École Centrale de Nantes **Université de Nantes** **École des Mines de Nantes**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2006

N° attribué par la bibliothèque

Contribution à l'étude des langages de transformation de modèles

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

Présentée
et soutenue publiquement par

Frédéric JOUAULT

*Le 26 septembre 2006 à l'*UFR *Sciences & Techniques, Université de Nantes,*
devant le jury ci-dessous

Président	:	Paul LE GUERNIC	INRIA
Rapporteurs	:	Charles CONSEL, Professeur	Université Bordeaux I
		Louis FÉRAUD, Professeur	Université Paul Sabatier Toulouse III
Examinateurs	:	Jean BÉZIVIN, Professeur	Université de Nantes
		Paul LE GUERNIC, Directeur de recherche	INRIA
		Olivier ROUX, Professeur	École Centrale de Nantes
		Patrick VALDURIEZ, Directeur de recherche	INRIA

Directeur de thèse : Pr. Jean BÉZIVIN

Laboratoire: **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 0366-264

CONTRIBUTION À L'ÉTUDE DES LANGAGES DE TRANSFORMATION DE MODÈLES

Contribution to the study of model transformation languages

Frédéric JOUAULT

⋈

favet neptunus eunti

Université de Nantes

Frédéric JOUAULT

Contribution à l'étude des langages de transformation de modèles

IV+X+176 p.

Ce document a été préparé avec LaTeX2e et la classe `these-LINA` version v. 2.7 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse : `http://login.irin.sciences.univ-nantes.fr/`.
Cette classe est conforme aux recommandations du ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche (circulaire n° 05-094 du 29 mars 2005), de l'Université de Nantes, de l'école doctorale « Sciences et Technologies de l'Information et des Matériaux »(ED-STIM), et respecte les normes de l'association française de normalisation (AFNOR) suivantes :

- AFNOR NF Z41-006 (octobre 1983)
  Présentation des thèses et documents assimilés ;
- AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure ;
- AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Impression : main.tex – 17/10/2006 – 15:04.

Révision pour la classe : these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp

Résumé

Les techniques classiques de développement logiciel consistent généralement en l'écriture du code source d'un système par des programmeurs à partir d'une spécification comportant des modèles. Ces derniers sont souvent des dessins qui ne peuvent pas être traités automatiquement. On parle donc de modèles contemplatifs. L'ingénierie des modèles, dont le MDA (*Model Driven Architecture*) est une variante, est un nouveau paradigme de l'ingénierie du logiciel qui considère les modèles comme entités de première classe. Les modèles ne sont donc plus limités à la documentation d'un système mais peuvent faire partie de sa définition, au même titre que le code source. Ainsi, des techniques de transformations de modèles peuvent être mises en œuvre afin de générer automatiquement des parties du système à partir de modèles.

Cette thèse contribue à faire avancer les connaissances sur l'ingénierie des modèles et en particulier sur la transformation de modèles. Trois langages sont proposés : un langage de métamodélisation appelé KM3 (*Kernel MetaMetaModel*), un langage de transformation de programmes en modèles et modèles en programmes appelé TCS (*Textual Concrete Syntax*) et un langage de transformation de modèles appelé ATL (*ATLAS Transformation Language*). Une plateforme de modélisation appelée AMMA (*ATLAS Model Management Architecture*) basée sur ces trois langages est définie. Un ensemble de cas d'étude implémentés avec AMMA et couvrant différents domaines sont décrits.

Mots-clés : ingénierie des modèles, MDE, MDA, métamodélisation, transformation de modèles

Abstract

Traditional software development techniques generally involve programmers writing source code of a system from a specification including models. These models are often drawings that cannot be automatically processed. They are therefore called contemplative models. Model engineering, of which MDA (Model Driven Architecture) is a variant, is a new paradigm of software engineering, which considers models as first-class entities. Thus, the models are no longer restricted to the documentation of a system but can be part of its definition like source code. Model transformation techniques can then be used in order to automatically generate parts of a system from models.

This thesis contributes to increasing the knowledge on model engineering, and more particularly on model transformation. Three languages are proposed: a metamodelisation language called KM3 (Kernel MetaMetaModel), a language for transformation of programs into models and of models into programs called TCS (Textual Concrete Syntax), and a model transformation language called ATL (ATLAS Transformation Language). A modeling platform called AMMA (ATLAS Model Management Architecture) based on these three languages is defined. A set of case studies implemented using AMMA and covering various domains are described.

Keywords: model engineering, MDE, MDA, metamodelisation, model transformation

Remerciements

Mes remerciements vont tout d'abord à Jean Bézivin qui m'a donné l'opportunité de travailler sur un sujet passionnant et qui a dirigé mes recherches. Je lui suis reconnaissant pour son soutien et pour son accompagnement amical et avisé.

Je remercie Charles Consel et Louis Féraud qui m'ont fait l'honneur d'évaluer mon travail de thèse et d'en être les rapporteurs.

Je remercie Paul Le Guernic, Olivier Roux et Patrick Valduriez, membres du jury, qui ont accepté de juger mon travail.

Je remercie mes collègues de l'équipe ATLAS et du LINA et plus particulièrement Freddy Allilaire, Marcos Didonet Del Fabro et Ivan Kurtev pour leur participation essentielle au développement des idées et des outils de la plateforme AMMA.

Je remercie les étudiants et stagiaires de l'équipe ATLAS qui ont notamment contribué activement au développement des bibliothèques de métamodèles KM3 et de transformations ATL, montrant ainsi l'applicabilité des idées présentées dans cette thèse à de nombreux domaines.

Enfin, je remercie mes parents, ma famille et mes amis pour leur soutien tout au long de mes études et spécialement pendant ces trois années de thèse.

Sommaire

Glossaire

ADT acronyme anglais de *ATL Development Tools* signifiant outils de développement ATL.

AM3 acronyme anglais de *ATLAS MegaModel Management* signifiant gestion de mégamodèle ATLAS.

AMMA acronyme anglais de *ATLAS Model Management Architecture* signifiant architecture de gestion de modèles ATLAS.

AMW acronyme anglais de *ATLAS Model Weaver* signifiant tisseur de modèles ATLAS.

ATL acronyme anglais de *ATLAS Tranformation Language* signifiant langage de transformation ATLAS.

ATLAS nom du groupe de recherche (INRIA et LINA) dans le cadre duquel se sont déroulés les travaux décrits dans ce document.

ATP acronyme anglais de *ATLAS Technical Projectors* signifiant projecteurs techniques ATLAS.

CMOF acronyme anglais de *Complete Meta Object Facility* signifiant MOF complet. Cet acronyme s'applique à la version 2.0 de MOF mais pas à ses versions antérieures.

DDMM acronyme anglais de *Domain Definition MetaModel* signifiant métamodèle de définition de domaine.

DSL acronyme anglais de *Domain Specific Language* signifiant langage dédié, la traduction littérale étant : langage spécifique de domaine.

Eclipse plateforme ouverte pour le développement d'applications développée par la fondation du même nom.

Ecore nom du métamétamodèle sur lequel est basé EMF. Ecore est proche d'EMOF 2.0 mais n'est pas exactement conforme à cette recommandation de l'OMG.

EMF acronyme anglais de *Eclipse Modeling Framework* signifiant environnement de modélisation pour Eclipse.

EMOF acronyme anglais de *Essential Meta Object Facility* signifiant partie essentielle de MOF. Cet acronyme s'applique à la version 2.0 de MOF mais pas à ses versions antérieures.

IDM acronyme de *Ingénierie Dirigée par les Modèles*, synonyme de IdM et équivalent à MDE en anglais.

IdM acronyme de *Ingénierie des Modèles*, synonyme de IDM et équivalent à *model engineering* en anglais.

INRIA acronyme de *Institut National de Recherche en Informatique et en Automatique*.

LINA acronyme de *Laboratoire d'Informatique de Nantes Atlantique*. Il s'agit du laboratoire au sein duquel les travaux décrits dans ce document ont été réalisés.

MDA acronyme anglais de *Model Driven Architecture* signifiant architecture dirigée par les modèles. C'est ainsi que l'OMG appelle son approche de modélisation basée sur ses recommandations. Le MDA est une variante de l'ingénierie des modèles (IdM).

MDE acronyme anglais de *Model Driven Engineering* signifiant ingénierie dirigée par les modèles, équivalent à IDM en français.

MOF acronyme anglais de *Meta Object Facility* qui est le nom du métamétamodèle promu par l'OMG. La version 2.0 de MOF définit deux versions de complexité différente : EMOF et CMOF.

OCL acronyme anglais de *Object Constraint Language* signifiant langage de contrainte sur les objets. Il s'agit d'une recommandation de l'OMG définissant un langage d'expression de requêtes et de contraintes sur les modèles UML et généralisé pour l'utilisation sur n'importe quel modèle MOF.

OMG acronyme anglais de *Object Management Group* qui est le nom d'un organisme qui publie notamment des recommandations pour la modélisation dans le cadre du MDA telles que : MOF, UML, OCL, QVT, XMI.

QVT acronyme anglais de *Query / View / Transform* signifiant requête / vue / transformation. C'est le nom donné par l'OMG à sa recommandation MDA pour la transformation de modèles.

TS acronyme anglais de *Technical Space* signifiant espace technique.

UML acronyme anglais de *Unified Modeling Language* signifiant langage de modélisation unifié.

XMI acronyme anglais de XML *Model Interchange* signifiant échange de modèles en XML. C'est le nom donné par l'OMG à sa recommandation MDA pour la représentation des modèles dans des fichiers. XMI est basé sur XML.

XML acronyme anglais de *eXtensible Markup Language* signifiant langage extensible à balise.

CHAPITRE 1

Introduction

1.1 Contexte et enjeux

Les systèmes logiciels sont de plus en plus complexes. Le code source, qui est encore le principal élément de leur définition, permet de moins en moins de gérer cette complexité. Il s'agit souvent d'une définition des actions que le système doit effectuer afin de réaliser la fonction attendue. Cette définition prend généralement la forme de textes écrits par des programmeurs dans un ou des langages de programmation. On parlera alors de "programmation directe". Un processus de programmation directe s'appuie sur la traduction d'un langage de haut niveau (capable d'exprimer des fonctionnalités abstraites) en un langage de bas niveau (capable d'être exécuté par une machine). Les langages de programmation ont évolué afin de pouvoir gérer la complexité croissante des logiciels. Différents paradigmes ont été utilisés : procédural, fonctionnel, objet, etc. De nombreux langages reposant sur chacun d'entre eux ont été spécifiés.

En parallèle à l'évolution des langages, des méthodologies d'analyse et de conception de logiciels ont été définies. Ces méthodologies permettent généralement de prendre en compte ce que plusieurs chercheurs nomment actuellement les "early aspects". Il s'agit des phases amont, préparatoires à l'écriture du code, et des phases aval, par exemple ce qui concerne les activités de maintenance. La plupart de ces méthodologies utilisent des éléments plus abstraits que le code source, souvent appelés modèles. Le terme de modèle est souvent utilisé dans ce contexte pour désigner un diagramme informel destiné à guider et à inspirer le programmeur dans sa tâche de production de code exécutable. Nous qualifions ces modèles de "contemplatifs" car ils ont pour rôle essentiel de fournir la source d'inspiration au programmeur qui réalise une synthèse complexe.

Une nouvelle manière d'envisager la production et la maintenance des systèmes logiciels consiste à s'appuyer essentiellement sur ces modèles, qui sont alors considérés comme entités de première classe. Il s'agit de l'ingénierie des modèles (ou IdM), qui est une branche de l'ingénierie des langages. Les modèles sont maintenant représentés à l'aide de formats précis dont la manipulation peut être automatisée. Chacun de ces modèles est défini en utilisant un langage spécifiant un ensemble de concepts et leurs relations. Chaque langage a généralement aussi une syntaxe concrète, par exemple textuelle ou visuelle, permettant de représenter les modèles. L'IdM vise à définir un système logiciel à l'aide d'un ensemble de modèles utilisant différents langages. L'un des intérêts de l'IdM est de pouvoir considérer les modèles sur lesquels le programmeur raisonne comme faisant partie à part entière de la définition du logiciel. Chacun d'entre eux représente une partie du système. Cependant, toutes les informations contenues dans ces modèles ne sont pas toujours strictement nécessaires à l'exécution des actions que le système doit effectuer.

L'IdM s'inscrit plus en continuité et en complémentarité qu'en rupture avec les approches classiques de programmation directe. Un processus IdM de production de logiciel va généraliser l'approche de programmation directe en s'appuyant sur un nombre très important de langages capables de décrire les différents artefacts produits et consommés au cours du cycle de vie du logiciel. D'une part, l'IdM emprunte à la programmation directe son traitement rigoureux des aspects linguistiques. D'autre part, l'IdM

emprunte aux méthodes d'analyse et de conception leur prise en compte des aspects amont et aval du cycle de vie. L'IdM se différencie donc de la programmation directe par le fait qu'elle s'applique au cycle de vie complet du logiciel. Au delà d'une différence de terminologie qui renomme *programmes* en *modèles* et *grammaires* en *métamodèles*, l'aspect nouveau (et difficile) de l'IdM est la considération coordonnée, cohérente et automatisée d'un grand nombre de langages définis de manière rigoureuse et complémentaire. Un apport essentiel de l'IdM est l'unification du processus de production et de maintenance du logiciel sous la forme de chaînes automatisées d'opérations sur des opérandes rigoureusement définis appelés modèles. Un grand nombre de ces opérations sont des transformations.

