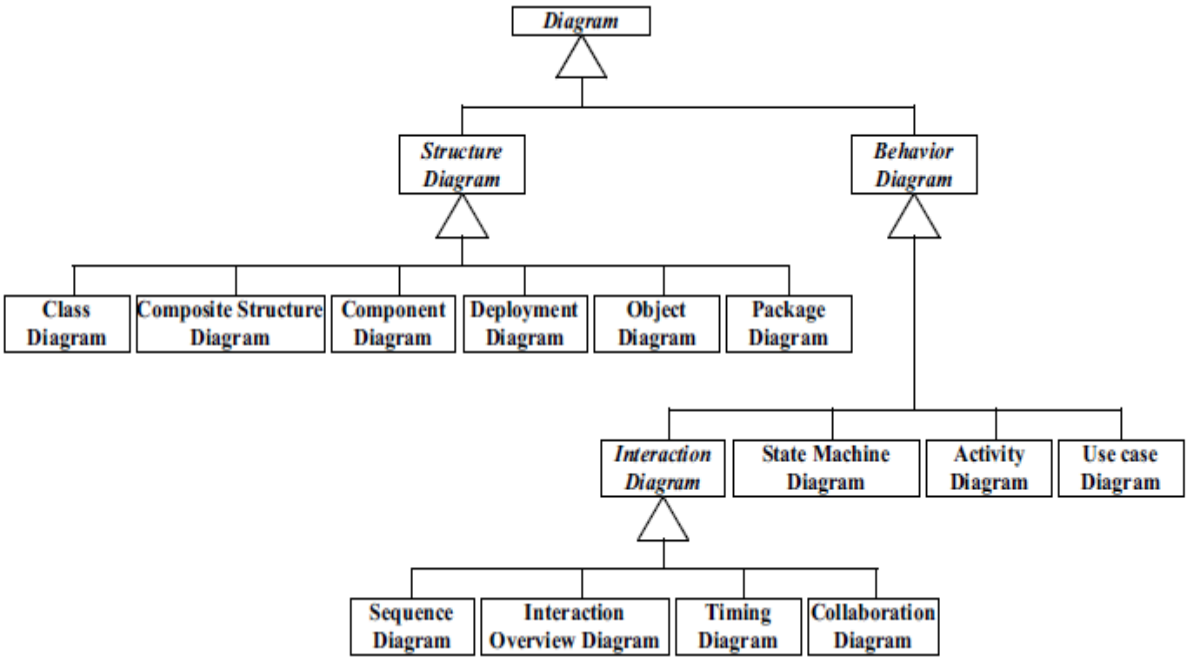


Figure 2.2. Les différents diagrammes UML.

UML est un langage de modélisation dans le sens où il définit un ensemble de concepts (i.e. éléments de modélisation) permettant la réalisation des modèles UML. En effet, le concept sur lequel UML est fondé est le métamodèle. Un métamodèle peut être défini comme une description formelle de tous les concepts d'un langage. Il permet de définir non seulement les éléments de modélisation, mais aussi la sémantique de ces éléments (i.e. leur définition et le sens de leur utilisation), ce qui limite grandement l'ambiguïté et incite la construction d'outils. Un métamodèle UML définit la structure qui doit avoir tout modèle UML. En d'autre terme, on dit un modèle UML tout modèle qui conforme au métamodèle UML. Il faut noter qu'un métamodèle UML est lui-même décrit par un méta métamodèle.

MOF

MOF (*Meta Object Facility*) est un formalisme de modélisation [Mof 14]. Un formalisme définit les éléments de modélisation nécessaire à l'élaboration des modèles ainsi que les relations entre ces éléments. On peut donc faire une analogie entre le formalisme de

modélisation et le métamodèle. MOF est considéré ainsi comme un *métaformalisme* dans le sens où il offre la possibilité de modéliser les formalismes de modélisation eux-mêmes (i.e. un formalisme de modèles de formalisme). MOF est appelé aussi un *métamétamodèle*. On distingue donc trois niveaux de modélisation : le niveau modèle, le niveau métamodèle et le niveau métamétamodèle. MDA ne monte pas dans les niveaux méta car il suffit d'utiliser un métamétamodèle MOF pour exprimer lui-même. En d'autre terme, un niveau méta plus haut n'est plus nécessaire. La figure 2.3 illustre les différents nivaux de MDA qui forme une architecture appelée *l'architecture à quatre niveaux* de MDA.

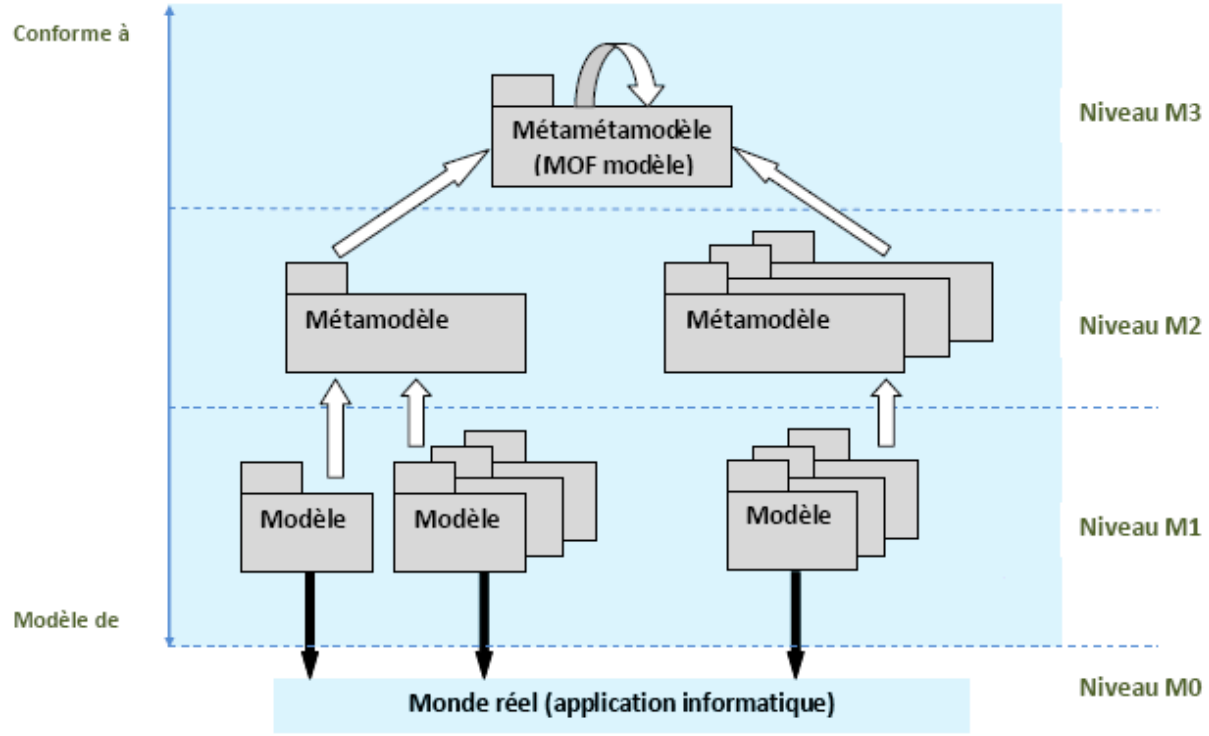


Figure 2.3. L'architecture à quatre niveaux de MDA.

Le niveau M0 contient les entités du monde réel à modéliser. Le niveau M1 comprend les différents modèles décrivant les entités du monde réels. Dans le niveau M2 se trouve les métamodèles exprimant les modèles du niveau M1. Le niveau M3 contient le méta-métamodèle qui permet non seulement de définir les métamodèles du niveau M2 mais aussi de décrire lui-même. Il reste à noter que MDA considère tous les éléments appartenant aux niveaux M1, M2 et M3 comme des modèles.

MOF représente les métamodèles sous forme de diagrammes de classes. Pour des raisons de différentiation, les classes qui constituent un métamodèle sont appelées métaclasses, et les objets instances des métaclasses sont appelés méta-objets. Une métaclasse possède un nom et contient des attributs et des opérations, aussi appelés respectivement méta-attributs et méta-opérations. Les métaclasses sont reliées par des méta-associations. Les méta-attributs et les paramètres des méta-opérations possèdent un type de donnée (*dataType*) qui peut être un booléen, un tableau, une chaîne de caractères, etc. La construction d'un type non primitif est possible grâce aux constructions fournies par MOF (énumérations, alias et structures). Les navigations entres les méta-objets est possible grâce au concept de référence. Il permet de naviguer à travers les méta-associations. Le concept de package (métapackage) regroupe les différents éléments d'un métamodèle.

Les concepts cités précédemment possèdent un nombre de propriétés :

- La métaclasse : a un nom, peut hériter d'une ou plusieurs classes, peut être abstraite, peut être considérée comme racine (*root*) ou feuille (*leaf*) d'un arbre d'héritage, etc.
- Le méta-attribut : a un nom, a un type (métaclasse ou type de données), possède une portée (*scope*), peut être ou non modifiable (*IsChangeable*), peut être considéré comme dérivé (*isDerived*), possède une multiplicité (*multiplicity*), peut être ordonné ou non (*is_ordered*) et l'information *is_unique* spécifie que l'ensemble des valeurs du méta-attribut ne contient pas de doublon.
- La méta-opération : possède un nom, possède une portée, contient zéro ou plusieurs métaparamètres, peut définir un type de retourne (métaclasse ou type de données), peut jeter une ou plusieurs exceptions, etc.
- La méta-association : possède un nom, contient forcément deux extrémités, etc.
- Le métapackage : a un nom, peut contenir zéro ou plusieurs métaclasses, contient zéro ou plusieurs autres métapackages, méta-associations qui relient les métaclasses et types de données nécessaires, peut hériter d'un ou plusieurs autres métapackages, peut importer un autre package, etc.
- La référence : a un nom, un type, une multiplicité, etc.

Vu qu'il n'existe pas une notation spécifique pour élaborer un métamodèle, on peut utiliser la notation UML (diagramme de classes UML). La figure 2.4 représente une partie de MOF (version1.4) sous forme de diagramme de classes. Ce diagramme définit la structure que doit avoir tout métamodèle, c'est-à-dire, il représente le métamétamodèle (modèle de MOF).

Dans sa version 2.0, le métamodèle MOF est constitué de deux parties : EMOF (*Essential MOF*), pour l'élaboration des métamodèles sans association, et CMOF (*Complete MOF*) pour les métamodèles avec associations. Comme il est presque impossible de faire la distinction entre un diagramme de classes représentant un métamodèle MOF et un diagramme de classes représentant un modèle UML, il est évident d'essayer de rapprocher ces deux standard au niveau des diagrammes de classes ce qui contribue à l'élaboration d'un métamodèle commun entre MOF et UML nommé UML2.0 Infrastructure [Bar 05].

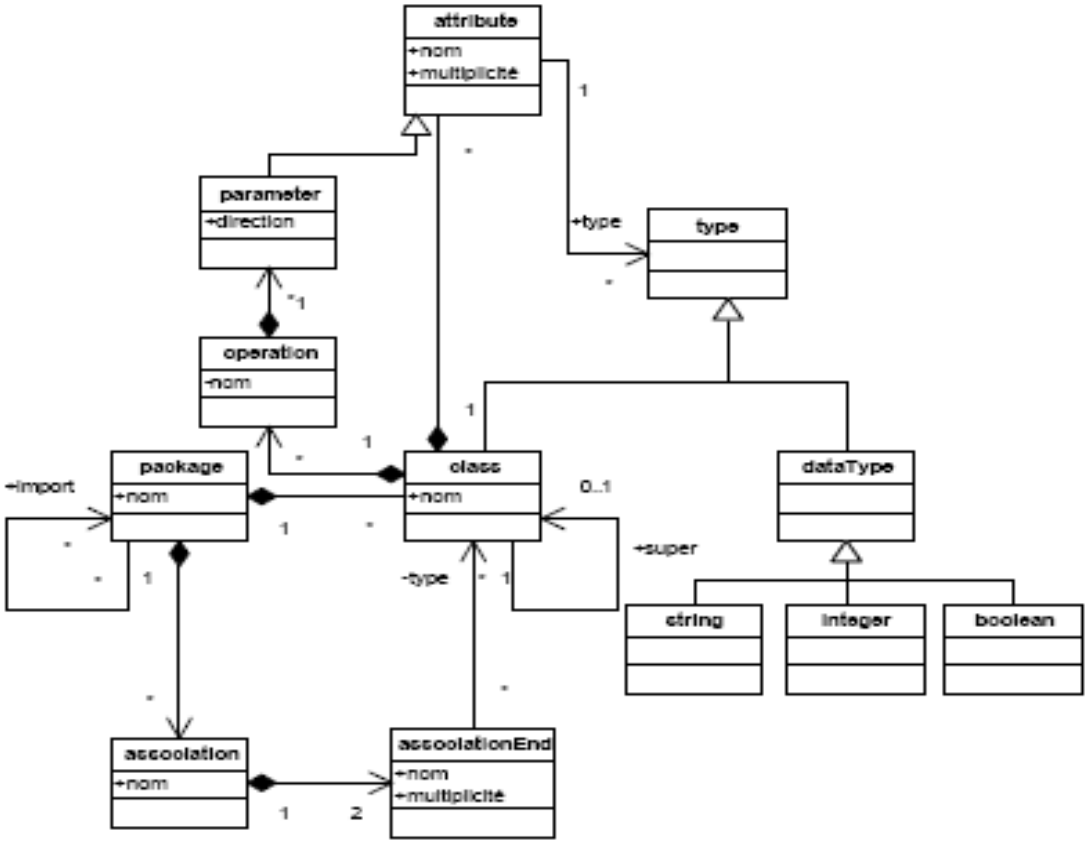


Figure 2.4. MOF (v1.4) sous forme de diagramme de classe [Bla 05].

En résumé, MOF est un standard de métamodélisation qui définit les éléments nécessaires de modélisation, la structure et la syntaxe des métamodèles utilisés pour réaliser des modèles.

OCL

OCL (*Object Constraint Language*) est un langage qui vise l'ajout de contraintes pour exprimer la sémantique statique des modèles et métamodèles [Ocl 12]. Un métamodèle (ou modèle) n'a pas toujours l'expressivité suffisante pour décrire toutes les contraintes sur les méta-éléments qu'il contient. L'OMG préconise l'utilisation du langage textuel OCL pour décrire des contraintes non capturées dans le métamodèle, sous forme d'expressions. L'ajout de ces contraintes permet de créer des modèles et métamodèles clairement et précisément définies et non ambiguës. Les expressions OCL sont rattachées à un contexte (i .e. un élément de modélisation) et peuvent être évaluées pour retourner une valeur (vraie ou faux). Elles n'ont aucun effet de bord et ne permettent ni de créer ni de supprimer ou modifier les objets d'un modèle. Les expressions OCL servent dans plusieurs situations :

- Dans le contexte d'une class UML, par exemple, une contrainte OCL peut être utilisée pour exprimer un invariant (mot-clé *inv*). Cela signifie que la contrainte OCL doit être tout le temps vraie pour toutes les instances de la classe.
- OCL fournit la possibilité de définir des pré-conditions (mot-clé *pre*) et post-conditions (mot-clé *post*) sur les opérations pour que le système modélisé reste dans un état cohérent quand on exécute ces opérations.
- OCL permet de contraindre la valeur retournée par une opération ou une méthode, et rend possible l'obtention de la valeur d'un attribue d'une classe (ou d'un méta-attribue d'une métaclasse).
- A travers le concept de collection, le langage OCL facilite la description des ensembles d'éléments et leur manipulation en fournissant un ensemble de fonctions. Les collections OCL ont plusieurs sortes selon l'ordre et le nombre d'apparition des éléments constituant l'ensemble décrite par la collection (Bag, Set, OrderedSet et Sequence).
- …

OCL fournit donc une bonne solution pour exprimer des contraintes sur les modèles mais pas pour décrire le comportement ou la sémantique des modèles (i.e. l'état de modèle reste inchangé). Il est important de noter que le langage OCL est indépendant de toute plateforme d'exécution et les contraintes OCL sont indépendantes des langages de programmation.

XMI

XMI (*XML Metadata Interchange*) est un standard qui permet de représenter concrètement des modèles sous forme de documents XML bien formé [Xmi 15]. Un document XML bien formé est construit moyennant le langage XML (*eXtensible Markup Language*) qui fournit un ensemble de concepts (e.g. les balises XML) permettant la structuration du document. Le document XML est un document textuel et est dit bien formé dans le sens où il est bien structuré. La traduction d'un modèle en un document XML est appelée sérialisation du modèle au format XML.

XMI fournit la possibilité de décrire les métamodèles en utilisant les DTDs XML (*DTD* pour *Document Type Definition*). Un DTD XML est un document textuel qui définit comment les balises XML sont structurées dans un document XML ainsi que leurs relations d'inclusion. Chaque modèle représenté par un document XML est conforme à un DTD correspondant (métamodèle). XMI définit un ensemble de règles qui permet de générer automatiquement une structuration de balises XML (e.g. DTD) à partir d'un métamodèle. La figure 2.5 illustre les opérations de généralisation et sérialisation.

CMW

CWM (*Common Warehouse Metamodel*) est un standard qui permet la définition des métamodèles qui décrivent à la fois des métadonnées métiers et des métadonnées techniques [Cwm 03]. Il facilite l'échange des métadonnées et les entrepôts de données (Warehouse), où se trouvent les métadonnées, entre les plateformes supports et les outils des entrepôts de données. CWM apporte l'interopérabilité entre divers logiciels par l'utilisation du format CWM de tel sorte qu'un logiciel émetteur traduit ses métadonnées vers le format CWM, et un logiciel récepteur extraire ses métadonnées par la traduction du format CWM reçu.

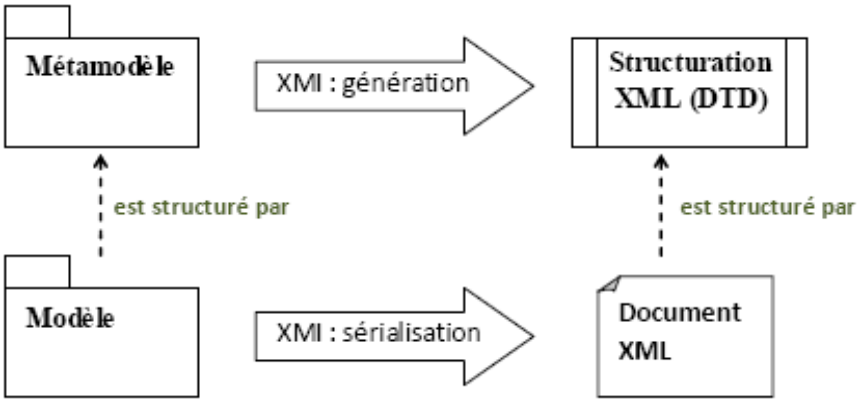


Figure 2.5. XMI et la structuration de balises XML [Bla 05].

Ces standards sur lesquels MDA s'est basé, s'enrichissent mutuellement et chacun peut apporter des concepts et des éléments dont les autres ont besoin. Le MOF fournit le concept de métamodèle et de métamétamodèle via lesquels nous pouvons définir la structure et la syntaxe de tous les métamodèles comme les métamodèle UML et CWM. Ainsi, MOF utilise le diagramme de classes défini par UML pour représenter les métamodèles. OCL, à son tour, enrichisse les diagrammes UML par une puissance d'expression afin de décrire des contraintes dont UML ne permet pas de les exprimées. OCL peut aussi être utilisé aux niveaux méta. Vu que les modèles sont des entités abstraites qui ne peuvent être ni stockées ni échangées sur un support informatique, XMI propose des solutions convenables pour résoudre ces problèmes dont les standards MOF et UML ne fournissent pas.

2.2.2 Processus MDA

Le principe clé de MDA consiste à utiliser les modèles aux différentes étapes du cycle de développement d'un système logiciel. MDA préconise la réalisation de différents types de modèles : les modèles d'exigences (*CIM*), les modèles d'analyse et de conception (*PIM*) et les modèles de code (*PSM*) [Bla 05].

- Le modèle d'exigences CIM (*Computation Independent Model*). Très tôt dans le processus de développement, MDA préconise l'utilisation des modèles. La première étape à effectuer dans tout processus de développement est la spécification des exigences. Un modèle d'exigence CIM est une représentation, de l'application à réaliser, qui exhibe les services fournis par cette dernière et les différents acteurs participants avec lesquels l'application

interagit. Un modèle d'exigence ne contient pas d'information sur comment réaliser l'application, ni sur les traitements, ni sur le fonctionnement et la structure de l'application. Il exprime tout simplement les demandes du client et permet donc de comprendre l'application et de vérifier, plus tard, si l'application élaborée est conforme aux besoins demandés ou non. Le modèle CIM est généralement représenté en utilisant un diagramme de cas d'utilisation défini par UML.

- Le modèle d'analyse et de conception PIM (*Platform Independent Model*). Les modèles PIM permettent de modéliser les étapes qui suivent la spécification des besoins, on parle de l'étape d'analyse et de conception. Les PIMs ont un haut niveau d'abstraction et sont indépendants de toute technologie d'implémentation (e.g. J2EE, .Net, PHP, etc.). Ils permettent d'exprimer les comportements et les fonctionnalités métiers de l'application, et ne s'intéressent pas aux détails concernant la technologie d'implémentation. Il existe plusieurs PIMs qui sont obtenus par des opérations de raffinement allant d'un PIM qui ne contient que les fonctionnalités métiers et les règles de gestion jusqu'à un PIM qui contient des informations sur la persistance, la gestion des transactions, la sécurité, la configuration, etc. Il sera donc plus facile de passer d'un PIM vers un modèle dépendant de plateforme. UML est considéré comme le plus approprié pour représenter les modèles PIMs par le fait qu'il est indépendant des plateformes d'exécution. Un modèle PIM peut être exprimé par un diagramme de classes qui peut être couplé avec un langage de contrainte tel que OCL. Il est possible aussi d'utiliser d'autres diagrammes pour matérialiser les différents aspects du modèle (e.g. le diagramme d'activité pour décrire le comportement).
- Le modèle de code PSM (*Platform Specific Model*). Une fois les modèles PIMs réalisés, l'étape qui aboutie à la génération de code commence. Les PSMs sont des modèles liés à une plateforme d'exécution et contiennent toutes les informations permettant l'exploitation de cette plateforme. Le rôle principal d'un PSM est de faciliter l'obtention du code à partir d'un modèle PIM. Les détails liés à la plateforme d'exécution ainsi que la représentation structurée décrivant le code rendent la génération de code à partir d'un modèle de code une tâche facile, voire triviale. Il existe plusieurs PSMs qui sont obtenus par transformations successives du premier modèle obtenu après la transformation de PIM en PSM. Un PSM sera donc raffiné continuellement jusqu'à l'obtention du code dans un langage de

programmation spécifique (Java, C++, etc.). MDA propose d'utiliser les profils UML pour élaborer des modèles de code, vu qu'ils sont adaptés à un domaine particulier et peuvent représenter adéquatement les PSMs.

Le processus MDA (figure 2.6) consiste en une succession d'étapes dont les modèles et les transformations des modèles jouent un rôle principale : construction du CIM, transformation en PIM, transformation en PSM, puis génération du code [Mes 06].

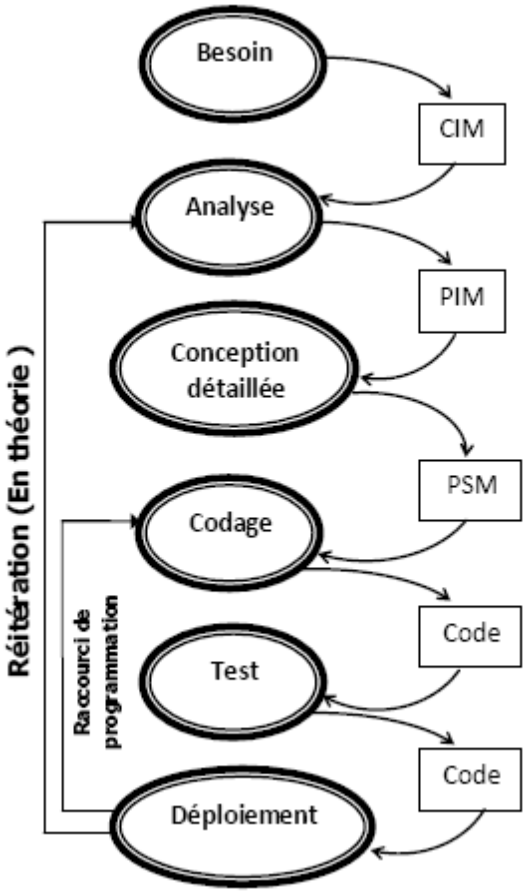


Figure 2.6. Processus de développement MDA [Mes 06].

1- Création de CIM. L'élaboration d'une nouvelle application commence toujours par une étape de spécification des exigences. Cette étape contribue à la construction d'un ou plusieurs modèles d'exigence (CIM).

2- Création de PIM. Une fois les modèles CIMs créés, on arrive à l'étape d'analyse et de conception abstraite. À ce niveau de développement, les modèles PIMs sont construits

partiellement à partir des CIMs. C'est dans cette étape que la logique métier de l'application est exprimée.

3- Création de PSM. Dans l'étape de conception détaillée, un mapping de PIM vers une plateforme d'implémentation est effectué. Le résultat de cette étape est la création des PSMs par la transformation des PIM en ajoutant les informations concernant la plateforme.

4- Génération du code. Les modèles de code élaborés à l'étape précédente sont suffisamment détaillés pour faciliter la génération du code (i.e. traduction en un formalisme textuel). Grâce à des outils de transformation, la génération du code peut être effectuée automatiquement.

Les avantages du processus MDA par apport au processus de développement classique résident dans le fait que l'utilisation de modèle comme un point de référence tout au long le cycle de développement a grandement consacrée la séparation des préoccupations, ce qui conduit à une pérennisation de la logique métier des entreprises. Ainsi, la possibilité d'automatiser les transformations de modèles a remarquablement augmentée la productivité de cette logique métier. Le passage de la conception à l'implémentation est plus facile grâce à l'intégration des informations concernant les plateformes d'exécution dans les transformations de modèles. Les transformations jouent donc un rôle central dans les processus MDA.

2.3 Transformation de Modèles

Les transformons de modèles sont au cœur de l'approche MDA. OMG définit la transformation de modèles dans le contexte de MDA par le processus qui convertit un modèle à un autre modèle d'un même système [Bie 10]. Une définition plus générale considère la transformation de modèles comme une fonction qui prend en entrée un ou plusieurs modèles sources et qui fournit en sortie un ou plusieurs modèles cibles [Bie 10]. Les éléments qui composent une transformation de modèles sont illustrés à la figure 2.7.

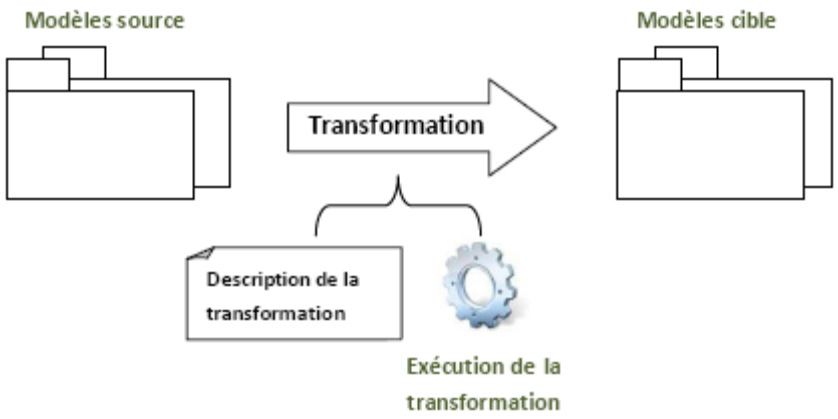


Figure 2.7. Les éléments d'une transformation de modèles.

Toute transformation de modèles doit contenir ces éléments [Bie 10]:

- Modèle source. Chaque transformation peut avoir un ou plusieurs modèles sources en entrée. Le modèle source est conforme à un métamodèle source.
- Modèle cible. Chaque transformation peut avoir un ou plusieurs modèles cibles. Le modèle cible est conforme à un métamodèle cible. Le terme modèle cible est seulement utiliser dans le contexte d'une transformation de type modèle-à-modèle.
- Description de la transformation. La description de la transformation est écrite dans un langage de transformation de modèles qui fournit un vocabulaire et une grammaire avec une sémantique bien définie pour réaliser des transformations de modèles. Le langage de transformation de modèles est basé sur une approche de transformation qui peut être impérative, opérationnelle, fonctionnelle, déclarative ou relationnelle. La plupart des langages de transformation sont basés règles. En d'autre terme, la transformation de modèle n'est qu'un ensemble de règles de transformation.
- Exécution de la transformation. L'exécution d'une transformation de modèles peut être effectuée par un outil de transformation (e.g. SmartQVT, ATL, etc.) qui interprète la description de la transformation. Il consiste à appliquer la description sur le modèle source pour obtenir le modèle cible. Les étapes typiques de l'exécution sont :

    1- Identifier les éléments à transformer dans le modèle source.

    2- Pour chaque élément identifié, produire l'élément cible associé.

3- Produire l'information de traçabilité qui relie les éléments source et cible affectés par cette règle.

La description de la transformation peut être exprimée par un modèle. MDA préconise la modélisation des transformations de modèles elles-mêmes. Tout comme les modèles d'entrée et de sortie, un modèle de transformation est conforme, à son tour, à un métamodèle de transformation qui permet d'exprimer les liens entre les concepts des métamodèles d'entrées et celles des métamodèles de sortie. La figure 2.8 montre les différents niveaux méta d'une transformation de modèles.

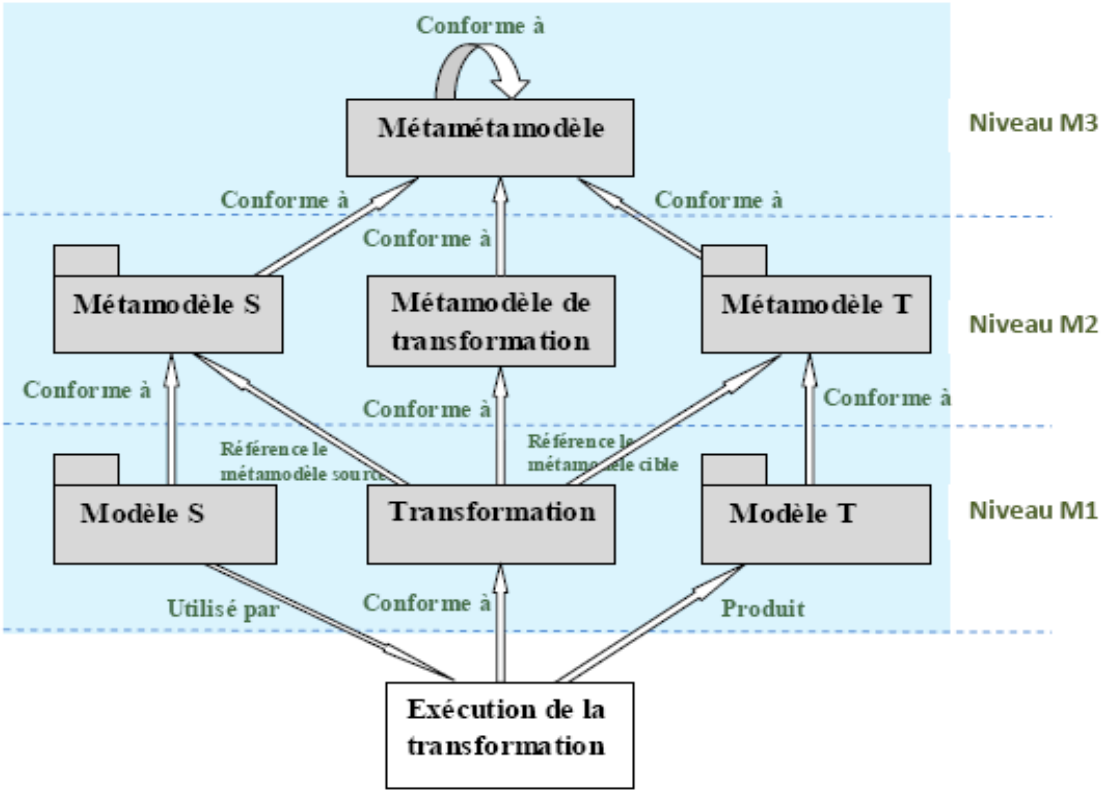


Figure 2.8. Les niveaux méta de la transformation de modèles.

2.3.1 Plan de classification pour les problèmes de transformation de modèles

Les langages de transformation de modèles ont été classifiés par Czarnecki et al [Cza 06], et par Mens et al [Men 06]. Ainsi, différents outils de transformation de modèles ont été évalués par Huber [Hub 08]. Il a conclu qu'il n'y a pas un outil de modélisation meilleur que

l'autre, mais presque pour chaque type de problème de transformation de modèles, il existe l'outil de transformation de modèles approprié. On désigne par problème de transformation de modèles le problème qu'on veut résoudre en utilisant une transformation de modèles. Un plan de classification de problèmes de transformation est alors proposé pour faciliter le choix du langage et l'outil de transformation les plus appropriés pour résoudre un problème donné [Bie 10]. La classification est basée sur un ensemble de propriétés qui caractérisent les problèmes de transformation :

Changement d'abstraction. La transformation de modèles peut changer le niveau d'abstraction entre le modèle source et le modèle cible. Selon le type de transformation, Le niveau d'abstraction des modèle peut être changé ou pas. On distingue deux types de transformation :

- La transformation verticale qui change le niveau d'abstraction. Le changement peut réduire le niveau d'abstraction par l'ajout de détails au modèle source, on parle d'une transformation de raffinage. Le changement peut aussi augmenter le niveau d'abstraction par la suppression des détails, on parle d'une transformation d'abstraction.
- La transformation horizontale ne change pas le niveau d'abstraction de modèle. Cependant, elle peut modifier sa représentation (conformément à un métamodèle différent).

Changement de métamodèles. Selon les métamodèles source et cible, on définit deux types de transformation :

- Transformation endogène. Les modèles source et cible de la transformation sont conformes au même métamodèle.
- Transformation exogène. Les modèles source et cible de la transformation sont conformes à des métamodèles différents.

Le nombre de modèles. Une transformation de modèle peut avoir plusieurs modèles sources et plusieurs modèles cibles. Le nombre minimum de modèles inclus dans la transformation est égale à un, dont les modèles source et cible sont les mêmes. La transformation ne change qu'une partie spécifique du modèle source (*In-place transformation*). La majorité des

transformations, cependant, incluent deux modèles. Elles génèrent un modèle cible différent du modèle source (*Out-place transformation*). Le dernier type de transformation peut inclure plusieurs modèles sources et combiner les informations trouvées dans ces derniers pour produire un modèle cible (intégration). Plusieurs modèles cibles peuvent aussi être produits par ce genre de transformation.

L'espace technologique. Les modèles sont représentés en utilisant différents espaces technologiques. Exemples : XML (XSD), MOF (OMG), EMF (Eclipse), DSL (Microsoft), etc. l'espace technologique des modèles limite l'ensemble d'outils de transformation qui peuvent être utilisés.

Le type cible. Le type cible d'une transformation peut être soit un modèle, ou bien, un texte.

Préservation des propriétés. Parfois, les modèles source et cible possèdent une propriété commune qui ne subit aucun changement lors de la transformation. Cette préservation peut être de divers types :

- Préservation de la sémantique. Il s'agit d'une transformation qui change un modèle en préservant son sens. Par exemple, la transformation qui vise l'amélioration des performances ou la refactorisation (la restructuration) est une transformation qui préserve la sémantique des modèles.
- Préservation de la syntaxe. La syntaxe définit une notation ou un format selon lequel le modèle est représenté. Généralement, la transformation qui préserve la syntaxe est horizontale et endogène.

2.3.2 Outils et Langages de transformation de modèles

Plusieurs outils et langages sont fondés pour réaliser les transformations et les mettre en œuvre.

QVT

QVT (*Query View Transformation*) est un langage standardisé de transformation de type modèle-à-modèle (*model-to-model transformation language*) adopté par OMG en 2005

[Qvt 11]. Il permet la définition de métamodèles permettant l'élaboration des modèles de transformation. QVT doit permettre de :

- Soumettre des requêtes (*Query*) sur des modèles. Les requêtes permettent la sélection des éléments sur un modèle. Le langage utilisé par QVT est OCL, légèrement modifié et étendu avec une syntaxe différente et simplifiée.
- Produire des vues des métamodèles (modèles déduits d'un autre pour en révéler des aspects spécifiques). Elles peuvent être définies en utilisant une query. Une vue est éventuellement conforme à un métamodèle restreint qui lui est spécifique.
- Opérer des transformations sur les modèles.

La syntaxe abstraite de QVT (métamodèle) est décrite en MOF et sa syntaxe concrète est textuelle ou graphique [Bla 05]. QVT supporte plusieurs scenarios d'exécution : transformations unidirectionnelles, transformations bidirectionnelles, etc.

Grâce à une architecture hybride, QVT propose les avantages combinés des approches déclaratives et des approches impératives. De part ses capacités d'expression, la partie déclarative permet dans de nombreux cas courant d'écrire facilement des transformations entre modèles, en particulier lorsque la transformation est constituée d'un ensemble de règles relativement simples. Néanmoins certains types de transformations se prêtent mal à un style déclaratif. Pour ce genre de transformation, QVT propose alors un style impératif.

Pour un style déclaratif, QVT utilise deux langages différents : Un langage de haut niveau nommé *Relations* qui consiste à effectuer une correspondance entre les éléments des modèles source et cible de la transformation, et un langage de bas niveau nommé *Core* qui est plus simple mais avec le même pouvoir d'expression.

Pour un style impératif, QVT utilise le langage impératif *Operational Mappings* qui permet la mise en œuvre d'une relation et l'ajout de primitives déclaratives inspirées en partie d'OCL. Operational Mappings définit des transformations unidirectionnelles exprimées de manière impérative. Il définit une signature indiquant les modèles impliqués dans la transformation et définit une opération principale pour son exécution (*main*). Ce langage étend le langage Relations avec des constructions impératives et des expressions OCL à effets de

bords. QVT inclut ainsi une implémentation nommée *Black Box* (boite noire) qui permet de connecter et d'exécuter du code externe durant l'exécution de la transformation. Ce mécanisme permet d'implémenter des algorithmes complexes et aussi la réutilisation de bibliothèques externes. La figure 2.9 illustre l'architecture de QVT. Dans la littérature, beaucoup d'outils implémentent le standard QVT. Parmi eux, nous citons : *SmartQVT*, *MediniQVT*, etc. [Dia 09].

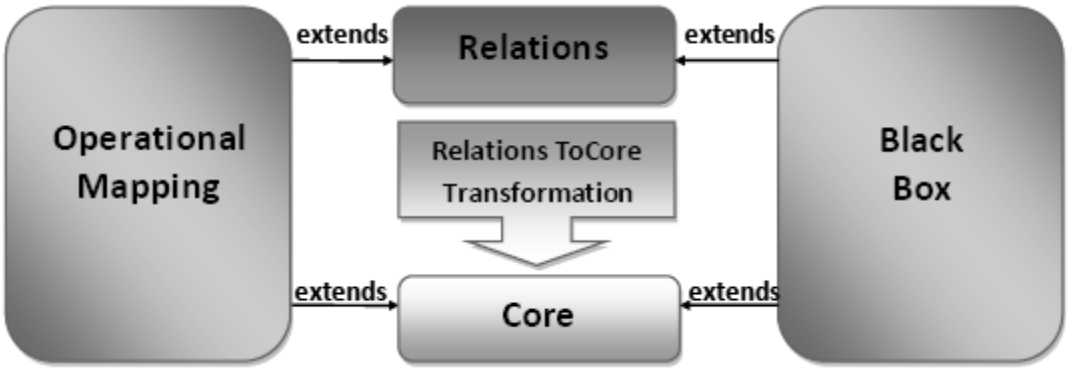


Figure 2.9. Architecture de QVT.

ATL

ATL (*Atlas Transformation Language*) est un langage de transformation de type modèle-à-modèle proposé par le groupe ATLAS [Atl 16]. ATL est considéré comme un langage hybride puisqu'il mélange les deux types de constructions, déclarative et impérative. Une transformation ATL est composée d'un ensemble de règles qui décrivent comment initialiser et créer les éléments constituant un modèle cible. Le langage est défini par un modèle MOF pour sa syntaxe abstraite et possède également une syntaxe concrète présentée sous forme textuelle.

ATL utilise les expressions OCL à effet de bord pour manipuler et accéder aux éléments d'un modèle. Ces expressions peuvent retourner une valeur de type simple ou complexe (élément d'un modèle, collection, etc.). Elles permettent la navigation entre les différents éléments d'un modèle et d'appeler des opérations sur ceux-ci. La navigation ne peut se faire que sur des éléments initialisés, on ne pourra jamais naviguer sur des éléments du modèle cible.

L'exécution d'une transformation est effectuée par un moteur de transformation ATL qui est une machine virtuelle. Comme la *Java Virtual Machine*, la machine virtuelle ATL dispose de son propre jeu d'instruction. Elle est indépendante du langage de transformation ATL et ne s'intéresse qu'au code de la transformation compilée.

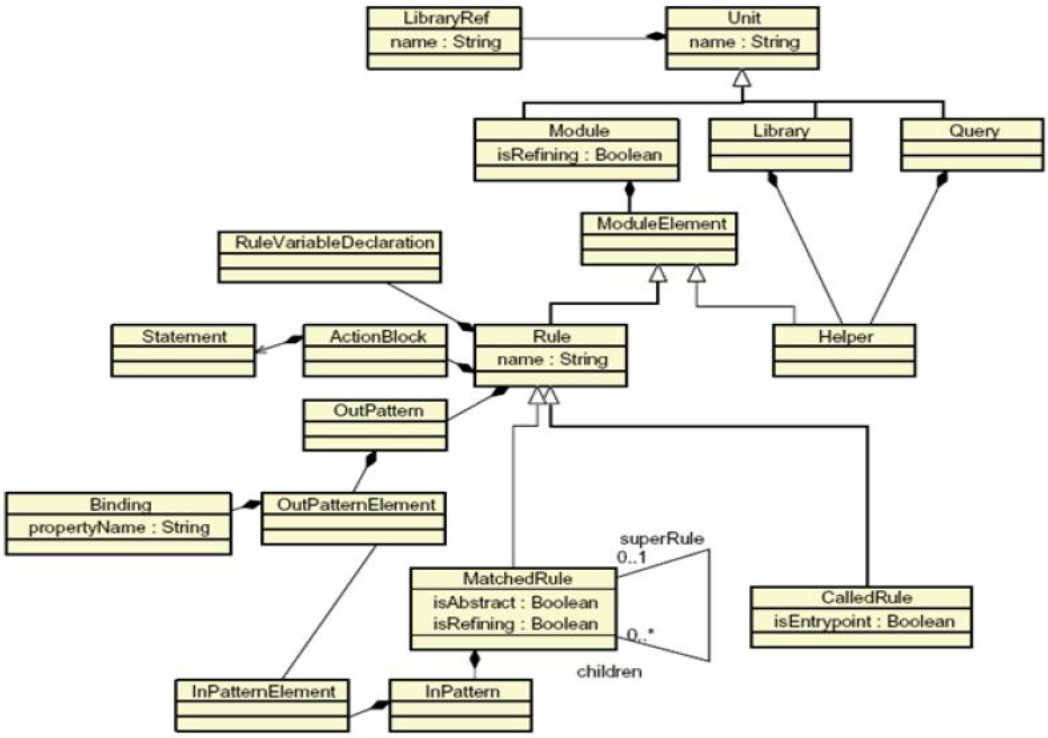


Figure 2.10. Métamodèle ATL simplifié.

Un modèle de transformation ATL est conforme au métamodèle ATL (Figure 2.10). La définition d'un modèle de transformation se fait dans un fichier portant l'extension " .atl ". L'entête de fichier se déclare de la manière suivante :

```
Module <nom de module>
Create OUT : <métamodèle cible> from IN: <métamodèle source>
```

Listing 2.1. Déclaration de l'entête de la transformation.

La section de l'entête définit le nom de module de transformation et le nom des variables qui correspondent aux modèles source et cible. Elle décrit ainsi le mode d'exécution de module. Le mode décrit dans le listing au-dessus est un mode normal. On peut exécuter le module en mode de raffinement (*refining mode*) dans le cas où la transformation souhaitée

nécessite la modification d'une petite partie du modèle. Les éléments qui n'ont pas subi de modification dans le modèle source seront copiés au modèle cible. Le développeur ne spécifie que les changements qui seront effectués sur le modèle source.

Pour décrire la transformation d'un élément du métamodèle source en un élément du métamodèle cible, on utilise une règle qui se définit selon la syntaxe :

```
rule <nom de la règle> {
from
<pattern source>
to
<un ou plusieurs patterns cibles>
}
```

Listing 2.2. Déclaration d'une règle de transformation.

Le pattern cible se constitue d'un ensemble de déclarations de variables qui correspondent aux éléments cibles à générer, suivies d'attachements qui indiquent pour chaque élément cible, comment instancier ses propriétés :

```
<nom de l'élément cible> :<type> mapsTo <élément source associé>(
<attachement>
 . . .
)
```

Listing 2.3. Pattern cible.

Un attachement a la forme :

```
<propriété de l'élément cible> <- <élément source>
```

Listing 2.4. La forme d'un attachement.

Implicitement, la propriété de l'élément cible généré prendra comme valeur le résultat de la transformation de l'élément source selon la règle qui lui correspond.

ATL fournit également la possibilité de créer et de détruire des éléments intermédiaires selon les besoins de la transformation, ainsi que des constructions conditionnelles (*if, switch*), et itératives (*while*) avec une syntaxe proche de Java.

Le concept de « *helper*» permet de définir des méthodes avec des paramètres et un type de retour, et des attributs (sans paramètres). Le corps d'un helper est défini à l'aide d'expressions OCL, il prend la forme suivante :

```
helper def :<nom de l'opération> (<paramètres> :<type>) : <type de
retour> =
 <expression OCL>;
```

Listing 2.5. Méthode en ATL.

```
helper def :<nom de l'attribut>: <type de retour>=
<expression OCL>;
```

Listing 2.6. Attribut en ATL.

ATL est disponible en tant que plugin dans le projet Eclipse M2M [Atl 16]. Il ne supporte que les transformations unidirectionnelles et est adapté pour interpréter des métamodèles décrits à l'aide d'EMF (*Eclipse Modeling Framework*) ou MDR (*Meta Data Repository* de NetBeans). Afin d'assurer l'indépendance d'ATL par rapport aux autres outils de modélisation, le plugin ATL met à disposition le langage KM3 (*Kernel Meta-Metamodel*), qui est une forme textuelle et simplifiée du EMOF qui permet de décrire des métamodèles et des modèles. Les modèles décrits au format KM3 peuvent être transformés en un format XMI Ecore, le format qui correspond dans l'EMF au modèle MOF. EMF, à son tour, fournit un éditeur qui permet de manipuler les modèles décrits au format XMI sous forme arborescente avec des possibilités de mise à jour.

Kermeta

Kermeta est un langage de métamétamodélisation développé à l'Université de Rennes [Ker 14]. Il dispose d'un environnement de développement de métamodèles basé sur EMOF dans l'environnement Eclipse. Il permet non seulement de décrire la structure des meta-modèles, mais aussi leur comportement. Dans Kermeta, un langage spécifique est défini pour écrire des transformations. La syntaxe de transformations en Kermeta est proche de Java et dispose d'une structure uniquement impérative pour écrire des transformations.

MIA-T

MIA-T (*Model-In-Action Transformation*) [Mia 16] est un outil de transformation de modèles développé par *Mia-Software*, filiale de *Sodifrance*. C'est à la fois un moteur de transformation, un dépôt de métamodèles et un environnement de développement. Cependant, il est possible d'utiliser le moteur de transformation en dehors de cet environnement via l'API disponible. Un projet de transformation est composé d'un ensemble de règles. Ces règles sont décomposées en deux parties : une requête qui sélectionne les éléments dans les modèles sources et une action qui créée et initialise les éléments dans les modèles cibles. Ces deux parties peuvent-être écrites dans des langages différents en fonction des besoins. MIA-T possède trois langages pour l'écriture des transformations : deux langages déclaratifs, à savoir *MIA-TL* et *RL-TL*, ainsi que Java qui peut-être appelé pour des traitements complexes nécessitant un langage impératif.

Autres

Plusieurs langages et outils de transformation de modèles existent actuellement [Bie 10], nous citons : *XMF-Mosaic*, *Moflon*, TCS (*Textual concrete Syntax*), *XText*, *Tefkat*, *Lxfamily*, *MOLA*, *MT*, *SiTra*, *ModfLog*, *GreAT*, *GenGen*, *BesBag*, *UMT*, *UMLX*, *ATOM VIATRA*, *BOTL*, *XDE Transformations*, *Codagen Architect Transformations*, *OptimaJ Transformations*, *MPS Transformation*, *Microsoft DSL Tool Transformations*, *AndroMDA*, *JET*, *FUUTje*, *GMT*, *Jamda*, *Fujaba Transformations*, *TXL*, *Stratego*, etc.

2.4 Conclusion

Dans ce chapitre, nous avons présenté l'approche MDA et les concepts de modèle, métamodèle, métamétamodèle et transformation de modèles. Nous avons expliqué comment l'utilisation de modèles tout au long le cycle de développement et de maintenance peut résoudre pas mal de problèmes (complexité, compréhension, productivité, etc.). Nous avons présenté aussi l'architecture générale de MDA et les différents standards d'OMG sur lesquelles cette architecture est basée. Ainsi, le processus de développement basé sur l'approche MDA est présenté et les différentes étapes et artefacts produits lors de ce processus sont identifiés. Nous avons clairement montré que la transformation de modèles est au cœur de

l'approche MDA. Elle permet d'aller d'un niveau d'abstraction très élevé (CIM) jusqu'à un niveau d'abstraction bas (code). La transformation de modèles peut être de différents types (horizontale, verticale, endogène, exogène, etc.) et le choix entre eux se fait selon le problème posé. Plusieurs outils et langages de transformation de modèles existent actuellement dont chacun est considéré comme approprié pour tel ou tel problème. Nous avons donc présenté un ensemble de langages et outils parmi les plus connus de nos jours.

Dans le chapitre suivant, nous présenterons les concepts de base de l'architecture logicielle et nous étudions quelques langages de description d'architecture.

3 Architectures Logicielles

L'un des paradigmes les plus récents dans l'industrie logicielle est l'ingénierie à base de composant (*CBSE* ou *Component Based Software Engineering*) [Bac 00]. Selon ce paradigme, la construction des systèmes logiciels se fait par l'assemblage des composants logiciels plus ou moins standardisés. La notion du composant logiciel n'est pas nouvelle. Elle a été présentée pour la première fois par Mcllroy en 1968 dans son article intitulé « Mass product Software Component » lors de la conférence NATO Software Engineering [Mcl 68]. Il a prévu une production massive de composants logiciels standards (composants sur l'étagère ou *COTS* pour *Component Off The Shelf*) qui seraient utilisés pour produire les logiciels. Selon Mcllroy, le développeur ne fait que combiner les composants au lieu de créer les parties logicielles voulues à nouveau. Néanmoins, ce n'est que dans les deux dernières décennies que l'intérêt pour cette approche s'est accentué. Cette orientation vers le composant est vue comme une évolution logique et naturelle du paradigme objet du fait que ce dernier, avec ses notions d'encapsulation, de polymorphisme et d'héritage, n'a résolu qu'un sous ensemble des problèmes engendrés par l'augmentation permanente de la complexité des logiciels (i.e. développement, maintenance et évolution de plus en plus difficiles) [Mey 00]. L'approche par composants vise donc la résolution des lacunes de l'approche par objets.

La notion de composant et celle de l'architecture logicielle sont étroitement liées de par le fait qu'une grande partie des travaux qui s'articulent autour des architectures logicielles ciblent la fourniture des briques de base réutilisables, assemblables et autonomes, i.e. composants logiciels, pour faciliter la construction des applications de haute qualité et de façon plus rapide et moins coûteuse [Bar 08]. Les architectures à base de composant mettent l'accent sur la description de l'assemblage des composants et leurs interactions. Plusieurs modèles sont proposés pour la description des composants et leur assemblage, aussi bien dans le monde académique que dans le monde industriel. Malgré la diversité des modèles proposés, on distingue des concepts clés sur lesquels reposent les différentes architectures logicielles, à savoir les composants, les connecteurs et les configurations [Med 00]. Ces concepts sont formalisés à travers l'utilisation des langages de description d'architecture (*Architecture Description Language* ou ADL) [Med 00] dont l'objectif principal est la spécification formelle

des architectures logicielles. Dans ce chapitre, nous passons en revue le vocabulaire associé à l'architecture logicielle puis nous citons quelques langages de description d'architecture parmi les plus connus dans la littérature.

3.1 Terminologies des Architectures Logicielles

L'architecture logicielle a émergé comme une sous-discipline importante du génie logiciel. Cette importance est essentiellement liée aux principes de génie logiciel dont l'architecture logicielle tend à consacrer, à savoir l'abstraction, la modularité et la réutilisation [Ghe 02]. Dans le contexte d'un processus de développement logiciel, une architecture logicielle peut être vue comme une passerelle entre l'étape de spécification des besoins et celle d'implémentation. Elle fournit une représentation abstraite d'un système en matérialisant sa structuration. Elle est essentiellement utilisée pour concevoir, développer, documenter, analyser et maintenir les systèmes complexes. Avec le niveau d'abstraction élevé qu'elle apporte, l'architecture logicielle permet de faciliter la compréhension et vaincre la complexité croissante des systèmes logiciels.

Il existe plusieurs définitions de l'architecture logicielle. Bien qu'elle désigne un concept complexe qui ne peut pas être capturé par une seule définition, les définitions existantes issues des différents acteurs académiques et industriels vont dans le même sens: la représentation abstraite d'un système logiciel. Selon la définition introduite par Perry et Wolf, une architecture logicielle est « un ensemble d'éléments architecturaux (ou, si vous préférez, de conception) qui ont une forme particulière » [Per 92]. On distingue trois classes différentes d'éléments architecturaux: les éléments de traitement, les éléments de données, et les éléments de connexion. Les éléments de données sont des composants qui contiennent les informations utilisées dans le système et sont transformées par les éléments de traitement pour produire d'autres éléments de données. Les éléments de connexion sont les composants responsables de la mécanique d'interconnexion qui permet de relier les différents composants du système ainsi que la circulation des informations (i.e. les éléments de données) entre les éléments de traitement [Per 92]. Booch et al. définissent l'architecture logicielle comme « un ensemble de décisions importantes concernant l'organisation d'un système logiciel, la sélection des éléments structuraux constituant le système et leurs interfaces, leur comportement collaboratif,

leur composition, et le style architectural qui guide cette organisation » [Boo 05]. Une définition avancée par Shaw et Garlan indique que l'architecture logicielle concerne « la description abstraite des éléments constituant les systèmes logiciels, de leurs interactions, des patrons de conception guidant leur composition, ainsi que des contraintes associées à ces patrons » [Sha 96].

Garlan et Shaw ont contribué à la formation d'un consensus sur les concepts clés de l'architecture logicielle et les termes à utiliser pour les désigner comme ceux de « composant » et de « connecteur ». Dans [Gar 93], ils définissent l'architecture logicielle comme « une collection de composants de calcul – ou tout simplement de composants – accompagnée d'une description des interactions entre ces composants – les connecteurs ». Une autre définition proposée par Buschmann et al. considère l'architecture logicielle comme « une description des sous-systèmes et composants d'un système logiciel et leurs interactions. Les sous-systèmes et les composants sont typiquement spécifiés selon différentes vues pour exhiber les propriétés fonctionnelles et non-fonctionnelles du système » [Bus 96]. L'organisation IEEE propose sa définition normalisée selon laquelle l'architecture logicielle est « une organisation fondamentale d'un système représenté par ses composants, la relation des composants du système les uns avec les autres et avec l'environnement, et les principes et les directives pilotant leur conception et leur évolution » [Iee 00].

Bien entendu, ces définitions, parmi d'autres, partagent un point commun malgré leur diversité et la différente terminologie employée : celui de considérer qu'une architecture logicielle est une description abstraite d'un système logiciel en termes de composants logiciels et d'interactions entre composants. D'autres termes tels que la structure architecturale, le style architecturale, les propriétés fonctionnelles et non-fonctionnelles des éléments architecturaux, etc. sont plus au moins utilisés dans certaines définitions pour désigner différents concepts de l'architecture logicielle, mais d'une manière générale, les concepts fondamentaux, qui sont différemment interprétés parfois, sont toujours présents dans presque toutes les définitions proposées. Dans la suite, nous citons les concepts d'architecture logicielle qui sont à la fois importants et essentiels pour la compréhension de notre travail.

3.1.1 Les Composants

Comme l'architecture logicielle, le concept de composant logiciel n'a pas atteint un consensus, néanmoins, les définitions proposées s'accordent sur un ensemble de concepts et d'éléments architecturaux qui le caractérisent. Dans la littérature, la définition de Szyperski [Szy 98] est parmi les plus acceptées par la communauté de génie logiciel, vu qu'elle souligne les caractéristiques fondamentales du composant. Selon Szyperski, un composant logiciel est « une unité de composition dotée d'interfaces contractuellement spécifiées, et de dépendances explicites au contexte uniquement. Un composant logiciel peut être déployé indépendamment et est sujet à composition par un tiers » [Szy 98]. Les caractéristiques dont nous avons parlé peuvent être tirées de cette définition :

- Un composant logiciel est une unité de composition, c'est-à-dire elle peut être composée avec d'autres composants pour construire un système logiciel complexe;
- Un composant logiciel est une unité indépendante de déploiement. Le déploiement consiste à mettre en service le composant au sein de l'environnement opérationnel cible. Cela est possible par le fait que l'existence du composant est totalement indépendante du contexte d'utilisation, ce qui simplifie sa réutilisation par différents clients dans différents contextes (ou applications) sans aucune modification;
- Un composant logiciel peut communiquer avec son environnement et avec les autres composants uniquement à travers des interfaces explicites. En d'autres termes, il peut être vu de l'extérieur comme une boite noire qui encapsule des fonctionnalités. Cette séparation entre les fonctionnalités et leur implémentation via interfaces supporte grandement la modularité. D'un autre côté, la communication se fait correctement en respectant des contraintes exprimées sous formes de contrats portant sur les interfaces du composant.

Dans le même sens, Councill et Heinmann définissent un composant comme « l'élément logiciel qui est conforme à un modèle de composant et peut être déployé indépendamment et composé sans modification selon un standard de composition » [Hei 01]. Cette définition est semblable à celle de Szyperski mais entraîne deux nouveau concepts : le modèle de composant et le standard de composition. Les deux représentent généralement un

ensemble de règles et patrons de conception qui décrivent les éléments architecturaux caractérisant un type de composant ainsi que les interactions et les coordinations entre les composants de ce type.

3.1.1.1 Perspectives sur le cycle de vie du composant

La définition de Szyperski [Szy 98] ne spécifie pas précisément le niveau d'abstraction des composants et fait référence indirectement au concept d'abstraction à travers la notion d'encapsulation. Plusieurs formes de composant à différents niveaux d'abstraction, depuis la phase de conception jusqu'à la phase d'exécution d'une application, ne sont pas prises en considération. D'autres définitions établissent une distinction claire entre les composants dans les différentes phases de leur cycle de vie. Dans [Bac 00], la définition suivante est avancée: « Un composant logiciel fusionne deux perspectives distinctes : Considérés comme des implémentations, les composants peuvent être déployés et assemblés pour construire des systèmes (ou sous systèmes) plus complexes; Considérés comme des abstractions architecturales, les composants expriment les règles de conception qui imposent un modèle standard de coordination sur tous les composants. Ces règles de conception prennent la forme d'un modèle de composant, ou un ensemble de standards et de conventions auxquelles les composants doivent se conformer ». Deux types de composants peuvent être identifiés :

- Les composants architecturaux : Appelés aussi *composants de conception*, comme indiqué dans la définition, ils regroupent des règles de conception et des conventions pour la construction et l'utilisation des composants. On parle alors de modèles de composants.
- Les composants d'implémentation : Ils représentent des programmes ou des sous-programmes qui peuvent être déployés et assemblés.

Encore dans un autre travail [Lue 02], un composant d'implémentation et un composant déployable sont définis comme étant deux aspects différents : (1) « un composant est un élément logiciel qui encapsule une implémentation réutilisable d'une fonctionnalité, peut être composé sans modification et adhère à un modèle de composant » (2) « Un composant déployable est un composant pré-paqueté, distribué indépendamment, facile à installer et désinstaller, et auto-descriptif ». Cette distinction se base essentiellement sur la propriété de déploiement qui marque le début d'une nouvelle phase du cycle de vie d'un

composant logiciel. Selon ce point de vue, un composant simple ne contient typiquement que des interfaces qui définissent leur fonctionnalité en termes de méthodes fournies et requises. Un composant déployable, en revanche, contient des données auto-descriptives assistant le processus de composition. On parle de méta-données (*meta-data*).

Chessman et Daniels introduisent d'autres formes de composant issues de plusieurs points de vue sur le composant durant le cycle de vie d'un projet [Che 01] :

- Standard de composant. Comme dans les autres domaines (e.g. automobiles, électroniques, mécanique), pour pouvoir utiliser un composant préfabriqué dans l'assemblage d'un produit, il doit se conformer à un standard. Le standard de composant est un ensemble de règles et conventions définissant une famille de composants logiciels (connues aussi sous l'appellation du *modèle de composant*, que nous avons déjà introduit). On trouve dans cette catégorie : les JavaBeans [Sun 97], les EJB [Sun 09], les composants .NET [And 05], les composants COM de Microsoft [And 05], etc.

- Spécification de composant. Choisir un composant, qui est conforme à un standard particulier, n'est pas suffisant pour pouvoir l'utiliser proprement. L'utilisateur doit connaître ce que le composant peut faire, sinon il risque de mener à une mauvaise utilisation. Il devient alors indispensable de fournir une spécification claire des composants.

- Interface de composant. Une partie majeure de la spécification d'un composant est la définition d'une interface (voir section 3.1.1.2).

- Implémentation d'un composant. Il s'agit d'une mise en œuvre de la spécification d'un composant. D'un point de vue assemblage, la spécification d'un composant est plus importante que sa mise en œuvre car l'assemblage dépend de la première. La séparation claire entre la spécification et l'implémentation est une caractéristique souhaitable qui supporte la possibilité d'avoir plusieurs implémentations pour la même spécification, dans n'importe quel langage de programmation. Cela signifie qu'un changement au niveau implémentation, tant qu'il préserve la spécification, n'altère en aucun cas la fonctionnalité de l'entité assemblée.

- Composant installé. Le déploiement d'une implémentation d'un composant sur une machine mène à la création d'un composant installé.
- Composant objet. En exécution, seules les instances de composants installés effectuent réellement un traitement. Chaque instance possède une identité et se caractérise par un état (i.e. le contenu du composant lors de l'exécution). On appelle cette instance *le composant objet*.

Une tentative d'identification des formes de composants dans leur cycle de vie par Marvie [Mar 02] a introduit quatre notions :

- Type de composant. Il est caractérisé par trois éléments : les interfaces, les modes de communication avec les autres types de composants et les propriétés configurables. Les interfaces sont indépendantes des implantations et peuvent être fournies ou requises. Les interfaces fournies représentent les services fournis par le type de composant et consistent en une énumération des signatures des opérations ainsi que les données échangées (comme les interfaces des objets). Les interfaces requises sont les services nécessaires pour que le composant puisse accomplir les services qu'il fournit. Ces interfaces définissent un mode de coopération avec les autres types de composants. On distingue trois modes : le mode synchrone, le mode asynchrone et le mode diffusion en continu. Les types de composants peuvent avoir diverses propriétés configurables. L'intérêt de ces propriétés se porte sur la précision du comportement d'une instance de composant dans un contexte donné par la fixation d'un état initial (i.e. des valeurs initiales des propriétés). La configuration permet aussi de fournir les services requis par l'instance à partir du système.
- Implantation de composant. L'implantation de composant comprend deux parties : la mise en œuvre de sa logique métier et la prise en charge des propriétés non-fonctionnelles relatives à l'exécution telles que la persistance, la sécurité, la gestion des transactions, etc. La logique métier doit être programmée, cependant, la réalisation de la deuxième partie de l'implantation consiste à décrire les propriétés non-fonctionnelles qui seront automatiquement prises en charge.
- Paquetage de composant. Le paquetage de composant est une unité diffusée et déployée sous forme d'une archive contenant la définition d'un type de composant, une ou

plusieurs implantations de ce type et une description du contenu du paquetage. Cette forme permet la diffusion des composants exécutables et facilite leur intégration dans les applications de telle sorte que seule la connaissance de leur type est requise.

- Instance de composant. De même que les instances d'une classe d'objets, une instance de composant est une entité s'exécutant dans un système. Elle se caractérise par une référence unique, un type de composant et une implantation de ce type.

La table 3.1 donne un aperçu sur les définitions que nous avons citées dans cette section et les diverses perspectives vis-à-vis le cycle de vie du composant que nous avons identifié.

<table>
<tr><td>Définitions</td><td colspan="6">Perspectives</td></tr>
<tr><td>[Szy 98]</td><td colspan="6">Composant logiciel</td></tr>
<tr><td>[Bac 00]</td><td colspan="3">Composant architecturale</td><td colspan="3">Composant d'implémentation</td></tr>
<tr><td>[Che 01]</td><td>Standard de composant</td><td>Spécification de composant</td><td>Interface de composant</td><td>Implémentation de composant</td><td>Composant installé</td><td>Composant objet</td></tr>
<tr><td>[Mar 02]</td><td colspan="3">Type de composant</td><td>Implantation de composant</td><td>Paquetage de composant</td><td>Instance de composant</td></tr>
<tr><td>[Lue 02]</td><td colspan="3">Modèle de composant</td><td>Composant</td><td>Composant déployable</td><td></td></tr>
<tr><td>Cycle de vie</td><td colspan="3">Conception</td><td>Implémentation</td><td>Déploiement</td><td>Exécution</td></tr>
</table>

Table 3.1. Perspectives sur le cycle de vie des composants.

3.1.1.2 Les interfaces

L'interface est le moyen à travers lequel un composant peut interagir avec les autres composants du système [Szy 98]. En d'autres termes, les dépendances et les interactions au niveau des composants sont contrôlées et formalisées par le mécanisme d'interface. C'est la partie visible du composant jouant le rôle d'une frontière qui le différencie de son

environnement. Les interfaces spécifient les points d'accès qui représentent l'ensemble des services fournis et requis par le composant [Szy 98]. On distingue donc deux types d'interfaces : les interfaces fournies et les interfaces requises. Il existe un troisième type d'interface appelé *interface de configuration* qui décrit la configurabilité du composant [Bos 00, Omm 02], mais il est très peu adopté par les modèles de composant.

Typiquement, une interface est présentée par une collection de noms d'opérations en offrant un fort découplage entre chaque opération et sa mise en œuvre, ce qui augmente l'adaptabilité du composant, dans le sens où l'ajout de nouvelle interfaces et donc de nouvelles implémentations peut se faire sans ajuster la structure externe actuelle de composant. Bien entendu, les interfaces représentent à la fois des services et des points d'interactions. Un point d'interaction est généralement appelé *port* lors de la modélisation des composants. La structure externe du composant est représentée par les interfaces ainsi que les informations qui décrivent son comportement. Les propriétés non fonctionnelles et les contraintes qui assurent le bon fonctionnement du système font aussi partie de cette structure, et se présentent souvent sous forme de *contrats* enrichissant la description syntaxique des interfaces. Les propriétés non fonctionnelles décrivent, par exemple, la sécurité, la performance, la configurabilité du composant et l'adaptabilité de son comportement durant l'exécution, etc.

En général, le terme *contrat* est utilisé dans le contexte des composants logiciels pour désigner un ensemble de contraintes associées aux interfaces. Beugnard et al. [Beu 99] proposent quatre niveaux différents de contrat :

- Contrats basiques (ou syntaxiques). Ce type de contrat décrit les propriétés syntaxiques des signatures des opérations (i.e. noms des opérations, nombre et type de paramètres, les exceptions possibles pouvant être rencontrées durant l'exécution). Typiquement, les contrats syntaxiques sont imposés par presque tous les modèles de composants actuels. Des langages dédiés à la définition des interfaces, comme le langage de définition d'interface IDL (*Interface Definition Language*) [Gud 01, Mic 96, Omg 91], ou même génériques, comme Java, peuvent être utilisés pour exprimer ces contrats.
- Contrats comportementaux. Il s'agit d'une spécification précise du comportement des opérations qui ne peut pas s'exprimer par des contrats syntaxiques. Elle consiste en des

conditions qui doivent être satisfaites pour garantir une bonne utilisation et une exécution correcte des opérations, et qui expriment les résultats attendu de cette exécution. Le langage Eiffel fût le premier à introduire ce type de contrat dans le cadre de l'approche *conception par contrats* [Mey 92]. Il s'agissait d'une méthode de conception logicielle qui spécifie les interfaces d'un composant (ou d'un module) logiciel en termes d'invariants, pré-conditions et post-conditions. Cette approche a été supportée par d'autres langages, tels que Java (JML [Lea 00]) et OCL d'UML [Ocl 12].

- Contrats de synchronisation. Les contrats comportementaux partent du fait que les services du composant sont atomiques et leur exécution est vue comme des transactions, ce qui est loin d'être le cas réellement. Il existe des dépendances entre les services des composants dont l'exécution peut être concurrente (e.g. les applications client/serveur), donc, les contrats de synchronisation sont utilisés pour décrire le comportement global d'un composant au moyen de synchronisations entre les appels d'opérations. Pour exprimer le comportement des composants dans un contexte concurrent, McHale [Mch 94], par exemple, utilise un formalisme d'expression de traces qui offre la possibilité de décrire un ensemble de politiques de synchronisation. Java, comme d'autres langages de programmation, supporte aussi le concept de synchronisation à travers l'utilisation du mot clé *synchronized*, qui permet l'exécution des opérations en exclusion mutuelle.
- Contrats de qualité de service. Ce type insiste sur la qualité de service offerte par un composant. La qualité de service est déterminée usuellement par des paramètres tels que le temps de réponse, la qualité du résultat (i.e. précision), la gestion de ressources, etc.

3.1.1.3 Modèles de composant

Les composants logiciels se conforment à un modèle de référence dit modèle de composant. Il permet d'utiliser uniformément les composants en définissant comment ils sont spécifiés et comment ils sont connectés. Depuis l'avènement de l'approche orientée composants, plusieurs modèles de composants ont été développés dans le but d'éliminer le problème de développement logiciel rapide, d'interopérabilité et d'évolution. Dans la suite, nous citons quelques modèles académiques et industriels ayant connu une vaste utilisation et qui sont largement étudiés dans les divers travaux de recherche.

JavaBeans

JavaBeans est un modèle de composant proposé par Sun Microsystems en 1996 pour intégrer l'approche orientée composants dans la plateforme Java [Sun 97]. La technologie JavaBeans consiste en un package Java (*Java.beans*) et une spécification JavaBeans, qui décrit comment utiliser les interfaces et les classes de ce package afin d'implémenter le concept de JavaBean. Le terme *Java Bean* ou simplement *Bean* signifie un composant logiciel qui conforme au modèle JavaBeans. Ce modèle est largement accepté et utilisé surtout dans le contexte de développement des applications Web.

La spécification donnée par Sun définit un Java Bean comme un composant logiciel réutilisable qui peut être visuellement manipulé dans un *builder tool* (qu'on peut traduire par outil de construction), i.e. un outil d'assemblage qui permet l'assemblage visuel des composants logiciels pour construire une application. La manipulation inclut la composition par le drag&drop des beans (i.e. faire glisser des beans, ayant une représentation symbolique, d'une palette, les déposer dans un espace de conception, puis les relier entre eux) ainsi que leur configuration, cela facilitera le développement rapide des applications. Un exemple remarquable qui montre l'avantage d'utiliser JavaBeans est celui du développement des interfaces graphiques (*GUIs* ou *Graphical User Interfaces*), où les beans représentent les composants graphiques des interfaces (e.g. boutons, panneaux, cases à cocher, etc.). L'aspect visuel est un point fort des beans. Dans un outil de construction, ils ont toujours une représentation graphique, cependant, durant l'exécution, ils peuvent être visibles ou inapparents. En d'autres termes, il est possible aussi d'implémenter des beans non-visuels.

Parmi les phases du cycle de vie, JavaBeans supporte l'implémentation et le déploiement de composants. L'implémentation est évidement effectuée en utilisant le langage Java. Le déploiement des beans a lieu en compilation après la distribution de ces derniers sous forme de packages. Les fichiers portant le format JAR sont utilisés pour regrouper les beans, dont chacun contient les informations et les fichiers sources nécessaires à leur utilisation. Typiquement, les JavaBeans se caractérisent par les propriétés, les évènements, les méthodes, la personnalisation, l'introspection et la persistance. Une propriété bean est un attribut paramétrant son apparence ou son comportement tel que la couleur, le label, la police, etc. Il est accessible à travers les méthodes fournies par le bean propriétaire (getters et setters) et

peut être de type primitif, classe ou interface. La personnalisation consiste à changer l'apparence et le comportement des beans, en modifiant les valeurs de leurs attributs, pour répondre aux besoins spécifiques des utilisateurs. Une autre caractéristique intéressante est celle de la persistance. C'est le processus par lequel on peut enregistrer, à un moment donné durant l'exécution, un objet et le récupérer ultérieurement.

Les évènements fournissent un mécanisme qui permet l'intégration des beans dans un outil de construction, et leurs interactions par le biais d'un modèle de communication par évènements (les beans utilisent le modèle d'évènements de Java). Un bean pour qu'il soit une source d'évènements, il doit disposer de méthodes qui permettent l'ajout et la suppression des *listeners* (i.e. objets écouteurs d'évènements) pour un type particulier d'évènements (i.e. celui défini par le bean). Dans le cas contraire, il doit implémenter une interface de listener (de type *java.util.EventListener*) pour être un consommateur d'évènements. Les méthodes d'un bean forment son interface et consistent en des méthodes publiques représentant ses services offerts, des méthodes permettant d'y accéder et de le manipuler, et des méthodes de traitement des évènements. L'introspection est l'analyse automatique d'un bean par l'outil de construction pour déterminer ses propriétés, ses évènements et ses méthodes, ce qui facilite son manipulation visuel.

EJB

Enterprise JavaBeans (EJB) est un modèle de composant développé par Sun Microsystems [Sun 09]. EJB adopte un modèle client-serveur et est essentiellement utilisé pour concevoir des applications distribuées, orientées objets, orientées transactions et base de données. Ce modèle est une extension logique de JavaBeans qui introduit les composants serveurs (*server components*). Ces derniers incorporent des fonctionnalités métiers réutilisables, archivées (i.e. dans des JAR) et parfois très complexes. Les composants EJB sont conçus pour être autonomes, portables et indépendamment déployés dans leur conteneur ou dans des serveurs d'application (i.e. logiciels d'infrastructure offrant un contexte d'exécution). Le conteneur d'EJB s'occupe de la gestion des composants EJB durant l'exécution (démarrage, arrêt, passivation ou activation) ainsi que des aspects non fonctionnels complexes tels que la sécurité, les transactions, la persistance, la concurrence, la fiabilité, la distribution et

la tolérance aux fautes. Le conteneur présente alors des avantages indéniables, quant au développement des applications complexes et distribuées, qui découlent du fait que les utilisateurs des composants EJB ne se focalisent que sur l'aspect métier de leurs applications.

La spécification d'EJB définit trois types de composants appelés *beans* : les composants entités (*Entity beans*), les composants de session (*Session beans*) et les composants orientés messages (*Message-Driven beans*). Un bean entité sert à représenter et gérer des données enregistrées dans une base de données. Il assure la persistance des données vu sa capacité d'être stocké dans un support physique entre deux sessions. Il existe deux types de bean entité : persistance gérée par le conteneur (*CMP : Container Managed Persistence*) et persistance gérée par le bean (*BMP : Bean Managed Persistence*). Avec un bean entité CMP, c'est le conteneur d'EJB qui assure la persistance des données. Il se charge de toute la logique des traitements de synchronisation entre les données du bean et les données dans la base de données. Un bean entité BMP, assure lui-même la persistance des données grâce à du code inclus dans les méthodes du bean. Plusieurs clients peuvent accéder simultanément à un même bean entité.

Les beans session effectuent des tâches pour des clients. À la différence des beans entité, leur partage n'est pas possible puisque chaque bean session représente un seul client dans le serveur d'application. Pour accéder à une application qui est en cours de déploiement sur le serveur, le client fait appel aux méthodes du bean session (mode de communication synchrone). Ce dernier effectue les tâches demandées tout en cachant leur complexité à son client. On distingue deux sortes de bean session : sans état (*stateless*) et avec état (*stateful*). Ceux avec état peuvent conserver l'état du bean dans des variables d'instance durant toute la conversation avec un client. Mais ces données ne sont pas persistantes (i.e. à la fin de l'échange avec le client, l'instance du bean est détruite et les données sont perdues). Les beans session sans état, de leur côté, ne peuvent pas conserver de telles données entre chaque appel du client.

Les beans orientés messages jouent le rôle d'un listener (écouteur) d'un certain type de messages (e.g. *JMS* ou *Java Message Service* API). La différence principale entre ce type et

les deux autres types d'EJB est le fait que la communication avec celui-ci est établie uniquement par des messages de façon asynchrone.

Fractal

Fractal est un modèle de composant développé initialement par France Telecom R&D et INRIA [Bru 06]. Actuellement, il fait partie d'un projet du consortium *ObjectWeb*. Son objectif principal est la fourniture d'un modèle générique, extensible et open source, qui peut être facilement adapté à une grande variété d'applications et de domaines, et utilisé avec n'importe quel langage de programmation. Il couvre le cycle de vie du développement des systèmes logiciels complexes, allant de la conception, à l'implémentation, jusqu'à le déploiement et la configuration dynamiques. Ses principales caractéristiques sont :

- La composition récursive des composants Fractal en un seul composant dit *composite*.
- Les composants Fractal sont réflectifs de telle sorte que leurs propriétés peuvent être exposées automatiquement via l'introspection.
- Le partage d'un composant Fractal entre plusieurs autres composants.
- Les composants de liaison (*binding components*) permettant la connexion (synchrone ou asynchrone) entre les interfaces de composants Fractal.
- La personnalisation des propriétés non fonctionnelles des composants à travers la notion d'attribut (*attribute*) accessible et changeable.

Chaque composant peut contenir, en plus de sa logique métier, une partie non fonctionnelle représentant des capacités de contrôle (i.e. des possibilités permettant aux autres composants de le contrôler). Ces dernières ne sont pas stables dans le modèle Fractal, mais sont néanmoins extensibles et adaptables aux besoins spécifiques et aux contraintes de l'utilisateur. La capacité de contrôle offerte par un composant Fractal appartient à l'un des niveaux suivants :

- Au niveau le plus bas, un composant Fractal n'offre aucune capacité de contrôle. Seules les invocations de ses méthodes permettent sa manipulation et son utilisation.

- A un niveau plus élevé (appelé niveau d'introspection), un composant Fractal offre des opérations d'analyse pour permettre l'introspection de sa capacité de contrôle et ses services fournis et requis.
- A un niveau supérieur (niveau de contrôle), non seulement l'introspection est possible mais aussi la possibilité de contrôler et reconfigurer la structure interne d'un composant Fractal (i.e. contrôler ses sous-composants et leurs liaisons).

Selon le niveau de la capacité de contrôle, on distingue trois types de composants Fractal : Les composants composites (niveau de contrôle), les composants primitifs (niveau d'introspection) et les composants de base (le niveau le plus bas). Les interfaces jouent un rôle essentiel dans le modèle Fractal selon lequel une interface est un point d'accès à un composant. Elles peuvent être fournies (serveurs) ou requises (clients), fonctionnelles ou de contrôle. Alors que l'interface de contrôle correspond à un aspect non fonctionnel comme l'introspection, la configuration et la reconfiguration, l'interface fonctionnelle correspond à un service fourni ou requis.

A l'intérieur, un composant Fractal est formé de deux parties : Un contrôleur (ou membrane) et un contenu. Le contenu d'un composant est constitué d'un ou plusieurs composants, appelés *sous-composants*, qui sont sous le contrôle du contrôleur contenu dans le composant composite qui les englobe. Le contrôleur peut avoir des interfaces internes et externes. Les interfaces externes sont accessibles de l'extérieur et les interfaces internes sont accessibles exclusivement des sous-composants. Le contrôleur concrétise son contrôle en offrant une représentation explicite de la structure de composition des sous-composants, en interceptant les invocations entrantes et sortantes dont les sous-composants sont cibles et sources respectivement, en superposant un comportement spécifique aux sous-composants, y compris leur suspension, la reprise de leurs activités, etc.

CCM

CCM (*Corba Component Model*) est le modèle de composant de CORBA, qui fait partie du standard CORBA3 défini par l'OMG [Bol 01]. Son objectif principal est de réduire l'effort de développement et de déploiement des applications CORBA. Les composants CCM

peuvent être implémentés dans n'importe quel langage de programmation et sur n'importe quelle plateforme tant qu'ils utilisent le middleware CORBA. Un package de composant dans CCM consiste en des programmes compilés (e.g. fichiers de bibliothèque C++ ou fichiers de classe Java) et un descripteur de composants. Le descripteur de composant est un fichier XML qui contient des informations sur les interfaces et les services que le composant supporte.

En exécution, les composants CCM sont déployés dans des *conteneurs*. Le conteneur représente un support d'exécution qui fournit des services spécifiques à la plateforme d'exécution (e.g. la gestion du cycle de vie du composant, le nommage, les transactions et la sécurité) de telle sorte que les composants soient déchargés de leur implémentation. Par exemple, les opérations de la gestion du cycle de vie du composant sont implémentées par des objets appelés *maisons de composants* (*component home*), qui s'occupent de l'instanciation des composants. En effet, chaque type de composant doit avoir une maison de composants qui lui est associée. Le processus de développement de CCM supporte aussi la génération automatique de code par le framework d'implémentation de composants : CIF (*Component Implementation Framework*). À cette fin, CCM définit un langage déclaratif appelé CIDL (*Component Implementation Definition Language*) pour décrire les composants. La compilation d'une description CIDL génère des parties de l'implémentation de composant et des descripteurs de composants utilisés dans les composants paquetagés.

Il existe deux niveaux de composants CCM : le niveau basique (*basic*), qui permet d'empaqueter des objets CORBA ainsi que l'intégration et le mapping entre le modèle CCM et le modèle EJB (i.e. les composants basiques et les composants EJB sont compatibles), et le niveau étendu (*extended*), qui est beaucoup plus riche et fournit plusieurs fonctionnalités. Les interfaces d'un composant étendu sont définies par des ports dont le type peut être :

- Une facette : déclare une interface fournie par un type de composant;
- Un réceptacle : décrit explicitement les interfaces utilisées par un type de composant;
- Une source d'évènement : représente une interface qui permet l'émission d'évènements;
- Un puit d'évènement : représente une interface qui permet la réception d'évènements;
- Un attribut : est une valeur nommée destinée à la configuration du composant.

En utilisant ces types de ports, il devient possible de décrire les communications synchrones et asynchrones entre les services fournis et requis des composants communiquant entre eux. Les interfaces des composants basiques et étendus sont définies en IDL.

En effet, CCM ne supporte pas la composition hiérarchique, cependant, il définit quatre catégories de composants :

- Session : il s'agit d'un objet temporel utilisé à chaque connexion d'un client sur un serveur. une fois la session terminée, l'objet sera détruit. Ce type de composant est sans état;
- Service : c'est un objet temporel sans état qui fournit des services aux clients. Sa durée de vie correspond à la durée du traitement des services invoqués;
- Entité : un composant de cette catégorie se trouve dans un état persistant géré par le conteneur de composant. Il est essentiellement utilisé pour représenter les informations qui sont enregistrées dans une base de données;
- Processus : similairement aux composants entité, les composants processus se caractérisent par un état persistant. Ils sont utilisés pour représenter les processus métiers dont l'état doit être enregistré de manière persistante.

3.1.2 Les Connecteurs

La définition du connecteur la plus référenciée dans la littérature est celle de Shaw et Garlan [Sha 96] : « les connecteurs gèrent les interactions entre les composants ; c'est-à-dire, ils établissent les règles qui gouvernent les interactions entre les composants et spécifient tous les mécanismes auxiliaires nécessaires ». Dans une architecture logicielle, et d'après cette définition, le connecteur est un élément architectural qui gère et modélise de manière explicite les interactions entre les composants logiciels. La complexité du connecteur peut varier d'une simple communication de type appel de méthode ou envoi de message jusqu'à la modélisation d'une interaction très complexe comme, par exemple, une communication transactionnelle ou un protocole d'accès aux bases de données.

Alors que les composants ont toujours été considérés comme des entités centrales de l'architecture, les connecteurs ont été souvent traités comme implicites et leur implémentation

était effectuée au niveau des composants. Il a fallu attendre l'avènement de nouvelles générations de systèmes logiciels complexes, constitués de composants totalement indépendants, dynamiques et distribués, qui mettaient en avant la nécessité de consacrer une séparation des préoccupations, de telle sorte que la mécanique d'interconnexion ne sera plus à la charge des composants, mais plutôt elle sera considérée comme étant une entité de première classe au même titre que les composants. Le fait de libérer les composants des taches relatives à l'aspect de connecteur (i.e. les règles gouvernant les interactions simples ou complexes) minimise les interdépendances de ces derniers, et rend plus facile leur adaptation à plusieurs contextes d'utilisations, leur réutilisation, ainsi que leurs interactions, même pour ceux provenant de différentes sources. De plus, la réutilisation et l'ajustement des technologies d'interconnexions mises en œuvre, sera aussi plus facile.

Le connecteur, comme le composant, possède une interface qui représente sa partie externe visible, dont les points de connexions avec les composants sont appelés *rôles*. Le rôle identifie les participants à l'interaction décrite par le connecteur. L'interface comporte aussi des propriétés non-fonctionnelles (e.g. sécurité, performance, etc.) et des contraintes d'utilisation du connecteur garantissant la cohérence du système.

3.1.3 Les Configurations

La configuration (ou *topologie*) décrit la manière dont les composants et les connecteurs sont assemblés dans une architecture. Elle représente la structure de l'architecture et peut être présentée par un graphe connecté de composants et de connecteurs. La configuration aide à tacler des problèmes tels que la cohérence, la concurrence, la distribution, la performance, etc. car elle fournit une vue abstraite des composants et connecteurs constituant le système, leur type, leurs associations, les contraintes de conception imposées sur l'architecture, etc. Par exemple, la configuration permet de déterminer si les composants sont correctement connectés, par des connecteurs dont les rôles sont convenables, et si leurs interfaces sont compatibles, et leurs sémantiques combinées produisent le comportement désirable.

Certains modèles de composants définissent des composants composites dont chacun peut être vu comme une configuration de sous-composants et sous-connecteurs. Dans ce cas,

la configuration est celle ayant le niveau d'abstraction le plus élevé. La configuration dispose de contraintes et propriétés non-fonctionnelles semblables à celles caractérisant les composants et les connecteurs. Elles sont exprimées au niveau de la configuration, non pas au niveau des composants ou connecteurs. La raison à cela est le fait qu'il existe des propriétés et des contraintes qui concernent uniquement l'architecture dans sa globalité (e.g. propriétés concernant l'environnement de son déploiement, contraintes exprimant les relations entres composants, etc.).

3.2 Les Langages de Description d'Architectures

Les langages de description d'architecture mettent en évidence les concepts de base de l'architecture logicielle et prennent en charge la spécification rigoureuse des composants constituant un système logiciel, leur interconnexion et leurs propriétés [Med 00]. Ils ont pour objectif principal de fournir une notation formelle, représentée textuellement ou graphiquement, selon laquelle les architectures logicielles vont être décrites. La communauté scientifique a proposé une diversité de langages issus de plusieurs centres de recherches, entreprises et universités [Med 00]. La prolifération des ADLs montre l'intérêt important de l'architecture logicielle et est le résultat logique de la couverture continue des aspects non abordés précédemment. Par exemple, Darwin [Mag 96] s'oriente vers la description d'architectures distribuées et la prise en compte de leur reconfiguration dynamique. ACME [Gar 00] cible la fourniture d'un standard qui définit les éléments architecturaux communs aux ADLs existants. Wright [All 98] est plus axé sur la description du comportement dynamique et la formalisation des connecteurs, tandis qu'Olan [Bal 98] représente une solution aux problèmes de conception des systèmes fortement distribués ainsi que leur déploiement.

Les ADLs sont considérablement étudiés et explorés, et, au vu de leur nombre important, plusieurs études comparatives ont été effectuées pour exposer leurs propriétés, leurs points forts et leurs limitations [Cle 96, Med 00, Ous 05, Ves 93]. Dans cette section, nous étudions quelques ADLs en prenant une direction différente, dans laquelle nous nous focalisant sur deux concepts particuliers : l'outillage et la métamodélisation. Le premier concerne l'ensemble d'outils supports accompagnant l'ADL (e.g. outils de modélisation, de simulation, d'analyse, etc.) et le second concerne sa représentation au niveau méta (voir la

section 2.2). Un métamodèle d'un ADL définit explicitement sa syntaxe abstraite et sa sémantique statique. Alors que la syntaxe abstraite définit les constructions du langage et leurs relations, donc sa structure, la sémantique statique détermine les critères (des contraintes) avec lesquelles les expressions du langage sont jugées bien formées. Le métamodèle facilite l'intégration du langage au sein des outils de modélisation existants, et sert de brique de base pour développer des outils supports. Dans la suite, nous décrivons, pour chaque ADL, son métamodèle (s'il existe, présenté sous forme d'un diagramme de classes UML) et nous citons les outils supports disponibles, tout en déterminant leur état actuel vis-à-vis les besoins des utilisateurs.

3.2.1 ACME

Acme est un ADL générique, considéré comme un langage pivot qui tient compte des caractéristiques communes de l'ensemble des ADLs existants [Gar 00]. Acme peut être utilisé comme format d'échange entre les différents outils de conception d'architecture, et peut même fournir un fondement pour développer de nouveaux ADLs grâce à sa structure extensible.

3.2.1.1 Métamodèle

La figure 3.1 présente un métamodèle d'ACME. Il contient seize métaclasses décrivant les constructions du langage. Un fichier Acme est modélisé par l'élément *ACMEFile*. Ce dernier possède zéro ou plusieurs entrées, représentées par la métaclasse abstraite *ACMEEntry*. Il existe deux types d'entrées : *System* et *ComponentType*. *Component*, *Connector* et *System* sont les éléments architecturaux fondamentaux d'Acme. La métaclasse Component représente les éléments de traitement et de stockage dans les systèmes logiciels (i.e. les composants), alors que Connector représente les interactions entre ces éléments (i.e. les connecteurs). Les composants exhibent leurs fonctionnalités à travers leurs ports (modélisés par l'élément *Port*). Un port est un point d'interaction entre un composant et son environnement. Les connecteurs, à leur tour, incluent un ensemble d'interfaces sous forme de rôles (modélisés par l'élément *Role*). Un rôle définit quels sont les participants (i.e. les ports) pouvant participés dans une connexion.

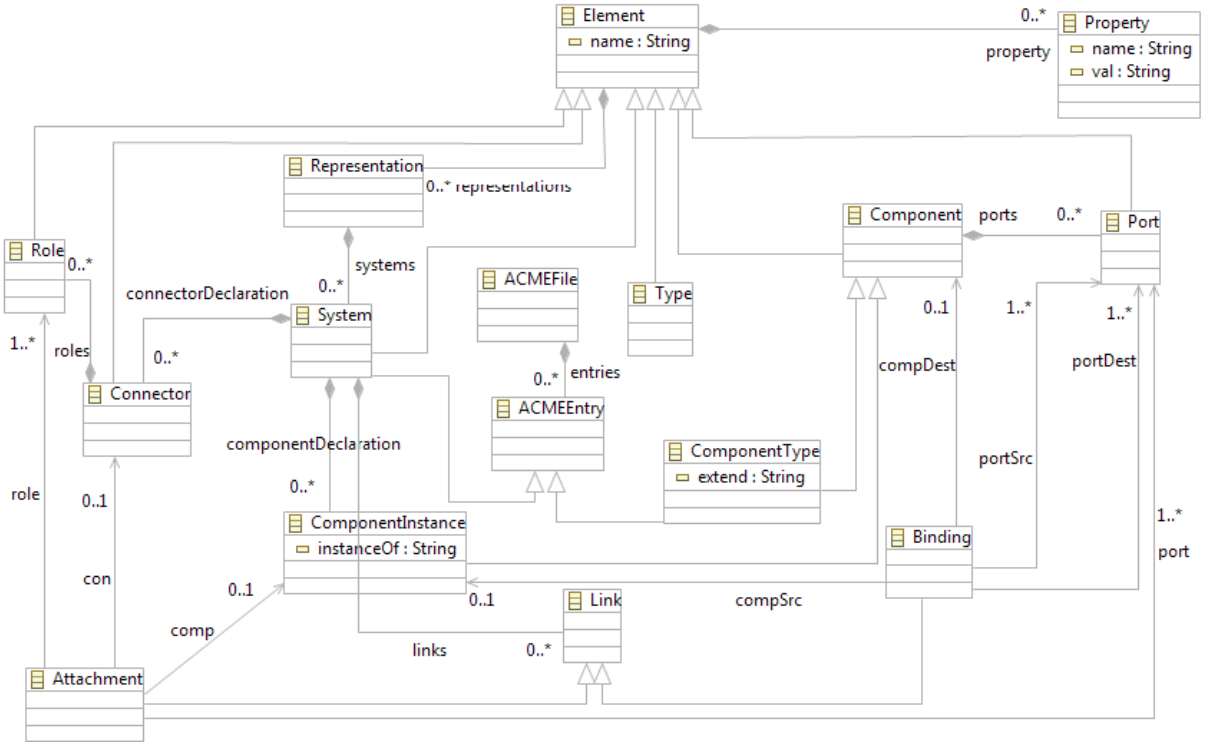


Figure 3.1. Métamodèle ACME.

System correspond à la configuration des composants et connecteurs. Acme supporte la description hiérarchique des architectures. Chaque composant et chaque connecteur peuvent être représentés par une ou plusieurs descriptions détaillées ayant un niveau hiérarchique plus bas. Les descriptions sont modélisées par l'élément *Representation*. Ce mécanisme permet de décomposer un élément en des sous-systèmes. D'une manière générale, les éléments Acme, modélisés par la métaclasse *Element*, peuvent avoir plusieurs représentations où chacune correspond à une vue conceptuelle différente ou à une des façons alternatives dont un élément est décomposé.

L'attachement des ports aux rôles est représenté par la métaclasse *Attachment*, alors que l'association d'un port d'un composant avec un port dans une représentation correspondante est modélisée par la métaclasse *Binding*. Un élément Acme, que ce soit un rôle, un attachement, un système ou autre, se caractérise par des propriétés. Chaque propriété possède un nom et une valeur. Les propriétés d'un composant, par exemple, décrivent typiquement ses traitements, mais peuvent aussi être utilisées pour décrire ses propriétés comportementales telles que la performance, la capacité de stockage, la tolérance aux fautes,

etc. De façon similaire, un connecteur inclut des propriétés relatives essentiellement à l'aspect de connexion. Par exemple, une propriété peut définir la manière avec laquelle les tâches à effectuer par le connecteur sont distribuées dans ce dernier. De même, les propriétés du système définissent des propriétés globales concernant tous ses éléments, des propriétés caractérisant l'environnement où il s'opère, ou des propriétés émergeant de l'ensemble de propriétés de ses composants et connecteurs.

3.2.1.2 Outils Supports

Acme est assisté par des outils logiciels qui permettent le développement de nouveaux outils de conception et d'analyse architecturale. Le langage Acme et la bibliothèque AcmeLib fournissent une infrastructure générique et extensible pour décrire, représenter, générer, manipuler et analyser les descriptions d'architecture. AcmeLib comprend des parseurs et génrateurs qui peuvent être utilisés pour développer de nouveaux outils pour Acme. De plus, un environnement gratuit de développement et de visualisation appelé AcmeStudio[1], doté d'un éditeur graphique (figure 3.2) et textuel, est développé, en utilisant AcmeLib, pour la conception et l'analyse des architectures décrites en Acme. Il permet d'utiliser des styles d'architecture prédéfinis ou de créer de nouveaux styles qui désignent des familles de systèmes partageant un vocabulaire et des propriétés communes, et imposant des règles de configuration et des contraintes sur les éléments du vocabulaire (i.e. les types de composants et connecteurs définissant le style).

AcmeStudio est gratuit et disponible sous forme de plugin de l'environnement Eclipse, et donc peut être étendu par de nouvelles fonctionnalités, et utilisé avec plusieurs plateformes. Monroe [Mon 99] a proposé l'environnement Armani, qui utilise un éditeur graphique commercial appelé Visio et implémente le vérificateur de contraintes initial pour les spécifications Acme (évaluateur des prédicats logiques), qui a été intégré par la suite dans l'environnement AcmeStudio pour décrire et vérifier les règles de conception.

[1] http://www.cs.cmu.edu/~acme/AcmeStudio/index.html

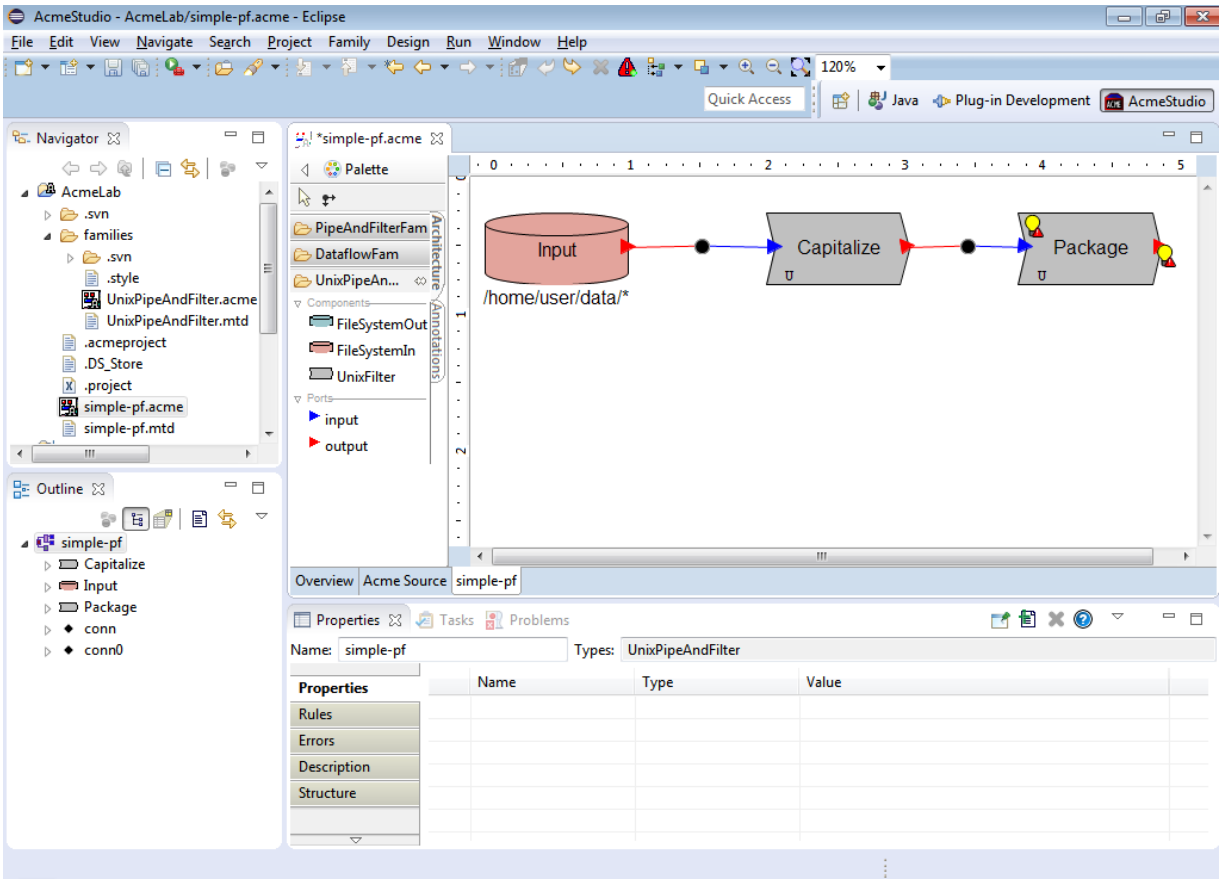


Figure 3.2. AcmeStudio.

En ce qui concerne la spécification active, le processus de conception réactive est partiellement adopté par Acme à travers le vérificateur de contraintes qui permet de signaler les erreurs à l'utilisateur en cas de violation des règles de conception (contraintes définies dans le langage de contrainte Armani et attachées aux éléments d'architecture). Ainsi, tout changement dans la représentation textuelle de l'architecture sera automatiquement reflété dans la représentation graphique et vice versa. Malheureusement, Acme et AcmeStudio ne permettent pas de décrire le comportement du système et se concentrent uniquement sur l'aspect structurel (la syntaxe) et non pas sur l'aspect sémantique. Cependant, les éléments d'une architecture peuvent être annotés pour représenter ce type d'informations (pourtant le fait qu'elles ne sont pas interprétables par les outils d'Acme). Actuellement, la génération d'un système exécutable à partir d'une description Acme n'est pas supportée par AcmeStudio.

Pour étendre les capacités d'Acme, soit pour décrire formellement les propriétés non-fonctionnelles du système et son comportement ou pour supporter d'autres fonctionnalités

(e.g. multiples vues, la traçabilité, le raffinement, la génération de code, etc.), la bibliothèque AcmeLib doit être utilisée et étendue afin d'éviter de réinventer la roue. C'est une API java (disponible en C++ aussi) qui comprend un parseur et d'autres outils de vérification et de manipulation de description d'architecture.