Les langages de définition de modèles peuvent être de nature très variée. Il peut s'agir de langages généralistes capables de représenter tous les aspects d'un système. Mais il peut aussi s'agir de langages dédiés dont chacun ne peut capturer qu'une partie limitée d'un système. L'intérêt de ces langages dédiés est qu'ils abandonnent la généralité au profit d'une plus grande expressivité dans un domaine donné. L'un de leurs inconvénients est qu'il est généralement nécessaire d'en avoir plusieurs afin de couvrir tous les domaines requis pour représenter un système donné. Or, il est généralement plus coûteux de construire des systèmes de représentation pour de multiples langages plutôt que pour un seul. L'IdM offre certes la possibilité d'utiliser des langages généralistes. Cependant, elle est particulièrement adaptée à l'utilisation de langages dédiés car elle permet, de par son essence même, une représentation précise des modèles les utilisant. Une combinaison de ces deux sortes de langages peut aussi être utilisée.

L'IdM est connue sous différents noms. Une de ses appelations en langue anglaise est MDE (*Model Driven Engineering* pour ingénierie dirigée par les modèles). Les principes du MDE sont appliqués dans différents standards. L'approche MDA (*Model Driven Architecture* ou architecture dirigée par les modèles) est un exemple d'application du MDE. Le MDA est recommandé par l'OMG (*Object Management Group*) et est basé sur d'autres recommandations de ce même organisme.

L'automatisation de la manipulation des modèles est donc réalisée par des opérations sur ces modèles. La principale opération est la transformation de modèles qui consiste à créer de nouveaux modèles à partir de modèles existants. La transformation n'est alors plus limitée à la traduction du code source mais peut opérer sur tous les modèles décrivant un système. De plus, les modèles créés par transformation ne sont pas nécessairement du code exécutable mais peuvent être très variés. Dans son approche MDA, l'OMG recommande QVT (Query / View / Transformation ou requête / vue / transformation) qui propose une famille de langages de transformation.

Au moment d'écrire ce document, les possibilités offertes par l'IdM sont encore méconnues et de nombreuses de questions restent encore ouvertes. Ce travail n'a l'ambition que d'aider à mieux comprendre une nouvelle approche de la production et de la maintenance du logiciel.

1.2 Contributions de la thèse

Les travaux présentés ici ont pour objectif de contribuer à améliorer les connaissances actuelles sur l'ingénierie des modèles ainsi qu'à développer des outils et techniques qui aideront à la rendre applicable dans différents contextes. La transformation étant une opération essentielle, un accent a été mis sur la conception d'un langage de transformation de modèles : ATL. D'autres éléments ont aussi été définis afin de positionner ce langage dans une approche IdM. Ainsi, afin de définir précisément ces modèles à transformer, une définition de la notion de modèle dans le contexte de l'ingénierie des modèles est nécessaire.

L'approche langage dédié a été privilégiée par rapport à une approche basée sur un langage généraliste pour deux raisons principales :

- L'ingénierie des modèles est particulièrement adaptée aux langages dédiés. L'existence d'un grand nombre de langages généralistes montre qu'ils n'ont pas réellement besoin de l'ingénierie des modèles pour exister. En revanche, l'approche basée sur les langages dédiés peut bénéficier de nouveaux outils conceptuels et opérationnels facilitant leurs définitions.
- L'ingénierie des modèles est basée sur des langages dédiés. Par exemple, le langage de métamodélisation recommandé par l'OMG : *MOF* (*Meta Object Facility*) est un langage dédié. De même, l'appel à proposition de l'OMG pour la définition du langage de transformation QVT spécifie que ce langage doit être dédié à la transformation de modèles.

Les contributions de cette thèse sont les suivantes :

1. Une définition formelle de la structure des modèles compatible avec l'approche MDA de l'OMG. Cette définition permet de mieux comprendre la structure des modèles. Elle peut aussi servir de base à la formalisation d'autres éléments tels qu'un langage de métamodélisation.
2. Un langage de métamodélisation appelé KM3 (pour *Kernel MetaMetaModel* ce qui signifie métamétamodèle noyau) s'appuyant sur cette définition. KM3 est compatible avec plusieurs autres langages similaires tels que *MOF*, le langage de métamodélisation recommandé par l'OMG, ou encore *Ecore*, une variante de *MOF* adaptée à son utilisation avec le langage Java dans l'environnement Eclipse. Une syntaxe textuelle simple pour KM3 a été définie. De plus, une définition formelle de KM3 est donnée.
3. Un langage de transformation de modèles appelé ATL basé sur un paradigme hybride entre déclaratif et impératif. ATL opère sur des modèles définis à l'aide de métamodèles eux-mêmes définis en KM3.
4. Un langage de transformation de programmes en modèles et de modèles en programmes appelé TCS. Ce langage permet, entre autres, la représentation des syntaxes textuelles de KM3, ATL et TCS lui-même.
5. Une plateforme de modélisation appelée AMMA reposant sur les langages KM3, ATL et TCS. Cette plateforme contient les primitives de gestion de modèles. D'autres composants tels que la gestion globale de modèles font l'objet de recherches toujours en cours.
6. Un ensemble de cas d'étude couvrant différents domaines applicatifs. Certains d'entre eux correspondent à l'implémentation de langages dédiés. Les langages KM3, ATL et TCS peuvent aussi être considérés comme cas d'étude car ils sont en partie mutuellement définis.

Des prototypes opérationnels pour les trois langages KM3, ATL et TCS cités ci-dessus ont été implémentés au cours de nos travaux. Ces implémentations ont permis de tester l'applicabilité de l'ingénierie des modèles et plus particulièrement de la plateforme AMMA à différents domaines. Ceci est illustré par la mise en œuvre des cas d'étude avec ces prototypes. Voici une brève description des prototypes :

- **KM3.** Un ensemble de transformations de modèles permettent d'utiliser les métamodèles définis en KM3 dans les systèmes de manipulation de modèles Eclipse/EMF et Netbeans/MDR. Il est ainsi possible de convertir les métamodèles KM3 depuis et vers *Ecore* et *MOF* 1.4.
- **TCS.** La fonctionnalité "programme vers modèle" de TCS est implémentée par une transformation de modèles ciblant un générateur d'analyseur syntaxique. La fonctionnalité opposée (modèle vers programme) est fournie par un interprète écrit en Java.
- **ATL.** Un moteur d'exécution modulaire permet d'exécuter les transformations de modèles définies en ATL. L'architecture de ce moteur est basée sur une machine virtuelle qui permet de transformer des modèles définis avec différents systèmes de manipulation de modèles. Deux de ces systèmes (Eclipse/EMF basé sur Ecore et Netbeans/MDR basé sur MOF 1.4) sont déjà utilisables et d'autres peuvent être ajoutés.

1.3 Plan de la thèse

Ce document est composé de huit chapitres en plus de ce chapitre d'introduction :

- Le chapitre 2 définit le problème que nous tentons de résoudre.
- Le chapitre 3 présente l'état de l'art dans le domaine de la transformation de modèles.
- Le chapitre 4 présente la plateforme de modélisation AMMA et introduit trois de ses composants essentiels. Il s'agit des langages KM3, ATL et TCS qui sont décrits en détail dans les chapitres suivants.
- Le chapitre 5 décrit le langage de métamodélisation KM3 et en donne une définition formelle.
- Le chapitre 6 présente le langage TCS de transformation de programmes en modèles et de modèles en programmes.
- Le chapitre 7 présente le langage ATL de transformation de modèles.
- Le chapitre 8 détaille trois cas d'étude de la plateforme AMMA : la gestion de la traçabilité par les transformations de modèles, la vérification et la mesure des modèles ainsi qu'une utilisation de AMMA pour l'implémentation de deux langages dédiés à la téléphonie par internet. Ces cas d'étude font usage des trois langages : KM3, TCS et ATL.
- Le chapitre 9 donne quelques conclusions de nos travaux et présente quelques perspectives d'évolution.

CHAPITRE 2

Présentation du problème

2.1 Introduction

Ce chapitre présente la problématique principale de nos travaux : la transformation de modèles. Cependant, il est difficile de considérer cette technique sans évoquer ses applications potentielles. Nous avons donc choisi de considérer un problème applicatif : la gestion de données hétérogènes. Par ailleurs, l'ingénierie des langages dédiés présente de nombreux points communs avec l'ingénierie des modèles. Nous allons donc aussi la présenter.

Il existe de nombreuses technologies dont la plupart sont basées sur des systèmes de représentation de données différents et généralement incompatibles. Par exemple, les systèmes basés sur les grammaires représentent les données sous la forme de chaînes de caractères dont la structure est contrainte par ces grammaires. Un autre exemple est XML (eXtensible Markup Language) dont le système de représentation est basé sur des arbres particuliers. La structure de ces arbres peut être définie à l'aide de schémas XML. Bien qu'il soit possible de traduire des données entre ces technologies, les techniques classiques nécessitent souvent le développement d'outils ad hoc. Un nouvel outil doit souvent être créé pour chaque paire de formats, par exemple pour une grammaire et un schéma XML donnés. Les techniques de transformation de modèles peuvent être utilisées pour le développement d'outils génériques de conversion. Il est en effet possible d'automatiser une certaine partie du travail à réaliser sous la forme de transformations de modèles. Ceci nécessite cependant de pouvoir considérer les données à convertir comme des modèles. Nous introduirons donc la notion d'espace technique qui permet de faire cette hypothèse pour certaines technologies telles que celle des grammaires ou celle basée sur XML.

Contrairement aux langages généralistes (comme Java, C, Pascal), utilisables pour définir n'importe quelle solution, un langage dédié ne couvre qu'un domaine limité. Par exemple, SQL est dédié au requêtage de base de données et SVG [4] (*Scalable Vector Graphics*) à la description de dessins. L'intérêt principal des langages dédiés est qu'ils sont très expressifs dans leur domaine, au détriment de leur généralité. Puisque l'expressivité d'un DSL croît en général avec la réduction du domaine couvert, il est souvent nécessaire d'avoir beaucoup de ces langages pour définir un système complet. De nombreuses techniques utilisées pour le développement des langages généralistes sont donc peu utilisables car trop coûteuses. Le fait de considérer à la fois l'ingénierie des langages dédiés et celle des modèles peut permettre de faire des progès dans les deux. Un cas particulier est celui d'un langage de transformation de modèles qui est typiquement un langage dédié à cette tâche.

L'organisation de ce chapitre est la suivante. Les sections 2.2, 2.3 et 2.4 présentent respectivement les contextes de l'ingénierie des modèles, de la gestion de données hétérogènes et des langages dédiés. Chacune de ces sections se termine par l'identification d'un ensemble de besoins que nos travaux contribuent à satisfaire. Enfin, la section 2.5 donne les conclusions de ce chapitre.

2.2 L'ingénierie des modèles

L'ingénierie des modèles [7] est un nouveau paradigme de l'ingénierie du logiciel. Son principe essentiel consiste à considérer les modèles comme entités de première classe. De la même manière que le principe "tout est objet" a permis, dans les années 1980, de faire avancer la technologie objet, le principe "tout est modèle" peut aussi être essentiel pour l'établissement de l'ingénierie des modèles. Il s'agit de considérer a priori que toute entité manipulée par un système informatique est un modèle. Tenter d'appliquer ce principe à tout nouveau problème permet de tester les possibilités de l'ingénierie des modèles.

Les modèles, dont l'utilisation a longtemps été limitée à la documentation des logiciels, font désormais partie de la définition du logiciel. Il existe plusieurs implémentations de l'ingénierie des modèles telles que le MDA de l'OMG, le MIC (Model Integrated Computing), les usines à logiciel [34] (Software Factories), les sNets [8], etc. Nous allons commencer par présenter l'une d'entre elles : le MDA. Puis nous en généraliserons certains aspects. La formalisation de l'ingénierie des modèles que nous proposons sera en revanche détaillée dans le chapitre 4. Nous décrirons ensuite les relations entre l'ingénierie des modèles et la technologie des objets afin de mieux positionner ces deux paradigmes l'un par rapport à l'autre.

2.2.1 L'approche MDA de l'OMG

En novembre 2000, l'OMG a annoncé son initiative MDA™ qui est une variante particulière de l'ingénierie des modèles. UML (*Unified Modeling Language*) a d'abord été le moteur qui a permis de définir un système de représentation uniforme des modèles. En effet, dès 1995 l'OMG a lancé un appel à proposition pour un langage de modélisation réconciliant un grand nombre d'approches différentes existant à l'époque. Ce travail a abouti en 1997 à la définition de UML 1.1. Afin d'uniformiser la définition des modèles UML et du langage UML lui-même, MOF a été défini.