3.2.2 MetaH

MetaH est un ADL dédié initialement à l'expression et l'évaluation d'architecture des systèmes avioniques et de gestion de vol [Ves 96]. Il a été inventé non seulement pour la description et l'analyse, mais aussi pour l'intégration des composants logiciels au niveau des systèmes matériels embarqués. MetaH permet la portabilité, le développement rapide et l'évolution des systèmes critiques, temps-réel, tolérants aux fautes, multiprocesseurs et embarqués. Il offre la possibilité de spécifier comment un système est composé de modules logiciels tels que les sous-programmes et les packages, et d'objets matériels tels que les processeurs et les mémoires. MetaH a prouvé son efficacité à travers de nombreuses utilisations dans le domaine militaire [Ves 96].

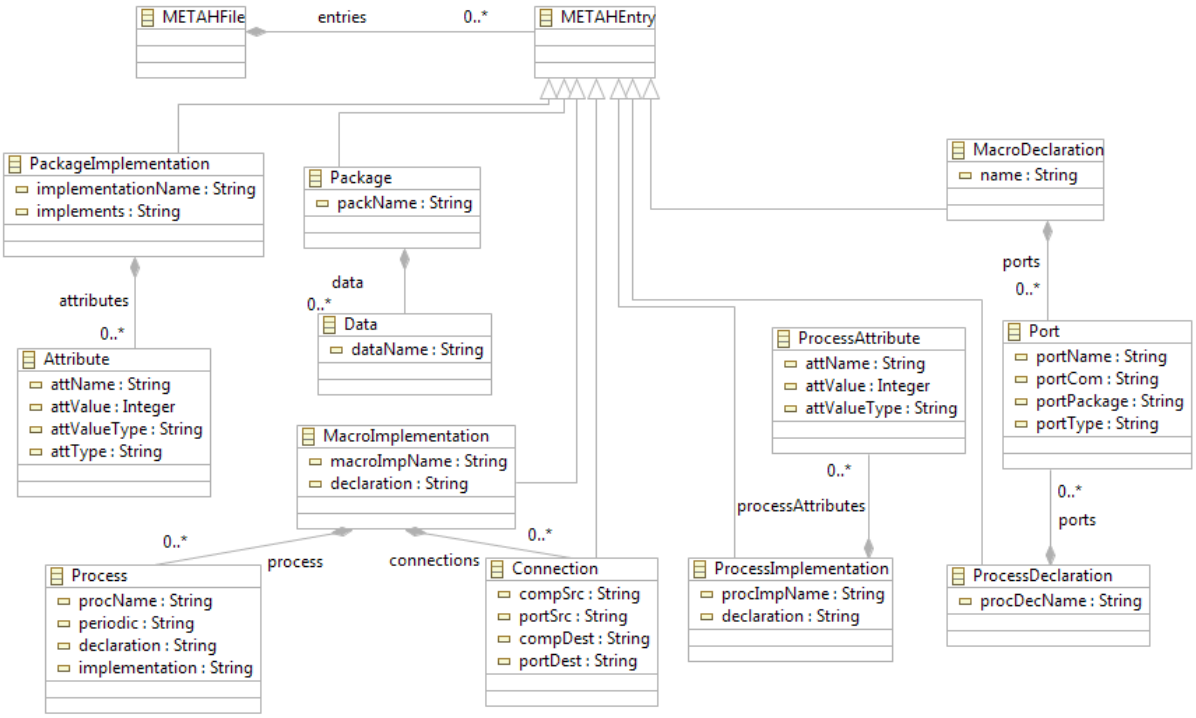


Figure 3.3. Métamodèle MetaH.

3.2.2.1 Métamodèle

Le métamodèle décrivant l'ADL MetaH est illustré par la figure 3.3. Un fichier MetaH est associé avec la métaclasse *MetaHFile*. Il y a plusieurs types d'entrées : *Package*, *Package Implementation*, *ProcessDeclaration*, *ProcessImplementation*, *MacroDeclaration*, *Macro Implementation*. Un package, modélisé par la métaclasse *Package*, est un module source utilisé pour construire les systèmes et regrouper des sous-programmes. Les packages sont partageables et peuvent contenir des ports. Les processus sont modélisés par la métaclasse *Process*, et leur implémentation et déclaration par *ProcessImplementation* et *ProcessDeclaration* respectivement. Ils regroupent les modules sources, qui doivent être ordonnancés soit comme des processus périodiques, soit comme des processus apériodiques. Les macros (représentées par *MacroImplementation* et *MacroDeclaration*) regroupent les processus et définissent les connexions (*Connection*) entre les ports (*Port*).

3.2.2.2 Outils Supports

MetaH est à la fois un ADL et une suite d'outils qui le supportent (figure 3.4). Il permet la spécification textuelle et graphique d'architecture ainsi que le passage entre les deux différentes représentations. MetaH fournit une interface graphique appelée *Dome* qui permet la description d'architecture ainsi que ses propriétés. Une fois la description terminée, Dome fournit la possibilité de l'exporter sous forme d'un fichier de description. Les spécifications graphiques devraient être transformées en texte avant de pouvoir les analyser syntaxiquement et sémantiquement. L'analyse syntaxique et sémantique repose, respectivement, sur un parseur et un compilateur. Le compilateur permet la génération des systèmes exécutables à partir des descriptions architecturales, à condition que les composants soient déjà implémentés.

Un outil intéressant de MetaH, appelé *Hardware/software binder*, fait la liaison entre le matériel et le logiciel. Par exemple, les processus sont liés aux processeurs, les ports de connexions sont liés aux canaux physiques intermédiaires et aux processeurs, et les modules partagés par plusieurs processeurs sont liés aux mémoires spécifiques, physiquement accessibles par tous ces processeurs. La suite d'outils contient aussi des générateurs de code qui produisent le code de liaison qui combine les différents modules sources pour former une application (outil de configuration exécutive). L'outil *Builder* compile est effectue la liaison

nécessaire pour créer des images exécutables et chargeables pour chaque processeur dans le système. MetaH fournit aussi des outils de vérification formelle d'architecture (e.g. la vérification de l'ordonnancement des tâches temps-réel, de la conformité des composants sources incorporés dans le système par rapport à leur spécification, de la fiabilité, de la partition du temps et d'espace, etc.). Néanmoins, MetaH ne supporte que le langage de programmation Ada (i.e. analyse et génération du code en Ada).

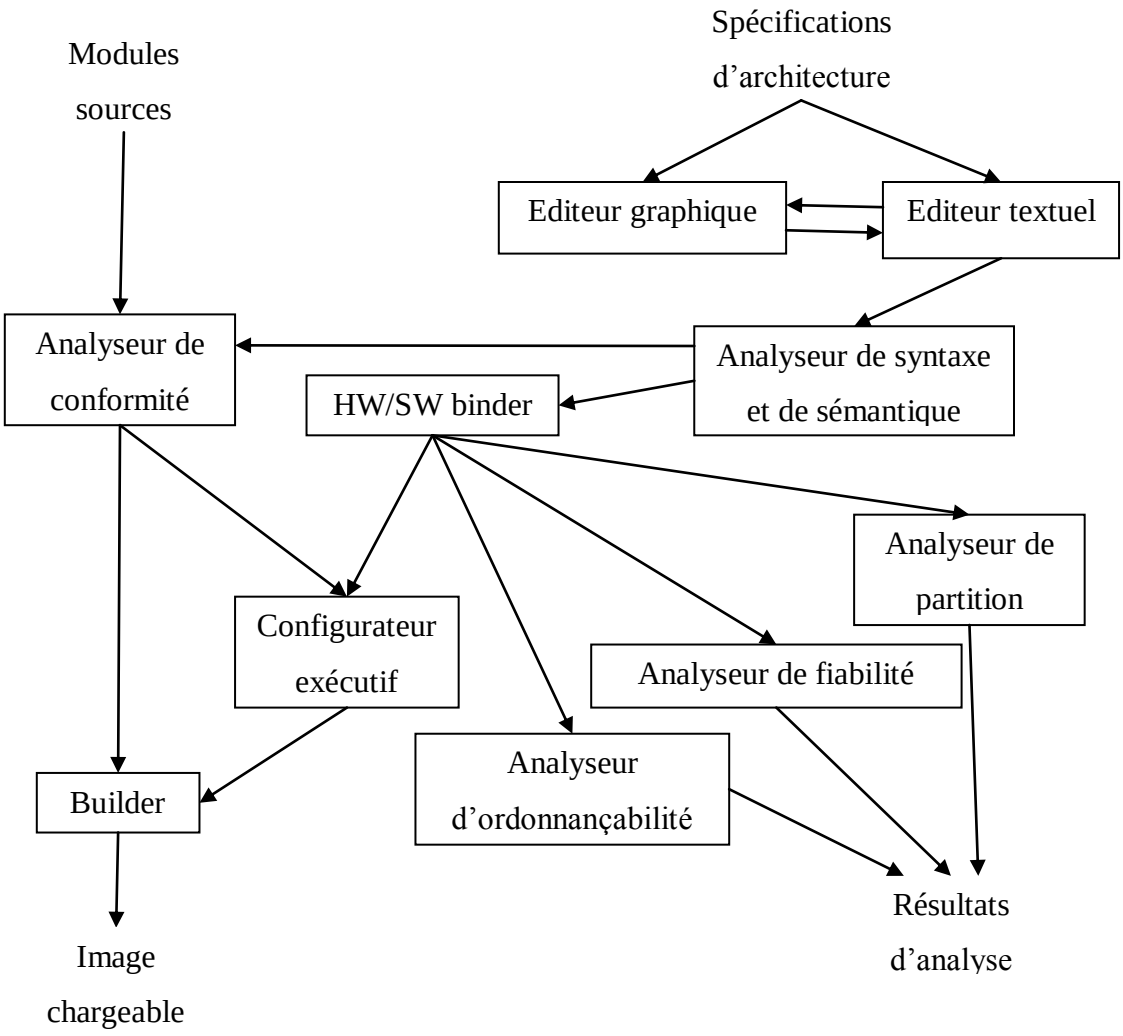


Figure 3.4. Outils MetaH.

3.2.3 AADL

AADL (*Architecture Analysis & Design Language*) est un standard proposé par l'organisation internationale SAE (*Society of Automative Engineer*) [Aad 12]. Découlé de MetaH, AADL est un langage d'analyse et de conception d'architecture qui a pour objectif la spécification, l'analyse, l'intégration et la génération automatique du code des systèmes distribués, temps-réel, et critiques comme l'aéronautique, l'automobile, et le spatial. D'ailleurs, sa première signification était *Avionics Architecture Description Language*. Il fournit un nouveau véhicule qui permet l'analyse des architectures logicielles, et supporte l'approche de développement dirigée par les modèles tout au long du cycle de vie du système.

AADL dispose de nombreux avantages acquis pendant plus de quinze ans d'expérimentations et de recherches extensives effectuées par des communautés industrielles :

- Il définit une syntaxe et une sémantique précise permettant la documentation et la modélisation des systèmes critiques, temps-réel et embarqués de manière assez détaillée;
- Il permet la spécification des architectures logicielles à grande échelle, selon différentes perspectives par un modèle analysable qui peut être réifié incrémentalement;
- Il supporte la modélisation des plateformes d'exécution et la représentation des propriétés non fonctionnelles qui décrivent, par exemple, le comportement du système (e.g. ordonnancement, sécurité, tolérance aux fautes, fiabilité, etc.) et l'allocation des ressources (e.g. mémoire, temps, etc.). En d'autres termes, AADL consacre la séparation des préoccupations ce qui permet aux développeurs d'applications de se concentrer sur l'aspect fonctionnel du domaine, tandis que l'architecte du système peut se focaliser sur les exigences non fonctionnelles. Ces capacités de modélisation facilitent l'analyse basée sur les modèles de ces propriétés ;
- Il est extensible, soit par des propriétés soit par des annexes. Les propriétés permettent l'association de nouvelles caractéristiques aux éléments d'une description tandis que les annexes permettent l'enrichissement de la syntaxe standard d'AADL.

3.2.3.1 Métamodèle

Le langage AADL et ses outils sont utilisé dans de nombreux projets académiques et industriels[1]. Les spécifications AADL peuvent être exprimées textuellement, graphiquement, en XML, ou en UML. Elles respectent un métamodèle qui décrit la syntaxe abstraite d'AADL. La puissance expressive de ce langage est exhibée par le nombre important des métaclasses qui composent son métamodèle. Il contient 254 métaclasses dont 56 sont abstraites. Vu sa complexité, il n'est pas inclus ici mais il est disponible en ligne[2].

Une description AADL est construite à partir de composants. C'est l'élément constitutif le plus important du métamodèle. AADL permet la description à la fois des parties logicielles et matérielles d'un système. Il définit des composants applicatifs (donnée, thread, groupe de threads, sous-programme, processus), des composants exécutifs (mémoire, bus, processeur, dispositif, processeur virtuel, bus virtuel) et des composants hybrides (système). Chaque catégorie de composants décrit des éléments bien identifiés dans l'ingénierie des systèmes :

- Les sous-programmes modélisent les fonctions et les procédures qui s'exécutent séquentiellement comme dans les langages C et Ada. Ils peuvent être appelés depuis un thread ou un autre sous-programme ;
- Les composants de type *donnée* (*data*) modélisent les données statiques qui peuvent être partagées par plusieurs threads et processus;
- Les threads représentent les unités d'exécution faisant partie d'un programme (la partie active du code), fonctionnant de façon autonome et parallèlement à d'autres threads, et gérés par un ordonnanceur. Les threads AADL peuvent avoir plusieurs modes d'exécution. Chaque mode peut décrire un comportement différent du thread (e.g. périodique,

[1] https://wiki.sei.cmu.edu/aadl/index.php/Projects_and_Initiatives

[2] http://www.aadl.info/aadl/osate/osate-doc/metamodel/

apériodique, hybride). Les groupes de threads sont utilisés pour créer des hiérarchies de threads, destinés typiquement à être ordonnancés par le même processeur;

- Les processus sont des espaces mémoire contenant les threads, les groupes de threads et les données. Les threads contenus dans un processus sont exécutés par le processeur auquel le processus est attaché;
- Les processeurs représentent les plateformes d'exécution. Généralement, ils modélisent les micro-processeurs et les ordonnanceurs qui gèrent et exécutent les threads. Ce type de composants peut contenir une mémoire, des bus, des bus virtuel, et des processeurs virtuels;
- Les mémoires modélisent les dispositifs de stockage tels que les disques durs et les RAMs. Elles peuvent contenir d'autres mémoires et nécessitent l'accès aux bus pour communiquer avec le reste du système;
- Les bus modélisent les différents canaux de communication, logiciels et matériels, qui permettent la circulation des informations de contrôle et les données entre les processeurs, les mémoires, et les dispositifs;
- Les dispositifs modélisent des appareils tels que les capteurs et les actuateurs servant d'interfaces avec l'environnement extérieur du système. Ils peuvent être considérés comme des composants logiciels de bas niveau qui s'exécutent sur des processeurs ayant accès aux dispositifs physiques. Ils peuvent avoir besoin d'accès aux bus pour communiquer avec le reste du système (e.g. le processeur). Un dispositif représente aussi le logiciel qui accède et gère le dispositif physique sous-jacent (i.e. un pilote);
- Les bus virtuels sont des abstractions logiques telles que les canaux virtuels ou les protocoles de communication. Un bus virtuel peut supporter la communication entre les composants applicatifs d'AADL, attachés au même processeur ou distribués aux plusieurs processeurs, à travers des bus;
- les processeurs virtuels sont des abstractions logiques des processeurs. En d'autres termes, ils représentent une ressource logique pour l'ordonnancement et l'exécution des systèmes. Ils peuvent être utilisés pour représenter une machine virtuelle telle que la JVM, un

domaine d'ordonnancement dans un processeur (i.e. une partition temporelle d'un processeur), ou un ordonnanceur dans une hiérarchie d'ordonnanceurs. Les processeurs virtuels sont éventuellement attachés aux processeurs. Ainsi, il est possible de les attacher à d'autres processeurs virtuels. Un processeur virtuel peut être déclaré comme un sous-composant d'un processeur physique ou un processeur virtuel contenu dans un processeur;

- Les systèmes ne représentent pas un concept concret. Ils permettent la composition et la structuration des différents composants applicatifs et exécutifs d'AADL pour former une description architecturale. En d'autres termes, ils représentent des composants composites de l'architecture.

Les composants AADL se caractérisent par un type et une ou plusieurs implémentations. Un type de composant décrit l'interface fonctionnelle et spécifie ce qui est visible à l'extérieur du composant. Une implémentation du composant décrit sa structure interne et comment ses interfaces fonctionnelles sont implémentées. Un type de composant peut avoir plusieurs implémentations. Cette séparation permet aux architectes de se concentrer sur les aspects architecturaux et facilite la modification des implémentations sans avoir changer l'interface du composant. Un type de composant décrit ses interfaces par *features, flows* (flux) et *properties (*propriétés*)*. *Features* comprennent les éléments suivants :

- Port : il représente un point logique de connexion entre les composants, pour le transfert des données, des évènements ou les deux. Il se caractérise par une direction qui peut être *in*, *out* ou *inout*. Les ports AADL modélisent des communications asynchrones et peuvent être regroupés dans des *groupes de ports* ;
- Sous-programme : les sous-programmes permettent la modélisation d'un point d'entrée vers l'exécution d'un composant thread, d'un appel de procédure comme dans les langages de programmation impératifs, ou d'une partie d'un composant donné. Les sous-programmes sont atomiques et peuvent être appelés depuis un thread ou un autre sous-programme ;
- Paramètres : Ils sont typés par un type de composant donnée, et orientés (i.e. soit *in*, *out* ou *inout*). Les paramètres sont semblables aux ports, mais sont spécifiques aux sous-programmes.

- Sous-composant : il représente l'instanciation d'un type ou d'une implémentation du composant, similairement aux objets étant des instances de classe dans le paradigme orienté objets. Ce feature permet l'imbrication des composants.

En plus des features, les types de composant peuvent être spécifiés par les propriétés et les flux. Un flux est une abstraction d'un canal de transfert d'information à travers ou entre les composants, et ne relie que des ports. À leur tour, les implémentations de composant spécifient sa structure interne par *connections* (connexions), *flows*, *modes*, *subcomponents* (sous-composants) et *properties*. Une connexion est l'élément qui relie les interfaces des différents composants (ou sous-composants). C'est un lien qui représente la communication de données et de contrôle entre les composants. AADL supporte trois types de connexions :

- Les connexions de ports : une connexion de port représente le flux de données et de contrôle entre deux threads, un thread et un processeur ou un dispositif ;
- Les connexions de paramètres : une connexion de paramètre est une abstraction du flux de données entre les paramètres d'une séquence d'appels de sous-programmes ;
- Les connexions d'accès : une connexion d'accès représente un accès aux composants de données partagés entre les threads ou les sous-programmes s'exécutant à l'intérieur d'un thread.

Le mode représente un état opérationnel d'un composant. Un composant peut avoir plusieurs modes de configuration de sous-composants et de connecteurs. La transition d'un mode à un autre permet de modéliser le comportement opérationnel dynamique du composant qui représente l'altération des configurations. Les sous-composants, comme pour les types de composant, permettent l'imbrication des composants, et les propriétés fournissent des informations sur les composants.

Les composants AADL peuvent être interconnectés de différentes manières selon leur catégorie. Comme nous l'avons déjà mentionné, les composants de connexions sont utilisés pour la communication des données et des évènements, et sont spécifiés par les éléments *connection*, *flow* et *port*. Les composants applicatifs peuvent être attachés aux plateformes d'exécution par des propriétés.

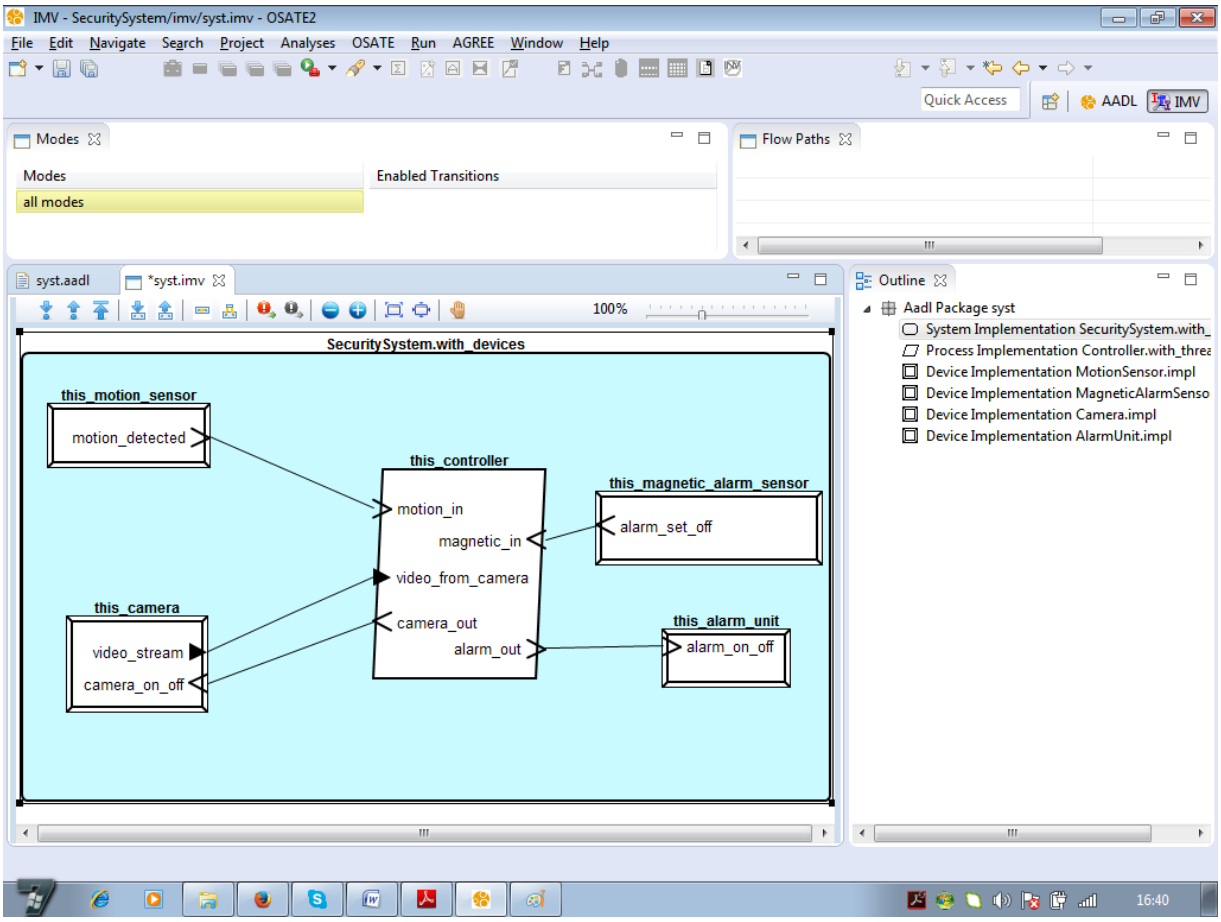


Figure 3.5. L'environnement OSATE.

3.2.3.2 Outils Supports

Grâce au métamodèle standard d'AADL, plusieurs outils supportant ce langage ont été créés[1]. Le support d'AADL par les outils a débuté par le développement de l'environnement OSATE (*Open-Source AADL Tool Environment*) pour Eclipse [Fei 05]. C'est un outil extensible à code source libre qui supporte le métamodèle standard d'AADL et qui permet l'analyse et la vérification syntaxique et sémantique des modèles AADL via Java et XML. L'environnement Eclipse avec OSATE forment une plateforme puissante pour le développement et l'intégration des outils d'analyse et de modélisation. Via ses définitions

[1] https://wiki.sei.cmu.edu/aadl/index.php/AADL_tools

XML standardisées, OSATE permet de s'interfacer avec des outils externes. La représentation intermédiaire des modèles AADL par le format d'échange XML/XMI facilite leur analyse par les outils compatibles externes. L'OSATE fournit un éditeur textuel et graphique, un analyseur, un générateur de texte, une vue Outline et d'autres outils facilitant les différents tâches de modélisation (figure 3.5).

Topcased (*Toolkit in Open Source for Critical Applications & Systems Development*), par exemple, fait partie de la chaine d'outils basés sur OSATE et AADL. C'est un atelier de développement des applications et systèmes critiques, en environnement Eclipse [Gau 05]. Il est soutenu par un ensemble de partenaires industriels et académiques croissant. Il s'appuie principalement sur des langages standardisés tels que AADL, UML et ECORE. Il comprend une suite d'outils incluant des éditeurs graphiques pour les outils de modélisation génériques (e.g. ECORE, UML) et dédiés aux systèmes embarqués et temps-réel (e.g. AADL), des outils de transformation de modèles, des générateurs de code et un générateur de documentation. En plus des outils open source, AADL est supporté aussi par des outils commerciaux tels que STOOD [Dis 04] et AADL Inspector[1].

3.2.4 Rapide

Rapide est un EADL (*Executable Architecture Description Language*) développé à l'université de Stanford [Luc 95]. Il permet le développement des systèmes logiciels temps-réel et concurrents de grandes tailles. Il peut être considéré à la fois comme un ADL et un langage de simulation. D'un côté, il fournit les aspects formels pour définir des architectures logicielles exécutables (i.e. architectures pouvant être traitées automatiquement). De l'autre, il permet la simulation et l'analyse comportementale de ces architectures plus tôt dans le processus de développement des systèmes. Rapide se fonde sur le modèle d'exécution POSET (*Partially Ordered event SET*) qui permet la représentation explicite de la

[1] www.ellidiss.com/products/aadl-inspector/

synchronisation, la concurrence et la communication entre les composants logiciels en termes d'évènements.

3.2.4.1 Métamodèle

Nous avons construit un métamodèle pour Rapide (c.f. Annexe A.8) à partir de sa description[1]. Il est relativement grand du fait que Rapide est conçu essentiellement pour être suffisamment riche afin de créer des architectures formelles exécutifs. De plus, Rapide prend en compte les aspects dynamiques ainsi que les contraintes liées à l'architecture. Cette richesse peut être clairement constatée en regardant les sous-langages qui le composent :

- Langage de type : il fournit les éléments de base pour définir les types d'interfaces et de fonctions, et pour créer de nouveaux types d'interfaces à partir de ceux existants grâce au mécanisme d'héritage;
- Langage d'architecture : il étend le langage de type par des constructions pour définir des architectures d'interfaces. Une architecture d'interface spécifie explicitement les communications entre les composants d'un système logiciel à travers l'établissement des liens entre leurs interfaces. Dans Rapide, une interface peut contenir des règles exprimant le comportement des modules qui l'implémentent. L'architecture peut définir aussi des contraintes sur les interfaces et les connexions qui les relient;
- Langage exécutable : il permet la définition des modules, des structures de contrôle, et des types et fonctions standards (i.e. types de données et fonctions que l'on trouve ordinairement dans plusieurs langages de programmation et de simulation);
- Langage de contrainte : il fournit les éléments architecturaux pour exprimer les contraintes sur le comportement des modules, des fonctions et des architectures;
- Langage de patterns : ce langage est fondamental pour les trois autres langages puisque il fournit des constructions utilisées par les *processus* dans le langage exécutable, les

[1] http://complexevents.com/stanford/rapide/

comportements et les *connexions architecturales* dans le langage d'architecture, et les *contraintes* dans le langage de contrainte.

Notre métamodèle omet quelques éléments ayant une granularité fine (e.g. des instructions du langage exécutable) et représente les entités de première classe de Rapide telles que l'architecture, le composant, la connexion, l'interface, le module et la contrainte. Une architecture (métaclasse *Architecture*) consiste en des composants (métaclasse *Component*), des connexions entre eux (métaclasse *Connection*) et des contraintes sur le comportement de l'architecture (métaclasse *Constraint*). Un composant d'une architecture peut être une interface (métaclasse *Interface*), un module (métaclasse *Module*) ou une architecture qui implémente une interface. Une architecture est en fait un type restreint d'un module.

Une interface définit le type d'un module. Plusieurs modules peuvent alors implémenter la même interface. Rapide permet la vérification statique de la conformité de modules à leurs interfaces pour garantir la consistance des instances d'architecture. Une interface se compose de plusieurs parties (métaclasse *InterfacePart*) contenant plusieurs *éléments d'interface* (métaclasse *InteraceItem)*. Un élément d'interface peut être une action (métaclasse *Action*), une fonction (métaclasse *Function*), un service (métaclasse *Service*), une énumération (métaclasse *Enumeration*), ou même une interface. La visibilité de ces constituants dépend de la partie d'interface où ils sont déclarés. On distingue la partit publique (métaclasse *PublicPart*), la partie privée (métaclasse *PrivatePart*) et la partie externe (métaclasse *ExternPart*). Les constituants publics représentent les services fournis par le module qui sont utilisables par d'autres modules. Les constituants externes représentent les services requis par le module. Les constituants privés sont les services fournis par le module qui sont visibles uniquement aux autres modules de même type.

Les fonctions sont des blocs d'instructions possédant un nom et pouvant être appelés par d'autres parties d'une description Rapide. Les fonctions modélisent les communications synchrones entre les modules. Les actions, quand à elles, indiquent que les modules de leur interface peuvent observer et générer des évènements. En d'autres termes, les actions modélisent les communications asynchrones entre les modules.

L'interface peut définir un comportement (métaclasse *Behavior*) sous forme de contraintes (i.e. des machines à état) sur l'exécution des modules qui l'implémentent. Une contrainte est une règle de transition (métaclasse *TransitionRule*) se caractérisant par un déclencheur (*trigger*) qui fait basculer le module d'un état à l'autre, et un corps (*body*) qui exprime le basculement. Le corps s'exécute lorsque le déclencheur se produit. Dans le cas de Rapide, le déclencheur et le corps sont des *patrons d'évènements* (métaclasse *Pattern*).

Les connexions définissent les relations entre les interfaces requises et les interfaces fournies des composants. La forme la plus simple d'une connexion est celle qui relie deux patrons et est appelée une *connexion de base*. Elle fait appel aux services fournis dès que les services requis correspondants sont demandés. Une connexion définit alors un flux causal d'évènements ou des appels de fonctions entre composants. Les connexions sont dynamiques du fait qu'elles peuvent posséder des paramètres d'exécution ou relier des éléments dynamiques qui varient durant l'exécution. La partie connexion d'une architecture Rapide contient des règles de connexion (métaclasse *ConnectionRuleList*) et de création (métaclasse *ConnectionGenerator*). Une règle de connexion peut connecter les composants entre eux, l'interface de l'architecture à ses composants, ou les deux. Elle se compose d'un déclencheur, un opérateur et un corps. L'opérateur spécifie le type de connexion (i.e. connexion de base, connexion pipe, connexion d'agent), alors que le déclencheur et le corps peuvent être des patrons d'évènements. La règle de création est une condition qui contribue à la création de nouveaux composants (i.e. l'exécution de son corps implique généralement la création d'un objet).

Bien entendu, les modules sont les objets d'interfaces. Il s'agit d'implémenter les services déclarés dans une interface pour créer un composant prototype exécutable. Le lien entre un module et son interface doit être explicite dans sa déclaration. Cette exigence, en plus de la possibilité de définir un comportement et d'imposer des contraintes au niveau d'interface, permet la vérification automatique de la conformité du module à son interface et la validité de l'architecture, plus tôt dans le processus de développement du système. Le module se distingue d'une architecture par une partie d'initialisation (métaclasse *InitializationPart*), une partie de finalisation (métaclasse *FinalizationPart*), des processus (métaclasse *Process*), et un handler (métaclasse *Handler*). Ces parties optionnelles s'inscrivent dans le cadre de

l'expression de l'aspect exécution de Rapide (le langage exécutable). Un module peut être vu comme un programme multi-thread dont l'exécution commence par sa partie d'initialisation, suivie par l'exécution simultanée de ses processus, et se termine par la partie de finalisation. Le processus consiste en une liste simple d'instructions qui s'exécute une seule fois. Le module peut inclure un handler pour gérer les exceptions qui peuvent survenir pendant l'exécution.

Les contraintes sont des instructions déclaratives qui imposent des restrictions sur le comportement des composants et des connexions de l'architecture. Elles représentent des patrons d'évènements qui contraignent le comportement visible d'un module et doivent ou ne doivent pas se produire pendant l'exécution (i.e. des patrons d'évènements acceptables ou non acceptables). Les contraintes d'une architecture peuvent, par exemple, spécifier une certaine séquence de communication entre les composants (e.g. exigée par un protocole particulier). C'est à l'exécution qu'un comportement d'une architecture ou d'un module est vérifié contre leurs contraintes. Chaque contrainte dispose d'un filtre (métaclasse *Filter*) et un corps. Le filtre décompose le calcul, qui consiste en des évènements, en des sous-ensembles d'évènements beaucoup plus faciles à contraindre. Le corps d'une contrainte dénote ce qui est acceptable parmi les évènements produits par le filtre. Finalement, il se peut qu'une contrainte soit imbriquée (i.e. contient d'autre contraintes). Dans ce cas, les contraintes internes s'opèrent sur les évènements générés par le filtre de la contrainte englobante.

3.2.4.2 Outils Supports

En premier lieu, Rapide est un EADL et une suite d'outils de simulation. Le processus de modélisation dans Rapide commence par la création d'une architecture logicielle. Un modèle Rapide initial peut être créé en utilisant des outils graphiques de construction d'architecture ou un éditeur de texte. Le simulateur de Rapide (i.e. compilateur, linker, bibliothèque d'exécution) exécute le modèle pour produire une simulation. Ce qui est produit exactement est un ensemble d'évènements causals du comportement de l'architecture. Plusieurs outils peuvent être utilisés pour analyser le résultat de simulation :

- Un navigateur d'évènements : fournit une représentation graphique des évènements générés par la simulation, et permet leur manipulation et leur filtrage;

- Outils d'animation : décrivent l'exécution dans un environnement d'animation graphique et temps_réel;
- Un analyseur de contraintes : vérifie la conformité de la simulation par rapport aux contraintes formelles définies par le modèle.

Comme plusieurs ADLs, les outils disponibles[1] qui permettent la construction, l'analyse et la simulation des modèles Rapide sont des prototypes qui nécessitent des refontes pour atteindre la maturation. Malheureusement, la spécification active n'est pas supportée et la génération du code se limite à la construction des simulations exécutables décrites par le sous-langage exécutable de Rapide.

3.2.5 Darwin

Darwin est un ADL qui permet de spécifier la structure des systèmes distribués en termes d'instances de composants et leurs interconnexions, structurés hiérarchiquement [Mag 96]. C'est un langage de configuration qui supporte la représentation textuelle et graphique des structures statiques, fixées durant l'initialisation du système, et dynamiques, qui évoluent durant l'exécution. Les composants (ou les types de composant) qui constituent une architecture Darwin peuvent être primitifs ou composites. Les composants composites encapsulent des composants primitifs et/ou composites. Les composants s'interagissent par l'accès aux services qu'ils fournissent. Afin de spécifier précisément le comportement d'un programme Darwin, une transformation de ses éléments vers un formalisme, tel que π-calculus, peut être effectuée. Ce type de langage permet d'analyser et de décrire les systèmes concurrents en termes de processus indépendants, communiquant via des canaux et dont la configuration est changeable.

[1] http://complexevents.com/stanford/rapide/tools-release.html

3.2.5.1 Métamodèle

Le métamodèle est montré dans la figure 3.6. Il encode directement la syntaxe officielle du Darwin[1]. Une spécification Darwin (métaclasse *DarwinSpecification*) contient des composants (métaclasse *ComponentDeclaration*), des interfaces (métaclasses *InterfaceDeclaration*), des déclarations de constantes (métaclasse *constantDeclaration*), des déclarations externes (métaclasse ExternalDeclaration) et des assertions (métaclasse *AssertDeclaration*). La déclaration d'un composant définit un type de composant qui peut avoir une ou plusieurs instances (métaclasse *ComponentInstance*). Chaque instance est associée à un et un seul type de composant. Le composant fournit et/ou requiert une ou plusieurs interfaces qui peuvent regrouper des portails (métaclasse *Portal*) formant un portail composite. Chaque portail peut avoir une direction (*provide*, *require*, *import* ou *export*). Les portails d'un composant peuvent être attachés soit aux portails de ses sous-composants (cas d'un composant composite), soit aux portails des autres composants du système. Darwin offre la possibilité d'effectuer, partiellement ou complètement, un sous-typage d'un composant existant. La déclaration partielle (métaclasse *PartialComponentDeclaration*) d'un type de composant s'effectue par l'omission d'un ou de plusieurs paramètres du composant existant.

Les liaisons (métaclasse *Binding*) établissent des liens entre les instances de composants en connectant les interfaces fournies aux interfaces requises de même type. Grâce à l'autoréférence *componentDeclaration*, des composants composites, hiérarchiquement structurés, peuvent être représentés. Ils se déclarent par la définition des instances de composants encapsulés ainsi que leurs liaisons. Les assertions, soient déclarées au niveau du composant ou au niveau d'architecture, sont des expressions booliennes qui contraignent l'élaboration des composants et génèrent des erreurs en cas de violation. Ainsi, l'élaboration de composants Darwin peut être conditionnelle ou itérative en utilisant, respectivement, la structure conditionnelle *When* et la structure itérative *Forall*.

[1] http://research.cs.ncl.ac.uk/cabernet/www.laas.research.ec.org/c3ds/trs/papers/08.pdf

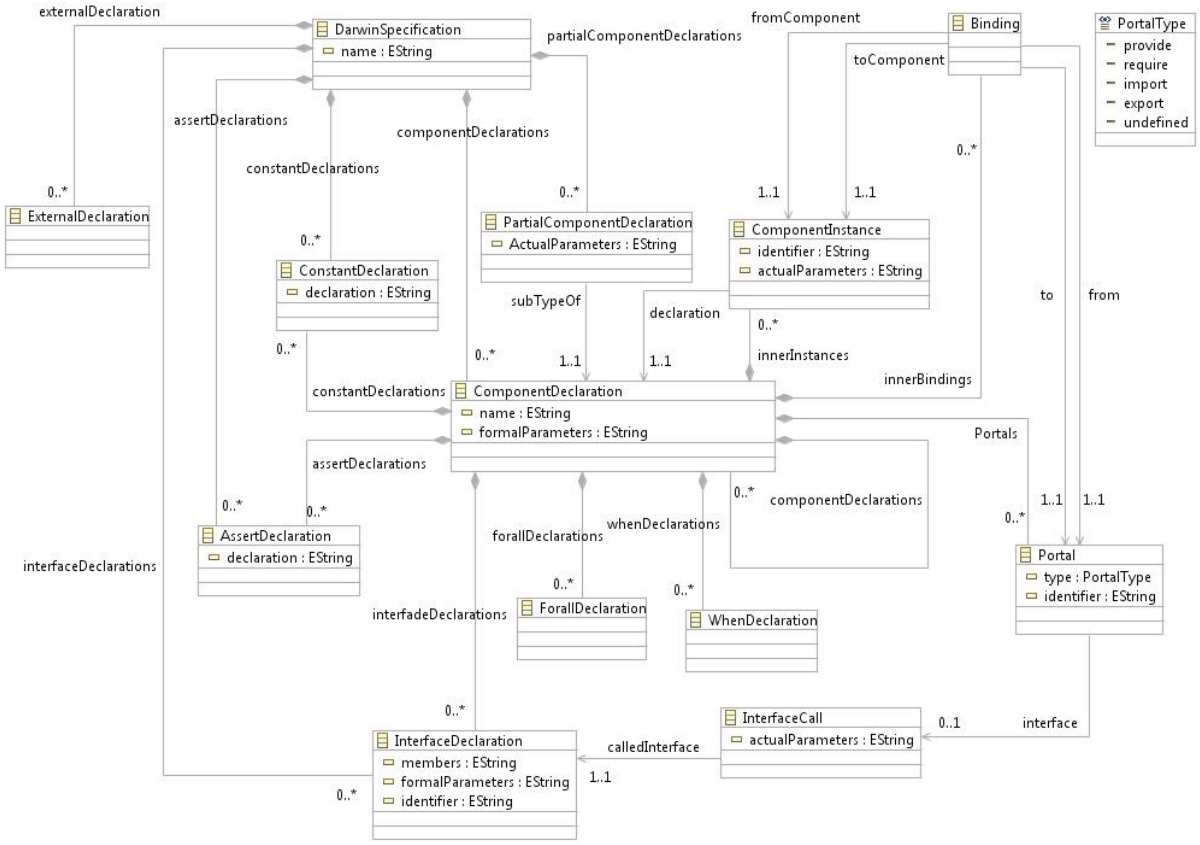


Figure 3.6. Métamodèle Darwin.

Notons qu'il est possible de supporter Darwin par d'autres formalismes (e.g. π-calculus) pour enrichir la description des systèmes distribués, modéliser leur comportement, et profiter des capacités d'analyse qu'ils fournissent. Le métamodèle en annexe A.9, par exemple, décrit la relation entre une spécification Darwin et une spécification écrite en FSP (*Finite State Process*) [Mag 99]. Ce métamodèle est composé, en fait, de deux métamodèles, un pour Darwin et l'autre pour FSP. Ce dernier représente un langage formel qui permet de modéliser l'aspect comportemental d'un système logiciel en termes de processus concurrents. La méta-association *relatedFSP* qui relie chaque composant (*ComponentSpecification*) à un processus à états finis (métaclasse *Process*), garantit la spécification comportementale des composants et permet de produire des automates analysables.

3.2.5.2 Outils Supports

Darwin est supporté par des outils d'aide à la construction, la vérification et la visualisation du système en cours de développement. En particulier, l'environnement de programmation *Regis* fournit un outil d'assistance proactive à la conception, la construction et la maintenance des systèmes parallèles et distribués [Ng 95]. Les programmes dans Regis consistent en des instances de composants interconnectées, dont la structure peut être décrite en utilisant Darwin, ou moyennant une représentation graphique. Le comportement du composant est implémenté en C++.

L'assistant permet de manipuler, via des éditeurs, les descriptions textuelles et graphiques de Darwin, de générer un code Darwin à partir d'une configuration graphique du programme, et de garantir la cohérence du système (e.g. si les interfaces fournies sont associées aux bonnes interfaces requises). Une fois la description d'architecture terminée, l'assistant passe à la compilation et l'exécution de la spécification Darwin/C++ pour construire l'application distribuée. La spécification proactive est supportée par l'assistant à travers l'ajout automatique des ports aux composants interconnectés ainsi que la propagation des modifications apportés aux types des ports vers les autres ports de l'interconnexion.

L'utilisation de π-calculus ou FSP pour modéliser les éléments de base de Darwin se révèle bénéfique lorsqu'il s'agit de décrire et vérifier les systèmes concurrents par des outils de vérification formelle (e.g. MWB[1], LTSA[2]).

3.2.6 Wright

Wright est un ADL qui permet de modéliser et d'analyser le comportement dynamique des systèmes concurrents [All 96]. Il se distingue par la définition explicite des connecteurs et par sa flexibilité du fait qu'il permet de spécifier à la fois la structure et le comportement de l'architecture logicielle. Sa sémantique formelle repose principalement sur le langage CSP

[1] http://www.it.uu.se/research/group/mobility/mwb

[2] http://www.doc.ic.ac.uk/ltsa/

(*Communicating Sequential Processes*) qui rend possible la modélisation du comportement et l'interaction de systèmes par des modèles algébriques de processus, et la concurrence via un mécanisme de synchronisation de processus par rendez-vous [Hoa 85]. La possibilité de vérifier formellement la cohérence et la complétude de l'architecture logicielle est l'une des caractéristiques les plus importantes de Wright. Il est primordial de vérifier si l'architecture logicielle est cohérente pour garantir la cohérence du système résultant. Savoir si la description d'architecture est complète préalablement à toute analyse est encore plus important, vu que l'omission d'un détail ou une information peut conduire à une mauvaise analyse.

3.2.6.1 Métamodèle

Dans [All 96], Wright est introduit avec une description formelle de sa syntaxe au format BNF, et ses concepts sont spécifiés informellement. Sur la base de ces informations, un métamodèle décrivant les aspects structuraux et comportementaux de Wright a été créé en adaptant UML2.0 [Gra 06]. Nous avons reconstruit ce métamodèle en utilisant ECORE (c.f. Annexe A.7). La partie qui décrit les aspects structuraux comporte les abstractions architecturales de base qui sont les composants (métaclasse *Component*), les connecteurs (métaclasse *Connector*) et les configurations (métaclasse *Configuration*). Un composant Wright décrit un calcul indépendant et localisé. Chaque composant contient deux parties : une interface et un calcul encapsulé (métaclasse *Computation*). L'interface consiste en un ensemble de ports (métaclasse *Port*). Chaque port représente une interaction à laquelle peut participer le composant. Sa spécification implique deux aspects concernant le composant : son comportement et ses besoins. Le port peut décrire partiellement le comportement du composant à travers la définition des propriétés que doit fournir le composant via ce point d'interaction, ou il peut indiquer ce que ce dernier attend de son environnement avec lequel il interagit (i.e. le système). La section de calcul fournit une description plus complète du comportement (i.e. l'implémentation des interfaces). Elle comprend les interactions qui sont décrites par les ports et la manière dont ils sont attachés pour former une totalité cohérente.

Le connecteur sert à modéliser l'interaction entre une collection de composants de manière explicite, pour accroître leur indépendance vis-à-vis les aspects relatifs à la

mécanique d'interconnexion et supporter l'analyse architecturale. Un connecteur Wright est décrit en termes de rôles (métaclasse *Role*) et de glues (métaclasse *Glue*). Il se caractérise par un type qui peut être instancié plusieurs fois dans différents contextes. En d'autres termes, il représente une connexion-type communément employée dans les architectures logicielles. Chaque rôle spécifie le comportement d'un participant à l'interaction puisque il indique comment ce dernier doit réagir durant la connexion. La glue d'un connecteur décrit comment les participants se communiquent pour créer une interaction. Comme la partie de calcul d'un composant, la glue d'un connecteur représente la spécification complète de son comportement.

La configuration décrit l'architecture complète du système, en termes de composants, de connecteurs, d'instances de composants et de connecteurs, et d'attachements (métaclasse *Attachement*). Chaque instance doit avoir un nom unique pour éviter toute ambiguïté dans l'architecture. Les attachements définissent la topologie de configuration en attachant chaque port d'un composant à un rôle d'un connecteur. Wright supporte la description hiérarchique, et en particulier celle de la partie de calcul d'un composant et la glue d'un connecteur. Le calcul (ou la glue) peut être une simple spécification ou une configuration. Ainsi, Wright prend en compte la notion de style qui permet la définition d'une famille de systèmes partageant les mêmes propriétés. Un style est défini par les types de composants, de connecteurs et d'interfaces déclarés dans la description, par le paramétrage et par la définition des contraintes. Le style architectural impose des contraintes de structuration (e.g. le type et le nombre de ports et de rôles dans une description) et dénote les décisions de conception et les caractéristiques communes aux architectures logicielles similaires.

La partie qui décrit les aspects comportementaux d'une architecture est basée sur les concepts de CSP. L'unité de base d'une spécification CSP est l'évènement (métaclasse *Event*). Elle représente une action importante qui peut se répéter plusieurs fois durant l'exécution. Un évènement peut être initialisé et observé par un processus. Un processus (métaclasse *Process*) est composé d'évènements décrivant le comportement d'un élément de Wright (e.g. la partie de calcul d'un composant, la glue d'un connecteur, le port et le rôle). Il existe plusieurs types de processus : terminaison avec succès, interblocage, parallélisme, choix déterministe et non déterministe, nommages des processus et ordonnancement d'évènements. Un processus peut fournir et recevoir des données portées par des évènements. Vu que la description des

éléments structurels de Wright est basée sur des expressions CSP, une relation de composition existe entre la partie structurelle et comportementale du métamodèle.

3.2.6.2 Outils Supports

Wright est un pur langage formel de spécification d'architecture qui ne dispose pas d'environnement d'exécution. Il existe, cependant, des traducteurs permettant l'analyse architecturale de façon indirecte, en traduisant les expressions Wright vers d'autres formalismes de spécification, accompagnés par des outils d'analyse[1]. À tire d'exemple, Wr2fdr est un traducteur qui permet de produire une spécification CSP à partir d'une description Wright. Le langage CSP possède un support outillé commercial nommé FDR[2] qui permet de vérifier et valider automatiquement les propriétés décrites en CSP. La plupart de ces traducteurs sont des prototypes ce qui met en doute leur fiabilité (e.g. [Ham 12]).

3.2.7 ArchJava

ArchJava est une extension du langage Java qui fournit un fort couplage entre l'architecture et son implémentation [Ald 02, Ald 03]. L'intégration des aspects architecturaux à Java a pour but d'assurer la conformité du code à son architecture, et de faciliter la conception des architectures pour les programmeurs habitués à java. Les ADLs classiques utilisent leur propre syntaxe pour décrire les contraintes architecturales et les communications entre les composants, cependant, ceux qui utilisent une spécification explicite des connecteurs au niveau architecture ne conservent pas au niveau implémentation cette explicité, de telle sorte que les connecteurs soient dispersés à travers les composants, et donc le concept clé des ADLs, qui considèrent le connecteur comme entité à part entière, ne sera plus tangible. ArchJava élimine ces problèmes par l'utilisation d'un mécanisme qui garantit l'intégrité des communications entre l'architecture d'un système et son implémentation [Ald 03].

---

[1] http://www.cs.cmu.edu/~able/wright/wright_tools.html

[2] https://www.cs.ox.ac.uk/projects/fdr/

3.2.7.1 Métamodèle

Définir un métamodèle pour ArchJava revient à définir un métamodèle pour le langage générique Java, ce qui est difficile, et à étendre ce dernier par les constructions architecturales. Même si quelques métamodèles Java existent actuellement (voir chapitre 4), l'intégration correcte des nouveaux éléments au sein d'un méga modèle contenant des dizaines d'éléments, n'est pas une tâche aisée à effectuer. C'est pourquoi, jusqu'à ce jour, il n'existe pas encore un métamodèle ArchJava.

3.2.7.2 Outils Supports

Un travail a été mené pour créer un compilateur qui met en œuvre les concepts de l'ADL ArchJava. Le compilateur est disponible sur Internet[1] et a été testé sur des applications de grosses tailles avec succès. Il permet de construire une architecture et d'assurer l'intégrité de ses communications par la génération du bytecode, destiné à la machine virtuelle Java, qui soit conforme aux contraintes architecturales encodées dans la description. Ainsi, Il présente une infrastructure dynamique qui permet une modification temps-réel des connexions entre les composants, c'est à dire la possibilité de connecter et déconnecter les composants pendant l'exécution. Le compilateur ArchJava est un outil en ligne de commande semblable au compilateur javac (le compilateur java). L'aspect visuel est aussi pris en considération par la création d'un outil simple de visualisation (n'est malheureusement pas disponible en ligne) qui permet de générer des diagrammes graphiques à partir du code source ArchJava [Ald 02]. Malheureusement, la composition graphique des composants n'est pas supportée et le codage reste le moyen principal pour créer l'architecture. Un IDE (*Integrated Development Environment*) intégré avec l'environnement AcmeStudio[1] a été proposé en 2004 pour exécuter et générer des programmes ArchJava à partir de descriptions Acme, mais il a été rapidement abandonné par les auteurs.

[1] http://archjava.fluid.cs.cmu.edu/software/index.html

3.3 Discussion

L'ensemble d'ADLs représentatifs que nous avons sélectionnés pour notre investigation inclut un ADL générique et standard (Acme), deux ADLs destinés à la description des systèmes distribués et temps-réel (MetaH et AADL), trois ADLs permettant l'analyse statique et dynamique d'architecture (Rapide, Darwin et Wright) et un ADL étendant un langage générique (ArchJava). Il existe d'autres ADLs que nous n'avons pas cru devoir mentionner (e.g. Unicon, C2, SADL, Aesop [Gar 95] et Weaves [Gor 94]), parce qu'ils ont presque les mêmes propriétés que ceux cités ci-avant.

Des études antérieures (e.g. [Med 00, Ves 93]) concluent que les différents ADLs partagent les notions de composant, de connecteur et de configuration. Notre étude vient confirmer cette conclusion. Les métamodèles qui décrivent les ADLs que nous avons étudiés (excluant l'ADL ArchJava qui n'en possède pas) contiennent des éléments qui représentent ces trois notions. Les principales constatations issues de cette étude sont résumées ci-après :

- La simplicité relative : par rapport aux langages génériques, les ADLs sont plus simples et plus petits à cause de leur spécificité au domaine d'architecture, ce qui restreint leur pouvoir d'expression et facilite leur manipulation. AADL et, à un degré moindre, Rapide sont complexes mais restent cependant plus réduits qu'un langage générique;
- La disponibilité du métamodèle : tous les ADLs, étant relativement simples, sont décrits au moins par une grammaire (i.e. des règles de production). La création de leur métamodèle ne pose donc pas un problème (sauf pour ArchJava qui est un cas spécial). Nous l'avons vu, les métamodèles de Rapide, Darwin, et Wright présentés ici, sont fondés sur leur grammaire (disponible en ligne). Pour certains ADLs, les métamodèles sont déjà construits et publiés dans la littérature scientifique (e.g. Acme[1], MetaH[2] et AADL[3]);

[1] https://github.com/grammarware/slps/blob/master/topics/grammars/metamodels/ACME/ACME.ecore

[2] https://github.com/grammarware/slps/blob/master/topics/grammars/metamodels/METAH/METAH.ecore

[3] http://www.aadl.info/aadl/testsite/tool/metamod.html

- Le mauvais état des outils d'assistance : dès la naissance des ADLs, qui remonte aux années 1970 avec l'apparition de SDL (*Specification and Description Language*) [Bel 89], beaucoup d'outils ont été développés pour les assister. On ne doit pas s'étonner qu'une grande partie de ces outils ne soient plus disponibles en ligne ou soient des prototypes, parce que, dans la plupart des cas, ils ont été abandonnés par leurs auteurs (cas des prototypes issus de la recherche académique). Pour utiliser un outil gratuit non disponible en ligne, ou obtenir son code source, la seule solution semble être de contacter les auteurs. À titre d'exemple, des défaillances ont été détectées dans le traducteur Wr2fdr [Ham 12], qui permet de produire une spécification CSP à partir d'une description Wright. Afin de les corriger et le faire évoluer, les auteurs de Wright ont été contactés pour récupérer son code source (représenté par 16000 lignes de code C++). L'exemple de Rapide[1] est encore plus indicatif. Ses outils d'accompagnement n'existent plus, le lien offert pour les télécharger est expiré[2], et même en contactant les créateurs de l'ADL, vous ne les obtiendrez certainement pas, parce qu'ils n'avaient pas l'intention, au départ, de créer des outils portables et réutilisables. Bien que Rapide ne soit pas directement supporté par des outils, ses concepts sont utilisés par la technologie de traitement d'évènements complexes *CEP* (*Complex Event Processing*) et beaucoup d'organisations ont développé leurs propres outils de CEP dont certains sont maintenant commercialisés. L'environnement Regis et d'autres outils supportant Darwin[3] sont dans un état similaire à celui de l'ensemble d'outils de Rapide (ils ne sont plus actifs). En contrepartie, AADL a bénéficié, pendant plus de quinze ans, des recherches extensives et de développement appuyés sur une collaboration entre des partenaires académiques et industriels, ce qui a abouti à la création de nombreux outils d'accompagnement commerciaux et gratuits[4]. De même, Acme, avec sa bibliothèque extensible AcmeLib, fournit une infrastructure complète de manipulation et d'intégration

[1] http://complexevents.com/stanford/rapide/tools-release.html

[2] ftp://pavg.stanford.edu/pub/Rapide-1.0/toolset/

[3] http://www.imperial.ac.uk/computing/research/dse/software/

[4] https://wiki.sei.cmu.edu/aadl/index.php/AADL_tools

de descriptions d'architecture dans son environnement AcmeStudio[1] (Dernière modification : Février 2009).

D'un point de vue général, les ADLs, tels qu'ils sont connus aujourd'hui, se caractérisent par le mauvais état des outils associés ainsi que la disponibilité de leur métamodèle et leur syntaxe concrète. La première caractéristique représente une difficulté qui entrave l'utilisation des ADLs, raison pour laquelle elle doit être surmontée, alors que la deuxième est un avantage que nous devons mettre à profit pour atteindre cet objectif.

3.4 Conclusion

Nous avons cité les concepts de base de l'architecture logicielle qu'on trouve dans la majorité des ADLs, à savoir les composants, les connecteurs et les configurations. Nous avons étudié quelques ADLs parmi les plus connus et nous avons concentré notre étude sur deux points : la métamodélisation et les outils supports. Le premier point est très important dans le contexte de modélisation et est en relation avec le deuxième, puisque un métamodèle décrivant un ADL représente un fondement solide et rigoureux pour la construction d'outils supports.

Dans le chapitre suivant, nous présenterons une approche dirigée par les modèles pour la construction d'outils manipulant les architectures. L'idée est de supporter les ADLs par un processus de développement d'outils, basé sur un environnement de développement de langages spécifiques et une représentation explicite de leur métamodèle. L'explication de l'approche sera soutenue par un cas d'étude détaillé.

[1] http://www.cs.cmu.edu/~./acme/AcmeStudio/index.html

4 Approche dirigée par les modèles pour les ADLs

L'approche MDA a engendré un changement radical dans le domaine de l'ingénierie du logiciel. Elle préconise l'utilisation des modèles tout au long du cycle de développement logiciel. Un des points clés de cette approche est la possibilité de transformer les modèles d'un niveau d'abstraction vers un autre. Cette possibilité a permis de décrire le processus de développement par des modèles et transformation de modèles. D'un autre côté, les ADLs sont inventés pour définir formellement des modèles abstraits décrivant l'architecture logicielles. Suite à une investigation menée sur l'état actuel des outils supports des ADLs, nous avons constaté que la construction de tels modèles, leur transformation, leur analyse, leur visualisation et leur intégration dans les outils de modélisation existants n'ont été supportées que partiellement par des outils qui sont, dans la plupart des cas, beaucoup plus des prototypes que des logiciel matures.

Les ADLs étant des DSLs, n'ont pas profité des environnements de développement des DSLs (language workbenches) et leur outillage support. Ce chapitre présente un travail de modélisation qui vise l'application des techniques de l'ingénierie des modèles, à savoir la métamodélisation et la génération du code, pour décrire les ADLs au niveau méta et les assister par des outils facilitant leur usage. Pour valider l'approche proposée, nous avons choisi l'ADL ArchJava qui nous semble le plus difficile à métamodéliser (i.e. créer un métamodèle qui le décrit) et qui soufre d'un manque d'outils d'accompagnement.

4.1 Notre Approche

Les limitations dont nous avons discutées précédemment, surtout le fait que peu d'outils d'analyse et de manipulation d'architectures ont été conçus, nécessitent une démarche qui tire profit des techniques de modélisation inexploitées. Afin de pouvoir intégrer le maximum d'outils de modélisations existants, nous avons proposé une approche dirigée par les modèles qui consiste, dans un premier temps, à créer un métamodèle décrivant l'ADL (sa syntaxe abstraite). Cette première étape sera ignorée dans le cas où l'ADL dispose d'un métamodèle. En effet, les ADLs sont des langages spécifiques au domaine d'architecture et sont plus simple par rapport aux langages génériques tels que Java et C++. Cette simplicité

explique le fait que les métamodèles de plusieurs ADLs existent déjà. Une fois construit, le métamodèle va servir de base à la création des outils et aux activités d'analyse et de manipulation d'architectures.

Dans un second temps, l'approche consiste à définir la syntaxe concrète de l'ADL. Dans ce cadre, le DSL *CS* de l'environnement EMFText est utilisé (voir chapitre 5, section 5.1). Une version initiale de la syntaxe concrète peut être générée automatiquement depuis le métamodèle mais nécessite certainement des modifications pour qu'elle décrive réellement la syntaxe du langage. À partir de la syntaxe abstraite (le métamodèle) et la syntaxe concrète (la spécification cs), EMFText génère un parseur, un générateur de code, un éditeur de texte, une API riche de manipulation et de personnalisation, ainsi que d'autres outils logiciels. Après l'étape de génération du code, une étape de personnalisation des outils générés aura lieu pour les adapter aux besoins. La figure 4.1 montre ces étapes.

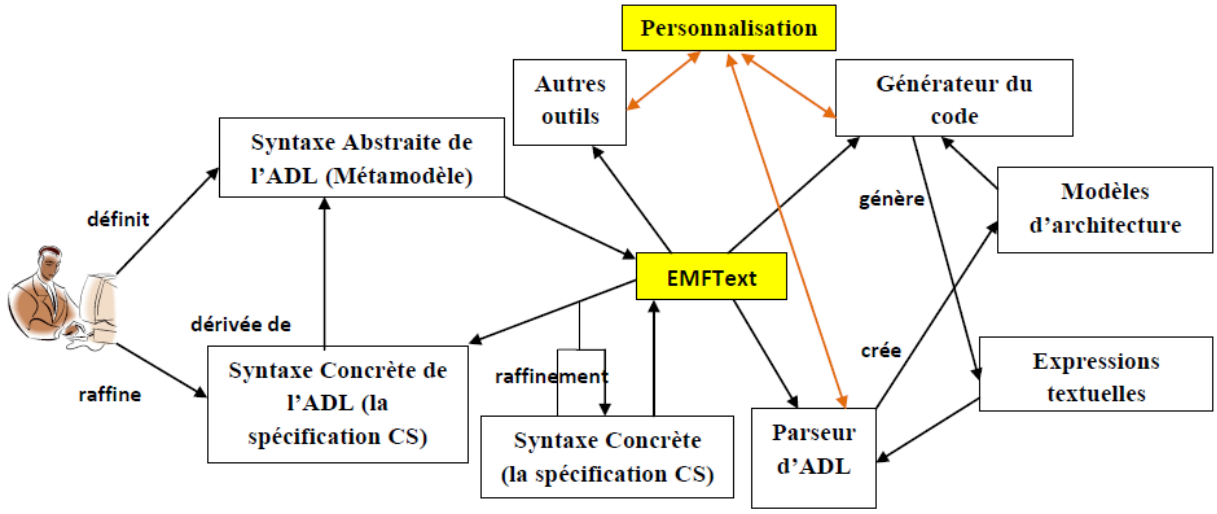


Figure 4.1. Approche de développement.

4.2 Cas d'Étude : ArchJava

Pour illustrer la validité de notre approche, nous nous basons sur l'ADL ArchJava. Le choix de ce langage est dû principalement, comme montré dans le chapitre précédent, à l'absence d'un métamodèle explicite qui définit sa syntaxe abstraite. C'est un cas extrême vu que la création d'un métamodèle d'un langage étendant Java est très difficile. Tout d'abord, un

métamodèle Java doit être défini puis une extension de ce dernier doit être réalisée. Au lieu de définir le métamodèle Java en partant de zéro, nous avons choisi de sélectionner, parmi les métamodèles Java que nous avons collectés (section 4.4.1), le plus approprié pour la tâche à accomplir, selon des critères de choix que nous allons introduire dans la section suivante.

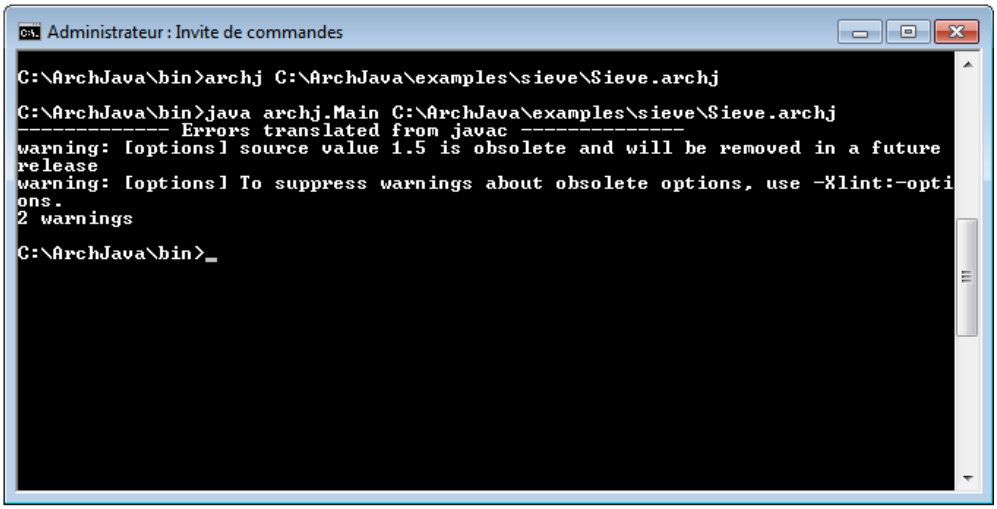


Figure 4.2. Compilateur ArchJava.

La deuxième raison justifiant ce choix est la difficulté de développer des outils manipulant les descriptions ArchJava et l'incapacité des outils associés[1] à les intégrer dans les outils de modélisation existants. Si nous prenons le cas du compilateur, qui est un outil en ligne de commande (figure 4.2), nous voyons que l'analyse du code est en fait découpée en deux phases principales : le code ArchJava est tout d'abord traduit en Java où chaque composant correspond à une classe Java portant le même nom, chaque port et chaque interface de port est compilé en une interface Java et chaque connexion se traduit en une classe de connexion qui implémente toutes les interfaces des ports connectés. Le compilateur javac est ensuite invoqué pour compiler les fichiers Java générés. Par exemple, le fichier *Sieve.archj* du l'exemple sieve (figure 4.3) téléchargeable avec le compilateur, qui contient une architecture ArchJava (c.f. Annexe B.2) décrivant le crible d'Ératosthène[2], se traduit en six fichiers java et

[1] http://archjava.fluid.cs.cmu.edu/software/index.html

[2] https://fr.wikipedia.org/wiki/Crible_d'Ératosthène

dix-sept fichiers class (bytecode) comme le montre la figure 4.4. Les erreurs de compilation ne sont détectées qu'après la compilation par javac (i.e. pendant la deuxième phase), ce qui est assez atténuant surtout avec l'absence d'un IDE qui facilite la visualisation et la correction instantanée des erreurs.

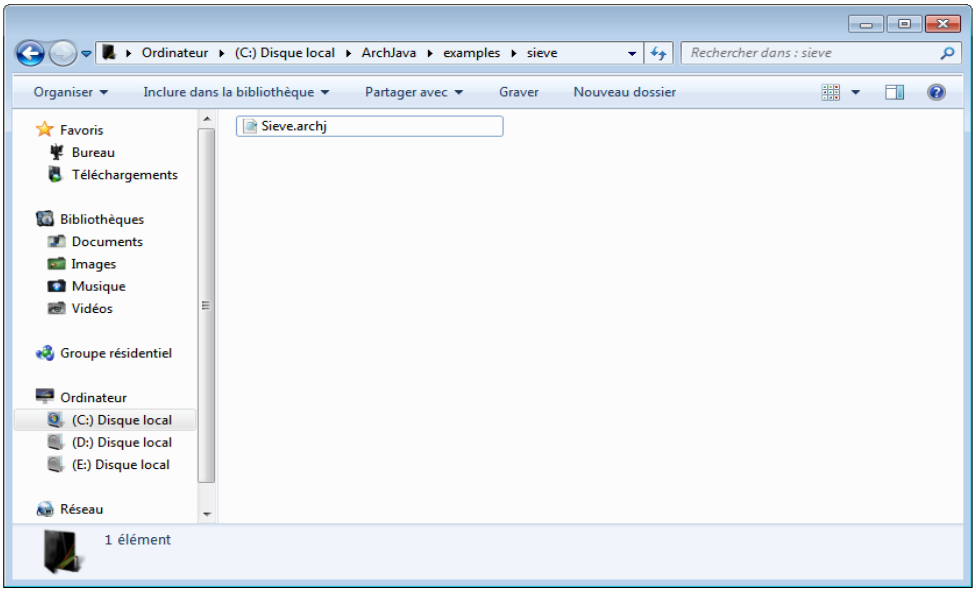


Figure 4.3. Répertoire sieve avant la compilation.

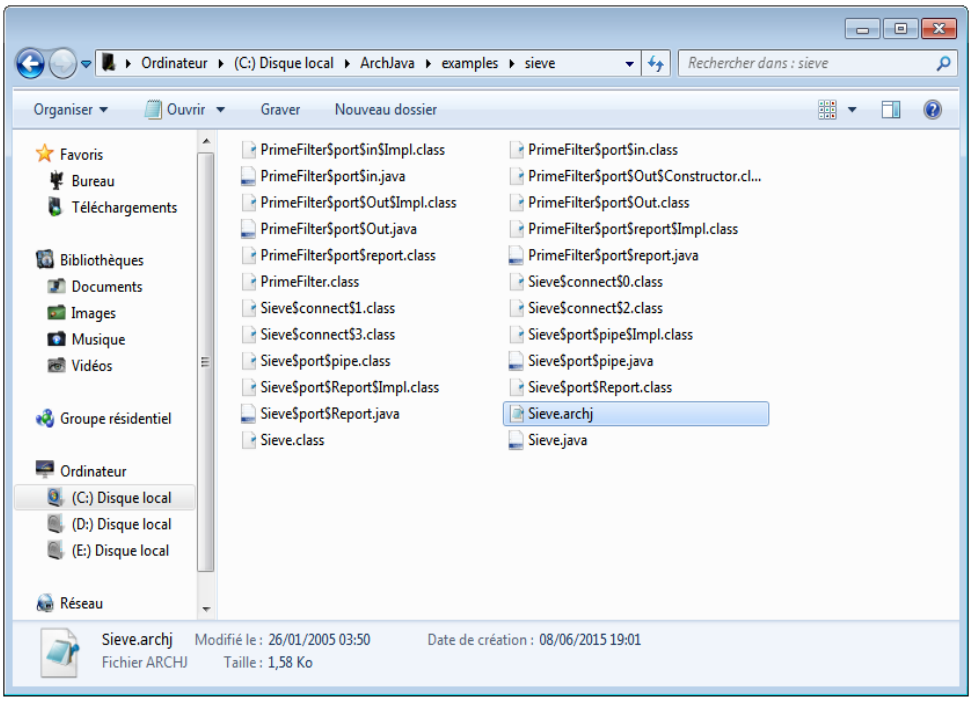


Figure 4.4. Répertoire sieve après la compilation.

4.3 Critères de Choix du Métamodèle Java

La sélection d'un métamodèle Java adéquat pour l'extension désirée passe par une étude profonde des métamodèles existants. Chaque métamodèle se différencie par des propriétés plus ou moins désirables. Dans cette section nous citons les critères que nous avons retenus pour comparer les métamodèles Java et choisir le plus convenable à employer dans la création du métamodèle ArchJava.

Niveau de Granularité

Les métamodèles Java peuvent être divisés en deux groupes selon leur niveau de granularité : métamodèles à granularité grossière et à granularité fine. Le premier groupe inclut les métamodèles qui définissent les constructions de base du Java mais ne prennent pas en compte le corps de la méthode (excluant les instructions d'accès aux variables et les invocations de méthodes). Vu que ArchJava définit de nouvelles instructions de méthode (e.g. l'expression *connect*), le métamodèle candidat doit avoir une granularité fine.

Modélisation des sémantiques

Les analyses sémantiques sont nécessaires dans le domaine des langages dédiés. La compilation est un exemple claire de l'analyse sémantique. La liaison de nom (i.e. l'assignation d'un nom à l'élément désigné), par exemple, qui est primordiale dans plusieurs activités telles que la compilation, ne peut être effectuée qu'avec une analyse sémantique. Ce type d'information doit être reflété dans le métamodèle Java pour permettre la création des modèles ArchJava complets à partir des représentations textuelles d'architectures, ainsi que les analyses sémantiques et la génération correcte et fiable du code à partir de ces modèles, sans aucune perte ou détérioration des informations.

Complétude

Il est préférable d'avoir un métamodèle qui couvre entièrement le langage Java. Idéalement, un modèle ArchJava doit préserver toutes les informations contenant dans la description textuelle d'une architecture, y compris les commentaires.

Représentation

Un métamodèle Java qui est représenté par un formalisme de modélisation standard fournit une meilleure intégration au sein des outils de modélisation compatibles.

Dépendance à Java

Il existe dans la littérature des métamodèles qui modélisent à la fois Java et d'autres langages de programmation. Ces métamodèle sont indépendants du langage Java, et sont inappropriés pour la création du métamodèle ArchJava car ils définissent souvent des constructions floues (i.e. l'utilisation des éléments pouvant représenter plusieurs concepts dans différents langages de programmation).

4.4 Étude comparative

Nous avons collecté les divers métamodèles qui peuvent être utilisés pour décrire tout ou partie du langage Java. Nous introduisons, pour chacun, une brève description de ses caractéristiques et ses limitations. La spécification du langage Java (*JLS* ou *Java Language Specification*) définit le métamodèle Java comme « une réflexion du langage Java » [Gos 14]. Cette définition est très vaste puisque le terme « réflexion » a besoin d'être clarifié. Qu'est ce qui est exactement reflété? Vu l'absence d'une définition claire de ce qu'est un métamodèle Java, l'inspection est basée essentiellement sur notre définition :

« *un métamodèle Java est un modèle qui définit la syntaxe abstraite du langage Java et décrit, au moins, ses constructions de base, leurs attributs et leurs relations. Une construction de base de Java est l'implémentation Java d'un concept de base de l'orienté objet* ».

Une chose importante à noter est le fait que le processus d'inspection adopté pour collecter les métamodèles Java, qui correspondent à cette définition, ne pose aucune restriction sur la manière dont les constructions de Java sont représentées et nommées dans un métamodèle donné. En d'autres termes, tout métamodèle dont les éléments peuvent être mappés, d'une manière ou d'une autre, aux constructions de Java, et tant qu'il répond à la définition, sera reconnu comme étant un métamodèle Java. Par exemple, deux métamodèles peuvent utiliser deux noms différents pour désigner le même concept (e.g. *attribute* et *field*, *method* et *operation*, etc.). Ainsi, l'existence de liens de conformité entre un métamodèle et les

concepts de Java ne signifie pas que le premier est un métamodèle Java, parce qu'il peut représenter, par exemple, les sémantiques dynamiques de Java. (e.g. diagrammes d'interaction d'UML). De plus, seulement les métamodèles ayant une représentation explicite sont considérés.

4.4.1 Métamodèles Java

Métamodèle UML

UML est un langage générique de modélisation créé par OMG pour la spécification, la visualisation, la construction et la documentation des artéfacts des systèmes logicielles [Boo 05]. Il permet la description d'un système selon différents perspectives et à différents niveaux d'abstraction. Cela est possible grâce à l'ensemble de diagrammes qu'il fournit. Chaque diagramme représente une facette du système. Deux types de diagrammes sont définis par le métamodèle UML : Diagrammes structurels et diagrammes comportementaux. Les diagrammes structurels (e.g. diagramme de classes, diagramme de composants, diagramme d'objets, diagramme de packages) représentent les informations structurelles qui montrent comment les éléments du système sont structurés et reliés entre eux. Les diagrammes comportementaux (e.g. diagramme de cas d'utilisation, diagramme d'activité, diagramme états-transitions, diagramme d'interaction) modélisent les interactions du système et une variété de ses propriétés durant l'exécution. Le métamodèle UML décrit les concepts d'UML, leurs attributs, leurs contraintes, et leurs relations. Il fournit une notation riche, ayant déjà fait ses preuves dans l'industrie, pour modéliser les éléments des systèmes logiciels.

Ce qui nous intéresse, parmi les artéfacts du système, est le code source écrit en Java. UML est utilisé pour représenter les programmes Java dans plusieurs travaux (e.g. [Ast 02, Coo 04, Har 00, Kat 05, kes 04, Par 11, Vee 14]) pour des raisons diverses et variées (e.g. analyse du code, transformation du code, visualisation du code, documentation du code, génération du code, l'ingénierie et la rétro ingénierie). Accepter le métamodèle UML comme une représentation du code Java au niveau méta a comme avantage l'exploitation des outils d'UML. Bon nombre d'outils logiciels, commerciaux et gratuits, supportent UML [Eic 09]. Ce dernier est devenu un standard de facto de la modélisation. De plus, le métamodèle est extensible (e.g. par le mécanisme de *profile UML*). Cependant, sa nature générique et le fait

que son métamodèle n'est pas spécifique à java, posent plusieurs problèmes lors de la description des constructions propres à Java. Premièrement, seule une petite partie du métamodèle peut être utilisée pour représenter un sous ensemble de constructions Java, i.e. le diagramme de classes qui reflète des concepts tels que les classes, les méthodes, les attributs et l'héritage (c.f. Annexe A.2). Deuxièmement, étendre le métamodèle pour combler le fossé entre Java et UML est une tâche difficile, surtout pour ceux qui ne maîtrisent pas les deux langages, parce que l'intégration de nouveaux éléments dans le métamodèle, si elle est mal effectuée, peut avoir un impact important sur les outils de support ainsi que sur la validité du métamodèle lui-même.

Extensions du métamodèle UML

Dans cette catégorie, nous regroupons les extensions et profiles UML qui peuvent être utilisés pour modéliser le langage Java. Selon la spécification d'UML, « un profil définit des extensions limitées d'un métamodèle de référence dont le but est l'adaptation de ce dernier à une plateforme ou un domaine particulier » [Uml 06]. OMG a introduit un profile UML qui adapte UML à la plateforme EJB (*Entreprise JavaBeans*) [Ejb 04]. Il couvre, en plus de l'architecture EJB, le langage Java. Plus particulièrement, il définit un sous ensemble de constructions Java (e.g. packages, classes, interfaces, champs, méthodes). L'utilisation de ce profile réduit l'effort nécessaire à l'adaptation du métamodèle UML, cependant, il ne spécifie pas le corps de la méthode.

Une autre extension appelée *GrammyUML* a été proposée par Gorp et al. [Gor 03] pour maintenir la consistance entre le modèle de conception et le code source correspondant. L'objectif étant de supporter la refactorisation dans tous les langages de programmation orientés objets, en offrant un modèle avec suffisamment de détails pour manipuler les informations au niveau des instructions. L'extension (figure 4.5) consiste à relier une méthode à son corps (ses instructions). Toutefois, le métamodèle est indépendant de Java et nécessite donc plusieurs modifications pour modéliser les constructions spécifiques à Java. À notre connaissance, aucune autre extension UML à ce jour n'est conforme à notre définition.

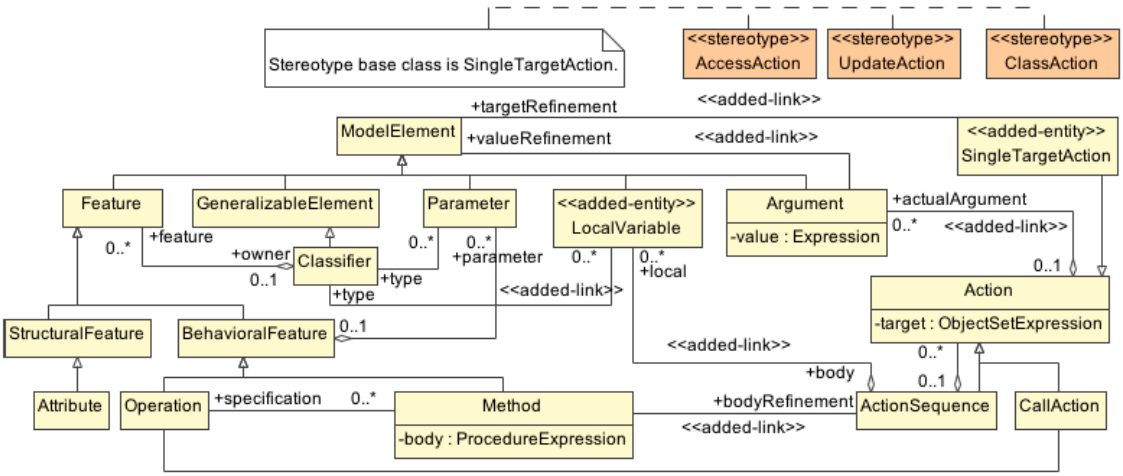


Figure 4.5. Métamodèle GrammyUML [Gor 03].

UNIQ-ART

Le métamodèle UNIQ-ART est une tentative vers l'unification de la représentation des dépendances structurelles des programmes orientés objets et procéduraux [Şor 13]. Il est essentiellement introduit pour supporter les activités de retro ingénierie. Ce métamodèle est indépendant de Java et fait abstraction de plusieurs concepts communs aux différents langages de programmation en utilisant les mêmes éléments structurels (figure 4.6). Par exemple, les classe/les modules et les méthodes/les fonctions sont représentés dans le métamodèle, respectivement, par *AbstractDataType* et *Function*. Les packages de Java et les directives de C sont définis par *UpperUnit*. Les relations se divisent en deux groupes : les agrégations et les dépendances. Les agrégations sont modélisées par la relation *isPartOf* alors que les dépendances comprennent les relations *import* (importation), *extends* (héritage) *isOfType* (typage) et *accesses* (accès).

Il est clair que UNIQ-ART peut être utilisé comme un métamodèle Java pour représenter les structures de dépendance du code Java. Sa nature générique favorise la réutilisation et permet la manipulation des programmes à un haut niveau d'abstraction. Néanmoins, il est superflu car il sépare les éléments structurels (*ProgramPart*) des relations (*AgregationRel* et *DependencyRel*) au niveau du métamétamodèle, ce qui signifie que l'adaptation de ses éléments aux concepts de Java ainsi que le fait de faire relier ses éléments

structurels par les relations, sont à la charge de l'utilisateur. De plus, ce métamodèle ne définit pas les instructions et les expressions.

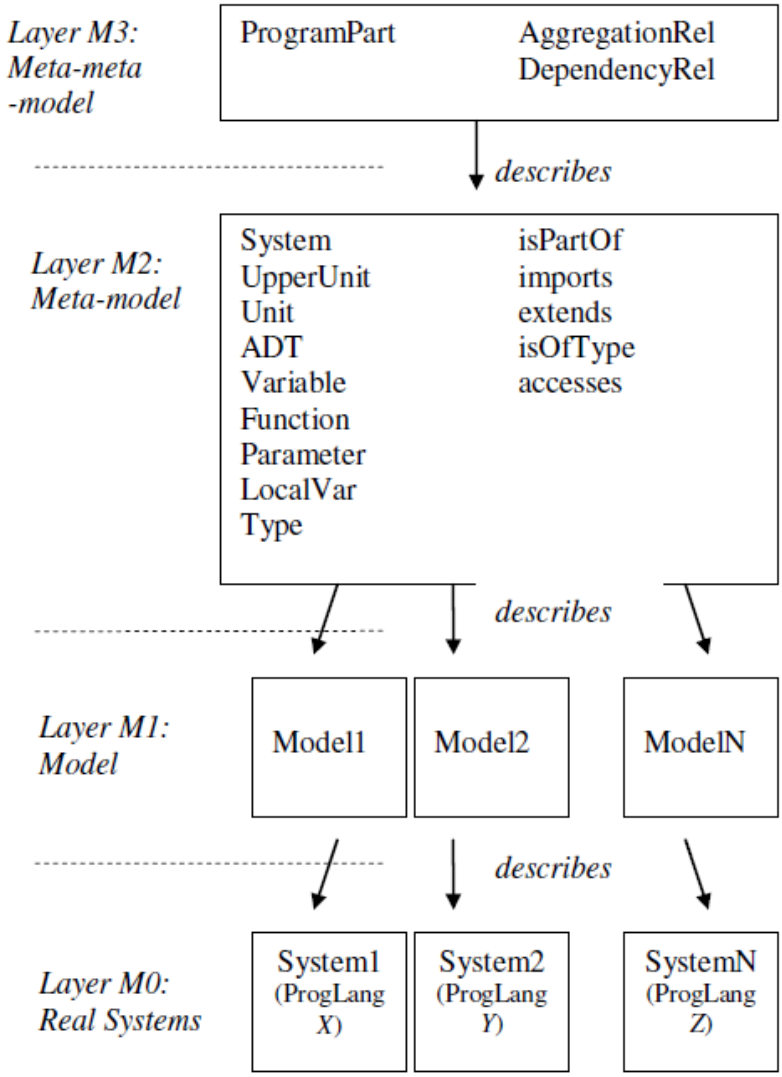


Figure 4.6. Architecture de quatre niveaux de UNIQ-ART [Şor 13].

DMM

DMM (*Dagstuhl Middle Metamodel*) est un modèle utilisé pour représenter les entités d'un programme dans les applications de retro ingénierie [Let 04]. C'est un format d'échange qui supporte l'interopérabilité entre les outils. Comme son nom l'indique, DMM est situé entre les arbres syntaxiques (*ASTs : Abstract Syntax Trees*) qui sont des modèles de bas niveau, et les modèles architecturaux de haut niveau. Ce métamodèle est divisé en quatre sous-

hiérarchies : un modèle de haut niveau qui définit les métaclasses racines des trois autres hiérarchies et leurs relations (figure 4.7), la hiérarchie *ModelObject* qui représente une vue abstraite d'un programme (figure 4.8), la hiérarchie *SourceObject* modélisant des blocs de codes qui définissent les entités conceptuelles d'un langage (figure 4.9), et la hiérarchie d'association qui modélise les relations entre les objets DMM (figure 4.10). La deuxième hiérarchie (*ModelObject*) comprend les éléments fondamentaux du code source tels que les packages, les classes, les attributs, les méthodes, les types, les paramètres, les valeurs, etc. La hiérarchie *SourceObject* définit les morceaux de code qui représentent les déclarations, les commentaires, les références, etc. La hiérarchie d'association énumère les différents types d'associations entre les éléments du métamodèle des deux hiérarchies précédentes.

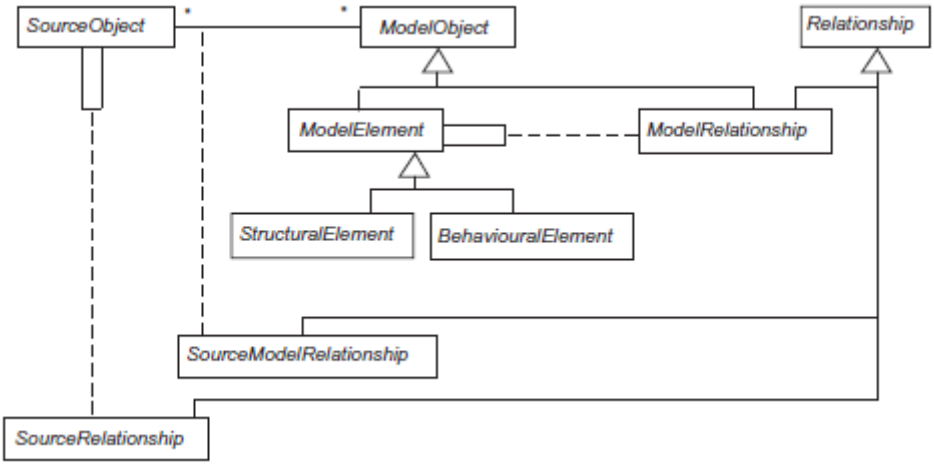


Figure 4.7. Modèle de haut niveau de DMM [Let 04].

DMM est représenté par UML et peut être utilisé comme un schéma pour les langages procéduraux et orientés objets courants. Dans [Mcq 06], DMM est utilisé pour mesurer les métriques logicielles sur les programmes Java. Ce travail, parmi d'autres, illustre la faisabilité de DMM entant qu'un métamodèle Java. De plus, comparé à UML, il est facile à étendre et à lire. Toutefois, DMM ne prend pas en compte le corps de méthode et, comme UNIQ-ART et UML, n'est pas spécifique à Java.

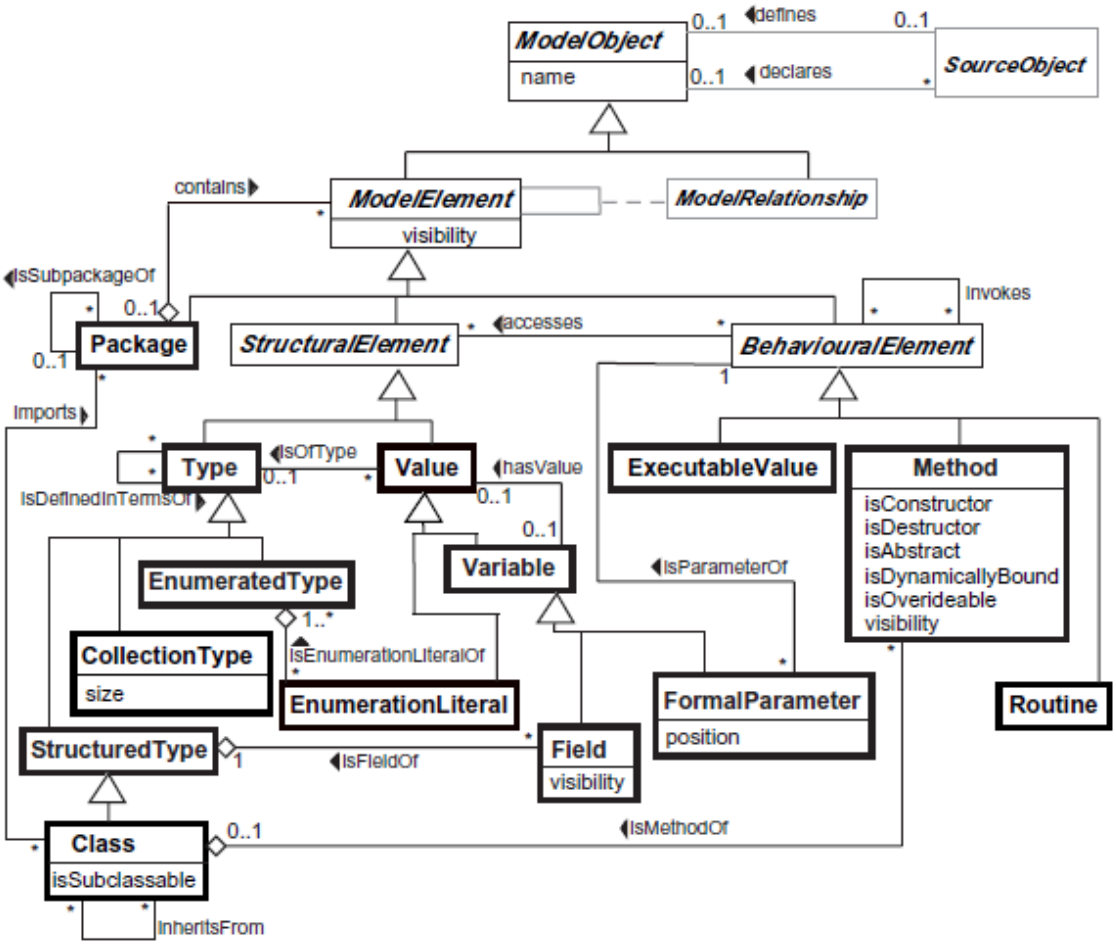


Figure 4.8. La hiérarchie ModelObject de DMM [Let 04].

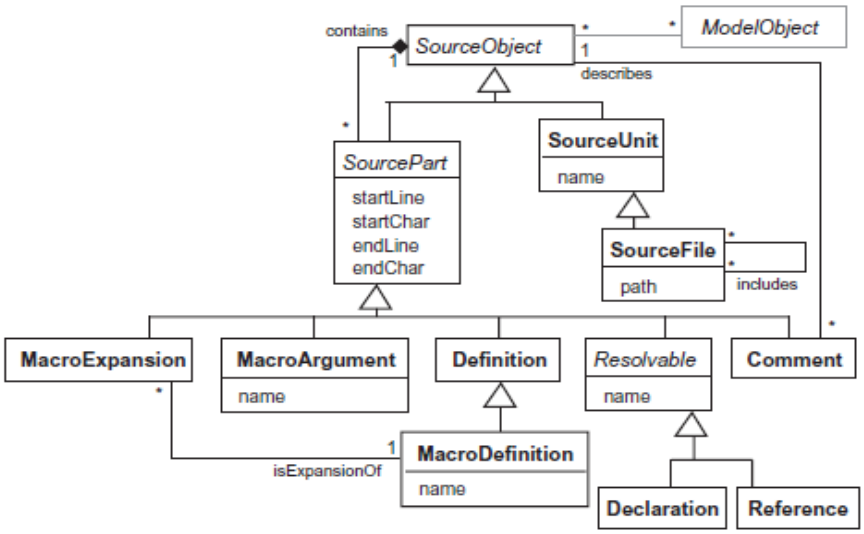


Figure 4.9.La hiérarchie SourceObject de DMM[Let 04].

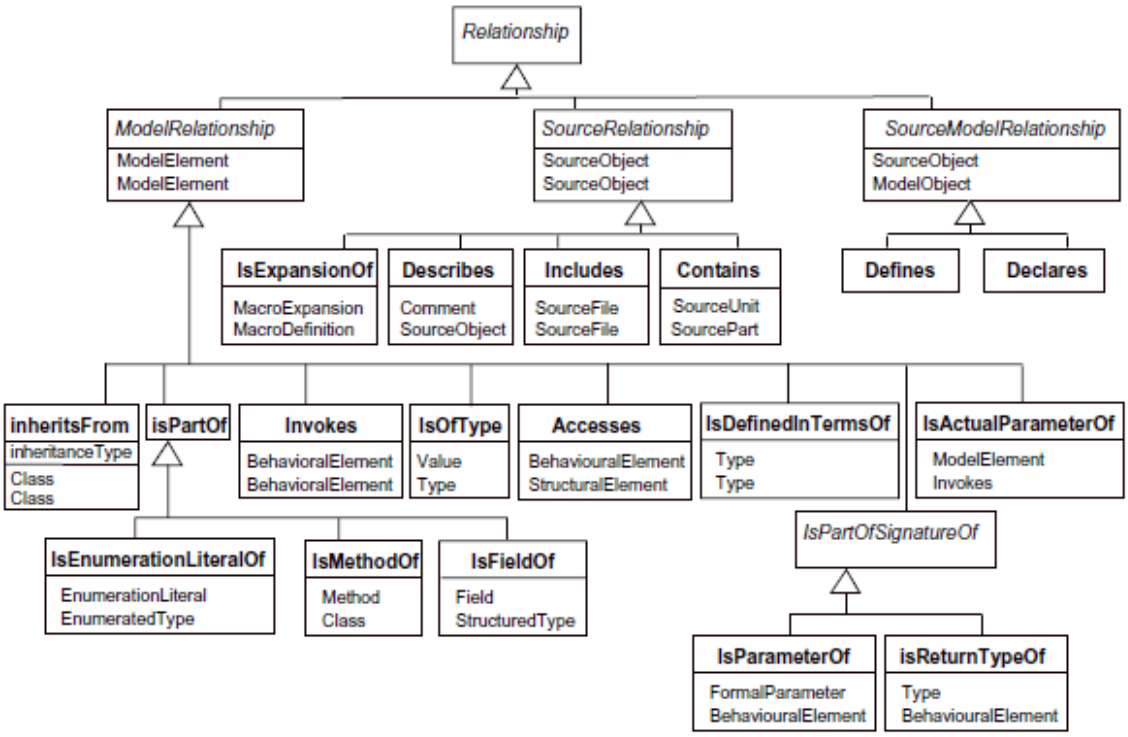


Figure 4.10. La hiérarchie d'association de DMM [Let 04].

Datrix

Datrix [Lap 01] a été introduit pour servir de format d'échange entre les outils de retro ingénierie. Les modèles qui sont conformes à Datrix peuvent représenter des graphes syntaxiques (*ASGs* : *Abstract Syntax Graphs*) et donc toute la sémantique statique des programmes. Cependant, la seule documentation [Led 99] de Datrix n'est plus accessible via le web vu que les références fournies sont devenues inactives, et, à part un diagramme de type Entité/Relation, il n'existe pas une autre représentation explicite de ce métamodèle. De plus, le diagramme E/R est particulièrement construit pour représenter des fragments de source C++ [Hol 00].

FAMIX

FAMIX [Dem 01] est un effort investi dans l'intégration de la rétro ingénierie et l'ingénierie dirigée par les modèles. Contrairement à UML qui met l'accent sur la représentation des systèmes au niveau conceptuel, FAMIX est conçu pour représenter le code source orienté objet. Son noyau est indépendant de toute plateforme et peut être étendu par des

concepts spécifiques aux langages (c.f. Annexe A.3). L'extension peut se faire par l'ajout de nouvelles entités (généralement en spécialisant les métaclasses de FAMIX), de nouveaux attributs et d'annotations au métamodèle. Par exemple, le concept de classe dans FAMIX (version 1.2) doit être adapté pour différencier une classe d'une interface, ce qui est réellement réalisé, dans le cadre d'une extension de FAMIX pour décrire Java, par l'ajout de l'attribut *isInterface():Boolean* dans l'entité *Class* du noyau FAMIX [Tic 01]. Une nouvelle version (3.0) de FAMIX est initiée dans [Duc 11]. Plusieurs éléments sont ajoutés mais le noyau n'a subi aucune modification. Le package nommé FAMIX-Java définit aussi des concepts spécifiques à Java tels que les exceptions, les énumérations et les génériques.

Malheureusement, FAMIX ne couvre pas tous les aspects de Java tels que les blocs et les expressions. La terminologie générique plus la présence d'éléments inutiles pour Java dans le métamodèle réduisent la lisibilité et font appel à des modifications et des interprétations par les utilisateurs.

MOON

MOON [Cre 00] (Minimal Object-Oriented Notation) est conçu pour représenter les constructions communes entre les langages de programmation orientés objet telles que les classes, les types, l'héritage, etc. MOON est décrit par des notations UML, et son modèle principal définit les éléments les plus importants, à savoir *Entity* (entité), *Class_Def* (définition de classe), *ATT_DEC* (déclaration d'attribut), et *METH_DEC* (déclaration de méthode). *Entity* représente tous les éléments ayant un type dans le code source (e.g. les méthodes, les attributs, les variables locales, etc.). *METH_DEC*, *LOCAL_DEC* et *INSTR* représentent les méthodes, les variables locales et les instructions d'une méthode, respectivement. Les instructions consistent en des créations, des affectations, des appels et des blocs d'instructions. Le métamodèle MOON a été initialement utilisé pour la refactorisation au niveau conceptuel, mais il peut être également utilisé dans d'autres activités de rétro ingénierie.

Malgré qu'il couvre les langages orientés objet, MOON doit être adapté pour qu'il soit utilisé comme un métamodèle Java. Des liens de correspondance entre ses aspects de bases et les aspects Java doivent alors être établis par l'utilisateur. Ainsi, MOON ne définit pas plusieurs instructions telles que les exceptions, les boucles et les structures conditionnelles. De

plus, la seule description détaillée de ce métamodèle peut être trouvé uniquement dans une thèse de doctorat écrite en espagnole [Cre 00].

Spoon

Spoon est un métamodèle Java (version 5) qui couvre toutes les constructions du langage [Paw 14]. Il est facile à lire, à comprendre et à manipuler (c.f. Annexe A.4). Ce métamodèle est au centre de la plateforme Spoon d'analyse et de transformation du code Java. Il est accompagné par une API riche pour la modification et la génération du code. Il modélise l'aspect structurel d'un système par le métapackage *structural*, les détails du code par le métapackage *code*, et les références par le métapackage *reference*. Chaque élément de Spoon représente un élément du temps de compilation d'un programme et correspond à un nœud de l'AST de ce dernier. Le métapackage *structural* définit les éléments structurels tels que les classes, les interfaces, les annotations, les méthodes, etc. Le métapackage *code* regroupe les instructions constituant le corps de méthode (i.e. les instructions et les expressions). Le métapackage *reference* définit les types de références (e.g. références de type, références de variables, etc.).

Dans Spoon, le préfixe Ct (*Compile time*) précède les noms des éléments du métamodèle ce qui réduit sa lisibilité. La version actuelle de Spoon supporte déjà Java 5 mais pas encore les closures et les expressions Lambda de Java 8.

JaMoPP

JaMoPP est un métamodèle qui décrits Java 5 [Hei 10]. Sa représentation standard repose sur le langage de modélisation ECORE [Ste 08]. Comparé à Spoon, JaMoPP est plus large vu qu'il contient 288 métaclasses réparties dans 18 métapackages (c.f. Annexe A.5). Contrairement à Spoon qui utilise, par exemple, un seul métapackage pour modéliser les détails du code, JaMoPP emploie plusieurs métapackages tels que *statements* (instructions), *expressions*, *operators* (opérateurs), *arrays* (tableaux) and *variables* (figure 4.11). Ce nombre élevé de métapackages et cette organisation basée packages conduisent à une meilleure compréhension et une meilleure lisibilité. En effet, JaMoPP est créé pour combler le fossé entre les modelware (l'espace technique des modèles) et le grammareware (l'espace technique des grammaires), et pour soutenir la dernière étape du développement dirigé par les modèles,

i.e. la génération du code. Ainsi, le fait que ce métamodèle est accompagné par un analyseur (un parseur) signifie la possibilité de l'utiliser dans le cadre d'activités de retro ingénierie.

La dernière version de JaMoPP implémente Java 5. Son parseur construit des modèles au format Ecore. Le niveau de granularité de ce métamodèle est très fin puisqu'il modélise non seulement les instructions mais aussi les commentaires. De plus, le parseur préserve le format du code (*layout*) en prenant en compte les espaces blancs qui sont usuellement ignorés durant l'analyse du code.