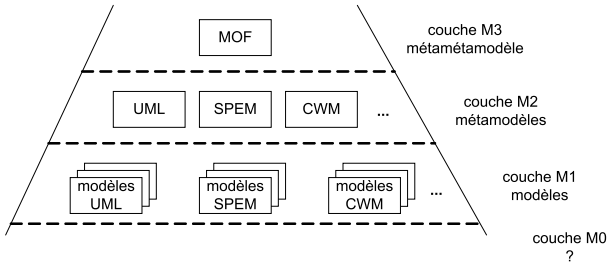

Figure 2.1 – L'architecture en quatre couches du MDA

L'architecture obtenue est représentée sur la figure 2.1. Les modèles UML apparaissent au niveau appelé M1. Le langage UML utilisé pour les définir est situé au niveau immédiatement supérieur : M2. En réalité, seuls les concepts de UML, avec leurs relations et propriétés, que les modèles peuvent utiliser sont définis au niveau M2. Leur signification ainsi que leur représentation visuelle (ou syntaxe) ne sont pas présentes. Nous pouvons, pour l'instant, considérer qu'ils ne sont pas pris en compte par l'architecture MDA et qu'ils ne sont donc définis que dans le texte de la recommandation UML, en anglais. Cette entité qui contient les concepts et relations de UML s'appelle son métamodèle. Le métamodèle UML lui-même est défini en utilisant un ensemble limité de concepts et relations qui sont similairement placés au niveau supérieur : M3. Cet ensemble servant à définir un métamodèle s'appelle un métamétamodèle.

Le niveau M0 est parfois décrit comme étant le “monde réel” que les modèles du niveau M1 représentent. Cependant, il n'y a pas de consensus total sur ce point. C'est pourquoi nous avons représenté M0 suivi d'un point d'interrogation sur la figure 2.1, sans en détailler le contenu. Notre interprétation de ce niveau M0 sera présentée dans la section 2.2.2.

Puisque les éléments du métamodèle UML sont des concepts et des relations, les éléments présents au niveau M3 sont en réalité essentiellement les notions de *concept* et de *relation*. Le métamétamodèle défini par l'OMG a été appelé MOF (*Meta Object Facility*), *concept* renommé en *Class* (i.e. classe) et *relation* en *Association* et *Attribute* (association et attribut). Ces choix, hérités de la technologie des objets, entraînent souvent un amalgame entre les notions d'objet et de modèle puisque les deux paradigmes utilisent un vocabulaire similaire. Un autre exemple de confusion linguistique est l'utilisation omniprésente du mot *instance* pour dénoter des relations souvent très différentes :

- Un object est une instance d'une classe.
- Un modèle est une instance d'un métamodèle.
- Un élément de modèle est une instance d'une classe de son métamodèle.
- etc.

Il est cependant essentiel de bien saisir les différences entre objet et modèle. Les relations entre les deux approches sont résumées à la section 2.2.3 afin d'aider le lecteur à discerner les points similaires et surtout les divergences. À partir de maintenant, nous dirons qu'un modèle est conforme à un métamodèle lorsqu'il utilise les concepts définis par ce dernier. Les définitions formelles de modèle, métamodèle, métamétamodèle et conformité qui seront données dans le chapitre 4 ne font appel qu'à la logique du premier ordre, sans référence à l'objet, et clarifient donc la situation.

L'apport essentiel du MOF est qu'il n'est pas limité à la définition de UML. En effet, MOF peut aussi servir à capturer les concepts et relations de tout langage. Ainsi, les concepts et relations de MOF lui-même sont définis en termes de MOF. Il n'est donc pas nécessaire d'avoir de niveau supérieur à M3. La figure 2.2 résume la situation en représentant la relation de conformance par une flèche appelée *conformsTo* (ce qui signifie *conforme à*) : chaque modèle UML est conforme au métamodèle UML, le métamodèle UML est conforme au MOF et le MOF est conforme à lui-même.

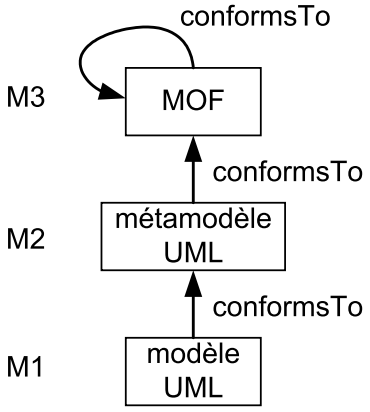

Figure 2.2 – La relation de conformance entre les trois niveaux supérieurs du MDA

D'autres métamodèles, tels que CWM^{TM} (Common Warehouse Metamodel) ou SPEM (Software Process Engineering Metamodel) ont alors été définis par l'OMG à l'aide de MOF et positionnés au niveau M2 comme le montre la figure 2.1. Les modèles définis à l'aide de ces nouveaux métamodèles se retrouvent tout naturellement à côté des modèles UML, au niveau M1. Plus récemment, il est apparu

que la transformation de modèles est une notion essentielle pour l'utilisation des modèles. L'une des applications mise en avant est la génération automatique de code exécutable à partir de modèles UML. En 2002, l'OMG a donc lancé un appel à proposition [55] pour la définition d'un langage de transformation appelé QVT (Query / View / Transformation), ce langage devant tout naturellement être capturé par un métamodèle MOF. La recommantation QVT a été adoptée en 2005 [61].

Après avoir présenté ce bref historique et ces quelques notions de bases sur le MDA, il reste à définir quels sont ses objectifs. Leur définition a bien entendu évolué avec le développement de la compréhension de l'ingénierie des modèles et l'apparition de nouvelles recommandations comme QVT. Voici la définition du MDA donnée par l'OMG le 26 août 2004 [52] :

> "MDA is an OMG initiative that proposes to define a set of non-proprietary standards that will specify interoperable technologies with which to realize model-driven development with automated transformations. Not all of these technologies will directly concern the transformations involved in MDA.
>
> MDA does not necessarily rely on the UML, but, as a specialized kind of MDD (Model Driven Development), MDA necessarily involves the use of model(s) in development, which entails that at least one modeling language must be used. Any modeling language used in MDA must be described in terms of the MOF language, to enable the metadata to be understood in a standard manner, which is a precondition for any ability to perform automated transformations."

Ce qui peut se traduire en :

> "Le MDA est une initiative de l'OMG qui propose de définir un ensemble de recommandations non-propriétaires qui spécifieront des technologies interopérables afin de réaliser le développement dirigé par les modèles (MDD) avec des transformations automatisées. Ces technologies ne concerneront pas toutes directement la problématique de transformation impliquée dans le MDA.
>
> Le MDA ne repose pas nécessairement sur UML, mais, en tant que variante spécialisée du MDD, le MDA comporte nécessairement l'utilisation de modèle(s) dans le développement, ce qui implique qu'au moins un langage de modélisation soit utilisé. Tout langage utilisé dans le MDA doit être décrit en termes de MOF, afin de permettre que les métadonnées soient comprises d'une manière standard, ce qui est une précondition à toute capacité à réaliser des transformations automatisées."

Remarques. Le fait que MOF soit conforme à lui-même ne signifie pas que le langage capturé par le métamétamodèle MOF soit auto-défini. De même que le métamodèle UML ne définit que les concepts, avec leurs relations, utilisés par les modèles UML, MOF ne définit que les concepts et relations utilisés par MOF et les métamodèles tels que UML, CWM ou SPEM. Les autres aspects essentiels (par exemple sémantique et syntaxe) des langages définis en MOF, tels que MOF et UML, sont généralement capturés informellement dans le texte des recommandations de l'OMG.

L'architecture présentée dans les figures 2.1 et 2.2 permet une représentation uniforme des modèles, les langages utilisés pour les définir étant capturés par des métamodèles. Cependant, l'OMG a défini un autre système de représentation à l'intérieur de ce premier système : les profils UML (*UML profiles* en anglais). Cette notion est définie au niveau du métamodèle UML et permet de capturer des langages non plus avec des classes et associations MOF mais avec des stéréotypes et valeurs marquées (*tagged values*). Il devient donc possible de représenter des langages et les modèles les utilisant à l'aide de modèles UML uniquement, sans avoir à définir de nouveau métamodèle. Bien que redondant, en théorie, ce système de profils a permis aux outils UML, dans lesquels le métamodèle n'était en réalité souvent pas défini à l'aide

de MOF mais codé en dur, de capturer langages et modèles sans nécessiter de changement fondamental. Même si cette technique est applicable à d'autres métamodèles sous des formes similaires, elle est en pratique limitée au métamodèle UML et contourne le système de représentation uniforme des couches M1, M2 et M3. Puisqu'il s'agit d'une technique dont la portée est limitée à un métamodèle et non à l'ingénierie des modèles en général, nous ne nous intéresserons pas particulièrement aux profils UML dans ce document. De plus, ceux-ci sont automatiquement pris en compte par les approches mettant en œuvre plusieurs métamodèles, telles que la transformation de modèles.

2.2.2 Principes de base

Après avoir présenté une application particulière de l'ingénierie des modèles, nous allons maintenant prendre du recul et détailler notre interprétation de l'ingénierie des modèles. Il ne s'agit pas de contredire le MDA mais plutôt de le préciser et de compléter certains points. Cette présentation reste informelle et sert à introduire notre problématique. Des définitions plus précises sont données au chapitre 4.

Commençons par étudier le problème de la relation entre les modèles et le "monde réel", qui correspond au niveau M0 du MDA. Pour ce faire, nous donnons deux définitions : une pour système et une pour modèle.

Définition 1. Un *système* est un ensemble d'éléments interdépendants formant un tout complexe.

Définition 2. Un *modèle* est une représentation plus ou moins abstraite d'un système et est créé dans un but précis.

Or, nous avons vu précédemment qu'un métamodèle ne représente pas un ou des modèles mais représente en fait les concepts et relations utilisés par des modèles. Les deux relations entre système et modèle, d'une part, et entre modèle et métamodèle, d'autre part, sont donc de nature différente. La figure 2.3 représente ces relations. Nous appelons la première relation, entre système et modèle, la relation de représentation : un modèle est une représentation d'un système, ce qui est dénoté par la flèche *representationOf* (c'est-à-dire *représentation de*) de modèle vers système. La seconde est appelée conformance : un modèle est conforme à un métamodèle, ce qui est représenté par la flèche *conformsTo*. Les deux flèches correspondant aux deux relations ne sont pas alignées sur la figure 2.3 car ces relations sont de natures différentes. En revanche, ces deux relations ne sont bien entendu pas indépendantes. Ainsi, il est possible de considérer un métamodèle comme un filtre qui délimite les aspects du système qu'un modèle s'y conformant peut représenter.

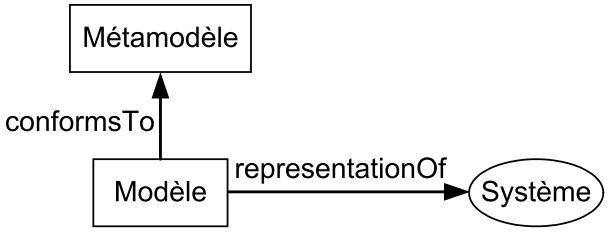

Figure 2.3 – Relations de base en ingénierie des modèles

Une représentation des couches de modélisation peut maintenant être donnée à la figure 2.4. Du fait de la différence de nature des relations entre les niveaux M0-M1, d'une part, et M1-M2, M2-M3 et M3-M3, d'autre part, on parle parfois de 3+1 couches au lieu de quatre. La ligne en pointillés présente

entre les niveaux M0 et M1 concrétise la séparation entre le “monde des modèles”, situé au-dessus et comportant les niveaux M1 à M3, et le “monde réel”, situé au-dessous et comportant les systèmes.

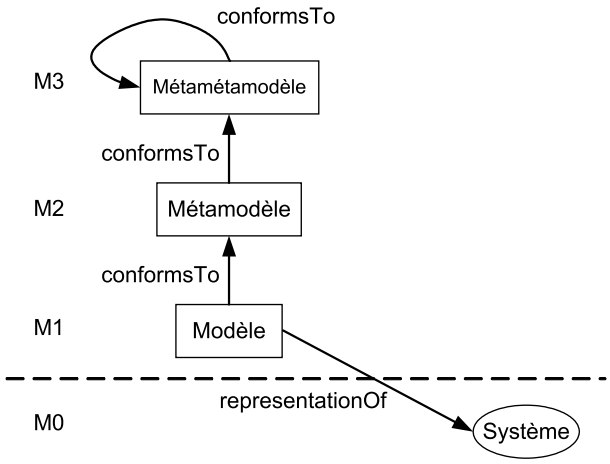

Figure 2.4 – Les quatre couches de modélisation et leurs relations

Un autre aspect important de l’ingénierie des modèles est la place qu’y prend la transformation de modèles. En effet, l’un des intérêts d’avoir un système de représentation uniforme des modèles est de pouvoir automatiser leur manipulation. Parmi les opérations automatisables, nous pouvons citer :

- **La traduction** d’un langage (ou métamodèle) à un autre. Cette opération peut servir, par exemple à traduire un modèle abstrait en modèle opérationnel ou exécutable. Il est ainsi possible de convertir un diagramme de classes en SQL ou encore une machine d’état en langage machine. De telles traductions peuvent aussi avoir pour objectif la définition d’une sémantique pour le métamodèle source en traduisant ses concepts vers un métamodèle cible dont la sémantique est connue (par exemple les réseaux de Petri).
- **La mesure** des modèles : comptage du nombre d’éléments en fonction de leur type, du nombre de cycles dans un graphe, etc. Le résultat de la mesure peut prendre la forme d’une liste ou d’un tableau et peut être représenté par un modèle.
- **La vérification de contraintes** sur les modèles [11]. Le résultat de cette vérification est aussi représentable par un modèle dont la complexité peut être adaptée aux besoins. Il peut s’agir d’une simple valeur booléenne indiquant si toutes les contraintes sont satisfaites par le modèle. Un entier peut représenter le nombre de violations de contraintes. Une structure plus complexe peut aussi capturer des informations plus complexes telles qu’un message explicitant la violation, une référence vers les éléments incriminés, etc.