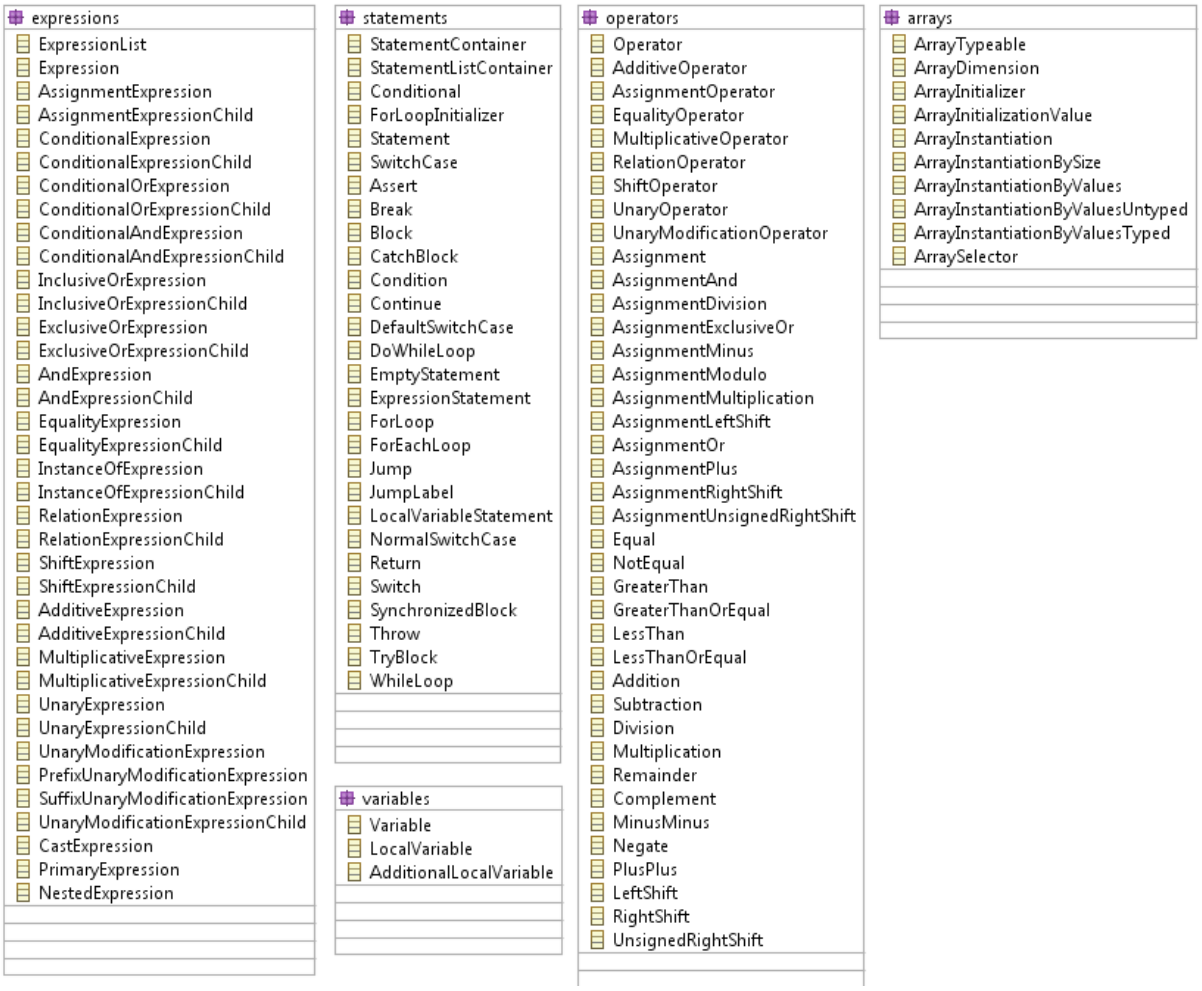


Figure 4.11. Extrait du métamodèle JaMoPP.

Stratego/XT

Ce métamodèle Java [Bra 08] est fourni avec les outils Stratego/XT au format SDF (Syntax Definition Formalism [Vis 97]). Stratego/XT est l'ensemble constitué du langage de

transformation Stratego et les outils XT. Le premier est utilisé pour transformer les ASTs alors que XT est utilisé pour l'analyse et la génération du code sur la base de la définition SDF (i.e. le métamodèle Java dans le cas du langage Java).

Similairement à Spoon, le métamodèle Stratego/XT décrit Java5 et doit être étendu pour définir les nouvelles structures de Java8. Pour ce faire, il est indispensable de lire et étendre la définition SDF ce qui est plus difficile que la lecture et l'extension des métamodèles hiérarchiques qui sont représentés graphiquement ou sous format arborescent (e.g. UML ou Ecore). Malheureusement, la représentation SDF obstrue l'intégration du métamodèle au sein des outils de modélisation existants.

MoDisco

MoDisco [Bru 14] a pour objectif de soutenir la modernisation des logiciels et les activités de retro ingénierie. Contrairement à FAMIX, l'idée principale ici est l'utilisation d'un métamodèle spécifique à une technologie particulière (e.g. Java) pour encapsuler formellement les informations extraites des systèmes patrimoniaux par un mécanisme de découverte. Les modèles qui sont conformes à MoDisco sont construits par les découvreurs (*discoverers*). Le projet MoDisco fournit un découvreur Java avec un métamodèle qui couvre l'AST du JDT (*Java Development Tool*) [Ecl 15]. Les modèles *découverts* par le découvreur Java sont des abstractions de bas niveau (i.e. descriptions détaillées d'instructions et expressions constituant le corps de méthode) qui assurent une extraction sans aucune perte d'informations.

Métamodèle du graphe Java

Dans [Hof 08], un métamodèle est proposé en tant qu'une spécification d'un graphe représentant le code Java. Il est introduit dans le contexte d'une implémentation de la refactorisation automatique des programmes, basant sur les transformations de graphes. Il exprime les constructions Java de base telles que les packages, les classes, les méthodes, les variables, etc. et modélise quelques expressions définissant le corps de méthode telles que les instanciations, l'instruction *return*, les instructions d'accès et les blocs (figure 4.12). Le métamodèle est décrit en utilisant la notation UML et a été utilisé avec succès comme un schéma d'un système de refactorisation implémentant les opérations *Encapsulate Field* (encapsuler le champ), *Move Method* (déplacer la méthode) et *Pull-up Method* (monter la

méthode). Le système lit, affiche et transforme les graphes représentant les programmes, qui sont sérialisés dans des fichiers GXL et sont conformes au métamodèle.

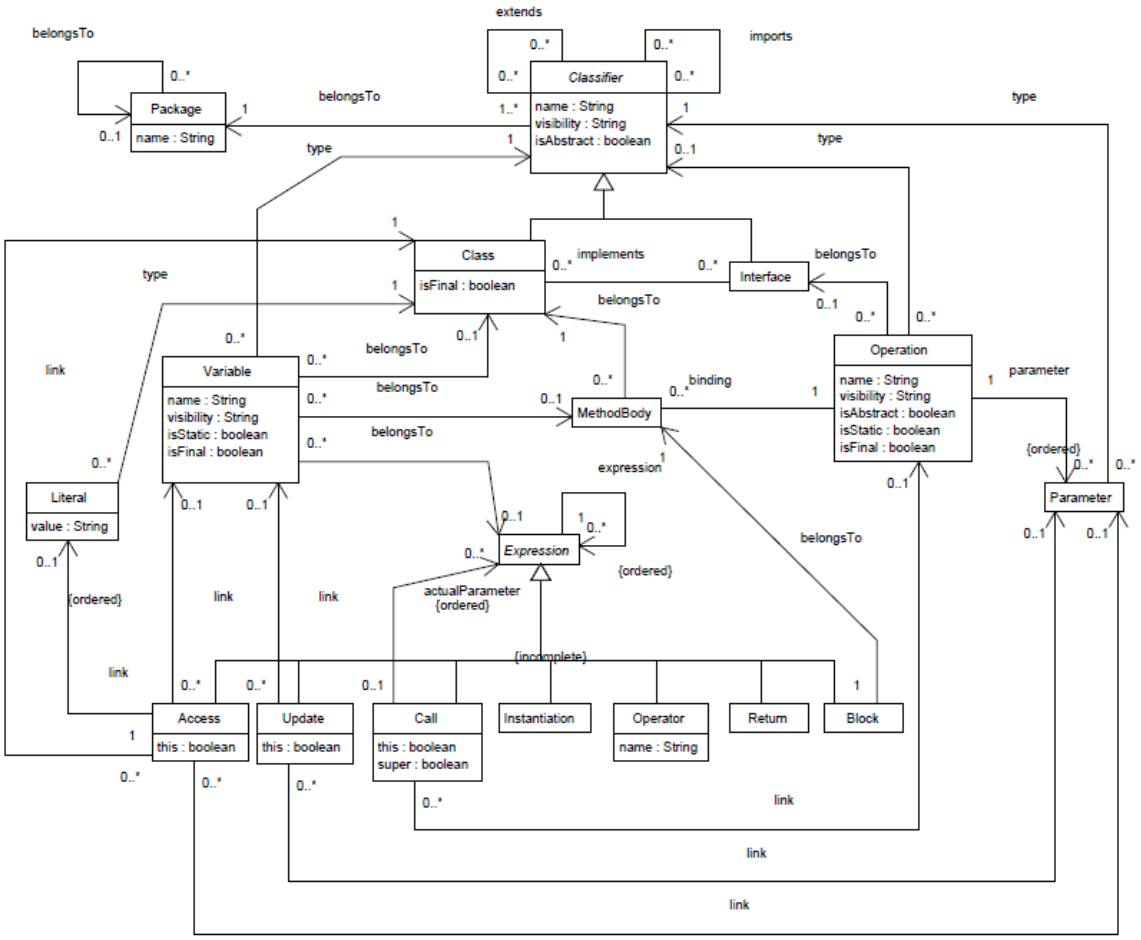


Figure 4.12. Métamodèle de la représentation graphe de Java [Hof 08].

Pour simplifier les graphes de programmes, des restrictions sur le code Java sont considérées. Le métamodèle ne modélise pas les concepts suivants: les classes internes et anonymes, la gestion des exceptions, les types paramétrés (generics), les annotations, la conversion de type et les nouveaux concepts de Java8. Ainsi, pour la simplicité, plusieurs contraintes garantissant la validité structurelle de Java ne sont pas explicitement modélisées. Par exemple, une classe n'hérite pas d'une interface. Cette contrainte, parmi d'autres, n'est pas incluse dans le métamodèle et doit être prise en compte.

ASTM

ASTM (*Abstract Syntax Tree Metamodel*) est un standard proposé par OMG pour définir une spécification de modélisation des éléments afin d'exprimer les ASTs dans une représentation qui est partageable entre plusieurs outils provenant de différents vendeurs [Omg 11a]. Il supporte le passage du niveau code aux modèles logiciels de bas niveau. ASTM est composé d'une spécification principale appelée GASTM (*Generic Abstract Syntax Tree Metamodel*), et un ensemble de spécifications complémentaires qui l'étend, SASTMs (*Specialized Abstract Syntax Tree Metamodels*). GASTM fournit un ensemble générique d'éléments de modélisation communs à plusieurs langages. Il peut être étendu par un SASTM pour ajouter des éléments spécifiques. ASTM complémente d'autres spécifications OMG qui fournissent des vues conceptuelles de haut niveau sur le logiciel, telles que UML, par la définition d'une relation complémentaire entre ces modèles de haut niveau et les modèles AST de bas niveau. GASTM est suffisamment riche d'expressions pour représenter les constructions de base de Java, et est décrit informellement et par des diagrammes UML. Ainsi, l'architecture MoDisco inclut une implémentation de référence basée EMF de GASMT [Mod 15].

ASTM n'est pas un métamodèle standard de Java, mais offre un moyen standard pour décrire les ASTs de Java. Malheureusement, une extension SASTM pour Java n'a pas encore été créée. De plus, il n'existe pas des outils pour la construction directe de modèles Java conformément à GASTM à partir des programmes Java.

KDM

KDM (*Knowledge Discovery Metamodel*) est un standard proposé par OMG pour représenter les logiciels existants [Omg 11b]. Comme UML, KDM assure l'interopérabilité et l'échange de donnés entre les divers outils construis par les différents vendeurs. KDM et ASTM sont complémentaires parce que KDM fournit une spécification pour les modèles de graphes sémantiques abstraits (ASGs), alors que ASTM établit une spécification pour les modèles AST. En d'autres termes, les modèles KDM sont à un niveau d'abstraction plus élevé que celui des modèles ASTM. Vu comme un métamodèle Java, KDM permet la représentation des éléments Java du niveau implémentation ainsi que leurs associations via l'ensemble d'éléments du métamodèle groupés dans les métapackages *Code* et *Action*. Les deux

composent un seul modèle KDM appelé *CodeModel*. Le métapackage *Code* reflète les éléments communs des langages de programmation tels que les classes, les procédures, les méthodes, les espaces de noms, etc. le métapackage *Action* définit les éléments de comportement et d'association, et permet la modélisation du corps de la méthode. La spécification de KDM est détaillée et clarifiée en utilisant les diagrammes de classes UML, et une implémentation basée Ecore est disponible dans le projet MoDisco [Mod 15].

La vaste portée de ce métamodèle qui couvre une grande variété d'applications, plateformes et langages de programmation, explique le nombre important de ses éléments de modélisation. Comme nous l'avons mentionné déjà, deux métapackages, parmi onze métapackages constituant le métamodèle, contiennent les métaclasses fondamentales qui peuvent être utilisées pour décrire les constructions de Java. MoDisco fournit un découvreur KDM permettant la création de modèles Java mais seulement les éléments de superstructure sont capturés, le corps de la méthode n'est pas pris en considération. Pour construire des modèles Java complets et conformant à KDM, des outils doivent être construis.

Métamodèles implicites

Dans cette catégorie, nous introduisons des métamodèles Java implicites, i.e. des réflexions de Java qui ne sont pas explicitement représentées et décrites par des formalismes de modélisation. Parmi les métamodèles Java implicites qui sont très utilisés mais inaperçus, les métamodèles des compilateurs Java. Un compilateur Java analyse le code Java puis construit un AST. Similairement aux modèles abstraits, les ASTs créés par les compilateurs se conforment à un métamodèle. Les compilateurs importants comme javac[1], EJC[2] (*Eclipse Compiler for Java*), GCJ[3] (*GNU Compiler for Java*), et Jikes[4], ont un métamodèle Java implicite, généralement décrit par un ensemble de classes Java. Cette représentation,

[1] http://eclipse.org/jdt/,

[2] http://java.sun.com,

[3] http://gcc.gnu.org/java/

[4] http://jikes.sourceforge.net/

cependant, ne permet pas l'intégration sémantique de Java dans les approches dirigées par les modèles.

Sun Microsystems fournit le JLS (*Java Language Specification*), une spécification du langage Java [Gos 14]. C'est une documentation qui illustre la signification de chaque concept de Java à l'aide des diagrammes syntaxiques. En d'autres termes, le JLS est un métamodèle Java implicite. La réflexion de Java, et comme son nom l'indique, est un autre métamodèle Java, fourni sous forme d'une API. Elle est essentiellement utilisée pour l'introspection des classes, pour collecter dynamiquement les métadonnées et pour contrôler le comportement du code.

Métamodèle	**Niveau de granularité**	**Représentation**	**Nombre de Métaclasses**	**Dépendance à Java**
UML 1.4	Grossière	UML/Ecore	122	Non
UML 2.0	Grossière	UML/Ecore	242	Non
EJB/JAVA UML PROFILE	Grossière	UML	242 (et 18 stéréotypes)	Oui
GRUMMYUML	Fine	UML	122 (et 3 stéréotypes)	Non
UNIQ-ART	Grossière	Enumération textuelle	14	Non
DDM	Grossière	UML	63	Non
FAMIX 2.1	Grossière	Notation UML	22	Non
FAMIX 2.1 POUR JAVA	Grossière	Notation UML	23	Oui
FAMIX 3.0	Grossière	Notation UML	50	Non
FAMIX 3.0 POUR JAVA	Grossière	Notation UML	60	Oui
MOON	Fine	UML	56	Non
SPOON	Fine	Ecore	140	Oui
JAMOPP	Fine	Ecore	233	Oui

STRATEGO/XT	Fine	SDF	161	Oui
MODISCO	Fine	Ecore/ KM3 [Jou 06]	126	Oui
JAVA GRAPH	Fine	UML	17	Oui
GASTM	Fine	Ecore/diagrammes de classes UML	193	Non
KDM	Fine	Ecore/diagrammes de classes UML	300	Non

Table 4.1. Comparaison des métamodèles Java.

4.4.2 Comparaison des métamodèles Java

La table 4.1 introduit une comparaison des métamodèles Java en fonction de quatre critères : le niveau de granularité, la représentation, le nombre de métaclasses et la dépendance à Java. Une métaclasse est une classe spéciale utilisée pour représenter une construction Java au niveau méta. Le nombre de métaclasses constituant le métamodèle et son niveau de granularité, peuvent révéler des informations concernant sa complexité et sa complétude. Le concept de métaclasse est différemment représenté dans les formalismes de modélisation. Par exemple, Ecore décrit une métaclasse par EClass, un diagramme E/R utilise Entity, alors que dans les métamodèles fournis au format SDF, la partie gauche d'une règle syntaxique (Sort) est l'équivalant de métaclasse. Nous avons choisi le métamodèle JaMoPP qui nous semble le plus complet. Sa représentation est standardisée (défini en Ecore), son niveau de granularité est fin, et il est spécifique à Java. De plus, JaMoPP modélise la sémantique statique du code (i.e. supporte la résolution des références). Par rapport à Spoon et MoDisco, JaMoPP est plus riche en termes d'éléments conceptuels, mieux structuré et plus facile à étendre.

4.5 Modèles Java

L'outil JaMoPP (Java Model Parser and Printer), qui entraîne un métamodèle complet pour Java (c.f. Annexe A.5), permet de convertir un code source Java à un modèle EMF [Hei 10]. Par conséquent, JaMoPP permet aux outils basés sur EMF de travailler sur les programmes Java. La figure 4.13 montre le modèle EMF (la partie droite) qui représente une

classe Java (la partie gauche) conformément au métamodèle JaMoPP. C'est la vue Outline de l'éditeur Java de EMFText de la classe *Movie* sous forme d'une structure arborescente. Chaque élément dans l'arbre représente une construction dans la représentation textuelle. Seuls les éléments importants sont présentés, i.e. les instances des métaclasses concrètes du métamodèle, les références, et les attributs ayant la multiplicité 1.

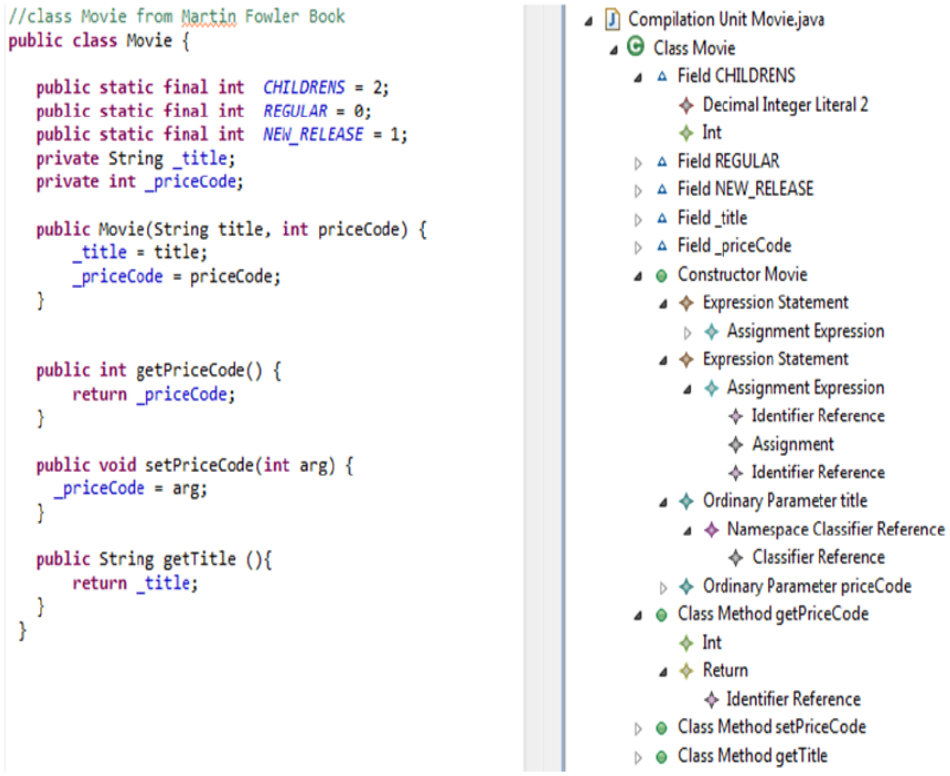


Figure 4.13. Classe Movie sous forme de modèle Java.

La relation entre les modèles et les métamodèles est illustrée par le standard MDA de l'OMG (figure 4.14) où les métamodèles sont définis par le métamétamodèle MOF. Au niveau méta, JaMoPP est utilisé pour définir les concepts de Java. Tous les modèles conformant à JaMoPP sont des modèles EMF. Ils fournissent une représentation abstraite des programmes Java qui se trouvent au niveau instance. JaMoPP permet de traiter un code Java comme un modèle et donc rend possible son intégration dans les outils de modélisation. Si nous arrivons à étendre JaMoPP pour prendre en compte des éléments architecturaux d'ArchJava, ce dernier va bénéficier d'une nouvelle infrastructure logicielle très riche qui le manipule.

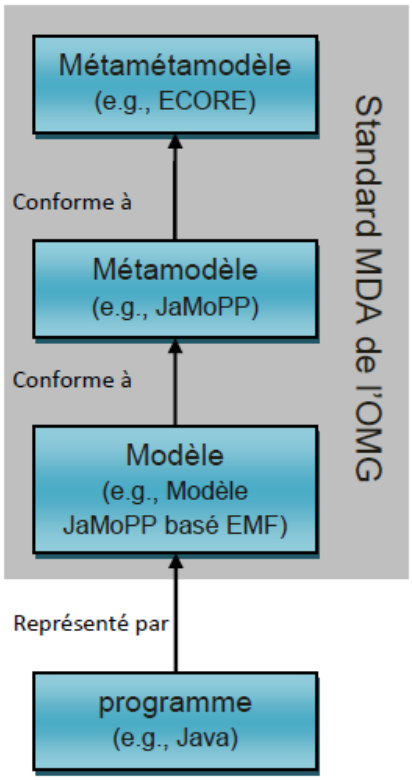


Figure 4.14. Architecture à quatre niveaux.

4.6 Métamodèle ArchJava

Dans cette section nous présentons en détail les nouveaux éléments d'ArchJava ainsi que la manière dont ils sont ajoutés au métamodèle JaMoPP. Chaque concept d'ArchJava est représenté par une nouvelle métaclasse. Les relations entre les métaclasses JaMoPP et celles ajoutées, sont des relations d'héritage, de composition et de référence simple. Le métamodèle complet qui définit les nouveaux éléments et leurs relations avec les métaclasses de JaMoPP, est disponible en annexe A. Il est important de noter que l'extension est fondée essentiellement sur les informations incluses dans le manuel de référence d'ArchJava[1].

[1] http://archjava.fluid.cs.cmu.edu/papers/archjava-language.pdf

4.6.1 Composants

Un composant ArchJava, comme tout composant architectural, interagit avec les autres composants à travers des ports. C'est une instance de la classe *Component* (classe composante) qui peut contenir des ports et des connexions. La figure 4.15 illustre la métaclasse qui modélise ce concept. *ComponentClass* est une sous-classe de la métaclasse *Class* du métapackage *classifiers* de JaMoPP. Cette dernière représente la notion de classe Java. Chaque classe, peut comporter des méthodes et des champs (attributs), appelés *Members* (membres de classe). C'est pourquoi une classe est un conteneur de membres (*MemberContainer*).

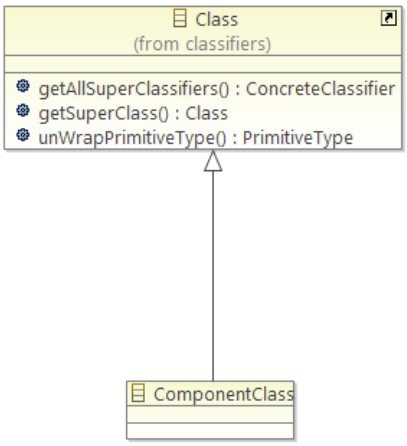


Figure 4.15. Métaclasse ComponentClass.

4.6.2 Ports

Un port est un canal logique de communication entre une instance d'un composant et son environnement. La métaclasse *Port* (figure 4.16) est introduite pour représenter les ports d'ArchJava. Il est à noter que l'ajout de la métaclasse Port est plus complexe que celui de ComponentClass. Un Port est à la fois un membre (Member) d'un ComponentClass et un conteneur de membres (i.e. il définit des méthodes et/ou des signatures de méthodes). Un port déclare trois sortes de méthodes : méthodes fournies, méthode requises et méthodes de diffusion; en utilisant les trois modificateurs : *provides, requires et broadcasts*, respectivement. Ainsi, les modificateurs d'accès *public*, *private* et *protected*, s'appliquent aux ports. De ce fait, *Port* est définit comme une sous-classe de *Modifiable*, qui représente les éléments Java

ayant un modificateur d'accès (e.g. classes, méthodes, attributs). En plus des modificateurs d'accès classiques, les métaclasses *Provides*, *Requires* et *Brodcasts* dérivant de la métaclasse *Modifier*, sont ajoutées pour décrire les nouveaux modificateurs.

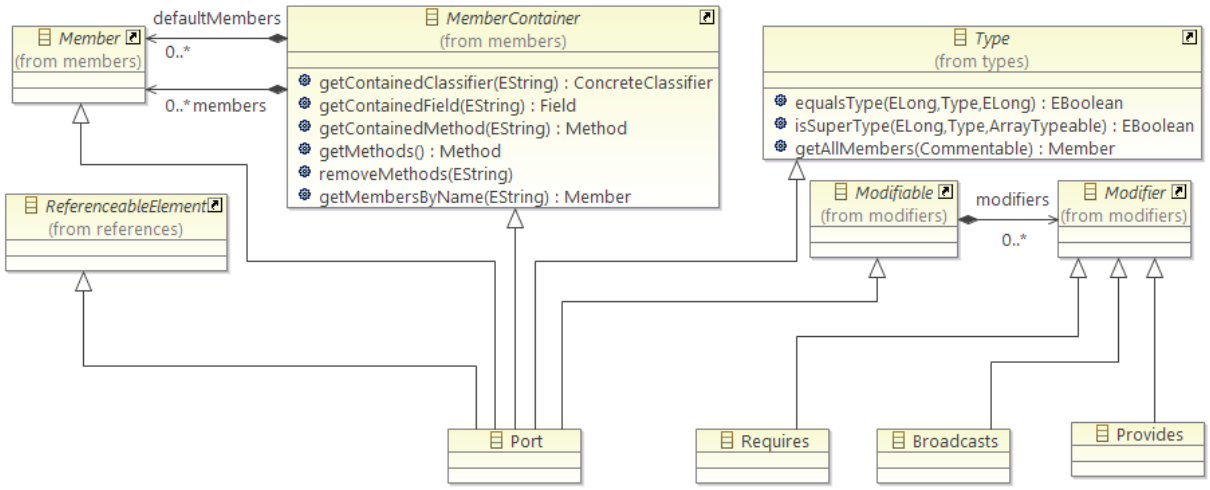


Figure 4.16. Métaclasse Port.

4.6.3 Méthodes de Port

La déclaration d'un port passe par la définition des méthodes ou/et des signatures des méthodes. Les méthodes de ports sont des méthodes Java ordinaires. En ce qui concerne les signatures de méthodes, une nouvelle métaclasse appelée PortMethod est crée pour les représenter. Comme le montre la figure 4.17, la métaclasse PortMethod est dérivée de ClassMethod de JaMoPP, qui modélise les méthodes d'une classe.

4.6.4 Connexions

La figure 4.18 montre les métaclasses qui décrivent l'aspect de connexion. ArchJava définit des connecteurs primitifs (représentés par Connection), qui sont implémentés dans les composants (ils sont alors des membres de ComponentClass) et permettent de relier les composants (ou sous composants) directement via leurs ports (ils référencent les ports par PortExpressionReference, qui est une TypeReférence. Cette dernière modélise les références de type). La connexion consiste à joindre une méthode requise à une méthode fournie ayant le même nom et la même signature. ArchJava permet aussi d'établir des liaisons entre des

composants créés dynamiquement en exécution par le biais de la construction connect expression. Il s'agit d'une expression comprenant une liste de référence de ports. Le dernier type de connexion est la glue (représentée par Glue). Elle permet de créer des connections de délégation aux sous-composants ou à d'autres ports. Comme l'instruction connect, glue fait référence aux ports (par PortExpressionReference) afin de les attacher les uns aux autres.

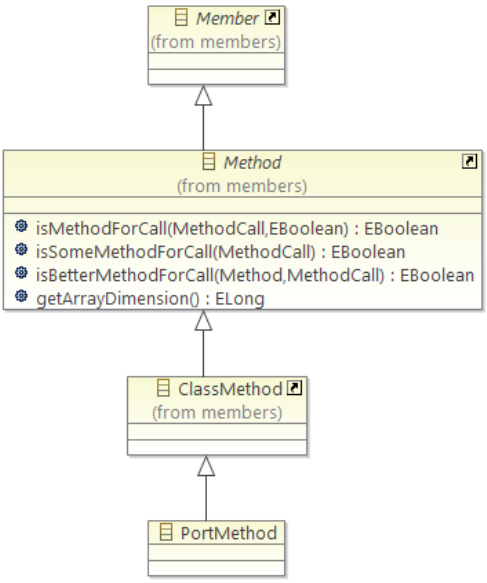


Figure 4.17. Métaclasse PortMethod.

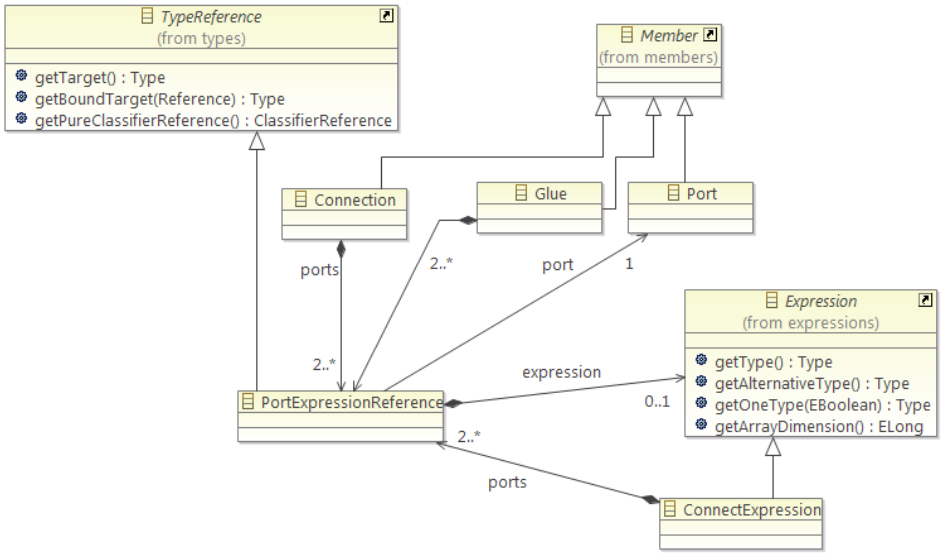


Figure 4.18. Métaclasses de connexion.

4.7 Conclusion

Nous avons présenté dans ce chapitre notre approche qui suggère l'application des techniques de modélisation et de construction des DSLs pour développer des infrastructures logicielles assistant les ADLs. L'idée préconisée est qu'un ADL, qui est en fait un DSL, doit profiter des environnements de développement des DSLs qui génèrent automatiquement des outils logiciels depuis leurs artéfacts (i.e. la syntaxe abstraite et concrète). Le concept de métamodèle a été longuement considéré comme secondaire dans les divers travaux sur les architectures logicielles. Ici, le métamodèle joue un rôle très important car il limite les ambiguïtés et encourage la construction des outils en offrant une description claire et explicite des concepts architecturaux. Dans ce chapitre, nous avons aussi montré comment le métamodèle ArchJava est construit en étendant le métamodèle JaMoPP que nous avons choisi après une étude comparative détaillée des différents métamodèles Java existants dans la littérature.

Dans le chapitre qui suit, nous introduisons les outils et les standards utilisés pour mettre en œuvre notre approche, et nous définissons la syntaxe concrète d'ArchJava. En suite, nous personnalisons les outils de support générés.

5 Mise en œuvre

Ce chapitre est consacré à la concrétisation de notre approche. Dans un premier temps, nous citons les outils et les langages nécessaires à la mise en œuvre d'*ArchJaMoPP* (*ArchJava Model Parser and Printer*). C'est une bibliothèque qui implante un métamodèle d'ArchJava, sa syntaxe concrète, des outils d'accompagnement et des fonctionnalités d'analyse et de manipulation de modèles d'architecture ArchJava. Dans un second temps, nous détaillons la syntaxe concrète d'ArchJava. En fin, nous décrivons les divers outils inclus dans ArchJaMoPP ainsi que les modifications apportées à leurs versions initiales (i.e. versions générées automatiquement).

5.1 Outils et technologies d'implémentation

La première étape de notre démarche vise à produire un métamodèle d'ADL. Dans le chapitre précédent, nous avons créé le métamodèle ArchJava qui est décrit en Ecore [Ste 08]. Ecore est l'implémentation de MOF (EMOF) de la plateforme de développement Eclipse. En d'autre terme, il représente le métamétamodèle de son processus de modélisation. Nous avons utilisé EMF (*Eclipse Modeling Framework*) pour définir le métamodèle ArchJava. EMF est l'environnement de la plateforme Eclipse dédié au MDA [Ste 08], sur laquelle s'appuient tous les plugins de transformation et de modélisation. Il fournit les outils pour créer des modèles Ecore et générer les éditeurs correspondants. EMF permet aussi le développement rapide et l'intégration de nouveaux plugins Eclipse.

EMFText [Hei 09] est au cœur de notre approche. Il offre la possibilité de définir la syntaxe textuelle pour les métamodèles basés Ecore. En fournissant la définition de la syntaxe textuelle d'un langage (qu'il soit dédié ou générique) et le métamodèle représentant ce dernier, EMFText génère l'outillage d'analyse (parseur) et de génération de texte (générateur). JaMoPP [Hei 10], qui est développé en utilisant l'environnement EMFText, fournit un métamodèle exhaustif et l'outillage pour analyser et générer le code Java, ce qui permet aux outils de modélisation (e.g. outils de transformation de modèles) de lire et modifier des programmes écrits en Java tout en conservant les liens entre le code source et son modèle. Il a été testé sur une grande masse de programmes écrits en Java 5 [Hei 10]. JaMoPP peut être

étendu (e.g. pour supporter les nouvelles versions de Java ou pour ajouter de nouvelles constructions) en modifiant le métamodèle et la définition de la syntaxe sans modifier le code. La figure 5.1 illustre l'architecture de plugins Eclipse d'ArchJaMoPP.

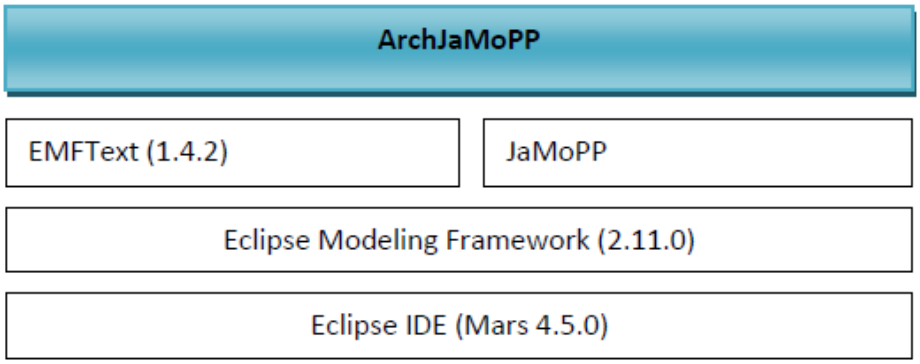


Figure 5.1. Architecture du Plugin ArchJaMoPP.

5.2 Syntaxe Concrète d'ArchJava

EMFText offre un mécanisme d'extension qui permet d'importer des métamodèles et des spécifications de syntaxe. Dans [Ham 15], nous avons exploité cette technique pour étendre la syntaxe de Java par la syntaxe d'un DSL appelé TOM. Sur la base de cette expérience, nous avons pu réaliser l'extension présentée dans cette thèse.

```
1  SYNTAXDEF archj
2  FOR <http://www.emftext.org/archjava>
3  START java.Containers.CompilationUnit
4  IMPORTS {
5    java : <http://www.emftext.org/java> WITH SYNTAX java
6    <../../org.emftext.language.java/metamodel/java.cs>
7  }
8  OPTIONS {
9      defaultTokenName = "IDENTIFIER";
10     usePredefinedTokens = "false";
11 }
12 RULES {
13 ComponentClass ::=
14   annotationAndModifiers* "component" "class" name[] ("<"
15   typeParameters("," typeParameters)* ">")?("extends" extends)?
16   ("implemets" (implements ("," implements)*))? 19 #1 "{" (!1
17   members)* !0 "}";
18 Port ::=
19    modifiers* "port" ("interface")? name[] #1 ("{" (!1 members)* 22    !0
20   "}" | ";");
21 PortMethod ::=
22   annotationAndModifiers* ("<" typeParameters ("," 
```

```
23   typeParameters)* ">")? (typeReference arrayDimensionsBefore*)
24   name[] "("(parameters ("," parameters)*)? ")"
25   arrayDimensionsAfter*  ("throws" exceptions ("," exceptions)*)?
26   ";";
27 PortExpressionReference ::=
28   (expression ".")? port[]";";
29 Connection ::=
30   "connect" ("pattern")? ports ("," ports)+  ";";
31 ConnectExpression ::=
32   "connect" "(" ports ("," ports)+")" ";";
33 Glue ::=
34   "glue" ports ("," ports)* ("to" ports ("," ports)*)? ";";
35 Provides ::=
36   "provides";
37 Requires ::=
38   "Requires";
39 Broadcasts ::=
40   "broadcasts";
41 }
```

Listing 5.1. La syntaxe concrète d'ArchJava en CS.

Le listing 5.1 donne la syntaxe complète d'ArchJava. La première ligne définit l'extension du langage (.archj), alors que la deuxième relie la syntaxe au métamodèle par son URI (*Uniform Resource Identifier*) d'espace de noms. En effet, ce qui est référencié n'est pas le métamodèle en format Ecore mais son modèle générateur de EMF (*.genmodel*). Ce dernier contient toutes les informations nécessaires pour que EMFText puisse générer l'outillage correspondant, telles que les noms complètement qualifiés des classes générées par EMF. La troisième ligne indique l'élément racine qui représente le symbole de départ. L'élément racine d'ArchJava.cs est la métaclasse *CompilationUnit* du métamodèle JaMoPP, qui modélise les unités de compilation (i.e. les fichiers sources Java). Les lignes de 4 à 7 présentent la section d'importation dans laquelle on importe le métamodèle et la spécification syntaxique de JaMoPP. L'importation permet de réutiliser les règles syntaxiques définies dans la spécification référenciée. Les lignes de 8 à 11 présentent la section d'options de génération de code. EMFText supporte 257 options parmi lesquelles nous avons retenu deux :

- *defaultTokenName* : cette option est utilisée pour spécifier le nom de token (lexème) donné par défaut (*IDENTIFIER* qui est un token de Java.cs de JaMoPP);
- *usePredefinedTokens* : il est mis à faux. Dans ce cas, EMFText n'offre pas automatiquement les tokens par défaut (*TEXT*, *WHITESPACE*, *LINEBREAK*).

Les règles syntaxiques sont introduites dans la section des règles (les lignes de 12 à 41). Chaque règle est reliée à une métaclasse concrète du métamodèle ArchJava, et définit sa représentation syntaxique, ses attributs et ses références. Nous avons défini donc dix règles réécrivant les règles des métaclasses mères de JaMoPP :

- *ComponentClass* : la seule différence syntaxique entre la déclaration d'une classe Java et un composant ArchJava est l'ajout du mot clé *Component* avant le mot clé *Class*;
- *Port* : cet élément n'a pas de spécification syntaxique héritée du fait que les métaclasses mères *Membre*, *MemberContainer*, *Modifiable*, *Type* et *ReferenceableElement* sont abstraites. La syntaxe associée aux ports définit pour chacun, un nom, des modificateurs et zéro ou plusieurs membres. La spécification utilise le mot clé optionnel *interface* pour déclarer les interfaces de port, qui peuvent être instanciées plusieurs fois afin de communiquer via des connexions distinctes durant l'exécution.
- *PortMethod* : la syntaxe d'une signature de méthode est semblable à celle d'une méthode Java sauf le fait que la première ne spécifie pas un corps et se termine par un point-virgule.
- *ConnectExpression* : une *expression connect* (*Connect Exprssion*) renvoie un objet de connexion (*Connection Object*) qui représente la connexion. Comme les objets Java, un objet de connexion peut, par exemple, être affecté à une variable du même type. La syntaxe d'une expression connect commence par le mot clé *connect* puis une liste d'arguments, entre parenthèses, de type *Port*.
- *Connection* : une connexion ArchJava est déclarée en utilisant le mot clé *connect* et une liste de ports séparés par des virgules. Le mot clé optionnel *pattern* désigne un modèle de connexion (*Connection Pattern*) qui décrit un ensemble de connexions pouvant être instanciées en exécution par des expressions connect. Chaque expression connect doit correspondre à un modèle de connexion déclaré dans le composant conteneur. La correspondance est établie si les ports connectés sont identiques et chaque instance de composant connectée est une instance du type spécifié dans le modèle.
- *PortExpressionReference* : la référence de port représente un port qui se déclare dans le même composant ou ailleurs. Dans le premier cas, la référence est le nom du port; dans le deuxième cas la référence est plus complexe dont la syntaxe est similaire à celle des

instructions d'accès aux variables d'instance en Java. C'est pourquoi cet élément définit une référence de composition de type *Expression* de JaMoPP, appelé *expression*. Elle permet de spécifier à la fois les références complexes (i.e. qui comportent l'opérateur d'accès « . ») et les références simples. La spécification d'une référence de port contient alors zéro ou plusieurs expressions et se termine par un port (représenté par la référence *port* de type Port).

- *Glue* : Une connexion *Glue* définit une liste de ports ou deux listes de ports séparées par le mot clé *to*. Les ports d'une liste doivent être séparés par des virgules.
- *Provides, Requires et BroadCasts* : Ces modificateurs sont spécifiés par les mots clés *provides, requires* et *brodcasts*, respectivement.

5.3 Outillage de Support

Maintenant que le métamodèle et la syntaxe concrète sont disponibles, nous pouvons créer l'outillage de support moyennant EMF et EMFText. Tout d'abord, le modèle générateur *ArchJava.genmodel* est dérivé du métamodèle ArchJava.ecore (figure 5.2). C'est vrai que le modèle Ecore contient déjà une grande partie de l'information nécessaire à la génération des classes Java d'implémentation (i.e. leur nom, leurs attributs et leurs références), mais il existe bien d'autres informations qui doivent être fournies au générateur EMF (figure 5.2, la vue *Properties*), telles que l'emplacement où ces classes seront générées et le mot préfixant leur nom. EMF regroupe ces informations dans le modèle générateur. L'avantage de cette séparation réside dans la pureté du métamodèle de toute information qui est pertinente seulement pour la génération de code.

A partir du modèle ArchJava.genmodel, EMF génère pour chaque EClass (i.e. métaclasse) une interface java et son implémentation dans deux package différents (*org.emftext.language.archjava* et *org.emftext.language.archjava.impl*, respectivement), comme le montre la figure 5.3. Par exemple, la métaclasse ComponentClass implique la génération de l'interface *ComponentClass.java* et la classe *ComponentClassImpl*.java qui l'implémente. La séparation entre les interfaces et les implémentations s'inscrit dans le cadre

du modèle de conception adopté par Eclipse qui la considère comme la meilleure pratique en ce qui concerne la création d'APIs.

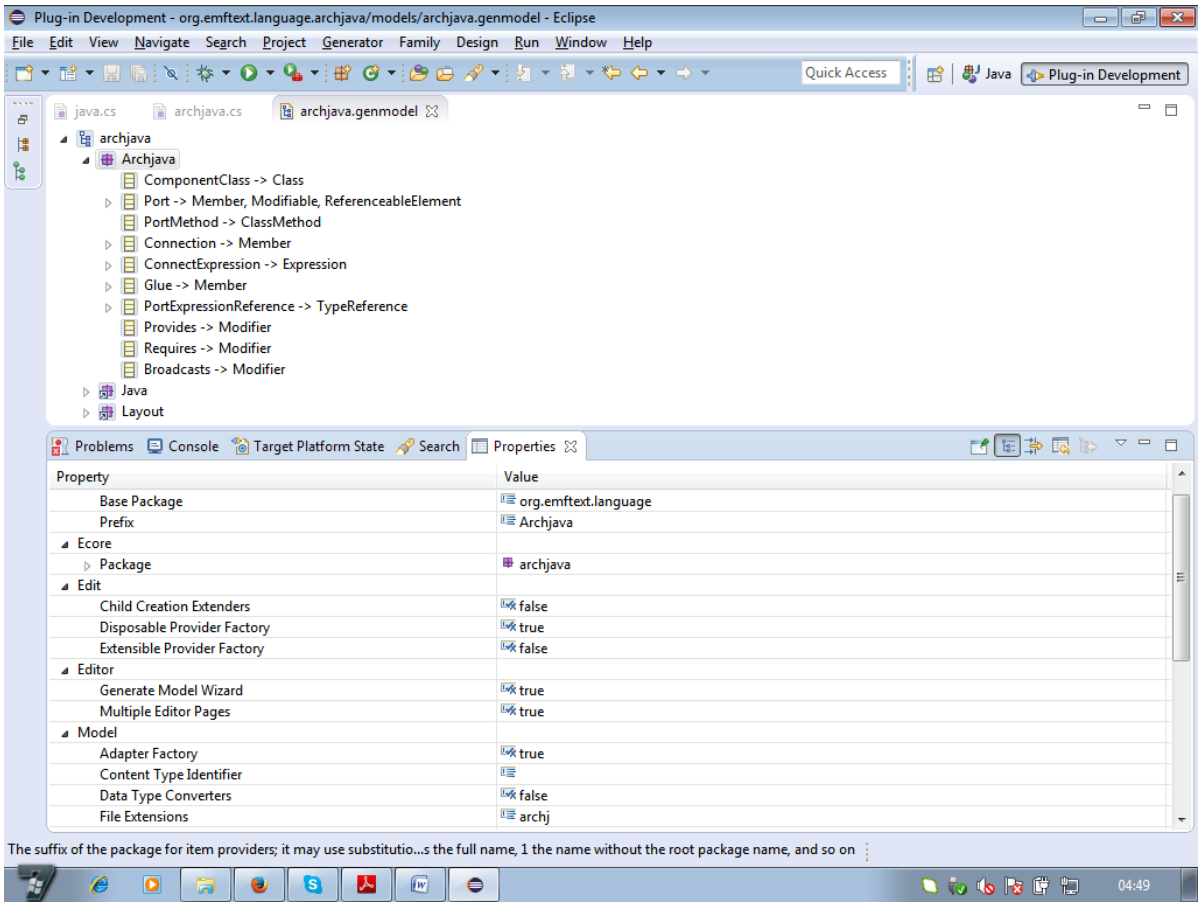


Figure 5.2. Le modèle générateur ArchJava.genmodel.

L'API générée sert d'entrée pour EMFText qui construit, à partir de la spécification syntaxique complète *archjava.cs*, un éditeur de texte avancé et une infrastructure personnalisable du langage ArchJava. Cette infrastructure forme, après une étape de modification et de personnalisation, l'outil ArchJaMoPP. La modification consiste à ré-implémenter des interfaces, injecter des blocs de code dans l'analyseur pour éliminer les ambiguïtés et redéfinir les résolveurs de références. Ces adaptations sont essentielles, bien que difficiles à réaliser sans une compréhension approfondie de ce que EMFText génère exactement.

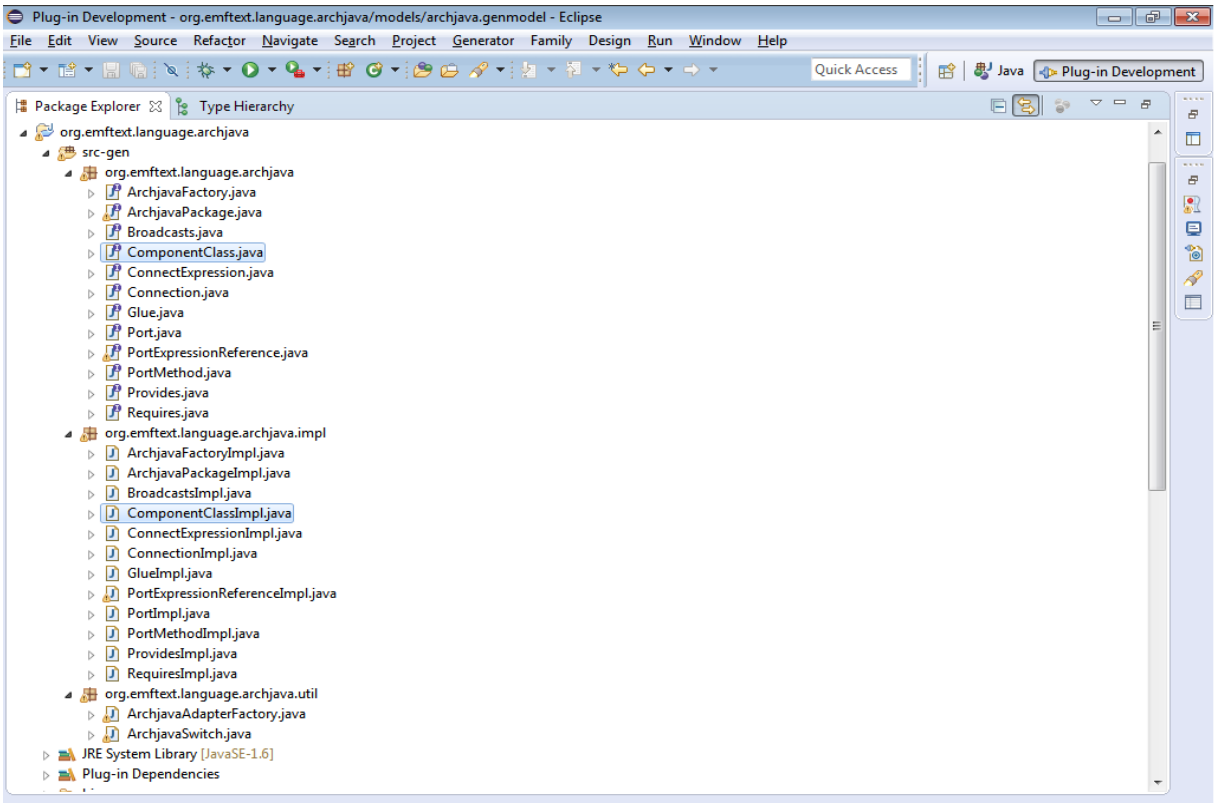


Figure 5.3. API java du métamodèle ArchJava.