Toutes ces opérations sont en réalité des transformations de modèles correspondant à la définition suivante :

Définition 3. Une *transformation de modèles* est une opération qui crée automatiquement un ensemble de modèles cible à partir d’un ensemble de modèles source.

La figure 2.5 représente le contexte opérationnel de la transformation de modèles. Le modèle M_B, conforme au métamodèle MM_B, est obtenu par l’application de la transformation MM_A *à* MM_B au modèle M_A, conforme au métamodèle MM_A. De plus, en suivant le principe “tout est modèle”, la transformation elle-même est un modèle dont le métamodèle est MM_T. On dit alors que MM_A *à* MM_B

est un modèle de transformation. Le langage de transformation, ou plus précisément ses concepts et leurs relations, est donc capturé par ce métamodèle. Les trois métamodèles MM_A, MM_B et MM_T sont conformes au métamétamodèle qui est conforme à lui-même.

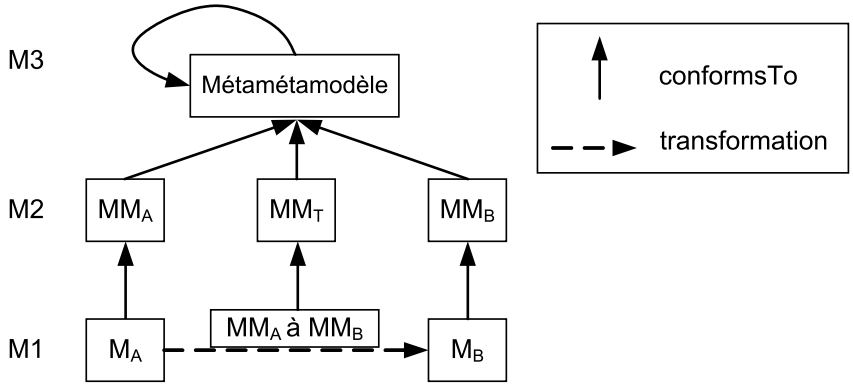

Figure 2.5 – La transformation de modèles

La plupart des principes présentés dans ces deux premières sections sur le MDA et sur les principes de bases sont partagés par plusieurs mises en œuvre de l'ingénierie des modèles. Par exemple, le MIC possède un métamétamodèle appelé MetaGME et un langage de transformation nommé GReAT (pour *Graph Rewriting and Transformation* ce qui signifie réécriture de graphe et transformation) qui sont implémentés dans l'environnement GME [33] (pour *Generic Modeling Environment* ce qui signifie environnement générique de modélisation). Similairement, les usines à logiciel sont supportées par l'environnement *DSL Tools* (outils pour langages dédiés) de Microsoft qui intègre son propre métamétamodèle. Les principales différences entre ces environnements de modélisation se situent au niveau des standards ou recommandations utilisés et des choix techniques, pas des principes.

2.2.3 Relations avec la technologie des objets

La technologie des objets a en son temps occasionné une rupture avec les techniques pré-existantes (procédural, fonctionnel, etc.). Le concept d'objet associé aux concepts de classe, de méthode ainsi qu'aux relations d'héritage et d'instanciation offre en effet des notions très différentes de celles de la technologie procédurale. Similairement, l'ingénierie des modèles se base sur des concepts fondamentalement différents de ceux de l'objet. Mais dans ce cas aussi, ceci ne signifie pas nécessairement que les modèles vont remplacer les objets à tous les niveaux. Certains modèles peuvent même être utilisés pour représenter des systèmes à objets (par exemple UML), auquel cas les deux approches se complémentent.

La figure 2.6 représente les concepts et relations de base en technologie des objets. Un objet ou instance (*Objet*) est une instance d'une classe (*Classe*). Ceci est représenté par la flèche *instanceOf* qui signifie *instance de*. Une classe hérite d'une classe mère, ce qui est représenté par la flèche *inheritsFrom* qui signifie *hérite de*.

De même qu'il existe des ponts entre technologies objet et procédurale, il existe aussi des ponts entre ingénierie des modèles et technologie objet. Ainsi JNI (Java Native Interface ou interface Java native) est un système permettant à du code écrit en Java d'appeler du code écrit en n'importe quel langage, y compris des langages procéduraux comme le C. Un autre pont, de nature différente puisque reliant des technologies différentes, existe pour relier le langage Java à l'ingénierie des modèles : JMI [68] (Java Metadata Interface), basé sur la version 1.4 du MOF et proposé par Sun. Il ne s'agit pas ici d'appeler du

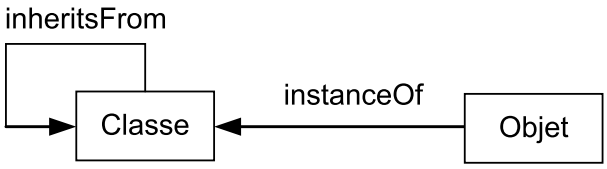

Figure 2.6 – Relations de base en technologie des objets

code “modèle” depuis du code Java, ce qui n’aurait pas vraiment de sens étant donné que les modèles ne sont justement pas du code. Mais, JMI permet à des programmes écrits en Java de manipuler des modèles. Ceci peut se faire soit de manière réflexive, c’est-à-dire indépendamment du métamodèle. Il est aussi possible de générer des interfaces et classes Java permettant de manipuler les éléments de métamodèles comme des classes et les éléments de modèles comme des objets. La technologie JMI est notamment implémentée par l’outil Netbeans/MDR. D’autres technologies similaires existent, par exemple Eclipse/EMF qui est basé sur le métamétamodèle Ecore, qui est une variante de EMOF 2.0.

Des langages généralistes tels que Java sont aujourd’hui utilisés pour construire des outils pour l’ingénierie des modèles. Il faut toutefois éviter de faire certains racourcis :

- Bien que JMI représente une classe MOF par une classe Java, ces deux notions sont fondamentalement différentes. Le fait que la notion de *concept* soit appelé *Class* en MOF contribue malheureusement à la confusion. Il y a de nombreuses différences entre classe MOF et classe Java. Nous en citons deux à titre d’illustration :
  - Une propriété (extrémité d’association ou attribut) en MOF ne peut pas être redéfinie dans une classe fille. Il est donc interdit de donner le même nom à deux propriétés accessibles depuis la même classe par héritage. En revanche, Java autorise la redéfinition de champ dans les classes filles : le nouveau champ masque le champ de la classe parente, qui est alors indirectement accessible à l’aide du mot-clé *super*.
  - MOF autorise l’héritage multiple de classes mais pas Java. JMI définit donc une correspondance entre classe MOF et interface Java et représente l’héritage de classe MOF par l’implémentation d’interfaces en Java, qui peut être multiple (une interface fille implémentant plusieurs interfaces mères). Les classes Java correspondant aux classes MOF implémentent alors ces interfaces. De plus, l’héritage Java à une influence sur le comportement des objets. Or, les éléments de modèles n’ont pas réellement de comportement propre.
- De même, JMI représente un élément de modèle par une instance Java (i.e. un objet) et la relation entre élément de modèle et élément du métamodèle par la relation d’instanciation Java. Mais, une classe Java n’est pas elle-même une instance d’une autre classe. L’API réflexive de Java fournit certes une classe *java.lang.Class* dont les instances représentent les classes Java, mais il n’y a pas identité entre les instances de cette classe et les classes Java. Or, les relations entre élément de modèle et élément de métamodèle, d’une part, et élément de métamodèle et élément de métamétamodèle, d’autre part, sont en fait les mêmes. Ce point sera détaillé dans le chapitre 4 et correspond au fait que la même relation de conformance existe entre modèle, métamodèle et métamétamodèle.

Ce bref parallèle entre Java et le MDA illustre la différence fondamentale qui existe entre technologie des objets et ingénierie des modèles. Afin d’éviter les confusions improductives, nous essayons donc d’utiliser des mots différents pour dénoter des concepts ou relations différents. Les noms donnés aux relations de conformance et de représentation vont dans ce sens. En revanche, il y a certains concepts qu’il est plus difficile de renommer de par leur omniprésence. Ainsi, nous conservons le concept de classe MOF tout en notant bien qu’il ne s’agit pas du concept homonyme de classe dans le monde de l’objet.

Ce qui est également apparu dans nos remarques précédentes est que les concepts sont différents mais que Java reste un standard de mise en œuvre. Il est donc nécessaire d'avoir une certaine correspondance entre les concepts d'objet et de modèle.

2.2.4 Besoins

En tant que technologie émergente, l'ingénierie des modèles n'est pas encore mature. Nos travaux contribuent à améliorer les connaissances sur cette technologie. Nous nous sommes concentrés sur la transformation de modèles qui est une opération essentielle. Certes, l'OMG dispose aujourd'hui de la recommandation QVT qui fournit trois langages de transformation proposant les paradigmes impératif et déclaratif. Mais aucun outil n'implémente encore QVT. De plus, nos travaux ont commencé au moment de l'émission de l'appel à proposition pour la définition de QVT [55]. Ils ont aussi contribué au débat autour de QVT, ce qui s'est entre autres concrétisé par la publication d'une version préliminaire de nos travaux dans la réponse faite à l'appel à proposition de l'OMG par le consortium *OpenQVT* en 2003 [63].

D'autres langages de transformation de modèles basés sur différents paradigmes ou différentes approches existent aussi (voir chapitre 3). Ceci montre que différentes approches sont utiles. QVT tente de résoudre ce problème en fournissant différents langages. Nous pensons que cela peut aussi signifier qu'il n'y a pas une unique solution au problème de la transformation de modèles. Il est ainsi possible d'envisager l'existence de multiples langages de transformation, chacun d'entre eux étant adapté à un type particulier de transformation. Il nous semble donc que la recherche sur les techniques de transformation de modèles n'est pas terminée et que de nombreuses contributions restent à faire.

Parmi les autres aspects de l'ingénierie des modèles qui sont encore peu développés, certains peuvent aussi freiner le développement de la transformation de modèles. Ainsi, il n'y a pas encore de formalisation de la notion de modèle ni de définition formelle d'un métamétamodèle. Or, ces deux éléments sont importants pour la compréhension des techniques de transformation de modèles.

2.3 Gestion de données hétérogènes

Il existe de plus en plus d'outils manipulant des données de plus en plus variées. De nombreux formats sont aussi utilisés pour représenter ces données. Or, peu d'outils se suffisent à eux-mêmes et il est très souvent nécessaire de travailler avec plusieurs sources de données, plusieurs systèmes de traitement de ces données et de transmettre les résultats obtenus à différentes destinations imposant des formats précis. L'approche préconisée par le W3C (World Wide Web Consortium) est d'utiliser un unique système de représentation des données : XML. C'est peut-être en effet la solution qui prévaudra dans l'avenir. Le MDA lui-même repose, avec XMI (XML *Model Interchange*), sur XML pour sérialiser ses modèles. La dernière version de cette recommandation est XMI 2.1 [60]. Cependant, il y a aujourd'hui beaucoup de systèmes existants (dits systèmes patrimoniaux ou *legacy* en anglais) qui traitent des données non-XML. De plus, XML étant relativement verbeux, il est fort probable que certains systèmes continuent malgré tout d'utiliser des formats plus adaptés aux utilisateurs tels que des langages de programmation basés sur les grammaires. Il est donc nécessaire d'avoir des outils de conversion de données hétérogènes.

Nous commençons par présenter un découpage possible de ce problème en deux étapes à la section 2.3.1. Ceci permet de faire intervenir l'ingénierie des modèles et plus particulièrement la transformation de modèles. Nous introduisons ensuite, à la section 2.3.2, la notion d'espace technique qui permet de prendre en compte différentes technologies. Puis, la section 2.3.3 donne quelques exemples d'espaces techniques. Enfin, nous identifions les besoins à la section 2.3.4.

2.3.1 Découpage du problème

Les problèmes de traduction de données (d'une source vers une cible) peuvent généralement être décomposés en deux étapes :

1. **Transformation des concepts.** Les concepts utilisés par la source et la cible peuvent être différents. Considérons, par exemple, la traduction du relationnel vers l'objet. Il est nécessaire de traduire des tables et colonnes en classes et attributs. De plus, la correspondance est rarement simple.
2. **Changement de système de représentation.** Lorsque les technologies utilisées pour représenter les données source et les données cible sont différentes, il est nécessaire de changer de système de représentation. Ainsi, dans l'exemple de la traduction du relationnel vers l'objet, il faut passer du système de représentation relationnelle à un programme conforme à la grammaire Java.

Certains cas peuvent se ramener à une unique étape. Ainsi, traduire des données bibliographiques de BIBTEX vers BIBTEXML [35] ne nécessite que de changer de système de représentation, les concepts étant identiques. De même, traduire du BIBTEXML en XHTML pour des besoins de présentation consiste uniquement en une traduction des concepts, la source et la cible utilisant toutes deux XML. Mais, il n'est pas toujours simple de faire ce découpage. Cependant, lorsqu'il est possible, il permet souvent de faire des simplifications car chaque étape à un problème mieux délimité. De plus, les outils traitant chaque étape peuvent parfois être réutilisés séparément.