5.3.1 ArchJaMoPP

ArchJaMoPP est un parseur et générateur de modèles ArchJava. Il est constitué d'un ensemble de plugins Eclipse qui peuvent être utilisés pour transformer une description textuelle ArchJava en des modèles basés sur EMF et vice versa. À la base d'ArchJaMoPP, on trouve le code généré automatiquement dans l'étape précédente. La figure 5.4 montre l'ensemble de plugins générés par EMFText. Nous les présentons ci-après :

- *org.emftext.commons.antlr3_4_0* : ce projet introduit la bibliothèque dont dépend l'analyseur d'ArchJava;
- *org.emftext.language.archjava* : c'est le plugin de base du langage ArchJava (figure 5.3). Dans le dossier *src-gen* se trouve le code Java généré à partir du modèle générateur. Le dossier *métamodèle* contient le métamodèle Ecore, la spécification CS et le modèle générateur. Ce plugin est inaltérable par le générateur EMFText;

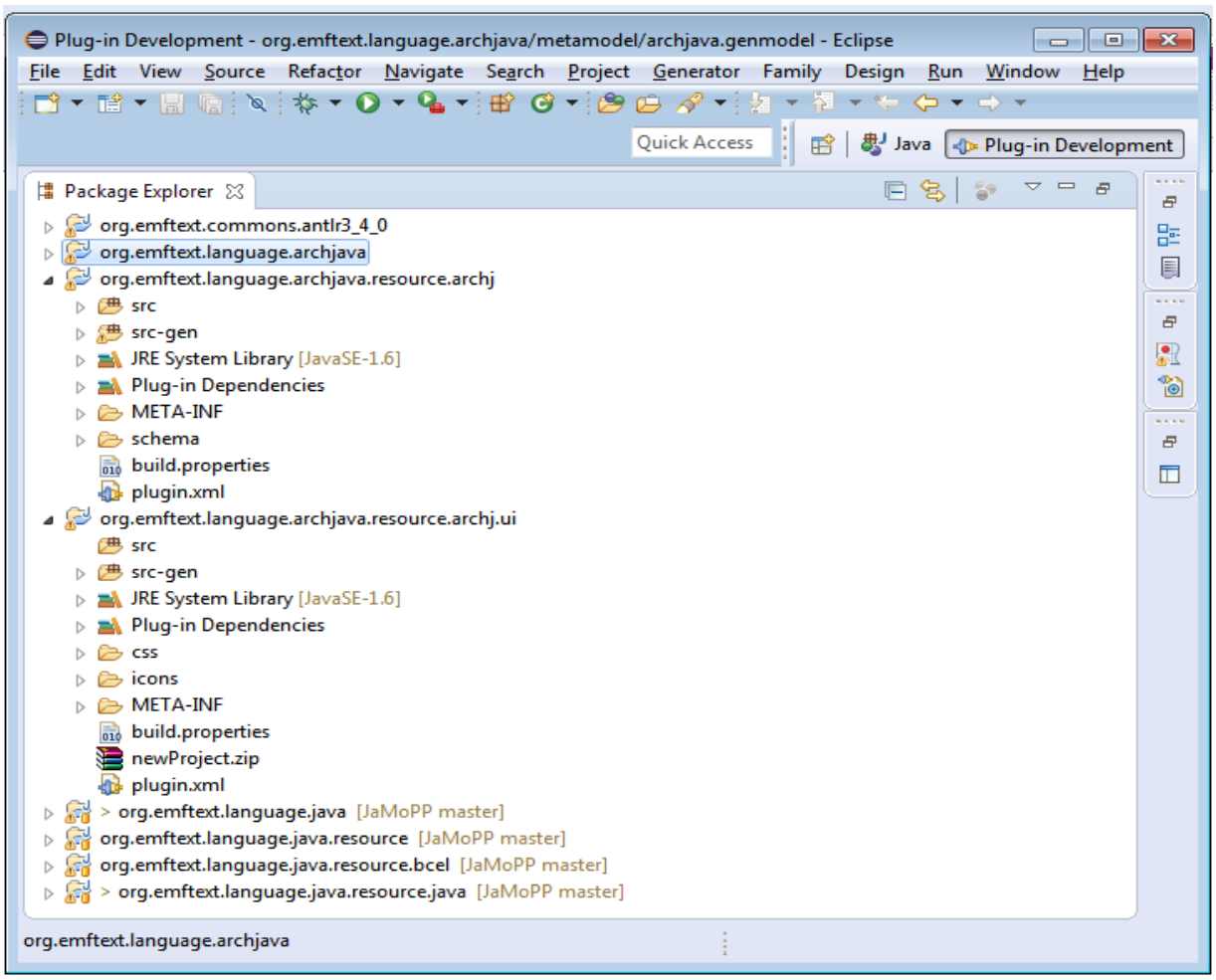


Figure 5.4. Projets générés par EMFText pour réaliser l'outillage.

- *org.emftext.language.archjava.resource.archj*: c'est le plugin qui contient le parseur, le générateur et l'infrastructure d'ArchJaMoPP. Le projet contient deux répertoires sources (src et src-gen). Contrairement à l'implémentation contenue dans le plugin de base et le répertoire src, EMFText redéfinit à chaque génération de code les fichiers sources de src-gen. src est destiné à la personnalisation manuelle du code, et contient par défaut les résolveurs de références. En plus des fichiers implémentant les outils d'ArchJava, un ensemble de points d'extensions spécifiques au langage sont générés dans le répertoire schema;
- *org.emftext.language.archjava.resource.archj.ui*: ce plugin contient les classes d'implémentation appartenant à l'interface d'utilisateur (UI) basée Eclipse. Cette

séparation de l'aspect interface de l'aspect implémentation facilite le détachement de l'infrastructure du langage de l'UI de Eclipse.

ArchJaMoPP est construit sur JaMoPP (i.e. il existe des relations d'héritage entre les deux implémentations). Sa génération, pour quelle soit correcte, doit réutiliser les plugins implémentant JaMoPP. Vu la sensibilité de cette opération, la version des plugins importés et de EMFText, ainsi que l'ordre de l'importation doit être choisis attentivement pour éviter l'obtention d'un code erroné. Plus précisément, nous avons importé les quatre plugins de base de JaMoPP (figure 5.4) : *org.emftext.language.java*, *org.emftext.language.resource.java*, *org.emftext.language.java.bcel* et *org.emftext.language.java.resource.java*, en utilisant le service web d'hébergement GitHub[1].

On peut utiliser ArchJaMoPP de deux manières différentes : dans l'environnent Eclipse ou en dehors. Pour l'utiliser dans Eclipse, il suffit d'installer ses plugins de base (figures 5.4). Pour l'utiliser en dehors de Eclipse, soit on utilise les plugins de base (en excluant le plugin d'UI Eclipse), soit on emploie l'outil extérieur *archjamoppc.jar* (c.f. Annexe B.3) qui fournit les mêmes fonctionnalités d'analyse via la ligne de commande.

5.3.2 Éditeur ArchJava

L'éditeur ArchJava fournit les mêmes fonctionnalités de l'éditeur Java de Eclipse : l'autocomplétion du code, le correctif rapide, la refactorisation, le coloriage du code, liens hypertexte et texte de survol pour la navigation rapide, la mise en évidence des parenthèses, accolades et crochets, le pliage de code, la vue Outline, indication instantanée des erreurs, …. La figure 5.5 montre un fichier ArchJava ouvert dans l'éditeur. Il porte l'extension *archj* et est contenu dans le package *Connector*. Il contient la classe composante *Worker* tirée du dossier *examples* de l'archive Archjava contenant les fichiers sources de la version 1.3.2 (voir le site officiel[2]). Elle étend la classe *Thread* et contient deux méthodes Java classiques (*fib()* et *run()*)

[1] https://github.com/DevBoost/JaMoPP/tree/master/Core

[2] http://archjava.fluid.cs.cmu.edu/software/

et un port nommé *barrier* spécifiant trois méthodes requises *start()*, *middle()* et *end()*. La méthode *run()*, par exemple, contient des instructions de type PortExpressionReference qui établissent des références internes vers les méthodes de port déclarées dans la même classe composante (e.g. *barrier.start();*).

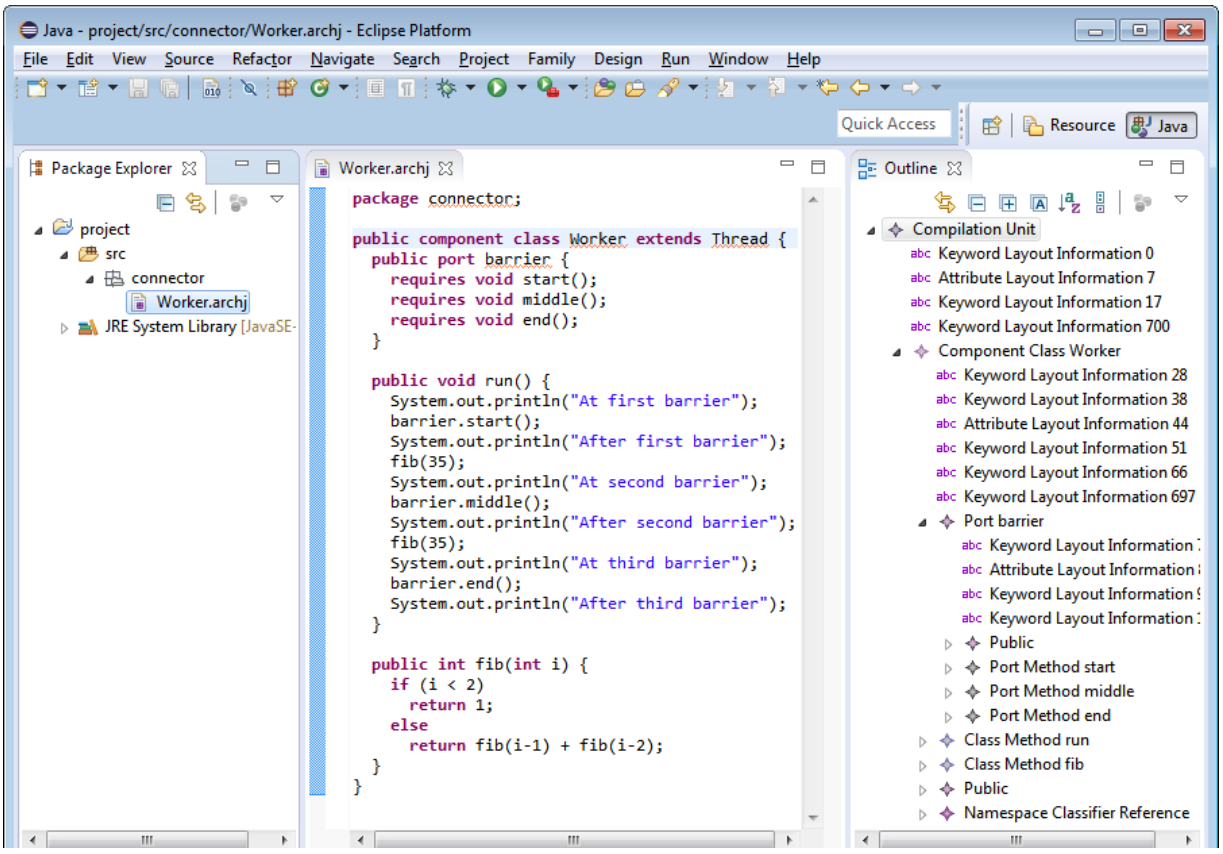


Figure 5.5. Editeur ArchJava.

Comme le montre la figure 5.5, le style du coloriage du texte de l'éditeur Archjava est similaire à celui de Java de Eclipse. Dans la vue Outline, les éléments de base de la classe Worker sont présentés sous forme arborescente. Cet éditeur joue un rôle très important dans l'usage d'ArchJava puisque c'est le seul moyen pour visualiser et corriger instantanément les erreurs commises lors de la spécification d'architecture. Il a connu plusieurs modifications depuis sa version initiale. Toutes les fonctionnalités de l'éditeur dont nous avons citées sont personnalisables à travers des techniques de personnalisation fournies par EMFText. Par exemple, le texte et l'icône à afficher pour l'autocomplétion du code sont ajustables à travers la désactivation de l'option *overrideProposalPostProcessor*.

5.3.3 ArchJaMoPPC

ArchJaMoPPC (c.f. Annexe B.3) est une classe Java qui implémente un outil de ligne de commande qui permet la transformation de fichiers ArchJava en modèles au format XMI conformément au métamodèle ArchJava. Reprenons l'exemple du composant Worker introduit plus haut. La ligne de commande suivante permet d'en extraire le modèle *Worker.archj.xmi* :

```
Java -cp c:\archjamoppc.jar org.emftext.language.archjava.archjamoppc.ArchJaMoPPC
c:\example c:\output
```

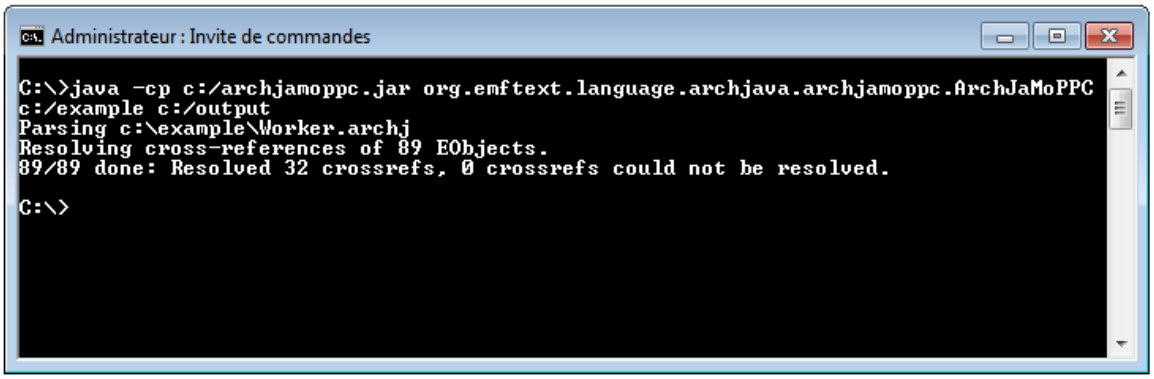


Figure 5.6. Exécution d'archjamoppc.jar.

L'exécution du fichier archjamoppc.jar se lance par la commande java en indiquant le nom qualifié de la classe, le dossier source qui contient le fichier Worker.archj ainsi que le dossier cible qui va contenir les modèles extraits. La fenêtre d'invite de commande montrée dans la figure 5.6 affiche le résultat d'analyse. Le modèle EMF construit est composé de 89 objets EMF (EObjects) dont 32 sont des objets référenciés. Le listing 5.2 présente le modèle XMI simplifié de la classe Worker (voir modèle complet en annexe B.7). ArchJaMoPPC permet aussi d'enregistrer les fichiers jar qui contiennent les classes java importées et de regrouper plusieurs unités de compilation en un seul modèle XMI.

```
<containers:CompilationUnit>

   <namespaces>connector</namespaces>

   <classifiers xsi:type="archjava:ComponentClass" name="Worker">

     <members xsi:type="archjava:Port" name="barrier">

       <modifiers xsi:type="modifiers:Public"/>
```

```
  <portMethods name="start">
    <typeReference xsi:type="types:Void"/>
    <annotationsAndModifiers xsi:type="archjava:Requires"/>
  </portMethods>
  ...
</members>
<members xsi:type="members:ClassMethod" name="run">
  <typeReference xsi:type="types:Void"/>
  <annotationsAndModifiers xsi:type="modifiers:Public"/>
    ...
  <statements xsi:type="statements:ExpressionStatement">
    <expression xsi:type="references:IdentifierReference">
      <next xsi:type="references:MethodCall">
        <target xsi:type="archjava:PortMethod"
                        href="//@classifiers.0/@members.0/@portMethods.0"/>
      </next>
      <target xsi:type="archjava:Port" href="//@classifiers.0/@members.0"/>
    </expression>
  </statements>
  ...
  <statements xsi:type="statements:ExpressionStatement">
    <expression xsi:type="references:MethodCall" target="//@classifiers.0/@members.1">
      <arguments xsi:type="literals:DecimalIntegerLiteral" decimalValue="35"/>
    </expression>
  </statements>
  ...
</members>
<members xsi:type="members:ClassMethod" name="fib">
  ...
</members>
<annotationsAndModifiers xsi:type="modifiers:Public"/>
<extends xsi:type="types:NamespaceClassifierReference">
```

```
      <classifierReferences>
        <target xsi:type="classifiers:Class"
                    href="java/lang/Thread.class.xmi#//@classifiers.0"/>
      </classifierReferences>
    </extends>
  </classifiers>
</containers:CompilationUnit>
```

Listing 5.2. Modèle simplifié de Worker en XMI.

5.4 Personnalisation

La dernière étape de notre approche consiste à personnaliser l'outillage généré. EMFText fournit trois mécanismes de personnalisation[1] : la redéfinition des artéfacts générés, l'utilisation des points d'extension générés et la redéfinition des classes de méta-informations. Dans cette section, nous présentons les modifications manuelles que nous avons apportées à la version initiale d'ArchJaMoPP (i.e. la version générée). Nous n'avons pas besoin d'utiliser les trois mécanismes d'EMFText car le changement concerne des classes générées par défaut dans le dossier *src*. Cela permet de garantir la conservation des changements lors de la régénération du code.

5.4.1 L'Assistant de Création de Projets et de Fichiers

Par défaut, l'assistant de création de projets généré par EMFText appartient à la catégorie *EMFText Project*, et son nom est *EMFText archj Project* (*EMFText* suivi par l'extension du DSL et le mot *Project*). De même, l'assistant de création de fichiers ArchJava, nommé *EMFText .archj file*, appartient à la catégorie *EMFText File* (voir figure 5.7). Pour les changer, le plugin *org.emftext.language.archjava.resource.archj.ui* de l'interface d'utilisateur basée Eclipse doit être modifié. Ce type d'informations (i.e. noms, catégories, icônes) peut être trouvé dans le fichier *Plugin.xml*, plus spécifiquement dans le point d'extension

[1] www.emftext.org/

org.eclipse.ui.newwizards. Nous avons changé le nom de l'assistant ainsi que sa catégorie. La figure 5.8 montre le résultat de nos modifications. La nouvelle catégorie de l'assistant de fichiers et de projets est *Other* et leurs noms sont, respectivement, *ArchJava file* et *ArchJava Project*. On peut aussi créer une nouvelle catégorie (e.g. ArchJava) pour les contenir en utilisant l'assistant de Eclipse pour la création des catégories.

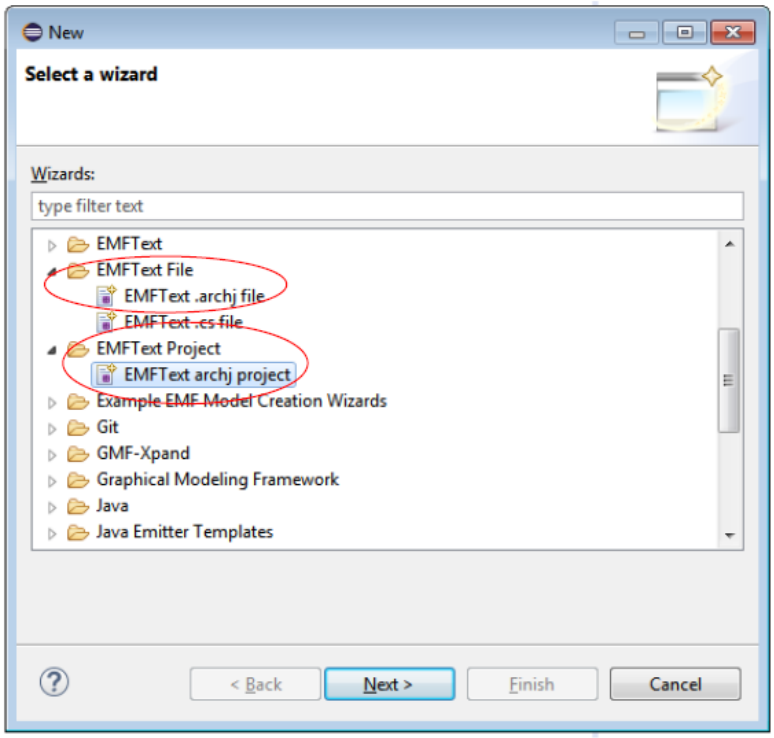


Figure 5.7. L'assistant de création de fichiers et de projets.

Bien que EMFText génère des assistants, ces derniers sont identiques pour tous les DSLs. Il s'agit d'un assistant de création de fichiers représenté par la classe *XyzNewFileWizard* et un assistant de création de projets représenté par la classe *XyzNewProjectWizard*, où xyz est l'extension des fichiers du DSL. Ces deux classes héritent de la classe *org.eclipse.jface.wizard.Wizard* et fournissent les services suivants :

- XyzNewFileWizard : fournit un champ pour spécifier l'emplacement du projet qui sera créé avec un fichier nommé *new_file.xyz* (i.e. nom donné par défaut). Ce dernier contient un programme minimal construit par la classe *XyzMinimalModelHelper*. Dans EMFText, le programme minimal est la représentation textuelle de l'élément racine indiqué dans le fichier cs du langage.

- XyzNewProjectWizard : demande le nom du nouveau fichier ainsi que l'emplacement de son conteneur (i.e. le répertoire ou le projet dans lequel il sera créé).

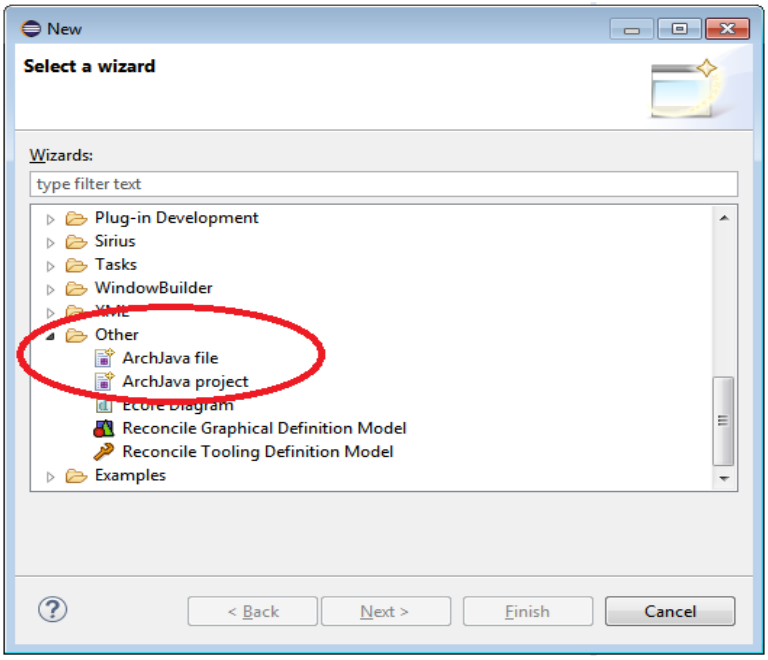


Figure 5.8. L'assistant personnalisé de création de fichiers et de projets.

Dans le cas d'ArchJava, le programme minimal généré par l'assistant de création de fichiers est une classe composante vide (figure 5.9). Il apparaît clairement que le comportement de l'assistant a besoin d'être modifié vu que le nom donné par défaut au composant et celui donné au fichier ne sont pas identiques. Nous devons donc changer ce comportement en redéfinissant la méthode *getMinimalModel(EClass eClass, EClass[] allAvailableClasses, String name)* de la classe ArchjMinimalModelHelper (c.f. Annexe B.4) qui construit le modèle minimal lors de la création du fichier. Cette dernière contient trois méthodes nommées getMinimalModel(), dont une permet de passer le nom du fichier comme paramètre. L'appel de méthode donné dans le listing 5.3 ignore le nom du fichier et oblige la création d'un composant qui porte le nom par défaut (i.e. *someName*). Pour obtenir le comportement voulu, on doit appeler la méthode getMinimalModel() avec comme troisième paramètre le nom du fichier, au lieu d'appeler une surcharge qui possède deux paramètres (le nom étant nul).

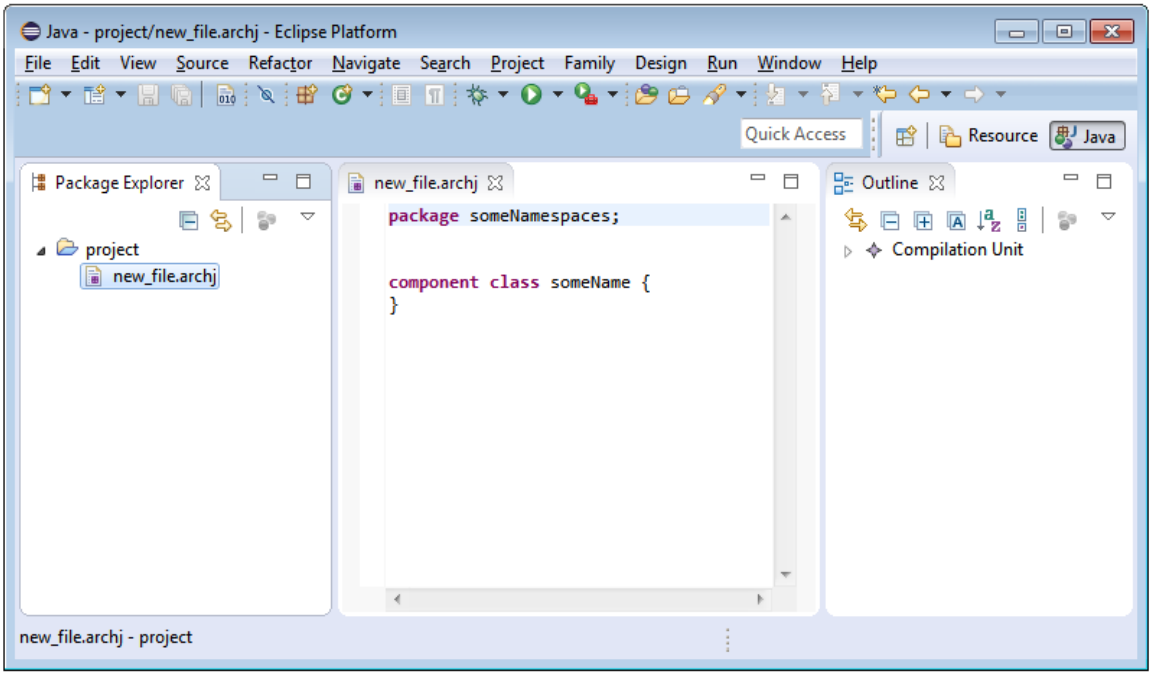


Figure 5.9. Programme minimal d'ArchJava.

```
EObject subModel = null;
if (reference.isContainment()) {
    EClass[] unusedClasses = getArraySubset(allAvailableClasses,
eClass);
    subModel = getMinimalModel(typeClass, unusedClasses);
}
```

Listing 5.3. Appel de la méthode getMinimalModel().

```
if (attribute.getName().equals("name") && name != null) {

    initialValue = name;
}
else if (attribute.getName().equals("namespaces") && conteneur !=
null){

    initialValue = conteneur;
}
else {
    initialValue = "some" +
org.emftext.language.archjava.resource.archj.util.ArchjStringUtil.ca
pitalize(attribute.getName());
}
```

Listing 5.4. Initialisation de l'attribut namespaces de la classe composante.

Notons qu'en ce qui concerne le nom du package, nous avons ajouté le bloc de code du listing 5.4 dans la même méthode (i.e. getMinimalModel()).Ainsi, un nouveau paramètre

nommé conteneur de type String est ajouté à la liste des paramètres. Le code recherche l'attribut namespaces de la classe composante puis affecte la valeur du paramètre dans la variable initialValue qui permet son initialisation. Cette valeur est obtenue de la variable *containerText* de la classe *ArchjNewFileWizardPage* (c.f. Annexe B.5). ContainerText est de type *org.eclipse.swt.widgets.Text* et contient l'emplacement du fichier .archj (i.e. le chemin absolu du répertoire dans le workspace) qui sera utilisé pour obtenir le nom du répertoire courant (i.e. le répertoire de plus bas niveau). Le listing 5.5 donne les changements à opérer sur la classe ArchjNewFileWizard (c.f. Annexe B.6) au niveau de la méthode *performFinish()* pour récupérer la valeur d'initialisation de l'attribut namespaces.

```
newContainerName= containerName;
int seperatorIdx= newContainerName.lastIndexOf('/');
newContainerName = containerName.substring(seperatorIdx+1);
```

Listing 5.5. Code ajouté à la classe ArchjNewFileWizard.

5.4.2 Résolution des Références

Dans le cas où le métamodèle contient des liens de références (i.e. des objets EMF de type *EReference* dont l'attribut *containment* est à faux), EMFText génère des résolveurs qui remplacent les identifiants symboliques des liens (des chaînes de caractères) par des identifiants uniformes de ressource (des URIs). Dans EMF, chaque modèle est associé à une ressource ayant un URI unique. Pour chaque lien de référence dans le métamodèle archjava, EMFText génère une classe de résolution dans le dossier *src* du package *org.emftext.language.archjava.resource.archj.analysis*. La classe *ArchjDefaultResolverDelegate* (c.f. Annexe B.8) utilise l'algorithme de résolution par défaut qui consiste à :

1- Rechercher dans la ressource (i.e. le fichier analysé) les objets ayant le même type que le lien de référence.

2- Comparer la valeur de l'attribut *ID* (identificateur), de l'attribut *name*, ou d'un attribut unique de type *EString* des objets trouvés (s'il existe) avec l'identifiant symbolique. Si les deux valeurs sont égales, l'objet sera marqué comme référencé. Cette étape est codée dans la méthode *tryToResolveIdentifierInObjectTree()*.

3- Si aucun objet n'est trouvé et si l'identifiant symbolique est un URI valide, EMFText charge la ressource sur l'URI. Si l'objet racine de cette ressource a le bon type, il sera considéré comme référencé. La méthode *tryToResolveIdentifierAsURI()* définit cette étape.

Cette stratégie de résolution ne prend pas en compte la spécificité et la complexité de certains types de référence (e.g. références aux classes). Ainsi, elle repose sur une recherche locale des objets (i.e. une recherche sur une seule ressource), ce qui n'est pas effectivement suffisant dans le cas de Java ou ArchJava, qui se caractérisent par une forte fragmentation de leurs modèles en plusieurs fichiers.

En ce qui concerne JaMoPP, pour permettre la résolution des références croisées (i.e. les liens de références qui mettent en évidence les relations entre différentes ressources), de nouveaux algorithmes sont utilisés en réimplémentant la méthode *resolve()* de l'interface *org.emftext.language.java.resource.java.IJavaReferenceResolver*[1]. JaMoPP utilise un registre global de ressources (*URIMap* de EMF), qui correspond à un classpath Java, pour les gérer avec leur emplacement physique sur le disque dure. Un classpath définit le chemin d'accès au répertoire où se trouvent les classes Java afin qu'elles soient exécutées par la machine virtuelle. JaMoPP utilise aussi l'analyseur de BCEL[2] (*Byte Code Engineering Library*) pour lire les classes disponibles uniquement en bytecode Java (e.g. la classe *java.lang.Object*) puis les traduire en modèles JaMoPP.

Initialement, les résolveurs de JaMoPP générés par EMFText faisaient appel à la méthode resolver() de la classe JavaDefaultResolverDelegate. Prenons l'exemple de la classe de résolution *ClassifierReferenceTargetReferenceResolver*, générée à partir de la référence *target* de la métaclasse *ClassifierReference* du métamodèle JaMoPP. ClassifierReference modélise les références qui ciblent les classifiers (i.e. classes, interfaces, annotations et énumérations).

[1] https://github.com/DevBoost/JaMoPP/tree/master/Core

[2] http://commons.apache.org/proper/commons-bcel/

La résolution de la référence *target* qui modélise le classifier cible se base sur l'algorithme par défaut de la classe JavaDefaultResolverDelegate comme montré dans le listing 5.6.

```
JavaDefaultResolverDelegate<ClassifierReference, Classifier> delegate =
 new JavaDefaultResolverDelegate<ClassifierReference,Classifier>();

public void resolve(String identifier,
 org.emftext.language.java.types.ClassifierReference container,
 EReference reference,
 int position,
 boolean resolveFuzzy,
 final org.emftext.language.java.resource.java.IJavaReferenceResolveResult
       <Classifier>  result) {

        delegate.resolve(identifier, container, reference, position,
                         resolveFuzzy, result);

}
```

Listing 5.6. La méthode resolve() de ClassifierReferenceTargetReferenceResolver.

Toutes les classes de résolution générées utilisent le même code, à quelques différences près (les arguments et les paramètres). L'instance *delegate* sert à déléguer l'appel de resolve() de ClassifierReferenceTargetReferenceResolver à JavaDefaultResolverDelegate. Le code qui définit le nouvel algorithme de résolution pour cet élément est donné en annexe B.9. Les étapes principales de l'algorithme sont les suivantes :

1- Si le conteneur de la référence est un objet de type *NamespaceClassifierReference* (la métaclasse qui représente les références ciblant un classifier dont le nom peut inclure un espace de noms), il sera recherché s'il existe un objet de type *ConcreteClassifier* dont le nom (la valeur de l'attribut name) correspond à l'identifiant symbolique dans l'espace de noms. Une fois trouvé, l'objet sera affecté à l'attribut target (la cible de référence) et marqué comme référencé.

2- Si le nom du classifier cible fait partie d'un nom absolu d'une classe (nom du package + nom de la classe), l'objet ayant le même nom sera directement affecté dans la variable target et marqué comme référencé, après une recherche dans le package conteneur via le classpath.

3- Si aucun classifier n'est trouvé et si le conteneur de la référence est un objet de type *NewConstructorCall* (métaclasse représentant les instructions d'instanciation de classes), qui est lui-même une référence, le contexte de recherche sera le classifier type de l'objet.

En résumé, la méthode générale de résolution n'est souvent pas propice à tous les scénarios vu que le comportement de resolve() se diffère d'une référence à une autre suivant le métamodèle du langage. De ce qui précède, il apparaît que l'importation de JaMoPP élimine le problème de résolution des références Java dans les programmes ArchJava, mais, en réalité, ArchJaMoPP, tel qu'il est généré par EMFText, n'utilise pas les résolveurs de JaMoPP, mais plutôt les résolveurs par défaut. En effet, la génération des résolveurs ne se limite pas seulement aux références du métamodèle ArchJava mais concerne aussi les références importées de JaMoPP.

Reprenons l'exemple de ClassifierReferenceTargetReferenceResolver. La version initiale d'ArchJaMoPP contenait deux versions de ce résolveur. La première est celle importée du package org.emftext.language.java.resource.java.analysis de JaMoPP, la deuxième est celle générée dans le package org.emftext.language.archjava.resource.archj.analysis. La méthode resolve() de la deuxième version du résolveur est similaire à celle montrée dans le listing 5.6. En d'autres termes, la version initiale utilisait l'algorithme de résolution par défaut. Nous avons donc réimplémenté resolve() pour produire le bon comportement. La solution est assez simple : remplacer l'appel de resolve() par défaut par celle du résolveur importé. Le code qui permet ceci est donné dans le listing 5.7.

```
   org.emftext.language.java.resource.java.analysis.
   ClassifierReferenceTargetReferenceResolver delegate =    new
   ClassifierReferenceTargetReferenceResolver();

 public void resolve(String identifier,
   org.emftext.language.java.types.ClassifierReference container,
   EReference reference,
   int position,
   boolean resolveFuzzy,
   final
org.emftext.language.archjava.resource.archj.IArchjReferenceResolveResult
<org.emftext.language.java.classifiers.Classifier> result) {

   delegate.resolve(identifier, container, reference, position,
                    resolveFuzzy, result);
 }
```

Listing 5.7. Appel de resolve() du résolveur importé.

5.4.3 Résolution des Ambiguïtés

Parfois, l'intégration d'un DSL dans un langage générique comme Java engendre des conflits dans la grammaire. Une grammaire est dite ambiguë lorsque deux ou plusieurs règles différentes peuvent s'appliquer en même temps pendant l'analyse syntaxique. À tire d'exemple, une méthode de classe Java et une méthode fournie de port ArchJava ont la même syntaxe concrète, ce qui produit un parseur qui ne peut pas les différencier. En effet, toutes les méthodes d'une classe Java seront identifiées comme des méthodes de port. Ce comportement est dû au fait que la métaclasse *PortMethod* hérite de la métaclasse *ClassMethod* de JaMoPP, sachant que les règles de grammaire de la spécification d'ArchJava sont prioritaires à celles de la spécification importée (i.e. les règles de Java). Avec plus de 50000 lignes de codes, la modification de la classe qui définit ce comportement est très difficile à ceux qui ne maîtrisent pas son code.

EMFText génère pour chaque élément d'ArchJava (représenté par une métaclasse dans le métamodèle), une méthode appelée *parse_element_name()* qui vérifie si le code analysé correspond à une syntaxe valide de ce dernier, puis construit l'objet EMF qui le décrit. Profitant de ces informations, l'ambiguïté peut être résolue si on arrive à empêcher l'appel de *parse_org_emftext_language_archjava_PortMethod()* lors de l'analyse du corps des classifiers (i.e. classes, interfaces, annotations, énumérations ou classes de composant). En effet, le corps d'un classifier est un ensemble de membres (représentés dans JaMoPP par la métaclasse Member). Puisque la méthode d'une classe est un membre, la méthode de port est alors un membre par héritage (selon ArchJaMoPP). Le listing 5.8 est un extrait de *parse_org_emftext_ language_ java_members_Member()* : la méthode qui permet l'analyse du corps. Il montre l'ordre de priorité qui cause le problème d'ambiguïté. Nous pouvons remarquer que l'opération d'analyse parse_org_emftext_language_archjava_PortMember() arrive avant l'opération parse_org_emftext_language_ java_members_ClassMethod().

```
   ......
   switch (alt242) {
       case 1 :
           {......}
       ......
       case 3 :
           {
pushFollow(FOLLOW_parse_org_emftext_language_archjava_PortMethod_in_parse_
```

```
            org_emftext_language_java_members_Member24963);
              c2=parse_org_emftext_language_archjava_PortMethod();
              state._fsp--;
              if (state.failed) return element;
              if ( state.backtracking==0 ) { element = c2; /* this is a
                    subclass or primitive expression choice */
              }
            }
            break;
        ......
        case 12 :
            {
pushFollow(FOLLOW_parse_org_emftext_language_java_members_ClassMethod_in_
            parse_org_emftext_language_java_members_Member25053);
             c11=parse_org_emftext_language_java_members_ClassMethod();
              state._fsp--;
              if (state.failed) return element;
              if ( state.backtracking==0 ) { element = c11; /* this is a
                     subclass or primitive expression choice */
              }
            }
            break;
        ......
   }
   ......
```

Listing 5.8. Extrait de la méthode parse_org_emftext_language_java_members_Member().

alt242 contient un entier qui représente le type d'opération d'analyse à appeler, alors que *c2* et *c11* sont deux variables, respectivement de type PortMethod et ClassMethod, qui vont contenir le résultat d'analyse. Il y a 15 types d'opérations correspondant à 15 types de membres (selon le métamodèle ArchJaMoPP). La valeur stockée dans alt242 est déterminée après une analyse lexicale et syntaxique. D'un point de vue syntaxique, la méthode de port n'est pas un membre de classifier puisque son conteneur est le port. Le code du bloc *case 2* devrait alors être remplacé par celui de *case 12* car ce dernier ne sera jamais exécuté (alt242 ne prend jamais la valeur 12). Le détail de modification est donné dans le listing 5.9.

```
   ......
   switch (alt242) {
        case 1 :
            {......}
        ......
        case 3 :
            {
pushFollow(FOLLOW_parse_org_emftext_language_java_members_ClassMethod_in_
            parse_org_emftext_language_java_members_Member25053);
              c11=parse_org_emftext_language_java_members_ClassMethod();
              state._fsp--;
```

```
            if (state.failed) return element;
            if ( state.backtracking==0 ) { element = c11; /* this is a
                    subclass or primitive expression choice */
            }
        }
        break;
        ......
        case 11 :
            {......}
        break;
        case 13 :
            {......}
        ......
    }
    ......
```

Listing 5.9. parse_org_emftext_language_java_members_Member() non-ambiguë.

5.5 Discussion

La création d'ArchJaMoPP part de la considération que le métamodèle du langage est le point de départ du développement. D'une part, il définit formellement et explicitement les concepts de l'ADL, ce qui limite les ambiguïtés et facilite le développement. D'autre part, il permet de générer une partie de l'infrastructure logicielle de manipulation de code. Le passage de l'espace technique des modèles (modelware) à l'espace technique des grammaires (grammarware) est aussi important pour définir les liens entre la syntaxe abstraite et la syntaxe concrète du langage, et donc entre le code et le métamodèle. Comme pour le métamodèle, la grammaire est un modèle utilisé pour générer l'outillage. Il est important de noter que l'utilisation d'EMFText est primordiale pour le succès de notre approche. Mais pouvons-nous utiliser un autre workbench ? Techniquement, Xtext[1] possède les mêmes capacités de génération de code que EMFText et peut être utilisé pour effectuer les mêmes tâches. Cependant, il n'est pas convenable pour l'implémentation de notre approche, car il considère la grammaire comme le point de départ du développement, alors que notre approche est basée métamodèle. De plus, nous pensons que dans ce contexte, la meilleure pratique de développement consiste à commencer par la modélisation à haut niveau de l'ADL (i.e. la

[1] https://eclipse.org/Xtext/

création de son métamodèle), et ensuite continuer à descendre dans les niveaux d'abstraction. En d'autres termes, la grammaire doit être basée sur le métamodèle, et non l'inverse comme dans Xtext. A notre connaissance, EMFText est le seul workbench qui adopte une telle pratique.

Globalement, ArchJaMoPP a été construit pour illustrer l'efficacité de notre approche. L'ADL choisi pose plusieurs challenges, que ce soit au niveau de la modélisation ou de l'implémentation des outils d'assistance. Les problèmes rencontrés et la manière dont ils ont été résolus nous ont donné un aperçu sur les capacités et les limitations de l'approche proposée. Nous pouvons les résumer dans les points suivants :

- La grande partie du code d'implémentation est générée automatiquement à partir des modèles (i.e. le métamodèle et la grammaire);
- EMFText est suffisamment puissant pour manipuler des langages génériques. Son mécanisme d'importation favorise la réutilisation des métamodèles et des spécifications syntaxiques. De plus, EMFText permet de séparer le code à réécrire automatiquement du code écrit manuellement dans le but de conserver ce dernier en cas de régénération. Ces possibilités de réutilisation simplifient la modification et l'évolution des ADLs et leurs outils de support;
- Les modèles semblent non seulement diriger le développement, mais aussi le documenter. Etant donné que le métamodèle de l'ADL et le modèle CS sont disponibles et fiables, il suffit de connaître comment fonctionne EMFText pour pouvoir comprendre le code d'implémentation;
- Les avantages de notre approche ne s'arrêtent pas à la construction du parseur, du générateur et de l'interface utilisateur pour un ADL donné. Elle offre également une meilleure intégration avec les outils de modélisation basés EMF;
- Malheureusement, notre approche dépend étroitement de l'environnement de développement de DSL. Son langage de spécification de syntaxe ainsi que ses mécanismes de génération doivent être maîtrisés afin de faciliter le développement des outils, et surtout leur personnalisation;

- La création d'une infrastructure d'ADL similaire à AcmeLib ne peut répondre que partiellement aux problèmes énoncés dans l'introduction. Cependant, elle nous fait avancer à grand pas vers la solution.

En effet, l'impact de notre approche sur le domaine de l'architecture logicielle fera en sorte que les problèmes liés à l'outillage et à l'automatisation des activités architecturales seront considérés comme des problèmes de transformation, d'analyse, de raffinement, de traçabilité et de comparaison de modèles EMF. Prenons le problème de génération de code à partir d'une spécification architecturale. Avec le parseur et le générateur créés en utilisant EMFText, ainsi que l'outil JaMoPP, la génération du code Java n'est plus une transformation de modèle à texte (Model to Text), mais plutôt une transformation de modèle à modèle (Model to Model). Le code montré dans le listing 5.10 est une transformation QVT simple qui permet de créer un fichier Java à partir d'un fichier ArchJava.

```
1   /*
2     Model type declaration
3   */
4   modeltype java uses "http://www.emftext.org/java";
5   modeltype archjava uses "http://www.emftext.org/archjava";
6
7   /*
8    The transformation
9   */
10  transformation archj2java(in archjModel: archjava, out javaModel: java);
11  main() {
12     archjModel.rootObjects()[java::containers::CompilationUnit] -> map cu2cu();
13  }
14  /*
15    Create a compilation unit
16  */
17  mapping java::containers::CompilationUnit::cu2cu():java::containers::CompilationUnit{
18
19     namespaces := self.namespaces;
20     var classifiers:= self.classifiers;
21     Sequence{1..classifiers->size()}->forEach(i){
22     let c= classifiers->asSequence()->at(i) in
23         if c.oclIsTypeOf(archjava::ComponentClass) then
24              result.classifiers+= map com2c(c.oclAsType(archjava::ComponentClass))
25         else if c.oclIsTypeOf(java::classifiers::Class) then
26              result.classifiers+= map c2c(c)
27         else if c.oclIsTypeOf(java::classifiers::Annotation) then
28              result.classifiers+= map a2a(c)
29         else if c.oclIsTypeOf(java::classifiers::Interface) then
30              result.classifiers+= map i2i(c)
31         else result.classifiers+= map e2e(c)
32         endif
34         endif
35         endif
36         endif;
37     }
38  }
39  /*
40    Create an Annotation from an Annotation
41  */
42  mapping a2a(in c:java::classifiers::ConcreteClassifier):java::classifiers::Annotation{
```

```
44
45      result.name := c.name;
46      result.members+= c.members;
47   }
48   /*
49     Create an Interface from an Interface
50   */
51   mapping i2i(in c:java::classifiers::ConcreteClassifier):java::classifiers::Interface{
52
53      result.name := c.name;
54      result.members+= c.members;
55   }
56   /*
57     Create an Enumeration from an Enumeration
58  */
59   mapping e2e(in c:java::classifiers::ConcreteClassifier):java::classifiers::Enumeration{
60
61      result.name := c.name;
62      result.members+= c.members;
63   }
64  /*
65     Create a Class from a Component
66  */
67   mapping com2c(in c:java::classifiers::ConcreteClassifier):java::classifiers::Class{
68
69      result.name := c.name;
70      result.members+= c.members;
71   }
72   /*
73     Create a Class from a Class
74  */
75   mapping c2c(in c:java::classifiers::ConcreteClassifier):java::classifiers::Class{
76
77      result.name := c.name;
78      result.members+= c.members;
79   }
```

Listing 5.10. La fonction de transformation QVT archj2java().

La transformation opérationnelle *archj2java* commence par la déclaration des types de métamodèles (java et archjava) en spécifiant leur URI (lignes 4 et 5). La signature de transformation (ligne 10) définit le processus de conversion d'un modèle source (*archjModel*) de type archjava vers un modèle cible (*javaModel*) de type java. La fonction *main()* représente le point de départ de la transformation. Elle définit la règle de transformation (ou de mapping) *cu2cu()* qui s'applique à tous les objets racines du modèle source (ligne 12). Sachant à l'avance que le modèle ArchJava contient un seul objet racine de type *java::containers:: CompilationUnit*, le rôle de la fonction cu2cu() serait donc de créer une unité de compilation dans le modèle cible à partir de l'unité de compilation du modèle source selon les étapes suivantes :

- L'espace de noms de l'unité source devient l'espace de noms de l'unité cible (ligne 19);

- Pour chaque classifier dans l'unité source, un classifier est créé dans l'unité cible (de 21 à 37). Si c'est une interface, une énumération, une annotation ou une classe, le classifier sera copié sans modification dans l'unité cible. Si c'est un composant, il sera transformé en une classe en utilisant la fonction de mapping *com2c()*;
- Quelque soit le type de classifier source, son nom et ses membres (i.e. son corps) sont copiés sans modification dans le classifier cible;

La dernière étape de la transformation indique que les constructions d'ArchJava tels que les ports et les connecteurs, demeurent intactes, c'est-à-dire que le fichier Java résultant peut contenir un code ArchJava. Pour changer ce comportement, l'instruction d'affectation `result.members+= c.members;` (lignes 46, 54, 62, 70 et 78) doit être changée. L'utilisation de QVT n'est pas le seul moyen de transformation. Une autre possibilité serait d'employer l'API Java d'ArchJaMoPP, notamment celle générée du modèle générateur. Le listing 5.11 donne le code Java équivalent à la transformation QVT du listing 5.10.

La seule différence réside dans le copiage du code ArchJava dans le fichier cible (*Example.java*). La ressource Java créée en utilisant la méthode *createResource()* n'accepte pas les objets EMF qui ne se conforment pas au métamodèle JaMoPP. En d'autres termes, tous les objets instances de métaclasses ArchJaMoPP du modèle source (*Example.archj*) sont automatiquement supprimés lors du copiage. Ce problème ne se pose que pour le copiage des membres, vu que les classifiers spécifiques à ArchJava (i.e. les composants) se transforment automatiquement en des classes Java. Pour résoudre ce problème, nous avons choisi de copier les membres de chaque classifier contenu dans la ressource Example.archj dans la variable *members* avant de les supprimer définitivement. Le résultat de l'exécution de la méthode archj2java() sera donc un fichier Java qui ne contient que des classifiers vide. Pour manipuler les membres autrement, la variable members peut être utilisée et l'instruction *EcoreUtil.remove(m);* doit être remplacée par le code de manipulation voulu.

Malgré sa simplicité, cet exemple montre l'apport de notre approche pour les architectures logicielles par l'intégration d'une architecture logicielle (Example.archj) avec les outils de modélisation (i.e. QVT et l'API riche de l'environnement de modélisation EMF). Les développeurs qui maîtrisent les outils de modélisation EMF seront alors capables de créer

facilement des outils de support, que ce soit pour la génération du code, l'analyse, le raffinement ou n'importe quel domaine d'architecture.

```
package codegeneration;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.emftext.language.archjava.resource.archj.mopp.ArchjResourceFactory;
import org.emftext.language.java.classifiers.ConcreteClassifier;
import org.emftext.language.java.members.Member;
import org.emftext.language.java.resource.JavaSourceOrClassFileResourceFactoryImpl;

public class Generator {

      public void archj2java() throws IOException  {

         // create resource set
         ResourceSet rs = new ResourceSetImpl () ;


         /* register the ArchJava and the Java Resource Factory to the ".archj"
            and ".java" extensions respectively */
         Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
                          "archj", new ArchjResourceFactory());
         Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
                          "java", new JavaSourceOrClassFileResourceFactoryImpl());

         // get the ArchJava Resource
         rs.getResource(URI.createFileURI
                         ("D:/archJava/codeGeneration/models/Example.archj"), true);

         // create a Java resource
         Resource javaResource = rs.createResource(URI.createFileURI
                                 ("D:/archJava/codeGeneration/models/Example.java"));
         Resource archjavaResource = rs.getResources().get(0);

         // get classifiers and clean their members
         if (archjavaResource != null) {
             Iterator<EObject> i = archjavaResource.getAllContents();
             while (i.hasNext()) {
                    EObject o = i.next();
                    if ( o instanceof ConcreteClassifier) {
                        ConcreteClassifier cc = (ConcreteClassifier) o;
                        List<EObject> members= new ArrayList<EObject>();
                        if (cc.getMembers().size()>0){
                            for (Member member : cc.getMembers()) {
                                members.add(member);
                             }
                        }
                        for(EObject m : members){
                                  EcoreUtil.remove(m);
```

```
                        }
                    }
                }

                // copy the ArchJava resource contents into a java resource
                javaResource.getContents().addAll(archjavaResource.getContents());
                List<Resource> result = new ArrayList<Resource>();
                result.add(javaResource);
                for (Resource jResource : result) {
                     Map<Object, Object> options = new HashMap<Object, Object>();
                     jResource.save(options);
                }
            }
        }

    }
```

Listing 5.11. La fonction archj2java () en Java.

5.6 Conclusion

Nous avons donné dans ce chapitre un aperçu sur les artefacts et les étapes suivies pour construire ArchJaMoPP. Nous avons cité les différents outils avec lesquels notre approche a été mise en œuvre. Nous avons choisi l'environnement Eclipse pour sa flexibilité et sa capacité à intégrer des plugins créés spécialement pour le domaine de modélisation. Les plugins EMF et Ecore Tools fournissent un environnement complet pour la création, la manipulation, la sérialisation, la représentation graphique et la transformation de modèles. Les plugins JaMoPP et EMFText nous ont largement servi pour définir la syntaxe concrète d'ArchJava et générer l'outillage support ArchJaMoPP. Une version en ligne de commande nommée ArchJaMoPPC est introduite avec son code source Java. Ainsi, nous avons présenté les types de modifications effectuées afin de personnaliser et adapter les outils générés aux spécificités d'ArchJava.

6 Conclusion et Perspectives

Nous avons étudié, dans le cadre de cette thèse, les techniques de l'ingénierie dirigée par les modèles et plus particulièrement l'initiative MDA qui fournit un ensemble d'avantages améliorant le processus de développement, d'évolution et de maintenance des logiciels et augmentant la métrise de la complexité. Ainsi, MDA offre la pérennité des spécifications métier des systèmes logiciels en se basant sur des formalismes stables avec une sémantique largement partagée par les personnes qui utiliseront les modèles (e.g. UML). Cet avantage est important par le fait qu'il permet une grande durée de vie des modèles représentant l'aspect métier des systèmes. L'augmentation de la productivité grâce à l'automatisation des transformations de modèles est un autre avantage important de MDA. Pour transformer les modèles, MDA exige que ces derniers soient précis, complets et bien définis.

Les architectures logicielles, à leur tour, sont des modèles qui représentent les systèmes logiciels. Une fois décrits et validés, ces modèles peuvent être utilisés pour générer le code source d'implémentation des systèmes. Leur description est basée sur un formalisme de modélisation qui définit leurs éléments ainsi que les relations entre eux. Nous avons étudiés les fameux ADLs qui formalisent la description architecturale en s'intéressant au particulier à l'aspect de métamodélisation et l'automatisation. Un métamodèle d'un ADL permet de définir explicitement ses constructions et facilite le développement d'outils d'assistance. L'automatisation concerne l'analyse, la manipulation et la génération de code par des outils logiciels. Nous avons inspecté donc pour chaque ADL étudié, l'état de ses outils.

6.1 Résumé des contributions

Les systèmes logiciels modernes se distinguent par une complexité accrue, qui doit être gérée à un niveau d'abstraction élevé. L'architecture logicielle à base de composants fournit une telle abstraction en décrivant la structure d'un système logiciel indépendamment des langages de programmation et de leurs plateformes d'exécution, ce qui permet de réduire sa complexité. Plusieurs travaux de recherches ont été consacrés aux architectures logicielles. En témoigne le nombre d'ADLs inventés pour les décrire formellement. Les ADLs sont souvent dotés d'outils d'accompagnement plus ou moins adaptés aux besoins des utilisateurs. En plus

de la description architecturale, chaque ADL tend à se concentrer sur un domaine spécifique et nécessite donc des outils implantant des fonctionnalités qui lui sont propres. Par conséquent, ces outils deviennent inappropriés voire incapables d'accomplir certaines tâches telles que la distribution, le déploiement, la dynamique, etc. Des formats d'échanges ont été proposés pour supporter l'interopérabilité entre ces outils et pour faciliter le passage d'un ADL à un autre tout en conservant les propriétés architecturales. Mais, en réalité, ce passage n'est pas aisé vu le manque d'outils pour le faire. L'extension de ces outils par de nouvelles fonctionnalités est une solution probable mais difficile à mettre en œuvre. D'un côté, ils nécessitent plusieurs améliorations avant qu'aucune extension ne soit possible, d'un autre côté, leur code source et la documentation de leur processus de développement sont difficilement accessibles, voire inaccessibles. Nous avons présenté dans ce travail une approche dirigée par les modèles pour les architectures logicielles dont l'un de ses objectifs importants est de surmonter ces limites.

Armée d'un environnement de développement de DSLs (un langage workbench), notre approche préconise l'utilisation des métamodèles représentant les ADLs pour guider le processus de développement d'outils supports. Les ADLs étant des DSLs, l'élaboration de leurs métamodèles est relativement simple. D'ailleurs, plusieurs ADLs disposent déjà d'un métamodèle explicite. Un métamodèle d'un ADL sert de base à la génération d'une API riche de manipulation en utilisant l'environnement Eclipse et EMF, dont la flexibilité et l'extensibilité sont grandement supportées, ce qui donne aux développeurs plusieurs opportunités de modélisation, d'évolution et de personnalisation. L'approche consiste dans un deuxième temps à utiliser un environnement de développement de DSLs (dans ce travail EMFText) pour créer une syntaxe concrète de l'ADL cible. À partir des artefacts créés dans les deux étapes précédentes (le métamodèle, son code dérivé et la syntaxe concrète), le workbench génère les outils supports. Une étape de modification et personnalisation doit être effectuée pour enrichir et adapter ces outils aux besoins.

L'ADL ArchJava constitue le cas d'étude pour montrer les apports de notre approche. En plus de son niveau bas qui est très proche du langage générique Java, il représente un cas d'étude extrême par rapport aux autre ADLs. La difficulté réside dans le fait que ArchJava ne possède pas une représentation abstraite au niveau méta. Nous avons donc créé un métamodèle ArchJava en se basant sur le métamodèle Java appelé JaMoPP. La concrétisation de

l'approche a contribué à la création d'ArchJaMoPP, une bibliothèque qui regroupe plusieurs artefacts et outils manipulant les modèles ArchJava.

6.2 Perspectives

Malgré que le cas d'étude présenté dans cette thèse n'est pas trivial, il ne suffit pas pour montrer les avantages de notre approche. Les modifications que nous avons apportées à la version initiale d'ArchJaMoPP nous ont permis de personnaliser l'éditeur de texte (e.g. le renommer) et d'éliminer les ambiguïtés que le parseur n'arrive pas à détecter. Ainsi, nous avons pu résoudre les problèmes de résolution des références d'ArchJava. Nous avons identifié trois perspectives à envisager, qui concernent l'évaluation et l'amélioration d'ArchJaMaPP, l'utilisation d'ArchJaMoPP dans le cadre de l'analyse et l'évolution architecturale et enfin l'application de notre approche aux ADLs les plus fréquemment étudiés.

Évaluation et Amélioration d'ArchJaMoPP

ArchJaMoPP se base principalement sur JaMoPP qui est exhaustivement testé avec succès en utilisant 79,017 fichiers Java (15,5 millions de lignes de code non vides contenant des commentaires) [Hei 10]. Nous n'avons pas testé ArchJaMoPP avec autant de fichiers car ArchJava est assez peu utilisé par rapport à Java, mais nous l'avons évalué manuellement avec quelques fichiers ArchJava téléchargeables avec le compilateur ArchJava[1]. En d'autres termes, la fiabilité d'ArchJaMoPP n'est pas garantie et beaucoup de travail reste à faire dans ce sens.

La structure explicite des composants constitutifs du plugin ArchJaMoPP (voir figure 5.1) rend facile l'ajout de nouvelles fonctionnalités. Par exemple, il est muni d'un outil pour l'extraction des modèles EMF représentant le code source de la description architecturale, mais cette fonctionnalité est implicite et se présente sous forme de classes Java. On peut alors développer un service nommé *Export as EMF Model* ou *Generate EMF Model* et l'intégrer à

[1] http://archjava.fluid.cs.cmu.edu/software/index.html

l'interface homme machine pour faciliter cette tâche à l'utilisateur. De plus, la représentation standard EMF permet une intégration facile des modèles ArchJaMoPP au sein des outils EMF de Eclipse. Par exemple, il sera intéressant de créer un éditeur graphique avancé en se basant sur GMF (*Graphical Modeling Framework*). Le potentiel d'extension des outils créés suivant notre approche est énorme. Même les cas radicales d'évolution, tel que l'extension de l'ADL, peuvent être facilement gérés grâce à l'extensibilité du métamodèle et de la syntaxe concrète, et au concept de régénération de code offert par le langage workbench.

A court terme, nous visons l'amélioration d'ArchJaMoPP par la création des interfaces graphiques exhibant les services présentés actuellement comme une API Java (e.g. l'analyse et la génération de code). Ainsi, nous comptons créer un éditeur graphique qui permet la description visuelle d'architectures où les composants ArchJava graphiques, comme les classes dans les diagrammes de classes UML, se caractérisent par des attributs, des ports et des opérations. Cette version visuelle de l'architecture peut être utilisée par la suite pour générer le code ArchJava.

Applications d'ArchJaMoPP

Cette deuxième perspective vise l'utilisation d'ArchJaMoPP dans le cadre de l'analyse et l'évolution statiques des architectures logicielles. Dans [Ham 15], les langages de modélisation et de transformation standards OCL et QVT sont utilisés pour implémenter les opérations de refactorisation du code Java. OCL permet la spécification des contraintes et requêtes d'inspection de modèles EMF qui représentent le code Java, et QVT performe leur transformation. De manière similaire, nous voulons créer des outils d'analyse et d'évolution architecturale statique en profitant de ces langages et de la présentation EMF des architectures offertes par ArchJaMoPP.

Généralisation de l'approche aux autres ADLs

La troisième perspective qui se dégage de ce travail renvoie à la création des outils semblables à ArchJaMoPP pour supporter les autres ADLs sur le long terme. Ce qui nous motive à aller dans ce sens est le nombre réduit des constructions d'ADLs, ainsi que la disponibilité de leurs métamodèles. L'accomplissement de ce travail va ouvrir la porte devant la définition des transformations de modèles permettant le passage d'un ADL à un autre. Cela

pourra avoir comme effet l'accroissement de l'interopérabilité entre les ADLs. Un autre axe de recherche que nous avons identifié est la génération de code à partir des architectures logicielles. Avec des outils comme JaMoPP qui rapproche le code Java et les modèles PSMs de MDA, nous pensons à la possibilité de résoudre le problème de la perte de la structure architecturale lorsqu'on passe de l'architecture au code source (e.g. les connecteurs explicites au niveau méta deviennent implicites au niveau code source).

Bibliographie

[Aad 12] SAE International Standard. Architecture Analysis and Design Language (AADL) AS-5506B. Technical report. (2012)

[Ald 02] Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting software architecture to implementation. In Proceedings of the Twenty Fourth International Conference on Software Engineering (ICSE'02), Orlando, FL, USA, 187–197. (2002)

[Ald 03] Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Language Support for Connector Abstractions. In Proceedings of the 2003 European Conference on Object-Oriented Programming (ECOOP'03), Darmstadt, Germany. (2003)

[All 96] Allen, R., Garlan, D.: The wright architectural specification language. Technical report CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science. (1996)

[All 98] Allen, R., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architecture. In Proceedings of the Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal. (1998)

[And 05] Andy, J., Wang, K. Q.: Component-Oriented Programming. John Wiley & Sons, Inc., Hoboken, New Jersey. (2005)

[Ast 02] Astels, D.: Refactoring with UML. In Proceeding of Int'l Conference on eXtreme Programming and Flexible Processes in Software Engineering, Alghero, Sardinia, Italy, 67–70. (2002)

[Atl 16] The Eclipse foundation, The Eclipse ATL website, https://eclipse.org/atl/.

[Bac 00] Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical Concepts of Component-Based Software Engineering. Software Engineering Institute. (May 2000)

[Bal 98] Balter, R., Bellissard, F., Boyer, F., Riveill, M., Vion-Dury, J.: Architecturing and Configuring Distributed Applications with Olan. In Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, UK, 241-256. (1998)

[Bar 05] Barbier, F.: UML 2 et MDE - Ingénierie des modèles avec études de cas. Dunod. (2005)

[Bar 08] Barais, O., Le Meur, A. F., Duchien, L. Lawall, J.: Software Architecture Evolution. In: Mens, T., Demeyer, S. (eds.): Software Evolution, Springer Verlag, 233-262. (2008)

[Bel 89] Belina, F., Hogrefe, D.: The CCITT-Specification and Description Language SDL. Computer Networks and ISDN Systems, Vol. 6, No. 4, 311–341. (March 1989)

[Beu 99] Beugnard, A., Jézéquel, J. M., Plouzeau, N., Watkins, D.: Making components contract aware. Computer, Vol. 32, No.7, 38–45. (July 1999)

[Béz 04] Bézvin, J: In Search of a Basic Principle for Model Driven Engineering. CEPIS, UP-GRADE, The European Journal for the Informatics Professional, Vol. 2, 21–24. (2004)

[Bie 10] Biehl, M.: Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology. (July 2010)

[Bla 05] Blanc, X.: MDA en Action : Ingénierie Logicielle Guidée par les Modèles. Ed. Eyrolles. (2005)

[Bol 01] Bolton, F.: Pure Corba. SAMS Publishing. (2001)

[Boo 05] Booch, G., Rumbaugh, J., Jackobson, I.: The Unified Modeling Language User Guide. 2nd Edition, Addison-Wesley Professional. (2005)

[Bos 00] Bosch, J.: Design and Use of Software Architectures - Adopting and Evolving a Product-Line Approach. Addison-Wesley. (2000)

[Bou 06] El Boussaidi, G., Mili, H.: Les Langages de Description d'Architectures. LATECE Technical Report, Montreal, Canada. (2006)

[Bra 08] Bravenboer, M., Kalleberg, K. T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Science of Computer Programming, Vol. 72, No. 1-2, 52-70. (2008)

[Bru 06] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stéfani, J. B.: The fractal component model and its support in java. Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems. (2006)

[Bru 14] Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a Model Driven Reverse Engineering Framework. Information and Software Technology, Elsevier, Vol. 56, No. 8, 1012-1032. (2014)

[Bus 96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons Ltd., Chichester, UK. (1996)

[Che 01] Cheesman, J., Daniels, J.: UML Components - A Simple Process for Specifying Component-Based Software. Addison-Wesley. (2001)

[Cle 96] Clements, P. C.: A survey of architecture description languages. In Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD), 16-16. (1996)

[Coo 04] Cooper, D., Khoo, B., von Konsky, B. R., Robey, M.: Java Implementation Verification Using Reverse Engineering. In Proceeding of the 27th Australasian Computer Science Conference - Volume 26 (ACSC '04), Australian Computer Society, Inc., Darlinghurst, Australia, Australia, Vol. 26, 203-212. (2004)

[Cre 00] Crespo, Y.: Incremento del potencial de reutilización del software me diante refactorizaciones. PhD thesis, Universidad de Valladolid. (2000)

[Cwm 03] Object Management Group (OMG), Common Warehouse Metamodel (CWM) Specification, Version 1.1, http://www.omg.org/spec/CWM/1.1/PDF. (March 2003)

[Cza 06] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal, Vol. 45, No. 3, 621-645. (2006)

[Dem 01] Demeyer, S., Tichelaar, S., Ducasse, S.: FAMIX 2.1 – The FAMOOS Information Exchange Model. University of Berne. (2001)

[Dia 09] Diaw, S., Lbath, R., Coulette, B.: État de l'art sur le développement logiciel basé sur les transformations de modèles. Université de Toulouse, TSI – 29. (2009)

[Dis 04] Dissaux, P.: Using the AADL for mission critical software development. In proceeding of the 2nd European *Congress* Erts (Embedded Real Time Software Toulouse), Toulouse. (2004)

[Duc 11] Ducasse, S., Anquetil, N., Bhatti, U., Hora, A. C., Laval, J.: MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. INRIA, Technical Report. (2011)

[Ecl 15] The Eclipse foundation, http://www.eclipse.org/ org/. (2015)

[Ejb 04] Object Management Group (OMG), Unified Metamodel and UML Profile for Java and EJB Specification Version 1.0, OMG Document Number formal/2004-02-02. (2004)

[Eic 09] Eichelberger, H., Eldogan, Y., Schmid, K.: A Comprehensive Survey of UML Compliance in Current Modelling Tools. SE 2009, LNI 143, Kaiserslautern, 39-50. (2009)

[Fei 05] Feiler, P. H., Greenhouse, A.: OSATE Plug-in Development Guide. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA. (2005)

[Gar 00] Garlan, D., Monroe, R., Wile, D.: Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, Leavens Gary et Sitaraman Murali (réd.) Cambridge University Press, 47-68. (2000)

[Gar 93] Garlan, D., Shaw, M.: An introduction to software architecture. In Advances in Software Engineering and Knowledge Engineering, Vol. 2, 1–39, Hackensack, NJ, USA: World Scientific Publishing Company. (1993)

[Gar 95] Garlan, D.: An Introduction to the Aesop System. Carnegie Mellon University, Pittsburgh, http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps. (1995)

[Gau 05] Gaufillet, P., Farail, P.: Topcased - un environnement de développement open source pour les systèmes embarqué, Technical report, Airbus France. (2005)

[Ghe 02] Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of software engineering. Second Edition, Prentice-Hall. (2002)

[Gor 03] Gorp, V. P., Stenten, H., Mens, T., Demeyer, S.: Towards Automating Source-consistent UML Refactorings**.** In Proceeding of the 6th International Conference on «UML» – The Unified Modeling Language, Springer,144-158. (2003)

[Gor 94] Gorlick, M., Quilici, A.: Visual Programming in the Large versus Visual Programming in the Small, In Proceeding of the IEEE Symposium on Visual Languages, St. Louis, Missouri, 137-144. (1994)

[Gos 14] Goslin, J., Joy, B., Steele, G. L.: The Java Language Specification. Java SE 8 Edition (Java Series), Addison-Wesley Professional. (2014)

[Gra 06] Graiet, M., Bhiri, M. T., Damak, F., Giraudin, J. P.: Adaptation d'UML2.0 à l'ADL Wright. Actes du 1ère Conférence francophone sur les Architectures Logicielles (CAL), Nantes, France. (2006)

[Gud 01] Gudgin, M.: Essential IDL: Interface Design for COM. Addison-Wesley. (2001)

[Ham 12] Hammami, M.: Maintenance de l'outil Wr2fdr de traduction de Wright vers CSP. Computing Research Repository (CoRR), Vol. abs/1208.0044. (2012)

[Ham 15] Hamioud, S., Atil, F.: Model Driven Java Code Refactoring. Computer Sci ence and Information Systems (ComSIS), Vol. 12, No. 2, 375-403. (2015)

[Har 00] Harrison, W., Barton, C., Raghavachari, M.: Mapping UML Designs to Java. SIGPLAN Not., Vol. 35, No. 10, 178-187. (2000)

[Hei 01] Heinemann, G. T., Councill, W. T.: Component-based software engineering. Addison-Wesley. (2001)

[Hei 09] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: EMFText and JaMoPP - Tool Presentation. Institut fur Software und Multimediatechnik Technische Universit, Dresden, Germany. (2009)

[Hei 10] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In Proceeding of the 2nd International Conference on Software Language Engineering (SLE'09), LNCS, Springer, Vol. 5969, 374-383. (2010)

[Hoa 85] Hoare, C. A. E.: Communicating Sequential Processes. Prentice Hall. (1985)

[Hof 08] Hoffman, B., Pérez, J., Mens, T.: A Case Study for Program Refactoring. In Proceeding of the 4th International Workshop on Graph-Based Tools (GraBaTs'08). (2008)

[Hol 00] Holt, R., Hassan, A. E., Laguë, B., Lapierre, S., LeDuc, C.: E/R Schema for the Datrix C/C++/Java Exchange Format. In Proceeding of the Seventh Working Conference on Reverse Engineering (WCRE'00), IEEE Computer Society, Washington, DC, USA, 284-286. (2000)

[Hub 08] Huber, P.: The model transformation language jungle - an evaluation and extension of existing approaches. Master's thesis, Technische Universitat Wien. (May 2008)

[Iee 00] Software Engineering Standards Committee of the IEEE Computer Society, IEEE Recommended practice for architecture description of software-intensive systems, IEEE Std 1471-2000, IEEE-SA Standards Board, http://standards.ieee.org/. (September 2000)

[Jou 06] Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In Proceeding of the 8th IFIP WG 6.1 International Conference (FMOODS 2006), LNCS, Springer, Vol. 4037, 171–185. (2006)

[Kat 05] Katayama, T., Yabuya, Y.: Proposal of a Method to Support Testing for Java Programs with UML. In Proceeding of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), IEEE Computer Society, Washington, DC, USA, 533-540. (2005)

[Ker 14] Triskell Team, Kermeta website, http://www.kermeta.org/. (2014)

[Kes 04] Keschenau, M.: Reverse Engineering of UML Specifications from Java Programs. In Companion to the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04), ACM, New York, NY, USA, 326-327. (2004)

[Lea 00] Leavens, G. T., Rustan, K., Leino, M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In OOPSLA 2000 Companion, Minneapolis, Minnesota, The Association for Computing Machinery, 105–106. (october 2000)

[Led 99] LeDuc, C., Laguë, B.: Datrix Abstract Symentic Graph Reference Manual. Bell Canada. (1999)

[Lap 01] Lapierre, S., Laguë, B., Leduc, C.: Datrix vM Source Code Model and its Interchange Format: Lessons Learned and Considerations for Future Work**.** ACM SIGSOFT Software Engineering Notes, Vol. 26, No. 1, 53-56. (2001)

[Let 04] Lethbridge, T. C., Tichelaar, S., Ploedereder, E.: The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. Electronic Notes in Theoretical Computer Science, Vol. 94, 7-18. (2004)

[Luc 95] Luckham, D. C., Kenney, J. J., Augustin, Vera, J., Bryan, D., Mann, W.: Specification and Analysis of Software Architecture using Rapide. IEEE Transactions on Software Engineering, Vol. 21, No. 4, 336-355. (1995)

[Lue 02] Luer, C., van der Hoek, A.: Composition environments for deployable software components. Technical Report, Department of Information and Computer Science, University of California. (2002)

[Mag 96] Magee, J., Karmer, J.: Dynamic Structure in Software Architecture. In Proceedings of ACM SIGSOFT'96 : Fourth Symposium Foundation of Software Engineering, San Francisco, CA, 3-14. (1996)

[Mag 99] Magee, J.: Dynamic Behavioral Analysis of Software Architectures Using LTSA. In Proceedings of Int'l Conf. on Software Engineering (ICSE), 634-637. (1999)

[Mar 02] Marvie, R., Pellegrini, M. C.: Modèles de composants, un état de l'art. RSTI-L'objet, No. 8, 61-89. (2002)

[Mch 94] McHale, C.: Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance. PhD thesis, Trinity College, University of Dublin. (1994)

[Mcl 68] McIlroy, M. D.: Mass produced software components. In Proceedings of the NATO Conference on Software Engineering, 138–155, Garmish, Germany. (October 1968)

[Mcq 06] McQuillan, J. A., Power, J. F.: Experiences of Using the Dagstuhl Middle Metamodel for Defining Software Metrics. In Proceeding of the 4th international symposium on Principles and practice of programming in Java (PPPJ '06), ACM, New York, NY, USA, 194-198. (2006)

[Med 00] Medvidovic, N., Taylor, R., N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, Vol. 26, No. 1, 70-93. (January 2000)

[Med 96] Medvidovic, N., Taylor, R. N., Whitehead, Jr. E. J.: Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In Proceedings of the California Software Symposium, Los Angeles, California, 28-40. (April 1996)

[Med 97] Medvidovic, N., Rosenblum, D. S.: Domains of Concern in Software Architectures. In Proceedings of the USENIX Conference on Domain Specific Languages, Santa Barbara, California, 199-212. (1997)

[Men 06] Mens, T., Gorp, P., V.: Applying a model transformation taxonomy to graph transformation technology. In Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), Vol. 152, 143-159. (2006)

[Mes 06] Meslati, D.: Mage : Une approche ontogénétique de l'évolution dans les systèmes logiciels critiques et embarqués. Thèse de doctorat d'état, Université de Annaba. (Février 2006)

[Mey 00] Meyer, B.: Conception et programmation orientées objet. Eyrolles. (juillet 2000)

[Mey 92] Meyer, B.: Eiffel : the language. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. (1992)

[Mia 16] MIA-Software. MIA-Transformation and MIA-Generation website. http://www. mia-software.com. (2016)

[Mic 96] Microsoft Developer Network Library, Common Object Model Specification, Microsoft Corporation. (1996)

[Mod 15] The Eclipse foundation, The Eclipse MoDisco website, https://eclipse.org/ MoDisco/. (2015)

[Mof 14] Object Management Group (OMG), Meta Object Facility (MOF) Core Specification, Version 2.4.2, http://www.omg.org/spec/MOF/2.4.2/PDF. (April 2014)

[Mon 99] Monroe, R. T.: Rapid Development of Custom Software Architecture Design Environments. PhD thesis, Carnegie Mellon University. (1999)

[Mor 95] Moriconi, M., Qian, X., Riemenschneider, R. A.: Correct Architecture Refinement. IEEE Transactions on Software Engineering (TSE), Vol. 21, No. 4, 356-372. (April 1995)

[Ng 95] Ng, K., Karmer, J., Magee, J., Dulay, N.: The Software Architect's Assistant - A Visual Environment for Distributed Programming. IEEE Computer Society, 254-263. (1995)

[Ocl 12] Object Management Group (OMG), Object Constraint Language (OCL), Version 2.3.1, http://www.omg.org/ spec/OCL/2.3.1/PDF. (January 2012)

[Omg 11a] Object Management Group (OMG), Architecture-Driven Modernization: Abstract Syntax Tree Metamodel (ASTM), Version 1.0, OMG Document Number formal/2011-01-05. (2011)

[Omg 11b] Object Management Group (OMG), Architecture-Driven Modernization: Knowledge Discovery Meta-model (KDM), Version 1.3, OMG Document Number formal/2011-08-04. (2011)

[Omg 16] Object Management Group (OMG), OMG Model Driven Architecture, http://www.omg.org/mda/. (2016)

[Omg 91] Object Management Group (OMG), The Common Object Request Broker: Architecture and Specification, Document Number 91.12.1. (December 1991)

[Omm 02] van Ommering, R., Linder, F., Kramer, J., Magee, J.: The koala component model for Consumer Electronics Software. IEEE Computer, Vol. 33, No. 3, 78-85. (2002)

[Ous 05] Oussalah, M.: Ingénierie des composants : concepts, techniques et outils. Editions Vuibert Informatique, Paris. (2005)

[Paw 14] Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon v2: Large Scale Source Code Analysis and Transformation for Java. INRIA, Technical Report, hal-01078532. (2014)

[Par 11] Parada, A. G., Siegert, E., de Brisolara, L.B.: Generating Java Code from UML Class and Sequence Diagrams. Computing System Engineering (SBESC), 2011 Brazilian Symposium on, 99-101. (2011)

[Per 92] Perry, D. E., Wolf, A. L.: Foundations for the Study of Software Architectures. SIGSOFT Software Engineering Notes, Vol. 17, No. 4, 40-52. (October 1992)

[Qvt 11] Object Management Group (OMG), Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, http://www.omg.org/ spec/QVT/1.1/PDF. (January 2011)

[Sha 95] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., Zelesnik, G.: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, Vol. 21, No. 4, 314-335. (April 1995)

[Sha 96] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Upper Saddle River, N.J. (1996)

[Şor 13] Şora, L.: Unified Modeling of Static Relationships between Program Elements. In Proceeding of the 7th International Conference, ENASE 2012, Communications in Computer and Information Science, Vol. 410, 95-109. (2013)

[Ste 08] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. 2ed Edition, Wesley Professional. (2008)

[Sun 09] Sun Microsystems, JSR-000318 Enterprise JavaBeans(tm) Specification, Version 3.1, www.oracle.com. (February 2009)

[Sun 97] Sun Microsystems, JavaBeans Specification, Version 1.01, http://java.sun.com/beans. (August 1997)

[Szy 98] Szyperski, C.: Component Software Beyond Object-Oriented Programming. Addison-Wesley. (1998)

[Veg 05] Vega, D.: Développement d'Applications à Grande Échelle par Composition de Méta-Modèles. Thèse de doctorat, Université de Grenoble. (décembre 2005)

[Uml 06] Object Management Group (OMG), Unified Modeling Language: infrastructure Version 2.0, OMG Document Number formal/05-07-05. (2006)

[Vee 14] Veeramani, A., Venkatesan, K., Nalinadevi, K.: Abstract Syntax Tree based Unified Modeling Language to Object Oriented Code Conversion. In Proceeding of the 2014 International Conference on Interdisciplinary Advances in Applied Computing (ICONIAAC '14), ACM, New York, NY, USA, 25:1-25:8. (2014)

[Tic 01] Tichelaar, S.: Modeling Object-oriented Software for Reverse Engineering and Refactoring. PhD thesis, Universität Bern. (2001)

[Ves 93] Vestal, S.: A Cursory Overview and Comparison of Four Architecture Description Languages. Technical Report, Honeywell Technology Center, Minneapolis. (1993)

[Ves 96] Vestal, S.: MetaH Programmer's Manual 1.09. Technical Report, Honeywell Technology Center, Minneapolis. (1996)

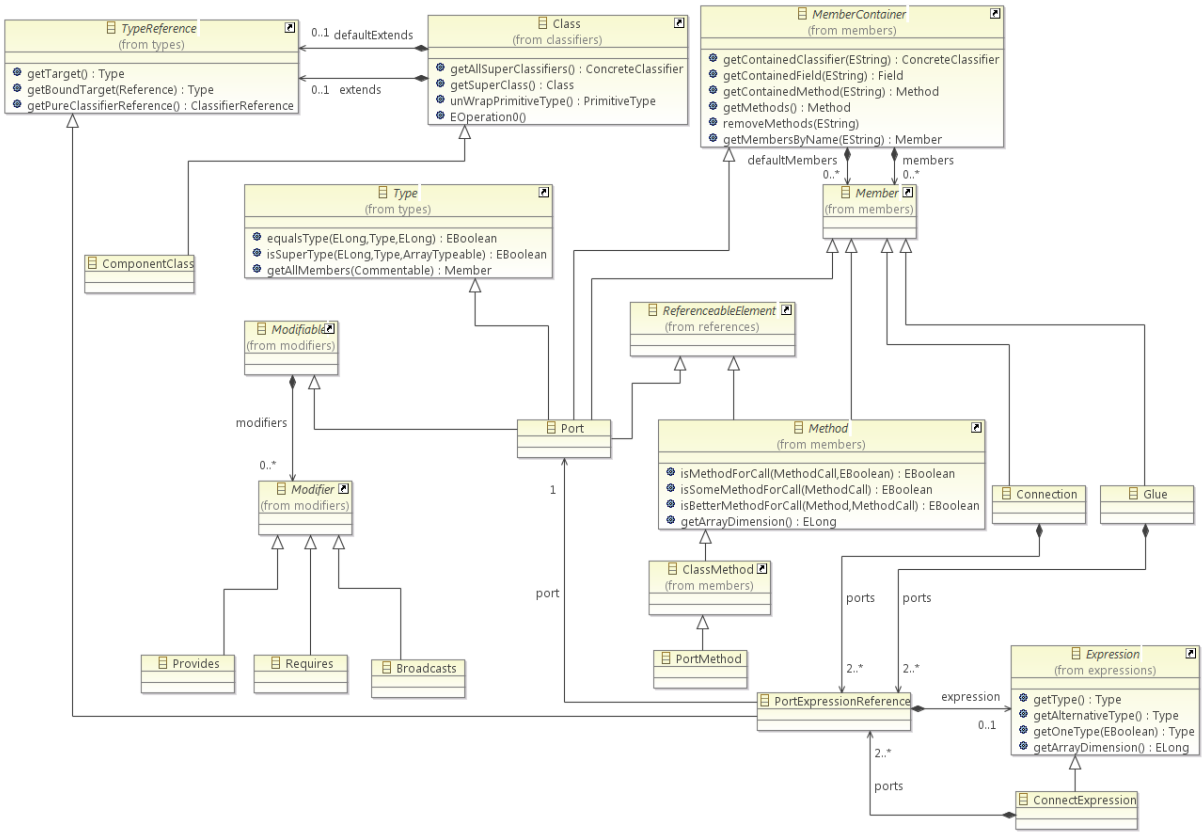
[Vis 97] Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam. (1997)

[Xmi 15] Object Management Group (OMG), XML Metadata Interchange (XMI) Specification, Version 2.5.1, http://www.omg.org/spec/XMI/2.5.1/PDF. (June 2015)

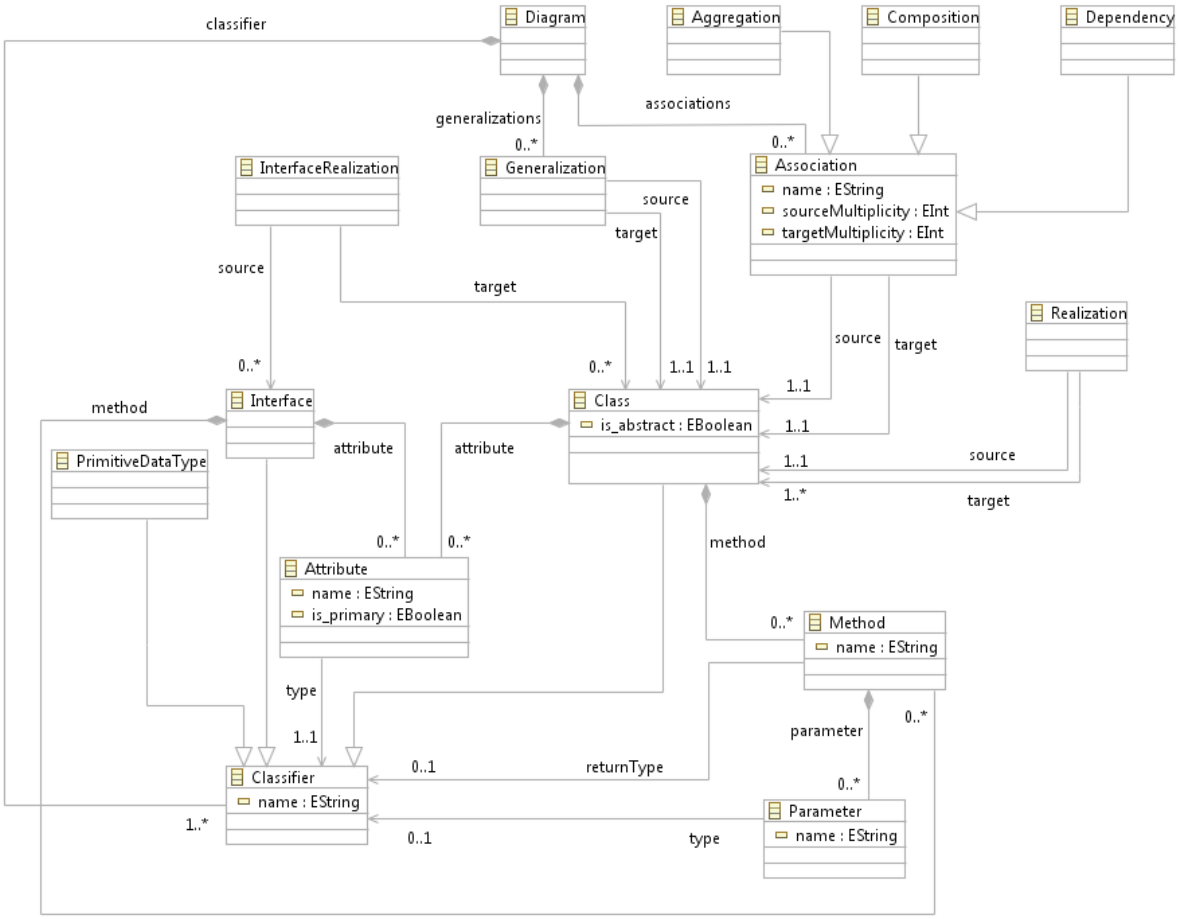
Annexe A : Liste de métamodèles

Pour des raisons de lisibilité, les métamodèles de grandes tailles sont présentés sans leurs métaréférences.

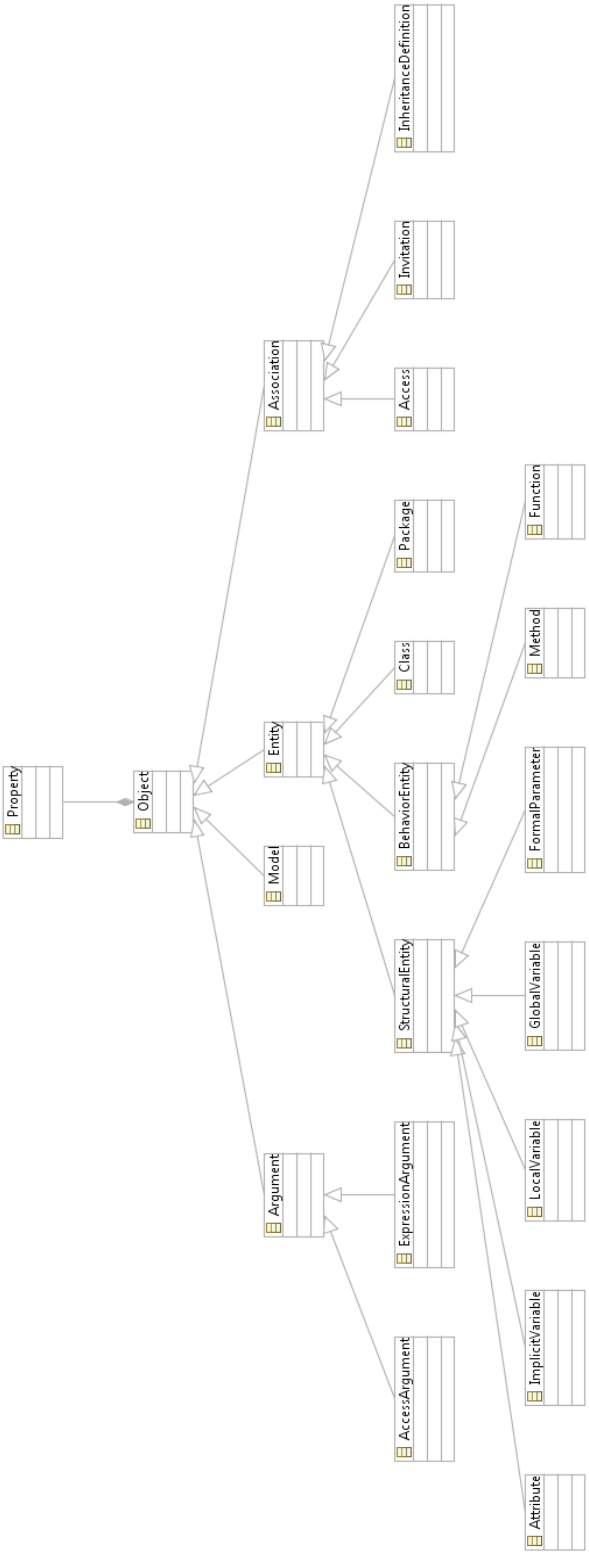
A.1. Métamodèle ArchJava



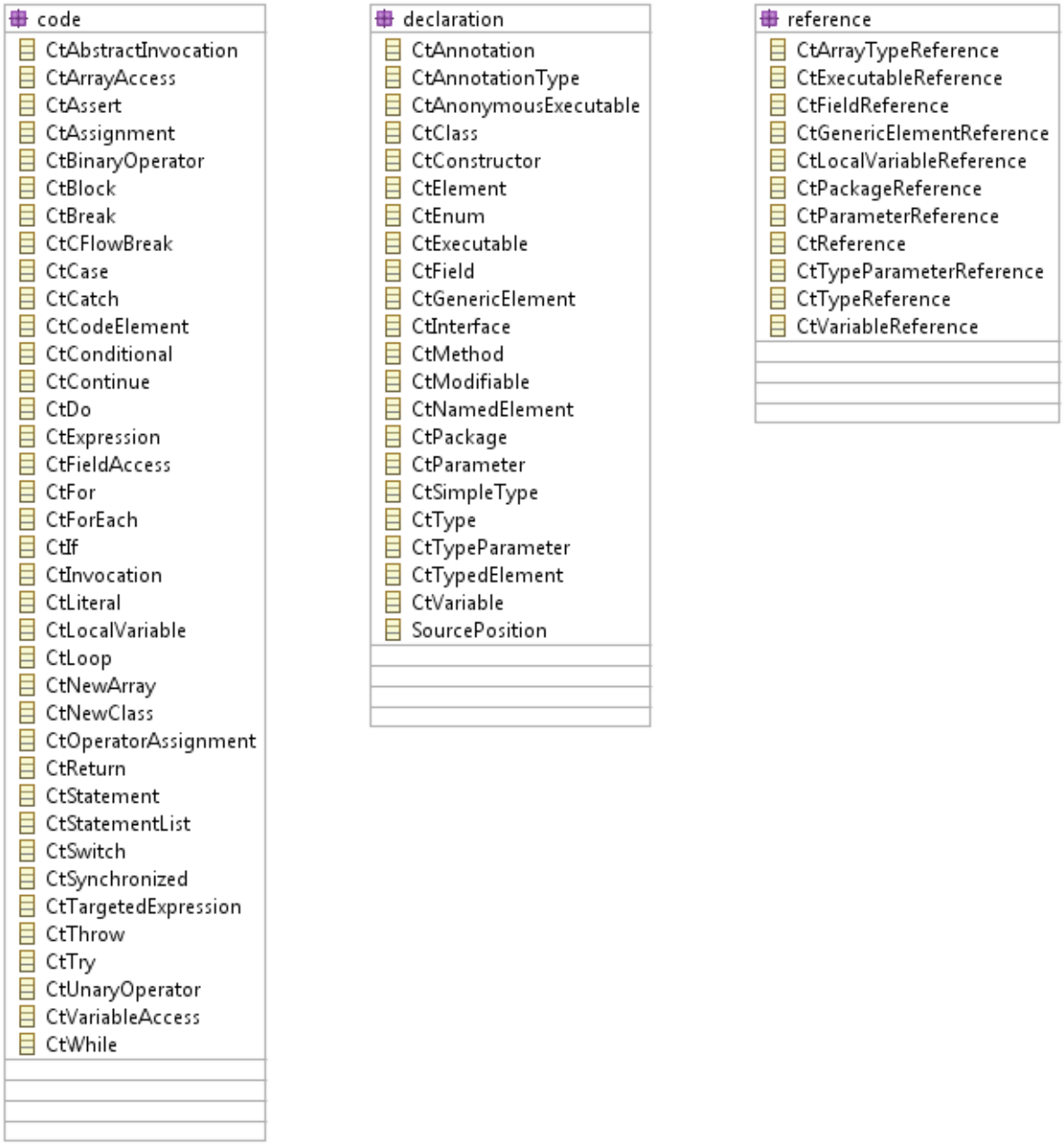
A.2. Diagramme de classes UML



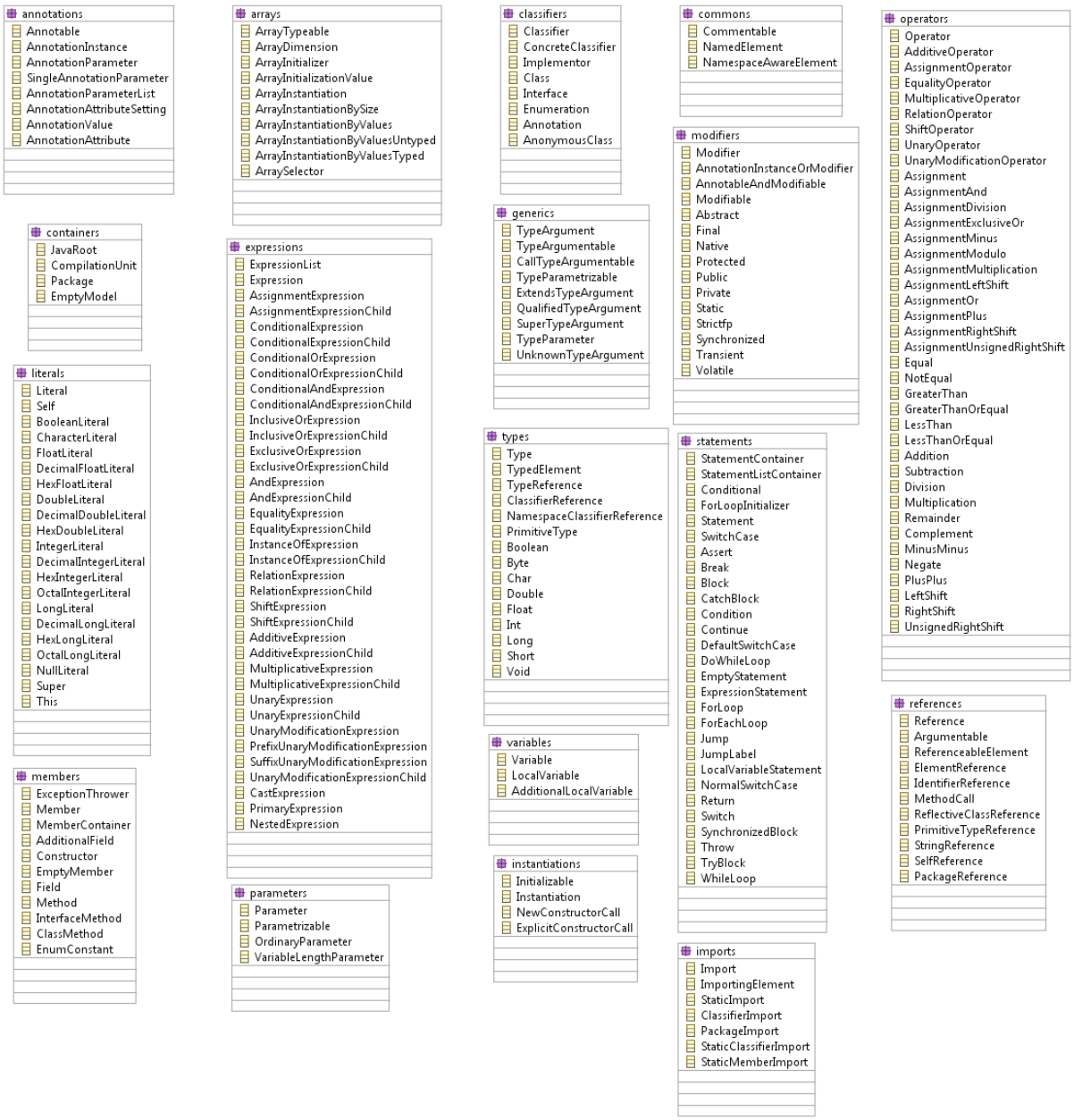
A.3. Métamodèle Famix 2.1



A.4. Métamodèle Spoon

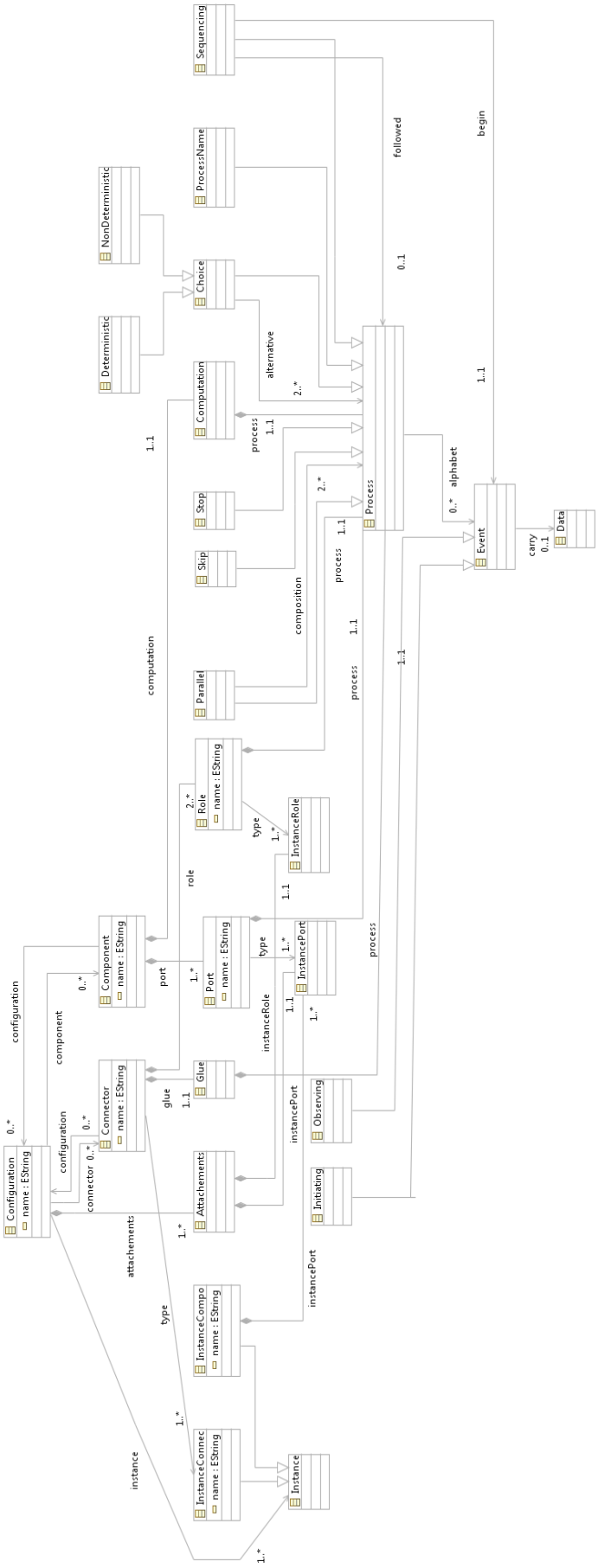


A.5. Métamodèle JaMoPP

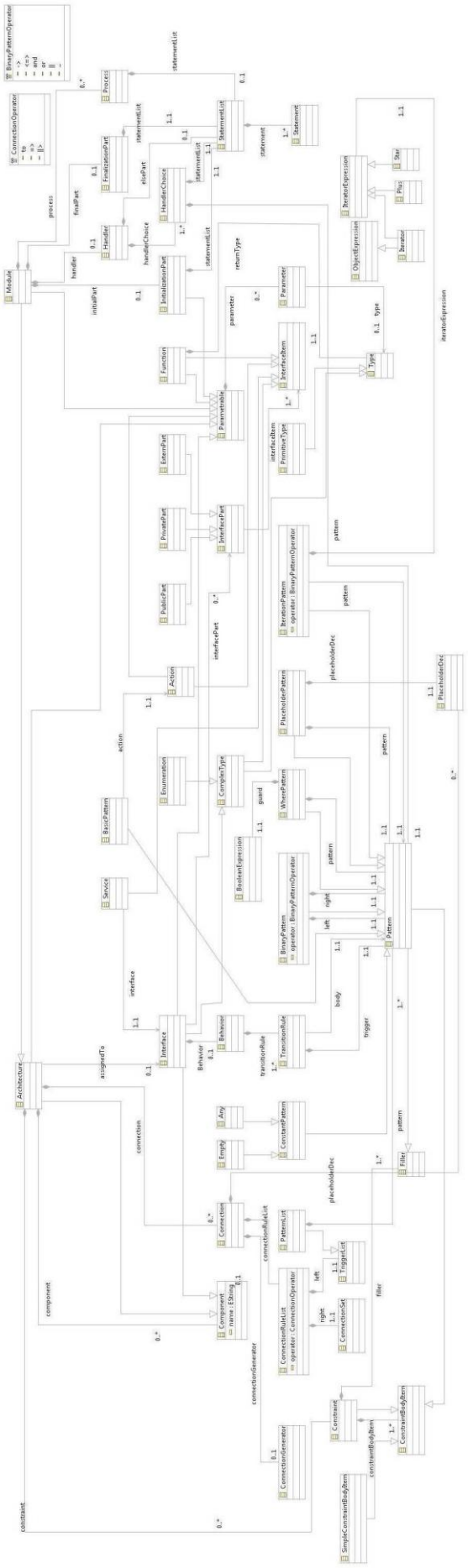


A.6. Métamodèle KDM

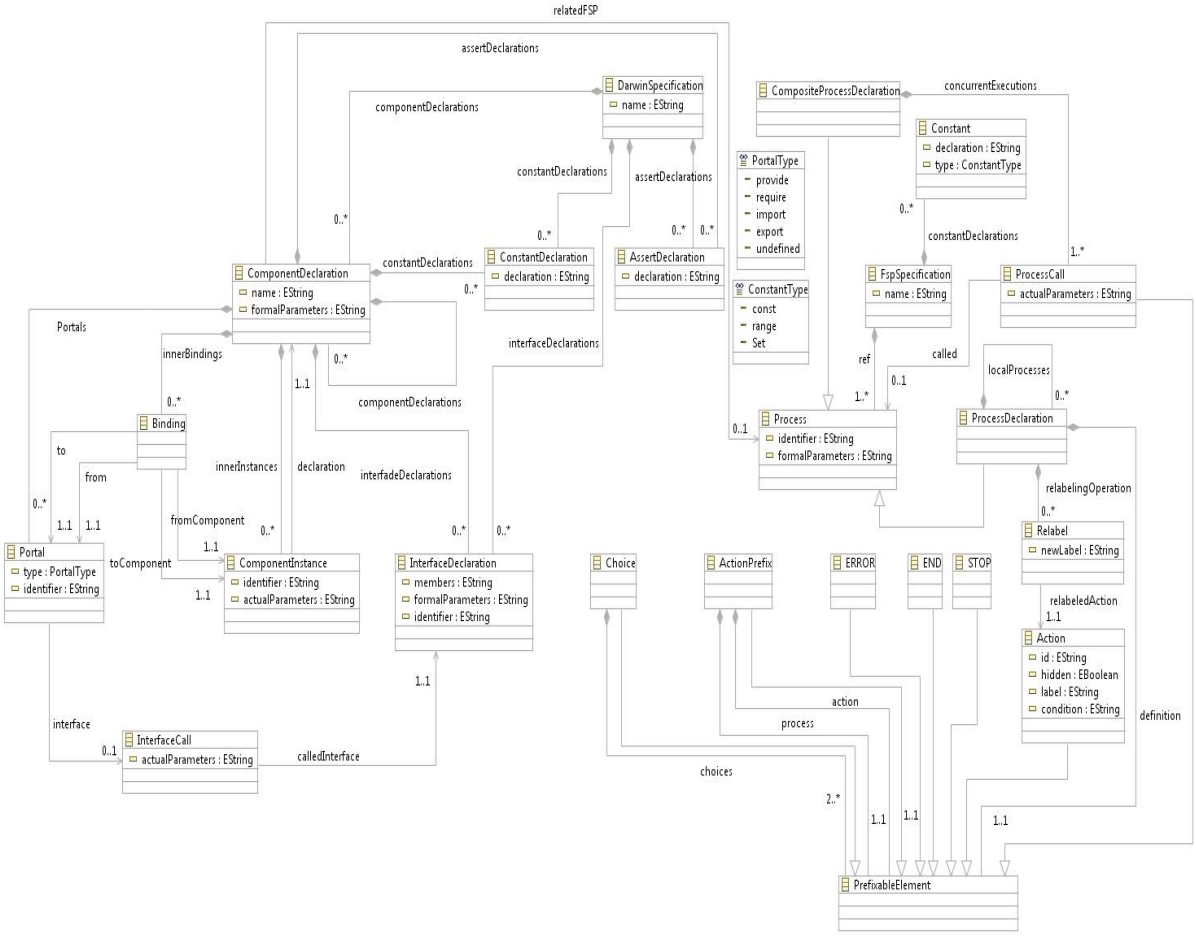
A.7. Métamodèle Wright



A.8. Métamodèle Rapide



A.9. Métamodèle Darwin/FSP



Annexe B : Liste des listings

B.1. Analyse d'une description Acme en Java

```
package dz.univ-annaba.acme;

import java.io.File;
import java.io.IOException;
import org.acmestudio.standalone.environment.StandaloneEnvironment;
import org.acmestudio.standalone.environment.StandaloneEnvironment.TypeCheckerType;
import org.acmestudio.standalone.resource.StandaloneResourceProvider;
import org.acmestudio.acme.core.resource.IAcmeResource;
import org.acmestudio.acme.core.resource.ParsingFailureException;
import org.acmestudio.acme.model.IAcmeModel;
import org.acmestudio.basicmodel.model.AcmeModel;

public class AcmeParser {

        public void parse(String filePath) throws ParsingFailureException, IOException{
           System.out.println ("Reading from " + filePath + " . . .");

        // Parse the Acme file

           StandaloneEnvironment.instance().useTypeChecker(TypeCheckerType.SYNCHRONOUS);
           System.out.println ("Parsing . . .");

        // Get Acme resource

           IAcmeResource resource = null;
           try {
                 resource =
                  StandaloneResourceProvider.instance().acmeResourceForString(filePath);
           } catch (IOException e) {
                System.out.println ("There was an error getting the resource.");
               System.exit (1);
           }
           StandaloneResourceProvider.instance().releaseResource(resource);
      }

      // Get Acme model from Acme Resource

      public AcmeModel getModel(IAcmeResource resource) {
           AcmeModel model = new AcmeModel(resource);
           return model;
      }
      public IAcmeModel getIAcmeModel(IAcmeResource resource) throws Exception {
           IAcmeModel model = resource.getModel();
           return model;
      }
}
```