L'ingénierie des modèles propose un système uniforme de représentation des modèles, qu'ils contiennent des données, du code, etc. Ces modèles sont conformes à des métamodèles qui définissent leurs syntaxes abstraites. La syntaxe concrète des modèles manipulés n'est pas imposée, mise à part l'existence de XMI qui est celle utilisée par défaut. L'ingénierie des modèles fournit aussi des systèmes de transformation manipulant les données en se reposant sur leurs métamodèles et donc indépendamment de leur syntaxe. Elle est donc dans une position favorable pour la transformation des concepts indépendamment de leur système de représentation.

2.3.2 Notion d'espace technique

La notion d'espace technique a été originellement définie dans [46] puis raffinée dans [13]. Nous réutilisons ici cette notion afin de structurer notre réflexion par rapport à la deuxième étape présentée ci-dessus : le changement de système de représentation. Certaines figures présentées dans cette section sont extraites de [13]. Nous reprenons ici la définition donnée dans [46] :

Définition 4. Un *espace technique* (ou *Technical Space* abrégé en TS) est un contexte de travail avec des concepts associés, un ensemble de connaissances, des outils, des compétences requises et des possibilités.

Cette définition fait apparaître des éléments humains puisque la plupart des techniques ont évolué dans des communautés humaines possédant certaines connaissances. De plus, une technique permet la création et la manipulation d'artéfacts. Certains de ces artéfacts sont des données dont le format est généralement imposé par la technique. Ce qui nous intéresse particulièrement ici est la structure de ces artéfacts et la manière dont ils sont associés. Ceci est en effet nécessaire afin de pouvoir traduire des données exprimées à l'aide d'une technique vers un format défini par une autre technique.

Dans [13], deux nouvelles hypothèses sont faites. Premièrement, il est supposé que les éléments de tous les espaces techniques peuvent être appelés modèles, grossièrement équivalente à celle d'artéfact. La figure 2.7 représente cette hypothèse. On y retrouve la notion de modèle représentant un système. Un

nouvel élément apparaît : l'espace technique qui fournit un système de représentation de modèles ainsi que des outils pour les manipuler.

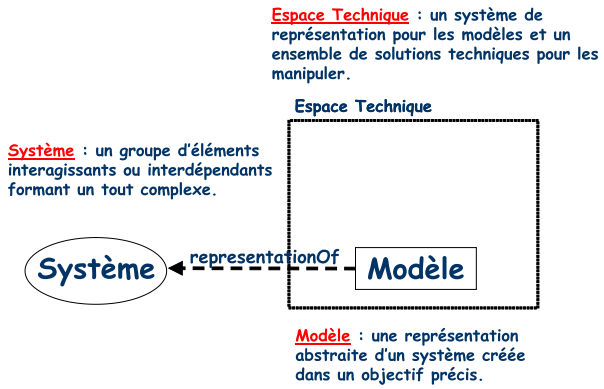

Figure 2.7 – Notion d'espace technique par rapport aux notions de modèle et de système

Puisqu'il existe un type de modèles par espaces techniques, il devient nécessaire de faire figurer l'espace technique dans l'appellation des modèles. On parle alors de λ-modèle, où λ dénote un espace technique (par exemple : XML-modèle, EBNF-modèle).

Deuxièmement, le fait de considérer les artéfacts comme des modèles permet de faire une seconde hypothèse. En effet, si certains artéfacts manipulés dans un espace technique donné sont des modèles, il est probable que d'autres artéfacts sont des métamodèles et peut-être même l'un des artéfacts est-il un métamétamodèle. Cette deuxième hypothèse est appelée conjecture des trois niveaux. Les auteurs de [13] supposent donc que plusieurs espaces techniques sont organisés comme celui de l'ingénierie des modèles (cf. figure 2.4).

La figure 2.8 donne une représentation de la conjecture des trois niveaux. Trois espaces techniques différents, appelés TS_A, TS_B et TS_C, sont représentés. Les modèles de chacun d'entre eux sont appelés respectivement *A-modèle*, *B-modèle* et *C-modèle*. Chaque modèle est conforme à un métamodèle du même espace technique. Enfin, il y a un unique métamétamodèle dans chaque espace technique auquel tous les métamodèles sont conformes. Il y a trois niveaux comme dans l'ingénierie des modèles : M1, M2 et M3.

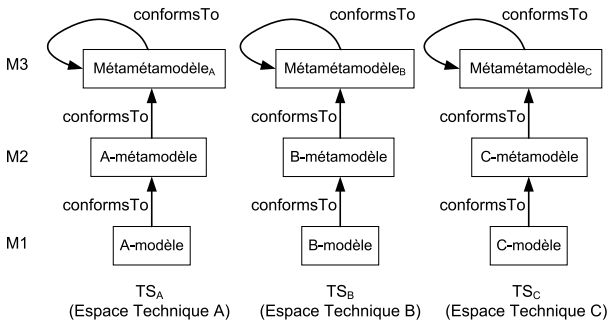

Figure 2.8 – La conjecture des trois niveaux

Il existe très probablement des technologies que la notion d'espace technique ne parvient pas à abstraire de manière satisfaisante ou ne peut tout simplement pas représenter. De même, la conjecture des

trois niveaux n'est probablement pas applicable à tout espace technique. Cependant, ces concepts permettent de considérer le problème de la gestion des données hétérogènes comme un pontage entre espaces techniques partageant la même structure, lorsque cela est possible. Dans les autres cas, des techniques ad hoc peuvent toujours être utilisées en attendant la définition éventuelle d'un principe unificateur plus fort que celui d'espace technique.

2.3.3 Exemples

Nous venons de voir la notion d'espace technique qui permet de considérer la gestion de données hétérogènes par rapport à une structuration relativement uniforme des différentes technologies. Nous allons maintenant voir quelques exemples d'espaces techniques. L'objectif est à la fois d'illustrer cette notion mais aussi de montrer qu'elle s'applique notamment à des technologies très répandues.

La figure 2.9 représente l'architecture en trois niveaux de quatre espaces techniques : MDA, XML, EBNF et RDF. La structure de cette figure est la même que celle de la figure 2.8, la différence étant qu'il s'agit ici d'espaces précis. Nous allons décrire les trois premiers en détail.

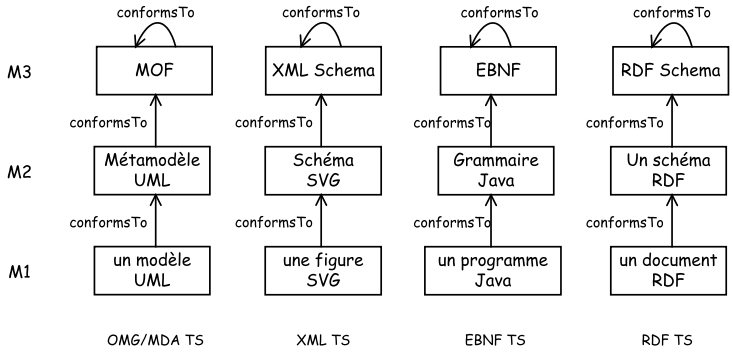

Figure 2.9 – Exemples d'espaces techniques à trois niveaux

2.3.3.1 L'espace technique de l'ingénierie des modèles

Le MDA, qui est une variante de l'ingénierie des modèles, a été décrit à la section 2.2.1 Sa représentation sur la figure 2.9 est quasi identique à celle donnée à la figure 2.2. Puisque la conjecture des trois niveaux est basée sur les principes de l'ingénierie des modèles, il est normal qu'elle s'applique sans problème à l'une de ses variantes. Le MDA est donc considéré ici dans le but de servir de référence pour la représentation des autres espaces techniques.

Afin de pouvoir aussi comparer la structure interne des modèles (i.e. leurs éléments) des différents espaces techniques, nous allons tout d'abord nous intéresser à celle d'une variante de l'ingénierie des modèles : les sNets. La figure 2.10 détaille la structure des modèles sNets. La différence essentielle avec le MDA réside dans le métamétamodèle. Sur cette version simplifiée, les seules différences sont terminologiques : la notion de concept s'appelle noeud (ou *node* en anglais) au lieu de classe et celle de relation entre concepts s'appelle lien (ou *link* en anglais). Chaque lien est dirigé : d'un noeud source (noté *inCom* pour *inComing* signifiant entrant) vers un noeud cible (noté *outGo* pour *outGoing* signifiant sortant).

Nous avons choisi de représenter les sNet plutôt que le MDA pour trois raisons :

- **Variété des illustrations.** Le MDA a en effet déjà été présenté. De plus, nous avons fait remarquer que sa terminologie hérite de certains concepts de l'objet tels que classe et attribut. Or, les entités appelées classe et attribut du MOF sont en réalité différentes de leurs homonymes du monde de l'objet. Puisque les sNets sont une variante de l'ingénierie des modèles qui n'est pas entachée par la confusion apportée par ces termes, il est intéressant de les prendre comme exemple. La notion de concept est ainsi appelée noeud (en anglais : *node*) et celui de relation entre concepts est appelé lien (en anglais : *link*).
- **Simplicité de la représentation.** Les sNets sont représentés à l'aide d'une syntaxe graphique. Chaque élément de modèle est représenté par un cercle (i.e. un noeud du graphe). L'étiquette propre (ou le nom) de chaque élément figure dans la partie basse du cercle. Le nom du concept dont un élément est une occurence figure dans la partie haute du cercle. Chaque occurence d'une relation entre éléments est représentée par une flèche (i.e. un arc du graphe) à côté de laquelle figure le nom de la relation. Cette syntaxe est très simple et s'applique de la même manière aux trois niveaux M1 à M3. Elle a l'inconvénient d'être verbeuse, c'est-à-dire qu'il faut rapidement beaucoup d'éléments visuels pour représenter même des modèles peu complexes. Elle est cependant suffisante pour notre illustration.
- **Multiplicité des solutions.** Le choix d'un métamétamodèle est important et il y a différentes solutions autres que le MOF. Le métamétamodèle est un langage qui peut être de différents types (objet, etc.).

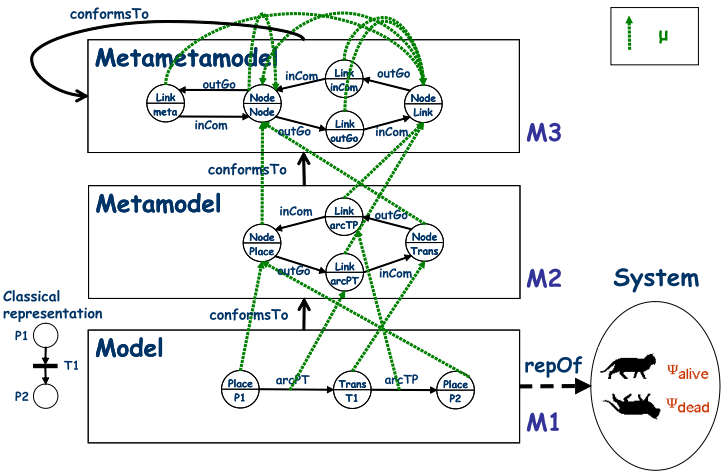

Figure 2.10 – Architecture à trois niveaux de l'espace technique sNet

Remarque. Une représentation identique pour les trois niveaux M1, M2 et M3 aurait aussi pu être utilisée pour le MDA. En effet, le MOF et UML peuvent être représentés par un diagramme de classes MOF et nous aurions pu limiter un exemple de modèle UML à l'utilisation des diagrammes de classes UML. Cette syntaxe visuelle est généralement moins verbeuse que celle des sNets car plusieurs éléments peuvent être représentés par un seul noeud du graphe. Par exemple, les différents attributs d'une classe ne sont pas représentés séparément de celle-ci mais en son sein. Il y a cependant deux problèmes qui nous ont décidés à éviter cette solution :

- Un diagramme de classes est certes une représentation graphique, mais il ne correspond pas directement au modèle. Ainsi, plusieurs éléments (par exemple une classe et ses attributs) sont en effet représentés par un seul noeud. De plus, un élément de type association est représenté par un arc.

C'est pour cela qu'un diagramme de classes est plus concis qu'une représentation sNet. Mais nous n'avons pas besoin de cette concision ici.

- Les diagrammes de classes utilisés pour représenter le métamodèle UML et le métamétamodèle MOF, d'une part, et un modèle UML, d'autre part, ne sont pas de même nature. En effet, une classe MOF représente un concept d'un métamodèle alors qu'une classe UML représente généralement une classe d'un système à objets (par exemple une classe Java).

Par ailleurs, la syntaxe des diagrammes d'objets est parfois utilisée pour représenter indifféremment des modèles, métamodèles ou le métamétamodèle. Il s'agit cependant d'une syntaxe concrète visuelle définie à l'origine pour les objets UML et son utilisation peut porter à confusion.

Nous allons maintenant décrire la figure 2.10. Nous considérons un système à deux états : chat vivant (que nous notons P1) et chat mort (noté P2). Le passage de P1 à P2 est possible mais le passage de P2 à P1 est considéré impossible. Nous choisissons de représenter ce système à l'aide du formalisme des réseaux de Petri. La présence d'un jeton dans la place P1 correspond à la présence d'un chat vivant et la présence d'un jeton dans P2 correspond à un chat mort. La transition T1 permet de passer d'un état à l'autre.

Au niveau M1, nous retrouvons trois éléments : les deux places (concept noté *Place*) P1 et P2 ainsi que la transition (concept noté *Trans*) T1. Deux arcs relient d'une part P1 à T1 (arc de type place vers transition noté *arcPT*) et d'autre part T1 à P2 (arc de type transition vers place noté *arcTP*). Notre choix de la représentation des sNets permet d'avoir une correspondance quasi directe entre le réseau de Petri (représenté en bas à gauche de la figure 2.10) et le modèle en sNet. Les flèches en pointillés représentent une relation différente de celles faisant partie des modèles : celle (appelée μ) liant un élément au concept dont il est une occurrence. Une définition précise de μ sera donnée au chapitre 4. Par exemple, P1 est relié au noeud *Place*, qui est relié au noeud *Node* (car c'est un concept), ce dernier étant relié à lui-même (le concept de concept). De même, l'arc entre P1 et T1 est relié au noeud *arcPT*, qui est relié au noeud *Link* (car c'est un lien), lui-même relié au noeud *Node*. La relation μ ne fait pas partie du graphe correspondant à chaque modèle, ce qui lui permet d'attacher aux arcs le noeud dont ils sont des occurences. Toutes les relations μ ne sont pas représentées dans l'objectif de simplifier la figure. Les relations entre arcs des niveaux M2 et M3 et noeuds de M3 sont notamment absentes.

2.3.3.2 L'espace technique XML

La représentation de l'espace technique XML donnée à la figure 2.9 se base sur le métamétamodèle XML Schema plutôt que sur les DTDs (*Document Type Definition* pour définition de type de document). Nous verrons dans le paragraphe suivant la raison de ce choix. Le schéma de XML Schema est lui-même, ce qui correspond à un métamétamodèle conforme à lui-même. La relation de conformance correspond ici grossièrement à la notion de validité d'un document XML par rapport à un schéma. SVG (*Scalable Vector Graphics* pour graphiques vectoriels passant à l'échelle) est un schéma conforme à XML Schema. SVG est utilisé pour définir des documents XML représentant des figures ou dessins vectoriels (i.e. composés de lignes et courbes plutôt que de points). Chaque figure définie en SVG est à son tour conforme au schéma SVG.

Si nous avions, au contraire, choisi de baser notre espace technique XML sur les DTDs, nous aurions rencontré un problème : la conjecture des trois niveaux ne pourrait pas s'appliquer. Les DTDs ne sont en effet pas des documents XML alors que les schémas le sont. Or, nous avons vu que l'architecture à trois niveaux de l'ingénierie des modèles est basée sur un système de représentation uniforme. Même si il est possible de dire qu'un document XML est conforme à une DTD, il n'est en revanche pas possible de dire qu'une DTD est conforme à une autre. Il n'est ainsi notamment pas possible d'avoir un métamétamodèle

conforme à lui-même. La structure des DTDs est en pratique souvent capturée par une entité provenant d'un autre espace technique : une grammaire[1].

La figure 2.11 donne une représentation avec la technologie XML du système qui a été présenté à la figure 2.10. Les places et transitions sont représentées par les éléments XML (respectivement `place` et `transition`) possédant un attribut `name` dont la valeur est leur nom. Les arcs sont représentés par les éléments `arcPT` (place vers transition) et `arcTP` (transition vers place) possédant deux attributs : `source` pour identifier la source de l'arc et `target` pour identifier sa cible. Il n'est pas possible de représenter le schéma complet pour des raisons de place. Nous voyons cependant sur l'exemple de l'élément `place` que celui-ci est défini comme élément à l'aide de l'élément `xs:element`. Le fait que `place` ait un attribut `name` est représenté dans son schéma par la présence d'un `xs:attribute` avec la valeur `name` pour son attribut `name`. De même, le schéma de réseau de Petri est conforme à XML Schema, qui est conforme à lui-même. L'élément `xs:element` est notamment une occurrence du concept `xs:element`.

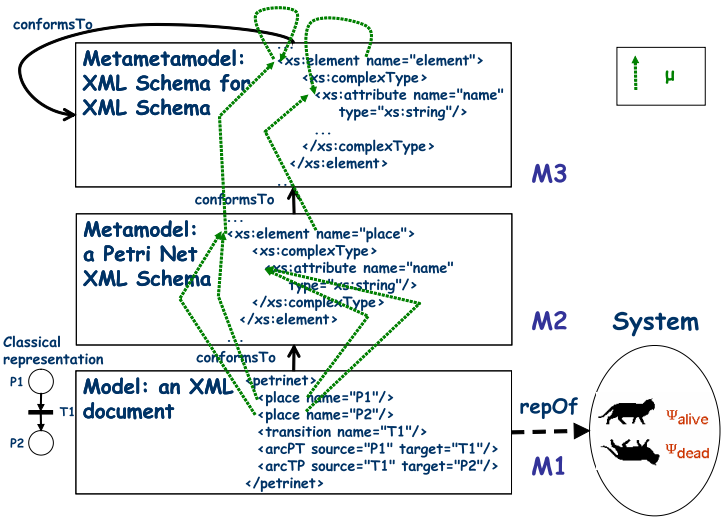

Figure 2.11 – Architecture à trois niveaux de l'espace technique XML

La technologie XML est en réalité plus compliquée que ce qui est présenté ici. Nous avons par exemple ignoré la notion d'espace de nommage (*namespace* en anglais). Il apparaît cependant que la conjecture des trois niveaux est applicable à cette technologie. Par ailleurs, nous avons vu l'importance de la relation de typage μ entre éléments du modèles et éléments du métamodèle.

2.3.3.3 L'espace technique EBNF

Nous appelons EBNF l'espace technique reposant sur la technologie des grammaires. Il existe en réalité plusieurs variantes de cet espace se différenciant par le choix du métamétamodèle. La notation EBNF est un choix possible, mais pas le seul. D'autres possibilités sont offertes, par exemple, par les grammaires attribuées.

L'utilisation de l'espace technique EBNF pour représenter le modèle déjà capturé par les figures 2.10 et 2.11 est donnée à la figure 2.12. L'application est très similaire à celle réalisée avec XML. Un

[1]Les DTDs dans l'espace technique XML comme les profils UML dans l'espace technique MDA peuvent être considérées similairement comme des solutions temporaires dans l'évolution progressive des technologies.

programme décrit le réseau de Petri à l'aide de mots-clés et symboles permettant d'identifier l'occurence d'une règle de production. Chaque règle est définie dans la grammaire de cette notation pour les réseaux de Petri. Cette grammaire est elle-même définie en utilisant la notation EBNF.

De même que MOF et XML Schema sont conformes à eux-mêmes, il est possible de donner une grammaire en EBNF de la notation EBNF. L'exemple de la figure 2.12 est certes simplifié, mais illustre l'applicabilité de la conjecture des trois niveaux à l'espace technique EBNF.

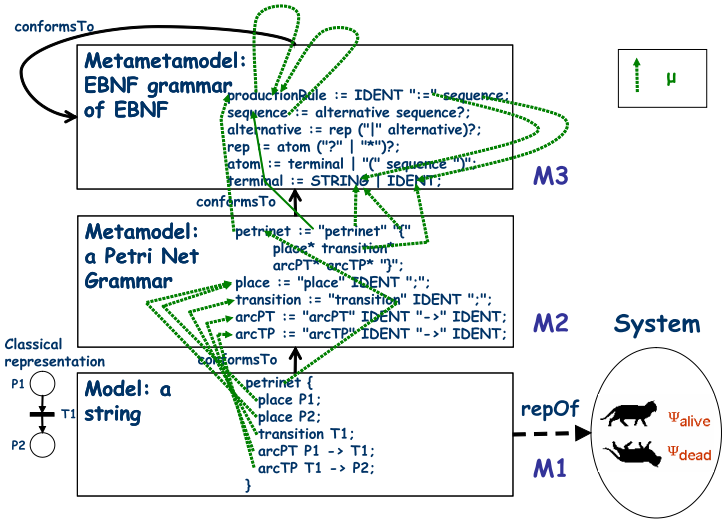

Figure 2.12 – Architecture à trois niveaux de l'espace technique EBNF

2.3.3.4 Autres considérations

Nous venons de voir la notion d'espace technique associée à la conjecture en trois niveaux. L'intérêt de ces concepts est qu'ils structurent différentes technologies suivant la même architecture. Nous avons ainsi montré que les technologies basées sur XML et sur les grammaires suivent cette structuration. D'autres technologies, telles que RDF, peuvent aussi être décrites de manière similaire.

Il y a cependant des différences entre les technologies évoquées. Ainsi, les technologies XML et EBNF sont basées sur des systèmes d'arbres alors que l'ingénierie des modèles est basée sur un système de graphes. Les arbres XML et les arbres EBNF sont différents. Par ailleurs, les technologies XML et EBNF s'intéressent particulièrement à la manière dont sont représentés leurs modèles sous forme textuelle. L'ingénierie des modèles, en revanche, ne dispose pas de représentation propre de ses modèles. Plusieurs de ses variantes, dont le MDA, le MIC et les usines à logiciel, utilisent en effet une représentation basée sur XML. Afin de prendre en compte ces traductions de modèles entre espaces techniques, les auteurs de [13] définissent les termes suivants : projecteur, injecteur et extracteur. Nous adaptons ici leurs définitions par rapport à celles données ci-dessus, notamment celle de la transformation de modèles.

Définition 5. Une *projection* est une transformation de modèles dont les modèles source et cible sont représentés à l'aide d'espaces techniques différents. Un projecteur est un outil réalisant une projection.

Définition 6. Une *injection* est une transformation dont le modèle cible est situé dans l'espace technique de référence choisi par son utilisateur. Un injecteur est un outil réalisant une injection.

Définition 7. Une *extraction* est une transformation dont le modèle source est situé dans l'espace technique de référence choisi par son utilisateur. Un extracteur est un outil réalisant une extraction.

Les définitions d'injection et d'extraction font appel à un espace technique de référence. L'objectif est de faciliter l'identification du sens de la projection dans le cas où on est amené à travailler avec un espace technique en particulier. Ainsi, nous choisissons l'espace technique de l'ingénierie des modèles comme référence. Une injection correspond donc dans ce document à une transformation résultant en un modèle de l'ingénierie des modèles. De la même manière, une extraction a pour source un modèle de cet espace technique.

Les outils basés sur la recommandation XMI sont donc des projecteurs. Ils font correspondre à chaque métamodèle MDA un schéma XML qui permet de faire correspondre à chaque modèle MDA un document XML. Un outil XMI est généralement bidirectionnel (il offre la possibilité d'injecter et d'extraire) car son utilité est de pouvoir sérialiser un modèle MDA sous une forme qui permet de le reconstruire ultérieurement. Un autre exemple de projection est la création d'un diagramme de classes représenté en SVG à partir d'un métamodèle. Il s'agit ici généralement d'une extraction car il est rare d'avoir besoin de créer un métamodèle à partir de rectangles et de flèches.

Remarque. La remarque donnée à la section 2.2.1 concernant la conformité de MOF à lui-même reste valable pour les autres techniques. Ainsi, bien que le document XML définissant XML Schema soit valide par rapport au schéma de XML Schema, ceci n'implique pas que le langage XML Schema soit auto-défini. En effet, la recommandation XML Schema définit en anglais les contraintes imposées par un schéma sur un document. Ces contraintes ne sont notamment pas prises en compte dans le schéma de XML Schema. Il en est de même pour la grammaire de la notation EBNF.

2.3.4 Besoins

Les notions d'espace technique et de projecteur permettent de gérer certains types de données hétérogènes. Il s'agit des données qui sont définies à l'aide de technologies pour lesquelles la conjecture des trois niveaux s'applique. Nous choisissons de ne considérer, pour l'instant, que ces technologies.

Bien que nous disposions de ces concepts pour structurer notre approche de la gestion des données hétérogènes, il nous manque encore des éléments. Nous appelons maintenant "projecteur" un outil de conversion de données entre espaces techniques. Il est cependant nécessaire d'aller plus loin que de nommer cet outil. Nous avons besoin d'outils pour définir ces projecteurs. Puisqu'une projection est une forme de transformation, il est possible que la transformation de modèles aide à résoudre ce problème.

2.4 Langages dédiés

Il y a plusieurs classifications possibles des langages informatiques : procéduraux ou à objets, déclaratifs ou impératifs, etc. L'une d'entre elles est la distinction entre langages généralistes (ou GPLs) tels que Java, C et langages dédiés (DSLs) tels que SQL, LaTeX et SVG. Les premiers sont utilisables pour résoudre tout type de problèmes tandis que les seconds sont limités à un domaine précis. Nous avons vu précédemment que l'ingénierie des modèles fournit un système de représentation uniforme pour les modèles mais aussi pour leurs métamodèles. Or, la notion de métamodèle est très proche de celle de langage. En effet, un métamodèle peut capturer les concepts et relations d'un langage. En revanche, d'autres aspects tels que la syntaxe, les contraintes et la sémantique ne sont généralement pas représentées dans un métamodèle.

Nous présentons donc ici l'ingénierie des langages dédiés en commençant par donner une définition à la section 2.4.1. Nous discutons ensuite des techniques d'implémentation des langages dédiés à la section 2.4.2. Enfin, nous identifions des besoins liés à l'ingénierie des langages dédiés à la section 2.4.3.