B.2. Crible d'Ératosthène en ArchJava

```
/** Implements the Sieve of Eratosthenes as an ArchJava architecture */

component class PrimeFilter {
    private int num;
    private Out out;

    public PrimeFilter (int n) {
       num = n;
    }
    public port report {
       requires void report(int i);
    }
    public port in {
        void process(int i) {
           if (num != 0 && (i % num) == 0)
              return;
           else if (out != null)
              out.process(i);
           else {
              report.report(i);
              out = new Out(i);
           }
       }
    }
    public port interface Out {
       requires connect(int i);
       requires void process(int i);
    }
}
public component class Sieve {
    private int numPrimes;
    private port /*interface*/ Report {
        void report(int i) {
            System.out.println(Integer.toString(i));
            numPrimes--;
       }
    }
    private port pipe {
       requires void process(int i);
    }
    private final PrimeFilter first = new PrimeFilter(0);
    connect pipe, first.in;
    connect Report, first.report;
    connect pattern Report, PrimeFilter.report;
    connect pattern PrimeFilter.Out, PrimeFilter.in {
       connect(PrimeFilter sender, int prime) {
           PrimeFilter newFilter = new PrimeFilter(prime);
           connect(Report, newFilter.report);
           return connect(sender.Out, newFilter.in);
       }
    };
    public void run(int n) {
        numPrimes = n;
       int next = 2;
       while (numPrimes > 0)
           pipe.process(next++);
    }

    public static void main(String args[]) {
        if (args.length != 1) {
            System.out.println("usage: java Sieve <numprimes>");
            return;
        }
        new Sieve().run(Integer.parseInt(args[0]));
    }
}
```

B.3. ArchJaMoPPC.java

```
package org.emftext.language.archjava.archjamoppc;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;
import org.eclipse.emf.common.notify.Notifier;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.InternalEObject;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.eclipse.emf.ecore.xmi.XMLResource;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
import org.emftext.language.java.JavaClasspath;
import org.emftext.language.java.JavaPackage;
import org.emftext.language.java.resource.JavaSourceOrClassFileResourceFactoryImpl;
import org.emftext.language.archjava.ArchjavaPackage;
import org.emftext.language.archjava.resource.archj.mopp.ArchjResourceFactory;
public class ArchJaMoPPC {
protected static final ResourceSet rs = new ResourceSetImpl();
public static void main(String[] args) throws IOException {
      if (args.length < 2) {
             printUsageAndExit();
      }
      setUp();
      File srcFolder = new File(args[0]);
      if (!srcFolder.exists()) {
             System.out.println("not found: " + args[0]);
             return;
      }
      JavaClasspath cp = JavaClasspath.get(rs);
```

```
	// register jar files
	for (int i = 2; i < args.length; i++) {
		File jarPath = new File(args[i]);
		if (!jarPath.exists()) {
			System.out.println("not found: " + args[i]);
			return;
		}
		System.out.println("Registering JAR " + jarPath.getCanonicalPath());
		cp.registerClassifierJar(URI.createFileURI(jarPath.getCanonicalPath()));
	}
	// load all archjava files into resource set
	loadAllFilesInResourceSet(srcFolder, ".archj", ".java");
	if (!resolveAllProxies()) {
		System.err.println("Resolution of some Proxies failed...");
		Iterator<Notifier> it = rs.getAllContents();
		while (it.hasNext()) {
			Notifier next = it.next();
			if (next instanceof EObject) {
				EObject o = (EObject) next;
				if (o.eIsProxy()) {
					try {
						it.remove();
					} catch (UnsupportedOperationException e) {
						e.printStackTrace();
					}
				}
			}
		}
	}
	URI srcUri = URI.createFileURI(srcFolder.getCanonicalPath());
	String outFileOrDir = args[1];
	if (outFileOrDir.endsWith(".xmi")) {
		saveToSingleXMIFile(srcUri, new File(outFileOrDir));
	} else {
		saveToFolder(srcUri, new File(outFileOrDir));
	}
}
private static void saveToSingleXMIFile(URI srcUri, File file)throws IOException {
	File parentDir = file.getParentFile();
	if (parentDir == null) {
		parentDir = new File(System.getProperty("user.dir"));
	} else if (!parentDir.exists()) {
		parentDir.mkdirs();
```

```
        }
        URI outUri = URI.createFileURI(file.getCanonicalPath());
        Resource xmiResource = rs.createResource(outUri);
        for (Resource archjavaResource : new ArrayList<Resource>(rs.getResources())) {
                xmiResource.getContents().addAll(archjavaResource.getContents());
        }
        // save the metamodels (schemas) for dynamic loading
        serializeMetamodel(parentDir);
        saveXmiResource(xmiResource);
}
private static void saveToFolder(URI srcUri, File outFolder)throws IOException {
        List<Resource> result = new ArrayList<Resource>();
        URI outUri = URI.createFileURI(outFolder.getCanonicalPath());
        for (Resource archjavaResource : new ArrayList<Resource>(rs.getResources())) {
                URI srcURI = archjavaResource.getURI();
                srcURI = rs.getURIConverter().normalize(srcURI);
                URI outFileURI = outUri.appendSegments(
                                        srcURI.deresolve(srcUri.appendSegment("")).segments())
                                        .appendFileExtension("xmi");
                Resource xmiResource = rs.createResource(outFileURI);
                xmiResource.getContents().addAll(archjavaResource.getContents());
                result.add(xmiResource);
        }
        // save the metamodels (schemas) for dynamic loading
        serializeMetamodel(outFolder);
        for (Resource xmiResource : result) {
                saveXmiResource(xmiResource);
        }
}
private static void saveXmiResource(Resource xmiResource)throws IOException {
        Map<Object, Object> options = new HashMap<Object, Object>();
        options.put(XMLResource.OPTION_SCHEMA_LOCATION, Boolean.TRUE);
        Set<EObject> danglingElements = new LinkedHashSet<EObject>();
        for (Iterator<EObject> i = xmiResource.getAllContents(); i.hasNext();) {
                for (EObject crossRefElement : i.next().eCrossReferences()) {
                        if ((crossRefElement.eContainer() == null)
                                        && (crossRefElement.eResource() == null)) {
                                danglingElements.add(crossRefElement);
                        }
                }
        }
        xmiResource.getContents().addAll(danglingElements);
        xmiResource.save(options);
```

```
}
private static void printUsageAndExit() {
	System.out.println("ARCHJaMoPPC Usage:");
	System.out.println("==============");
	System.out.println();
	System.out
	.println("To parse all files in a source folder and produce one model file per\n"
	+ "parsed compilation unit in the target folder, use:");
	System.out.println();
	System.out
	.println("  archjamopp <source folder path> <target folder path> <jar file paths>*");
	System.out.println();
	System.out
	.println("To parse all files in a source folder and produce one XMI file\n"
	+ "with the complete syntax graph, use:");
	System.out.println();
	System.out
	.println("  archjamoppc <source folder path> <target XMI file> <jar file paths>*");
	System.out.println();
	System.out.println("In the latter case, the second parameter has to end in \".xmi\".");
	System.exit(1);
}
protected static void setUp() {
	EPackage.Registry.INSTANCE.put("http://www.emftext.org/java",JavaPackage.eINSTANCE);
	Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
			"java", new JavaSourceOrClassFileResourceFactoryImpl());
	Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
			"archj", new ArchjResourceFactory());
	Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
			Resource.Factory.Registry.DEFAULT_EXTENSION,
				new XMIResourceFactoryImpl());
}
protected static void serializeMetamodel(File outFolder) throws IOException {
	URI outUri = URI.createFileURI(outFolder.getCanonicalPath());
	URI archjavaEcoreURI = outUri.appendSegment("archjava.ecore");
	Resource archjavaEcoreResource = rs.createResource(archjavaEcoreURI);
	archjavaEcoreResource.getContents().addAll(ArchjavaPackage.eINSTANCE.getESubpackages());
	archjavaEcoreResource.save(null);
}
protected static void loadAllFilesInResourceSet(File startFolder,String extension1, String extension2)
throws IOException {
	for (File member : startFolder.listFiles()) {
		if (member.isFile()) {
```

```
                        if (member.getName().endsWith(extension1) || member.getName().endsWith(extension2)) {
                                System.out.println("Parsing " + member);
                                parseResource(member);
                        } else {
                                System.out.println("Skipping " + member);
                        }
                }
                if (member.isDirectory()) {
                        if (!member.getName().startsWith(".")) {
                                System.out.println("Recursing into " + member);
                                loadAllFilesInResourceSet(member, extension1, extension2);
                        } else {
                                System.out.println("Skipping " + member);
                        }
                }
        }
}
protected static boolean resolveAllProxies() {
        boolean failure = false;
        List<EObject> eobjects = new LinkedList<EObject>();
        for (Iterator<Notifier> i = rs.getAllContents(); i.hasNext();) {
                        Notifier next = i.next();
                if (next instanceof EObject) {
                        eobjects.add((EObject) next);
                }
        }
        System.out.println("Resolving cross-references of " + eobjects.size()
                        + " EObjects.");
        int resolved = 0;
        int notResolved = 0;
        int eobjectCnt = 0;
        for (EObject next : eobjects) {
                eobjectCnt++;
                if (eobjectCnt % 1000 == 0) {
                        System.out.println(eobjectCnt + "/" + eobjects.size()
                                        + " done: Resolved " + resolved + " crossrefs, "
                                        + notResolved + " crossrefs could not be resolved.");
                }
                InternalEObject nextElement = (InternalEObject) next;
                for (EObject crElement : nextElement.eCrossReferences()) {
                        crElement = EcoreUtil.resolve(crElement, rs);
                        if (crElement.eIsProxy()) {
                                failure = true;
```

```
                                notResolved++;
                                System.out
                                        .println("Can not find referenced element in classpath: "
                                                + ((InternalEObject) crElement).eProxyURI());
                        } else {
                                resolved++;
                        }
                }
        }
        System.out.println(eobjectCnt + "/" + eobjects.size()
                        + " done: Resolved " + resolved + " crossrefs, " + notResolved
                        + " crossrefs could not be resolved.");
        return !failure;
  }
  protected static void parseResource(File file) throws IOException {
        loadResource(file.getCanonicalPath());
  }
  protected static void parseResource(ZipFile file, ZipEntry entry)
                throws IOException {
        loadResource(URI.createURI("archive:file:///"
                        + new File(".").getAbsoluteFile().toURI().getRawPath()
                        + file.getName().replaceAll("\\\\", "/") + "!/"
                        + entry.getName()));
  }
  protected static void loadResource(String filePath) throws IOException {
        loadResource(URI.createFileURI(filePath));
  }
  protected static void loadResource(URI uri) throws IOException {
        rs.getResource(uri, true);
  }
}
```

B.4. ArchjMinimalModelHelper.java

```
package org.emftext.language.archjava.resource.archj.util;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EAttribute;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.EClassifier;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EPackage;
import org.eclipse.emf.ecore.EReference;
import org.eclipse.emf.ecore.EStructuralFeature;
import org.eclipse.emf.ecore.InternalEObject;
/**
 * A helper class that is able to create minimal model instances for Ecore models.
 */
public class ArchjMinimalModelHelper {
	private final static org.emftext.language.archjava.resource.archj.util.ArchjEClassUtil eClassUtil =
new org.emftext.language.archjava.resource.archj.util.ArchjEClassUtil();
	public EObject getMinimalModel(EClass eClass, Collection<EClass> allAvailableClasses) {
		return getMinimalModel(eClass, allAvailableClasses.toArray(new
				EClass[allAvailableClasses.size()]), null);
	}
	public EObject getMinimalModel(EClass eClass, EClass[] allAvailableClasses) {
		return getMinimalModel(eClass, allAvailableClasses, null);
	}
	public EObject getMinimalModel(EClass eClass, EClass[] allAvailableClasses, String name) {
		if (!contains(allAvailableClasses, eClass)) {
			return null;
		}
		EPackage ePackage = eClass.getEPackage();
		if (ePackage == null) {
			return null;
		}
		EObject root = ePackage.getEFactoryInstance().create(eClass);
		List<EStructuralFeature> features = eClass.getEAllStructuralFeatures();
		for (EStructuralFeature feature : features) {
			if (feature instanceof EReference) {
				EReference reference = (EReference) feature;
				if (reference.isUnsettable()) {
```

```
			continue;
		}
		if (!reference.isChangeable()) {
			continue;
		}
		EClassifier type = reference.getEType();
		if (type instanceof EClass) {
			EClass typeClass = (EClass) type;
			if (eClassUtil.isNotConcrete(typeClass)) {
				// find subclasses
				List<EClass> subClasses = eClassUtil.getSubClasses(typeClass,
									allAvailableClasses);
				if (subClasses.size() == 0) {
					continue;
				} else {
					// pick the first subclass
					typeClass = subClasses.get(0);
				}
			}
			int lowerBound = reference.getLowerBound();
			for (int i = 0; i < lowerBound; i++) {
				EObject subModel = null;
				if (reference.isContainment()) {
					EClass[] unusedClasses =
							getArraySubset(allAvailableClasses, eClass);
					subModel = getMinimalModel(typeClass, unusedClasses);
				}
				else {
					subModel =
				typeClass.getEPackage().getEFactoryInstance().create(typeClass);
				// set some proxy URI to make this object a proxy
					String initialValue = "#some" +
org.emftext.language.archjava.resource.archj.util.ArchjStringUtil.capitalize(typeClass.getName());
					URI proxyURI = URI.createURI(initialValue);
			((InternalEObject) subModel).eSetProxyURI(proxyURI);
				}
				if (subModel == null) {
					continue;
				}
				Object value = root.eGet(reference);
				if (value instanceof List<?>) {
					List<EObject> list =
	org.emftext.language.archjava.resource.archj.util.ArchjListUtil.castListUnchecked(value);
```

```
                                        list.add(subModel);
                                } else {
                                        root.eSet(reference, subModel);
                                }
                        }
                }
        } else if (feature instanceof EAttribute) {
                EAttribute attribute = (EAttribute) feature;
                if ("EString".equals(attribute.getEType().getName())) {
                        String initialValue;
                        if (attribute.getName().equals("name") && name != null) {
                                initialValue = name;
                        }
                        else {
                                initialValue = "some" +
 org.emftext.language.archjava.resource.archj.util.ArchjStringUtil.capitalize(attribute.getName());
                        }
                        Object value = root.eGet(attribute);
                        if (value instanceof List<?>) {
                                List<String> list =
          org.emftext.language.archjava.resource.archj.util.ArchjListUtil.castListUnchecked(value);
                                list.add(initialValue);
                        } else {
                                root.eSet(attribute, initialValue);
                        }
                }
        }
        return root;
}
private boolean contains(EClass[] allAvailableClasses, EClass eClass) {
        for (EClass nextClass : allAvailableClasses) {
                if (eClass == nextClass) {
                        return true;
                }
        }
        return false;
}
private EClass[] getArraySubset(EClass[] allClasses, EClass eClassToRemove) {
        List<EClass> subset = new ArrayList<EClass>();
        for (EClass eClass : allClasses) {
                if (eClass != eClassToRemove) {
                        subset.add(eClass);
```

```
                }
            }
            return subset.toArray(new EClass[subset.size()]);
        }
    }
```

B.5. ArchjNewFileWizardPage.java

```
package org.emftext.language.archjava.resource.archj.ui;
import org.eclipse.core.resources.IContainer;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.IAdaptable;
import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.Path;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.ModifyEvent;
import org.eclipse.swt.events.ModifyListener;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.dialogs.ContainerSelectionDialog;
/**
 * The NewFileWizardPage allows setting the container for the new file, as well as
 * the file name. The page will only accept file names without extension OR with
 * an extension that matches the expected one.
 */
public class ArchjNewFileWizardPage extends WizardPage {
      private final String fileExtension;
      private Text containerText;
      private Text fileText;
      private ISelection selection;
      /**
       * Constructor for the NewFileWizardPage.
       */
      public ArchjNewFileWizardPage(ISelection selection, String fileExtension) {
             super("wizardPage");
      setTitle(org.emftext.language.archjava.resource.archj.ui.ArchjUIResourceBundle.NEW_FILE_WIZARD_PAGE_
TITLE);
      setDescription(org.emftext.language.archjava.resource.archj.ui.ArchjUIResourceBundle.NEW_FILE_WIZARD
_DESCRIPTION);
```

```
        this.selection = selection;
        this.fileExtension = fileExtension;
}
/**
 *
 * @see IDialogPage#createControl(Composite)
 */
public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        GridLayout layout = new GridLayout();
        container.setLayout(layout);
        layout.numColumns = 3;
        layout.verticalSpacing = 9;
        Label label = new Label(container, SWT.NULL);
        label.setText("&Container:");
        containerText = new Text(container, SWT.BORDER | SWT.SINGLE);
        GridData gd = new GridData(GridData.FILL_HORIZONTAL);
        containerText.setLayoutData(gd);
        containerText.addModifyListener(new ModifyListener() {
                public void modifyText(ModifyEvent e) {
                        dialogChanged();
                }
        });
        Button button = new Button(container, SWT.PUSH);
        button.setText("Browse...");
        button.addSelectionListener(new SelectionAdapter() {
                public void widgetSelected(SelectionEvent e) {
                        handleBrowse();
                }
        });
        label = new Label(container, SWT.NULL);
        label.setText("&File name:");
        fileText = new Text(container, SWT.BORDER | SWT.SINGLE);
        gd = new GridData(GridData.FILL_HORIZONTAL);
        fileText.setLayoutData(gd);
        fileText.addModifyListener(new ModifyListener() {
                public void modifyText(ModifyEvent e) {
                        dialogChanged();
                }
        });
        initialize();
        dialogChanged();
        setControl(container);
```

```
}
/**
 * Tests if the current workbench selection is a suitable container to use.
 */
private void initialize() {
        String name =
   org.emftext.language.archjava.resource.archj.ui.ArchjUIResourceBundle.NEW_FILE_WIZARD_FILE_NAME;
        if (selection != null && selection.isEmpty() == false && selection instanceof
                                                                        IStructuredSelection) {
                IStructuredSelection ssel = (IStructuredSelection) selection;
                if (ssel.size() > 1) {
                        return;
                }
                Object obj = ssel.getFirstElement();
                // test for IAdaptable
                if ((! (obj instanceof IResource)) && (obj instanceof IAdaptable)) {
                        obj = (IResource) ((IAdaptable) obj).getAdapter(IResource.class);
                }
                if (obj instanceof IResource) {
                        IContainer container;
                        if (obj instanceof IContainer) {
                                container = (IContainer) obj;
                        } else {
                                IResource resource = (IResource) obj;
                                container = resource.getParent();
                // we use the name of the currently selected file instead of 'new_file'.
                                name = resource.getFullPath().removeFileExtension().lastSegment();
                                name = name + "." + fileExtension;
                        }
                        IPath fullPath = container.getFullPath();
                        containerText.setText(fullPath.toString());
                }
        }
        // Select default name for new file
        fileText.setText(name);
        // Select file name without extension
        int indexOfDot = name.lastIndexOf(".");
        if (indexOfDot > 0) {
                fileText.setSelection(0, indexOfDot);
        }
}
public void setVisible(boolean visible) {
        super.setVisible(visible);
```

```
        if (visible) {
                fileText.setFocus();
        }
}
/**
 * Uses the standard container selection dialog to choose the new value for the
 * container field.
 */
private void handleBrowse() {
        ContainerSelectionDialog dialog = new ContainerSelectionDialog(getShell(),
                        ResourcesPlugin.getWorkspace().getRoot(), false, "Select new file container");
        if (dialog.open() == ContainerSelectionDialog.OK) {
                Object[] result = dialog.getResult();
                if (result.length == 1) {
                        containerText.setText(((Path) result[0]).toString());
                }
        }
}
/**
 * Ensures that both text fields are set.
 */
private void dialogChanged() {
        IResource container = ResourcesPlugin.getWorkspace().getRoot().findMember(new
                                                                        Path(getContainerName()));
        String fileName = getFileName();
        if (getContainerName().length() == 0) {
                updateStatus("File container must be specified");
                return;
        }
        if (container== null ||(container.getType() & (IResource.PROJECT | IResource.FOLDER))== 0) {
                updateStatus("File container must exist");
                return;
        }
        if (!container.isAccessible()) {
                updateStatus("Project must be writable");
                return;
        }
        if (fileName.length() == 0) {
                updateStatus("File name must be specified");
                return;
        }
        if (fileName.replace('\\', '/').indexOf('/', 1) > 0) {
                updateStatus("File name must be valid");
```

```
                return;
            }
            if (!fileName.endsWith("." + fileExtension)) {
                updateStatus("File extension must be \"" + fileExtension + "\"");
                return;
            }
            updateStatus(null);
        }
        private void updateStatus(String message) {
            setErrorMessage(message);
            setPageComplete(message == null);
        }
        public String getContainerName() {
            return containerText.getText();
        }
        public String getFileName() {
            return fileText.getText();
        }
        public String getPackageName() {
            return packageText.getText();
        }
}
```

B.6. ArchjNewFileWizard.java

```
package org.emftext.language.archjava.resource.archj.ui;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.InvocationTargetException;
import org.eclipse.core.resources.IContainer;
import org.eclipse.core.resources.IFile;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.resources.IWorkspaceRoot;
import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Path;
import org.eclipse.core.runtime.Status;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.operation.IRunnableWithProgress;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.wizard.Wizard;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.MessageBox;
import org.eclipse.ui.INewWizard;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPage;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.ide.IDE;
public class ArchjNewFileWizard extends Wizard implements INewWizard {
	protected String categoryId = null;
	protected org.emftext.language.archjava.resource.archj.ui.ArchjNewFileWizardPage page;
	protected ISelection selection;
	protected String newName = null;
	protected String newContainerName;
	public ArchjNewFileWizard() {
		super();
		setNeedsProgressMonitor(true);
setWindowTitle(org.emftext.language.archjava.resource.archj.ui.ArchjUIResourceBundle.NEW_FILE_WIZARD_WIND
OW_TITLE);
	}
	public String getCategoryId() {
```

```
		return categoryId;
	}
	public void setCategoryId(String id) {
		categoryId = id;
	}
	/**
	 * Adds the pages to the wizard.
	 */
	public void addPages() {
		page = new org.emftext.language.archjava.resource.archj.ui.ArchjNewFileWizardPage(selection,
			getFileExtension());
		addPage(page);
	}
	/**
	 * This method is called when 'Finish' button is pressed in the wizard. We will
	 * create an operation and run it using the wizard as execution context.
	 */
	public boolean performFinish() {
		final String containerName = page.getContainerName();
		newContainerName= containerName;
		int seperatorIdx= newContainerName.lastIndexOf('/');
		newContainerName = containerName.substring(seperatorIdx+1);
		final String fileName = page.getFileName();
		this.newName = fileName;
		seperatorIdx = newName.indexOf('.');
		if (seperatorIdx != -1) {
			newName = newName.substring(0, seperatorIdx);
		}
		final IFile file;
		try {
			file = getFile(fileName, containerName);
		} catch (CoreException e1) {
 org.emftext.language.archjava.resource.archj.ui.ArchjUIPlugin.logError("Exception while initializing new
file", e1);
			return false;
		}
		if (file.exists()) {
			// ask for confirmation
			MessageBox messageBox = new MessageBox(getShell(), SWT.ICON_QUESTION
					| SWT.YES | SWT.NO);
			messageBox.setMessage("File \"" + fileName + "\" already exists. Do you want to
				override it?");
			messageBox.setText("File exists");
			int response = messageBox.open();
```

```
                        if (response == SWT.NO) {
                                return true;
                        }
                }
                IRunnableWithProgress op = new IRunnableWithProgress() {
                        public void run(IProgressMonitor monitor) throws InvocationTargetException {
                                try {
                                        doFinish(containerName, fileName, monitor);
                                } catch (CoreException e) {
                                        throw new InvocationTargetException(e);
                                } finally {
                                        monitor.done();
                                }
                        }
                };
                try {
                        getContainer().run(true, false, op);
                } catch (InterruptedException e) {
                        return false;
                } catch (InvocationTargetException e) {
                        Throwable realException = e.getTargetException();
                        MessageDialog.openError(getShell(), "Error", realException.getMessage());
 org.emftext.language.archjava.resource.archj.ui.ArchjUIPlugin.logError("Exception while initializing new
 file", e);
                        return false;
                }
                return true;
        }
        /**
         * The worker method. It will find the container, create the file if missing or
         * just replace its contents, and open the editor on the newly created file.
         */
        protected void doFinish(String containerName, String fileName, IProgressMonitor monitor) throws
CoreException {
                // create a sample file
                monitor.beginTask("Creating " + fileName, 2);
                final IFile file = getFile(fileName, containerName);
                try {
                        InputStream stream = openContentStream();
                        if (file.exists()) {
                                file.setContents(stream, true, true, monitor);
                        } else {
                                file.create(stream, true, monitor);
```

```
                }
                stream.close();
            } catch (IOException e) {
            }
            monitor.worked(1);
            monitor.setTaskName("Opening file for editing...");
            getShell().getDisplay().asyncExec(new Runnable() {
                public void run() {
                    IWorkbenchPage page =
                            PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();
                    try {
                            IDE.openEditor(page, file, true);
                    } catch (PartInitException e) {
                    }
                }
            });
            monitor.worked(1);
        }
        protected IFile getFile(String fileName, String containerName) throws CoreException {
            IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
            IResource resource = root.findMember(new Path(containerName));
            if (!resource.exists() || !(resource instanceof IContainer)) {
                throwCoreException("Container \"" + containerName + "\" does not exist.");
            }
            IContainer container = (IContainer) resource;
            final IFile file = container.getFile(new Path(fileName));
            return file;
        }
        /**
         * Initializes file contents with a sample text.
         */
        protected InputStream openContentStream() {
            return new ByteArrayInputStream(new
org.emftext.language.archjava.resource.archj.mopp.ArchjMetaInformation().getNewFileContentProvider().getNew
FileContent(newName,newContainerName).getBytes());
        }
        protected void throwCoreException(String message) throws CoreException {
            IStatus status = new Status(IStatus.ERROR, "NewFileContentPrinter", IStatus.OK, message,
null);
            throw new CoreException(status);
        }
      /**
        * <p>
        * We will accept the selection in the workbench to see if we can initialize from
```

```
	 * it.
	 * </p>
	 * @see IWorkbenchWizard#init(IWorkbench, IStructuredSelection)
	 */
	public void init(IWorkbench workbench, IStructuredSelection selection) {
		this.selection = selection;
	}
	public String getFileExtension() {
		return new
		   org.emftext.language.archjava.resource.archj.mopp.ArchjMetaInformation().getSyntaxName();
	}
	public org.emftext.language.archjava.resource.archj.IArchjMetaInformation getMetaInformation() {
		return new org.emftext.language.archjava.resource.archj.mopp.ArchjMetaInformation();
	}
}
```

B.7. Worker.archj.xmi

```
<?xml version="1.0" encoding="ASCII"?>
<containers:CompilationUnit>
  <namespaces>connector</namespaces>
  <classifiers xsi:type="archjava:ComponentClass" name="Worker">
    <members xsi:type="archjava:Port" name="barrier">
      <modifiers xsi:type="modifiers:Public"/>
      <portMethods name="start">
        <typeReference xsi:type="types:Void"/>
        <annotationsAndModifiers xsi:type="archjava:Requires"/>
      </portMethods>
      <portMethods name="middle">
        <typeReference xsi:type="types:Void"/>
        <annotationsAndModifiers xsi:type="archjava:Requires"/>
      </portMethods>
      <portMethods name="end">
        <typeReference xsi:type="types:Void"/>
        <annotationsAndModifiers xsi:type="archjava:Requires"/>
      </portMethods>
    </members>
    <members xsi:type="members:ClassMethod" name="run">
      <typeReference xsi:type="types:Void"/>
      <annotationsAndModifiers xsi:type="modifiers:Public"/>
      <statements xsi:type="statements:ExpressionStatement">
        <expression xsi:type="references:IdentifierReference">
          <next xsi:type="references:IdentifierReference">
            <next xsi:type="references:MethodCall">
              <target xsi:type="members:ClassMethod"
href="java/io/PrintStream.class.xmi#//@classifiers.0/@members.43"/>
              <arguments xsi:type="references:StringReference" value="At first barrier"/>
            </next>
            <target xsi:type="members:Field"
href="java/lang/System.class.xmi#//@classifiers.0/@members.1"/>
          </next>
          <target xsi:type="classifiers:Class" href="java/lang/System.class.xmi#//@classifiers.0"/>
        </expression>
      </statements>
      <statements xsi:type="statements:ExpressionStatement">
        <expression xsi:type="references:IdentifierReference">
          <next xsi:type="references:MethodCall">
            <target xsi:type="archjava:PortMethod" href="//@classifiers.0/@members.0/@portMethods.0"/>
          </next>
```

```
            <target xsi:type="archjava:Port" href="//@classifiers.0/@members.0"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:IdentifierReference">
              <next xsi:type="references:MethodCall">
                <target xsi:type="members:ClassMethod"
href="java/io/PrintStream.class.xmi#//@classifiers.0/@members.43"/>
                <arguments xsi:type="references:StringReference" value="After first barrier"/>
              </next>
              <target xsi:type="members:Field"
href="java/lang/System.class.xmi#//@classifiers.0/@members.1"/>
            </next>
            <target xsi:type="classifiers:Class" href="java/lang/System.class.xmi#//@classifiers.0"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:MethodCall" target="//@classifiers.0/@members.1">
            <arguments xsi:type="literals:DecimalIntegerLiteral" decimalValue="35"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:IdentifierReference">
              <next xsi:type="references:MethodCall">
                <target xsi:type="members:ClassMethod"
href="java/io/PrintStream.class.xmi#//@classifiers.0/@members.43"/>
                <arguments xsi:type="references:StringReference" value="At second barrier"/>
              </next>
              <target xsi:type="members:Field"
href="java/lang/System.class.xmi#//@classifiers.0/@members.1"/>
            </next>
            <target xsi:type="classifiers:Class" href="java/lang/System.class.xmi#//@classifiers.0"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:MethodCall">
              <target xsi:type="archjava:PortMethod" href="//@classifiers.0/@members.0/@portMethods.1"/>
            </next>
            <target xsi:type="archjava:Port" href="//@classifiers.0/@members.0"/>
          </expression>
        </statements>
```

```
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:IdentifierReference">
              <next xsi:type="references:MethodCall">
                <target xsi:type="members:ClassMethod"
href="java/io/PrintStream.class.xmi#//@classifiers.0/@members.43"/>
                <arguments xsi:type="references:StringReference" value="After second barrier"/>
              </next>
              <target xsi:type="members:Field"
href="java/lang/System.class.xmi#//@classifiers.0/@members.1"/>
            </next>
            <target xsi:type="classifiers:Class" href="java/lang/System.class.xmi#//@classifiers.0"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:MethodCall" target="//@classifiers.0/@members.1">
            <arguments xsi:type="literals:DecimalIntegerLiteral" decimalValue="35"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:IdentifierReference">
              <next xsi:type="references:MethodCall">
                <target xsi:type="members:ClassMethod"
href="java/io/PrintStream.class.xmi#//@classifiers.0/@members.43"/>
                <arguments xsi:type="references:StringReference" value="At third barrier"/>
              </next>
              <target xsi:type="members:Field"
href="java/lang/System.class.xmi#//@classifiers.0/@members.1"/>
            </next>
            <target xsi:type="classifiers:Class" href="java/lang/System.class.xmi#//@classifiers.0"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:MethodCall">
              <target xsi:type="archjava:PortMethod" href="//@classifiers.0/@members.0/@portMethods.2"/>
            </next>
            <target xsi:type="archjava:Port" href="//@classifiers.0/@members.0"/>
          </expression>
        </statements>
        <statements xsi:type="statements:ExpressionStatement">
          <expression xsi:type="references:IdentifierReference">
            <next xsi:type="references:IdentifierReference">
```

```
              <next xsi:type="references:MethodCall">
                <target xsi:type="members:ClassMethod"
href="java/io/PrintStream.class.xmi#//@classifiers.0/@members.43"/>
                <arguments xsi:type="references:StringReference" value="After third barrier"/>
              </next>
              <target xsi:type="members:Field"
href="java/lang/System.class.xmi#//@classifiers.0/@members.1"/>
            </next>
            <target xsi:type="classifiers:Class" href="java/lang/System.class.xmi#//@classifiers.0"/>
          </expression>
        </statements>
      </members>
      <members xsi:type="members:ClassMethod" name="fib">
        <typeReference xsi:type="types:Void"/>
        <parameters xsi:type="parameters:OrdinaryParameter" name="i">
          <typeReference xsi:type="types:Int"/>
        </parameters>
        <annotationsAndModifiers xsi:type="modifiers:Public"/>
        <statements xsi:type="statements:Condition">
          <statement xsi:type="statements:Return">
            <returnValue xsi:type="literals:DecimalIntegerLiteral" decimalValue="1"/>
          </statement>
          <condition xsi:type="expressions:RelationExpression">
            <children xsi:type="references:IdentifierReference"
target="//@classifiers.0/@members.1/@parameters.0"/>
            <children xsi:type="literals:DecimalIntegerLiteral" decimalValue="2"/>
            <relationOperators xsi:type="operators:LessThan"/>
          </condition>
          <elseStatement xsi:type="statements:Return">
            <returnValue xsi:type="expressions:AdditiveExpression">
              <children xsi:type="references:MethodCall" target="//@classifiers.0/@members.1">
                <arguments xsi:type="expressions:AdditiveExpression">
                  <children xsi:type="references:IdentifierReference"
target="//@classifiers.0/@members.1/@parameters.0"/>
                  <children xsi:type="literals:DecimalIntegerLiteral" decimalValue="1"/>
                  <additiveOperators xsi:type="operators:Subtraction"/>
                </arguments>
              </children>
              <children xsi:type="references:MethodCall" target="//@classifiers.0/@members.1">
                <arguments xsi:type="expressions:AdditiveExpression">
                  <children xsi:type="references:IdentifierReference"
target="//@classifiers.0/@members.1/@parameters.0"/>
                  <children xsi:type="literals:DecimalIntegerLiteral" decimalValue="2"/>
                  <additiveOperators xsi:type="operators:Subtraction"/>
                </arguments>
```

```
            </children>
            <additiveOperators xsi:type="operators:Addition"/>
          </returnValue>
        </elseStatement>
      </statements>
    </members>
    <annotationsAndModifiers xsi:type="modifiers:Public"/>
    <extends xsi:type="types:NamespaceClassifierReference">
      <classifierReferences>
        <target xsi:type="classifiers:Class" href="java/lang/Thread.class.xmi#//@classifiers.0"/>
      </classifierReferences>
    </extends>
  </classifiers>
</containers:CompilationUnit>
```

B.8. Extrait de ArchjDefaultDelgateResolver.java

```
package org.emftext.language.archjava.resource.archj.analysis;
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.eclipse.emf.common.notify.Adapter;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EReference;
import org.eclipse.emf.ecore.EStructuralFeature.Setting;
import org.eclipse.emf.ecore.InternalEObject;
import org.eclipse.emf.ecore.plugin.EcorePlugin;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.util.EcoreUtil;
public class ArchjDefaultResolverDelegate<ContainerType extends EObject, ReferenceType extends EObject> {
      ......
      ......
      /**
       * This standard implementation searches for objects in the resource, which have
       * the correct type and a name/id attribute matching the identifier. If no
       * matching object is found, the identifier is used as URI. If the resource at
       * this URI has a root element of the correct type, this element is returned.
       */
      public void resolve(String identifier, ContainerType container, EReference reference, int position,
boolean resolveFuzzy,
       org.emftext.language.archjava.resource.archj.IArchjReferenceResolveResult<ReferenceType> result) {
            try {
                  EObject root = container;
                  if (!enableScoping) {
                        root = EcoreUtil.getRootContainer(container);
                  }
                  while (root != null) {
                        boolean continueSearch = tryToResolveIdentifierInObjectTree(identifier,
                        container, root, reference, position, resolveFuzzy, result, !enableScoping);
                        if (!continueSearch) {
                              return;
```

```
                        }
                        root = root.eContainer();
                    }
                    boolean continueSearch = tryToResolveIdentifierAsURI(identifier, container,
                                                reference, position, resolveFuzzy, result);
                    if (continueSearch) {
                        Set<EObject> crossReferencedObjectsInOtherResource =
                                                                findReferencedExternalObjects(container);
                        for (EObject externalObject : crossReferencedObjectsInOtherResource) {
                            continueSearch = tryToResolveIdentifierInObjectTree(identifier,
                    container, externalObject, reference, position, resolveFuzzy, result, !enableScoping);
                            if (!continueSearch) {
                                return;
                            }
                        }
                    }
                    if (continueSearch) {
                        continueSearch = tryToResolveIdentifierInGenModelRegistry(identifier,
                                                container, reference, position, resolveFuzzy, result);
                    }
                } catch (RuntimeException rte) {
                // catch exception here to prevent EMF proxy resolution from swallowing it
                    rte.printStackTrace();
                }
            }
            ......
            ......
            /**
             * Searches for objects in the tree of EObjects that is rooted at
             * <code>root</code>, which have the correct type and a name/id attribute matching
             * the identifier. This method can be used to quickly implement custom reference
             * resolvers which require to search in a particular scope for referenced
             * elements, rather than in the whole resource as done by resolve().
             */
            protected boolean tryToResolveIdentifierInObjectTree(String identifier, EObject container, EObject
root, EReference reference, int position, boolean resolveFuzzy,
org.emftext.language.archjava.resource.archj.IArchjReferenceResolveResult<ReferenceType> result, boolean
checkRootFirst) {
                EClass type = reference.getEReferenceType();
                boolean continueSearch;
                if (checkRootFirst) {
                    // check whether the root element matches
                    continueSearch = checkElement(container, root, reference, position, type, identifier,
                                        resolveFuzzy, true, result);
```

```
			if (!continueSearch) {
				return false;
			}
		}
		// check the contents
		for (Iterator<EObject> iterator = root.eAllContents(); iterator.hasNext(); ) {
			EObject element = iterator.next();
			continueSearch = checkElement(container, element, reference, position, type,
					identifier, resolveFuzzy, true, result);
			if (!continueSearch) {
				return false;
			}
		}
		// if the root element was already checked, we can return.
		if (checkRootFirst) {
			return true;
		}
		// check whether the root element matches
		continueSearch = checkElement(container, root, reference, position, type, identifier,
				resolveFuzzy, true, result);
		if (!continueSearch) {
			return false;
		}
		return true;
	}
	protected boolean tryToResolveIdentifierAsURI(String identifier, ContainerType container, EReference
reference, int position, boolean resolveFuzzy,
org.emftext.language.archjava.resource.archj.IArchjReferenceResolveResult<ReferenceType> result) {
		EClass type = reference.getEReferenceType();
		Resource resource = container.eResource();
		if (resource != null) {
			URI uri = getURI(identifier, resource.getURI());
			if (uri != null) {
				EObject element = loadResource(container.eResource().getResourceSet(), uri);
				if (element == null) {
					return true;
				}
				return checkElement(container, element, reference, position, type,
						identifier, resolveFuzzy, false, result);
			}
		}
		return true;
	}
```

```
	protected boolean tryToResolveIdentifierInGenModelRegistry(String identifier, ContainerType
container, org.eclipse.emf.ecore.EReference reference, int position, boolean resolveFuzzy,
org.emftext.language.archjava.resource.archj.IArchjReferenceResolveResult<ReferenceType> result) {
		EClass type = reference.getEReferenceType();
		final Map<String, URI> packageNsURIToGenModelLocationMap =
							EcorePlugin.getEPackageNsURIToGenModelLocationMap();
		for (String nextNS : packageNsURIToGenModelLocationMap.keySet()) {
			URI genModelURI = packageNsURIToGenModelLocationMap.get(nextNS);
			try {
				final ResourceSet rs = container.eResource().getResourceSet();
				Resource genModelResource = rs.getResource(genModelURI, true);
				if (genModelResource == null) {
					continue;
				}
				final List<EObject> contents = genModelResource.getContents();
				if (contents == null || contents.size() == 0) {
					continue;
				}
				EObject genModel = contents.get(0);
				boolean continueSearch = checkElement(container, genModel, reference,
						position, type, identifier, resolveFuzzy, false, result);
				if (!continueSearch) {
					return false;
				}
			} catch (Exception e) {
				// ignore exceptions that are raised by faulty genmodel registrations
			}
		}
		return true;
	}
	......
	......
}
```

B.9. resolve() de ClassifierReferenceTargetReferenceResolver.java

```
package org.emftext.language.java.resource.java.analysis;
import java.util.ArrayList;
import java.util.List;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EReference;
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.emftext.language.java.classifiers.Classifier;
import org.emftext.language.java.classifiers.ConcreteClassifier;
import org.emftext.language.java.containers.CompilationUnit;
import org.emftext.language.java.expressions.Expression;
import org.emftext.language.java.expressions.NestedExpression;
import org.emftext.language.java.generics.TypeParameter;
import org.emftext.language.java.instantiations.NewConstructorCall;
import org.emftext.language.java.references.Reference;
import org.emftext.language.java.references.ReferencesPackage;
import org.emftext.language.java.resource.java.IJavaReferenceResolveResult;
import org.emftext.language.java.resource.java.IJavaReferenceResolver;
import org.emftext.language.java.resource.java.analysis.decider.ConcreteClassifierDecider;
import org.emftext.language.java.resource.java.analysis.decider.IResolutionTargetDecider;
import org.emftext.language.java.resource.java.analysis.decider.TypeParameterDecider;
import org.emftext.language.java.resource.java.analysis.helper.ScopedTreeWalker;
import org.emftext.language.java.types.ClassifierReference;
import org.emftext.language.java.types.NamespaceClassifierReference;
import org.emftext.language.java.types.TypeReference;
......
......
public void resolve(String identifier, ClassifierReference container, EReference reference, int
                    position, boolean resolveFuzzy,IJavaReferenceResolveResult<Classifier> result) {
         List<IResolutionTargetDecider> deciderList = new ArrayList<IResolutionTargetDecider>();
         EObject startingPoint = null;
         EObject target = null;
         boolean hasNamespace = false;
         if (container.eContainer() instanceof NamespaceClassifierReference) {
            NamespaceClassifierReference ncr = (NamespaceClassifierReference) container.eContainer();
                int idx = ncr.getClassifierReferences().indexOf(container);
                if(idx > 0) {
                        hasNamespace = true;
                        startingPoint = ncr.getClassifierReferences().get(idx - 1).getTarget();
                }
                else {
```

```
                if(ncr.getNamespaces().size() > 0) {
                        hasNamespace = true;
                        EObject lastClassInNS =
                                        ConcreteClassifierDecider.resolveRelativeNamespace(
                                        ncr, 0, container, container, reference);
                        if (lastClassInNS != null) {
                                startingPoint = lastClassInNS;
                        }
                        else {
                                //absolute class starting with package
                                target = resolveFullQualifiedTypeReferences(identifier, ncr,
                                          container, reference);
                        }
                }
        }
}

if (target == null) {
        //new constructor call can be part of reference chain
        if (startingPoint == null && container.eContainer().eContainer() instanceof
             NewConstructorCall) {
                NewConstructorCall ncc = (NewConstructorCall)
                                        container.eContainer().eContainer();
                if (ncc.eContainmentFeature().equals(
                     ReferencesPackage.Literals.REFERENCE__NEXT)) {
                        startingPoint = ((Reference) ncc.eContainer()).getReferencedType();
                        //a ncc can be encapsulated in nested expressions
                        EObject outerContainer = ncc.eContainer();
                        while (outerContainer instanceof NestedExpression) {
                                Expression nestedExpression =
                                        ((NestedExpression)outerContainer).getExpression();
                                if (nestedExpression instanceof Reference) {
                                        outerContainer = (Reference) nestedExpression;
                                }
                                else {
                                        break;
                                }
                        }
                        if (outerContainer instanceof NewConstructorCall) {
                            NewConstructorCall outerNcc = (NewConstructorCall) outerContainer;
                            if (outerNcc.getAnonymousClass() != null) {
                                startingPoint = outerNcc.getAnonymousClass();
                            }
```

```
                        else {
                            startingPoint = outerNcc.getTypeReference().getTarget();
                        }
                    }
                }
            }
            if (startingPoint == null) {
                startingPoint = container;
            }
            if (hasNamespace) {
                if (startingPoint instanceof ConcreteClassifier) {
                    for(ConcreteClassifier cand :
                        ((ConcreteClassifier)startingPoint).getAllInnerClassifiers()) {
                        if (identifier.equals(cand.getName())) {
                            target = cand;
                            break;
                        }
                    }
                }
                else if (startingPoint instanceof TypeParameter) {
                    for(TypeReference extendsClassifierReference :
                                ((TypeParameter)startingPoint).getExtendTypes()) {
                        ConcreteClassifier extendsClassifier = (ConcreteClassifier)
                                        extendsClassifierReference.getTarget();
                        for(ConcreteClassifier cand :
                                    extendsClassifier.getAllInnerClassifiers()) {
                            if (identifier.equals(cand.getName())) {
                                target = cand;
                                break;
                            }
                        }
                    }
                }
            }
            else {
                deciderList.add(new ConcreteClassifierDecider());
                deciderList.add(new TypeParameterDecider());
                ScopedTreeWalker treeWalker = new ScopedTreeWalker(deciderList);
                target = treeWalker.walk(startingPoint, identifier, container, reference);
            }
        }
        if (target != null) {
            if (target.eIsProxy()) {
```

```
                        target = EcoreUtil.resolve(target, container);
                }
                if (!target.eIsProxy()) {
                        result.addMapping(identifier, (Classifier) target);
                }
        }
  }
......
......
```