2.4.1 Définition

Dans [72], les auteurs proposent la définition suivante de langage dédié :

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Ce qui peut se traduire de la manière suivante :

> Un langage dédié (DSL) est un langage de programmation ou langage de spécification exécutable qui offre, à l'aide de notations et abstractions appropriées, une puissance d'expression concentrée sur, et généralement limitée à, un domaine de problèmes particulier.

Cependant, il existe de nombreux langages non-exécutables et limités à des domaines particuliers. Ainsi, le langage BIBTEX sert à représenter des bibliographies et les langages LATEX et HTML servent à représenter des documents textuels. De tels langages sont considérés par certains [50] comme des langages dédiés malgré leur non-exécutabilité. Afin de pouvoir aussi tenir compte de ces langages, nous proposons la définition suivante, qui n'impose pas qu'un langage dédié soit exécutable :

Définition 8. Un *langage dédié* (en anglais : *Domain Specific Language*, abrégé en DSL) est un langage permettant la spécification de solutions aux problèmes existant dans un domaine précis, à l'aide de notations et d'abstractions appropriées.

La limitation à un domaine précis implique souvent l'impossibilité d'utiliser un langage dédié pour une tâche située dans un autre domaine. En revanche, elle permet d'utiliser des concepts et abstractions proches du domaine. Ceci autorise parfois certaines optimisations ou vérifications qui ne seraient plus possibles avec du code généraliste. Avec un langage généraliste, il est possible de résoudre n'importe quel problème mais il faut pour cela assembler des éléments génériques. Par exemple, la définition d'un rectangle en SVG se réalise à l'aide de l'élément `<rect>`. Cet élément possède des attributs permettant de définir la couleur du trait et du remplissage du rectangle. Avec un langage tel que Java, il faut typiquement appeler une méthode pour définir la couleur du trait (`setStroke()`), puis une méthode pour celle du remplissage (`setPaint()`) avant d'appeler la méthode `drawRect()` pour dessiner le rectangle. Ceci impose de prendre en compte le flux de contrôle du programme afin, par exemple, de s'assurer de l'ordre correct d'appel des méthodes.

L'exemple Java donné ci-dessus n'est cependant pas aussi compliqué qu'il aurait pu l'être. Ainsi, avec un langage fournissant moins de bibliothèques (e.g. en langage machine), il pourrait être nécessaire de dessiner le rectangle ligne par ligne, ou même pixel par pixel. La raison pour laquelle Java n'impose pas au programmeur de manipuler directement les pixels est la présence d'une bibliothèque adaptée, appelée Java 2D (pour les deux dimensions : hauteur et largeur d'un dessin). Certains considèrent en conséquence qu'une bibliothèque est une forme particulière de langage dédié [50]. L'intérêt de cette approche est qu'elle permet d'utiliser des concepts du domaine depuis un langage généraliste. Il est donc possible de combiner les concepts de plusieurs domaines en utilisant le langage généraliste avec différentes bibliothèques. En revanche, de tels langages dédiés n'ont plus vraiment de limite. Ainsi, il n'est généralement pas possible de s'appuyer sur le domaine de la bibliothèque pour analyser le code, par exemple afin de réaliser des vérifications ou optimisations.

Unix est un exemple de plateforme reposant sur les langages dédiés. Ses concepts unificateurs sont les notions de processus et de fichier, et non celui de modèle. Unix fournit un grand nombre d'outils simples de manipulation de fichiers. Beaucoup de ces outils reposent sur des langages dédiés. Ainsi, les outils `sed` et `awk` de filtrage de texte reposent chacun sur un langage dédié basé sur les expressions rationnelles (en anglais *regular expressions* abrégé en *regexp* ou *regex* et parfois traduites en expressions régulières). L'interpréteur de commande (ou *shell* en anglais) permet de construire des chaînes de transformations en combinant ces différents outils. Le shell est lui-même basé sur un langage dédié à cette tâche. Chacun de ces langages est parfois aussi appelé petit langage (*little language* en anglais).

2.4.2 Techniques d'implémentation

En raison de leur limitation à un domaine particulier, il est souvent nécessaire de disposer de plusieurs langages dédiés. Plus les domaines considérés sont restreints, plus l'expressivité des langages qui y sont dédiés est grande. Ceci signifie qu'une approche basée sur les langages dédiés doit fournir un grand nombre de petits langages. C'est notamment le cas de la plateforme Unix.

Or, les techniques d'implémentation utilisées aujourd'hui pour les langages généralistes sont relativement coûteuses. Les compilateurs et environnements de développement sont souvent développés à l'aide d'un langage généraliste. Par exemple, la suite de compilateur GNU (i.e. `gcc`) ou encore l'environnement de développement Eclipse, y compris le compilateur Java qui l'accompagne. C'est même souvent le cas des analyseurs syntaxiques qui pourraient pourtant s'appuyer sur la technologie des grammaires et générateurs d'analyseurs. Un tel choix est souvent justifié pour un langage généraliste car il apporte certains avantages. Nous en listons deux ici à titre d'illustration :

- Un programme écrit par un programmeur (e.g. un analyseur syntaxique) est généralement plus rapide qu'un outil similaire généré automatiquement (e.g. à partir d'une grammaire). Le temps passé à optimiser le code est cependant très long. Puisqu'un langage généraliste est utilisé pour définir de nombreux systèmes dans leur intégralité, il n'est pas déraisonable de dépenser beaucoup de ressources pour gagner en vitesse sur la compilation d'un langage généraliste. Ceci n'est pas nécessairement vrai pour tous les langages dédiés.
- Un programme généré automatiquement est souvent limité en fonctionnalités. Par exemple, un analyseur syntaxique écrit à la main peut être conçu pour être incrémental. C'est-à-dire qu'il peut n'analyser qu'une partie d'un fichier lors de modifications. Ceci est très important pour les environnements de développement disposant de complétion de code (e.g. proposant la liste des identificateurs commençant par les caractères déjà tapés).

En revanche, l'implémentation des langages dédiés ne peut souvent pas être aussi coûteuse. Elle nécessite donc l'utilisation de techniques générant au moins une partie des outils nécessaires à un langage dédié (e.g. compilateur, environnement de développement). La notion de *language workbench* (en français : *établi à langages* ou plateforme de construction de langages dédiés) correspond à ce besoin. Microsoft DSL Tools est un exemple d'une telle plateforme, basée sur l'approche des usines à logiciel [34].

2.4.3 Besoins

Les plateformes de construction de langages dédiés évoquées ci-dessus ne sont pas encore matures. Nous pensons que l'ingénierie des modèles peut fournir certains éléments pour le développement de telles plateformes. Il est cependant nécessaire d'explorer les possibilités ainsi offertes et de définir les architectures de ces "établis à langages".

2.5 Conclusion

Ce chapitre a présenté l'ingénierie des modèles, la gestion des données hétérogènes ainsi que l'ingénierie des langages dédiés. Les besoins suivants ont été identifiés :

- Les principes de base de l'ingénierie des modèles sont implémentés dans différents environnements, notamment dans différentes recommandations de l'OMG, mais ils ne sont pas encore formalisés.
- L'ingénierie des modèles a besoin de langages de transformation de modèles.
- Des systèmes de définition de projecteurs entre espaces techniques peuvent aider à gérer des données hétérogènes.
- Des plateformes de construction de langages dédiés sont nécessaires au développement de l'ingénierie des langages dédiés.

Nous avons également vu que grâce à la notion d'espace technique, l'ingénierie des modèles peut servir à "ponter" différentes technologies. L'ingénierie des modèles peut aussi servir à définir des plateformes de construction de langages dédiés.

CHAPITRE 3

État de l'art[a]

[a] Le travail présenté dans ce chapitre est partiellement adapté de [42].

3.1 Introduction

Ce chapitre présente un état de l'art des langages de transformation de modèles dans le contexte de l'ingénierie des modèles. En ce qui concerne la métamodélisation et les syntaxes concrètes, les chapitres 5 sur KM3 et 6 sur TCS présentent chacun un ensemble de travaux similaires auquels ces approches sont comparées.

La transformation de modèles a été définie à la section 2.2.2 et son contexte présenté à la figure 2.5. Nous considérons ici cinq approches de transformation auxquelles nous comparons ATL. Il s'agit de la recommandation QVT, du langage Tefkat et de trois approches basées sur la transformation de graphes : VIATRA2, GReAT (*Graph Rewriting and Transformation*) et AGG (*Attributed Graph Grammar*). De plus, QVT est composé de trois langages : *Relations*, *Core* et *Operational Mappings*. De plus, nous tenons compte à la fois d'ATL et du langage de sa machine virtuelle (ATL VM pour *ATL Virtual Machine*) qui seront présentés au chapitre 7.

Le chapitre est organisé de la manière suivante. La section 3.2 présente les critères de comparaison des langages. La section 3.3 présente la recommandation QVT. La section 3.4 décrit le langage Tefkat. Les approches basées sur la transformation de graphes sont détaillées à la section 3.5. Les différents langages sont comparés à la section 3.6. Enfin, la section 3.7 conclut.

3.2 Critères de comparaison

Afin de comparer les différents langages présentés dans la suite de ce chapitre, nous nous basons sur un ensemble de critères, organisés en différentes catégories. Un exemple d'une telle catégorie est le paradigme de programmation utilisé par un langage : impératif, déclaratif ou hybride. Un autre exemple est le langage de reconnaissance de motifs. Certains langages utilisent des motifs simples alors que d'autres utilisent des expressions OCL complexes. Une table de comparaison basée sur les critères présentés ici est donnée à la section 3.6.

La plupart des critères et catégories sont dérivés de [16], [31] et [42]. Ils sont présentés dans la liste suivante :

- **Scénarios de transformation.** Nous considérons les scénarios suivants, directement dérivés des langages analysés :
  - *Synchronisation de modèles.* Dans ce scénario, deux modèles existants sont synchronisés d'après un ensemble donné de relations. Les changements nécessaires sont appliqués aux modèles.
  - *Vérification de correspondance.* Dans ce scénario, le moteur vérifie si deux modèles existants satisfont un ensemble de relations. Les modèles ne sont pas modifiés.

  - *Transformation de modèles.* Dans ce scénario, un ensemble de modèles cible est produit à partir d'un ensemble de modèles source.
  - *Modifications en place.* Il s'agit d'un cas particulier de transformation de modèles dans lequel certains modèles source sont aussi des modèles cible.
  - *Transformation interactive.* Il s'agit ici aussi d'un cas particulier de transformation dans lequel l'utilisateur peut intervenir pendant l'exécution. Puisque le processus n'est pas entièrement automatique, il ne correspond pas tout à fait à la définition 3 donnée au chapitre 2. Nous tenons cependant compte de ce scénario ici car il est disponible dans l'un des langages considérés dans ce chapitre.
- **Paradigme.** Nous considérons les paradigmes *impératif*, *déclaratif* et *hybride* pour la définition de transformations.
- **Directivité.** Cette catégorie indique la direction dans laquelle une transformation peut être exécutée. Nous considérons les transformations *unidirectionnelles*, dans lesquelles les modèles ne peuvent pas changer de rôle (source ou cible), et les transformations *multidirectionnelles* dans lesquelles ces rôles des modèles peuvent être changés sans modifier la transformation.
- **Cardinalité.** La cardinalité indique le nombre de modèles source et cible d'une transformation.
- **Traçabilité.** Les liens de traçabilité permettent de mémoriser les correspondances entre éléments source et cible pendant (et parfois aussi après) l'exécution d'une transformation. En général, il y a deux types de traçabilité : *automatique* et *définie par l'utilisateur*. La traçabilité automatique est supportée par les constructions du langage et le moteur d'exécution. Quant à la traçabilité définie par l'utilisateur, elle impose au développeur de transformations de créer et d'utiliser lui-même les liens de traçabilité. Dans certains cas, les structures de données utilisées pour ces liens peuvent aussi être définies par l'utilisateur.
- **Langage de navigation.** Généralement, un langage de transformation fournit des moyens de sélectionner des éléments dans les modèles source et d'accéder à leurs propriétés. Nous appelons cette fonctionnalité : navigation dans les modèles. Un langage spécifique (tel que OCL) peut être utilisé, mais il ne s'agit pas nécessairement d'un langage défini séparément du langage de transformation.
- **Ordonnancement des règles.** Il s'agit de l'ordre d'application des règles en supposant que la notion de règle est l'unité de modularisation de base du langage. Il y a deux formes d'ordonnancement des règles : *implicite* et *explicite*. L'ordonnancement implicite est basé sur les dépendances implicites entre règles. On le retrouvre généralement dans les langages déclaratifs. L'ordonnancement explicite requiert une spécification explicite de l'ordre d'application des règles. Il en existe deux sortes : *interne* et *externe*. Dans le cas de l'ordonnancement explicite interne, le flux de contrôle est spécifié dans les règles. Au contraire, l'ordonnancement explicite externe sépare les règles de la spécification de leur ordre d'application. Cette dernière forme est souvent trouvée dans les langages de transformation de graphes.
- **Organisation des règles.** Cette catégorie concerne les relations entre règles. Il peut s'agir d'héritage de règles, de groupement de règles ou encore de relations de priorité.
- **Réflexion.** Certains langages offrent des possibilités de réflexion. Les différentes formes de réflexion observées dans les langages de transformations présentés ici seront détaillées au moment de décrire ces langages.

3.3 QVT

L'appel à proposition QVT [55] demandait un langage capable d'exprimer les requêtes, vues et transformations sur des modèles dans le contexte de l'architecture de modélisation MOF 2.0 [62]. Les différentes réponses qui y ont été donné ont convergé vers la recommandation QVT adoptée en 2005 [61]. Dans cette section, nous donnons un aperçu de QVT en nous concentrant sur son architecture et sur les points de conformité pour les outils QVT.

3.3.1 L'architecture de QVT

Le contexte opérationnel de QVT est celui qui a été présenté à la figure 2.5 avec MOF 2.0 comme métamétamodèle et le métamodèle QVT à la place de MM_T. La syntaxe abstraite de QVT est définie par ce métamodèle. Trois sous-langages pour la transformation de modèles y sont définis. OCL 2.0 [57] est utilisé pour naviguer les modèles. La création de vues sur les modèles n'est pas explicitement adressée dans la recommandation.

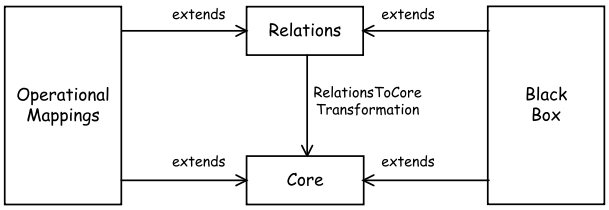

Figure 3.1 – L'architecture en couches des sous-langages de QVT

Les trois sous-langages de QVT forment collectivement un langage de transformation hybride : avec des constructions déclaratives et impératives. Ces langages sont appelés *Relations*, *Core* et *Operational Mappings*. Ils sont organisés en une architecture présentée à la figure 3.1.

Les langages *Relations* et *Core* sont tous deux déclaratifs mais placés à différents niveaux d'abstraction. La recommandation définit leurs syntaxes concrètes textuelles ainsi que leurs syntaxes abstraites. De plus, *Relations* a une syntaxe graphique. *Operational Mappings* est un langage impératif qui étend *Relations* et *Core*.

Le langage *Relations* offre la possibilité de spécifier des transformations comme des ensembles de relations entre modèles. Chaque relation contient un ensemble de motifs d'objets. Ces motifs peuvent être reconnus dans les modèles source, créés dans les modèles cible ou encore utilisés pour appliquer des changement aux modèles à la fois source et cible (i.e. pour des modifications en place). Le langage gère automatiquement la manipulation des liens de traçabilité et cache les détails associés au développeur.

Le langage *Core* est déclaratif et plus simple que *Relations*. Les définitions de transformations écrites en *Core* ont tendance à être plus longues que celles, équivalentes, écrites en *Relations*. Les liens de traçabilité sont traités comme des éléments de modèles ordinaires. Le développeur est responsable de leur création et utilisation. L'un des buts de *Core* est de fournir une base pour la spécification de la sémantique de *Relations*. La sémantique de *Relations* est donnée comme une transformation *Relations* vers *Core*. Cette transformation peut être écrite en *Relations*.

Il est parfois difficile de définir une solution complètement déclarative à un problème de transformation donné. Pour adresser cette question, QVT propose deux mécanismes pour étendre *Relations* et *Core* : un troisième langage appelé *Operational Mappings* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou *black box*).

Operational Mappings étend *Relations* avec des constructions impératives et des constructions OCL avec effets de bord. L'idée de base est que les motifs d'objets spécifiés dans les relations sont instanciés en utilisant les constructions impératives. De cette manière, les relations définies déclarativement sont implémentées impérativement. La syntaxe de *Operational Mappings* fournit les constructions communes des langages impératifs (boucles, instructions conditionnelles, etc.).

Le mécanisme boîte noire permet de brancher et d'exécuter du code externe au cours de l'exécution d'une transformation. Ce mécanisme permet l'implémentation d'algorithmes complexes dans n'importe quel langage et permet de réutiliser des bibliothèques existantes. Ceci rend certaines parties de la transformation opaques et représente un danger potentiel puisque la fonctionnalité apportée est arbitraire et non contrôlée par le moteur de transformation.

3.3.2 Points de conformité de QVT

La figure 3.1 ne suggère aucune implémentation particulière d'un moteur de transformation QVT. Les fournisseurs d'outils peuvent choisir différentes stratégies. Par exemple, *Core* peut être supporté par un moteur d'exécution et les transformations écrites en *Relations* peuvent être transformées en leur équivalents en *Core*. De cette manière, le moteur peut exécuter des programmes écrits dans ces deux langages. Une autre possibilité est que seuls *Relations* et *Operational Mappings* soient supportés par un outil donné. Dans ce cas, *Core* ne sert que de point de référence pour la définition de la sémantique de *Relations*.

Ces options d'implémentation peuvent produire des outils avec différentes capacités. Pour représenter les capacités des outils, la recommandation QVT définit un ensemble de points de conformité. Ces points sont organisés selon deux dimensions et forment une grille avec douze cellules. La table 3.1 montre ces dimensions et les points de conformité possibles.

		Dimension d'interopérabilité			
		Exécution de la syntaxe	Exécution du XMI	Export de la syntaxe	Export du XMI
Dimension des langages	*Core*				
	Relations				
	Operational Mappings				

Table 3.1 – Points de conformité des outils QVT

La dimension des langages définit trois niveaux correspondant aux trois langages de QVT : *Core*, *Relations* et *Operational Mappings*. Si un outil est confome à un certain niveau, cela signifie qu'il est capable d'exécuter les transformations écrites dans le langage correspondant.

La dimension d'interopérabilité concerne le format dans lequel une transformation est exprimée. Elle définit quatre niveaux :

- **Exécution de la syntaxe.** Un outil peut lire et exécuter les transformations écrites avec la syntaxe concrète donnée dans la recommandation QVT.
- **Exécution du XMI.** Un outil peut lire et exécuter les transformations sérialisées avec les règles XMI [60]. Les transformations QVT sont en effet conformes au métamodèle QVT et sont donc sérialisables en XMI.
- **Export de la syntaxe.** Un outil peut exporter les transformations dans la syntaxe concrète du langage correspondant.

- **Export du XMI.** Un outil peut exporter les transformations dans le format XMI.

Une condition additionnelle est posée : un outil capable d'exécuter la syntaxe ou le XMI pour un certain niveau de langage doit aussi être capable d'exporter respectivement la syntaxe ou le XMI.

3.4 Tefkat

Tefkat [47] est un langage de transformation déclaratif basé sur la réponse du DSTC [18] à l'appel à propositions QVT et sur le langage de transformation décrit dans [19]. Son contexte opérationnel est celui présenté à la figure 2.5 avec EMF/Ecore [14] comme métamétamodèle.

Les modèles sur lesquels opèrent les transformations Tefkat sont appelés *extents* en anglais. Ces *extents* sont des ensembles d'éléments de modèles qui correspondent aux modèles d'entrée et de sortie des relations de traçabilité. Il y a trois types d'*extents* : source, cible et traçabilité. Les *extents* source sont construits à partir des modèles source de la transformation. Ils peuvent seulement être navigués par les règles (i.e. ils sont en lecture seule). Les *extents* cible sont composés des éléments de modèles créés pendant l'exécution de la transformation. Ils sont en écriture seule. Un *extent* de traçabilité est un *extent* spécial qui peut être à la fois navigué et modifié par les règles de transformation. Son but est de maintenir les liens de traçabilité entre les *extents* source et cible. Les éléments des *extents* de traçabilité sont des instances des classes de traçabilité définies dans la transformation.

Les règles de transformation déclaratives et unidirectionnelles sont l'unité de modularisation de base en Tefkat. Elles ont deux parties formées par les motifs. Les motifs peuvent être définis séparément et utilisés dans plusieurs règles en les référençant par leur nom. Les motifs peuvent faire référence les uns aux autres et être récursifs. Les motifs sont soit reconnus sur les *extents* source et ils sélectionnent ainsi les éléments sur la base de certaines contraintes ou bien ils sont utilisés pour créer ou s'assurer de l'existence d'éléments de modèles dans les *extents* cible.

Les règles de transformation sont faiblement couplées : elles ne font pas directement référence les unes aux autres. Une règle peut utiliser les résultats produits par d'autres règles en naviguant les liens de traçabilité. Une classe de traçabilité peut être utilisée par de multiples règles.

Puisque les *extents* source sont en lecture seule, le scénario de modifications en place n'est pas supporté. Les transformations Tefkat sont unidirectionnelles. Le langage et le moteur supportent un scénario dans lequel aucun *extent* n'est créé mais seules les contraintes sont vérifiées sur les *extents* source.

Tefkat supporte la réflexion sous trois formes. La première est un accès générique aux propriétés et métaclasses des éléments de modèles pour laquelle Tefkat utilise l'API réflexive de EMF/Ecore. La seconde forme permet la spécification d'expressions aux endroits où une classe ou bien une propriété est attendue. La troisième forme utilise la construction *AnyType* qui permet de sélectionner n'importe quel élément de modèle indépendamment de son type concret.

Ces trois constructions réflexives permettent notamment la définition de transformations de copie de modèles de manière générique (i.e. fonctionnant pour tout métamodèle) et concise. Comme nous venons de le voir, les capacités réflexives de Tefkat ne concernent que les éléments de modèles. Nous n'avons pas d'information concernant d'autres capacités telles que la navigation de la transformation et de ses règles ou le changement du comportement de la transformation à l'exécution (i.e. intercession).

Le moteur de Tefkat peut être utilisé comme extension de la plateforme Eclipse. L'algorithme d'exécution s'inspire de l'unification de Prolog. Ceci est conforme à la théorie logique sur laquelle est basée le langage Tefkat.

3.5 Approches basées sur la transformation de graphes

Cette section donne d'abord une vue d'ensemble des approches basées sur la transformation de graphes en 3.5.1. Puis, les langages VIATRA2, GReAT et AGG sont repectivement présentés en 3.5.2, 3.5.3 et 3.5.4.

3.5.1 Vue d'ensemble

Les langages de transformation de graphes opèrent sur un graphe en le réécrivant. L'exécution d'une transformation est composée de pas opérant sur un graphe courant. Ce graphe courant est initialemment le graphe source puis évolue à chaque pas. Chaque pas correspond à l'application d'une règle. Quand aucune règle ne peut plus être appliquée, le graphe courant correspond au graphe cible.

Chaque règle est composée d'une partie gauche à reconnaître dans le graphe courant et d'une partie droite qui définit les changements à réaliser. En général, plus d'une règle est applicable au graphe courant. Le choix de la règle à appliquer peut être non-déterministe ou être dirigé par : des priorités, un flux de contrôle, un ordonnancement explicite des règles, etc. Chaque règle peut aussi avoir des conditions supplémentaires d'application.

Puisque les modèles sont des graphes, ces approches sont utilisables pour transformer des modèles. Certaines d'entre elles n'utilisent pas de métamodèle défini en MOF mais des traductions peuvent généralement être réalisées. Des parties du graphe courant sont conformes au métamodèle source alors que d'autres parties sont conformes au métamodèle cible. De plus, les liens de traçabilité entre éléments source et cible sont aussi représentés par des éléments du graphe courant. Ceci signifie que le graphe courant est conforme à un métamodèle commun composé des métamodèles source et cible ainsi que des éléments utilisés pour relier les éléments cible aux éléments source.

Les règles de transformation de graphes sont souvent représentées à l'aide de notations graphiques. Mais certains langages offrent aussi une notation textuelle. De plus, certaines parties des règles (e.g. leurs gardes) peuvent parfois être définies dans un langage textuel.

3.5.2 VIATRA2

VIATRA2 [5, 71, 74] est un langage de transformation unidirectionnel principalement basé sur des techniques de transformation de graphes. Il n'est pas basé sur les métamétamodèles MOF ou Ecore. Le langage opère sur des modèles exprimés à l'aide de l'approche de modélisation VPM [73]. Dans cette approche, il est possible d'avoir un nombre arbitraire de niveaux de modélisation et la relation de typage entre éléments de modèles et éléments de métamodèles est représentée explicitement. Il est cependant possible de représenter l'architecture à trois niveaux présentée au chapitre 2 avec VPM et donc de transformer des modèles l'utilisant (e.g. des modèles MOF).

VIATRA2 est composé de trois sous-langages qui forment un tout cohérent. Ces langages sont les suivants :

- **Langage de motifs de graphes.** Ce langage est utilisé pour exprimer les motifs à reconnaître pour sélectionner des éléments dans les modèles source. Ces motifs peuvent en réutiliser d'autres et former une récursion.
- **Langage de règles de transformation.** Il s'agit d'un langage pour exprimer les règles de réécriture de graphe. Chaque règle a une partie gauche et une partie droite qui sont toutes deux des motifs. Les règles sont unidirectionnelles. Une règle peut créer de nouveaux éléments, supprimer une partie des éléments reconnus et en conserver une autre partie.

- **Langage des machines d'état abstraites.** Il n'y a pas d'ordre prédéfini d'exécution des règles de transformation. L'ordre est spécifié en utilisant des machines d'état abstraites (*Abstract State Machines* ou ASMs). Le langage des ASMs fournit un ensemble de structures de contrôle de flux : séquencement, appel de règle, conditions, itération jusqu'à obtention d'un point fixe, etc. Ces constructions sont utilisées en ASMs d'une manière similaire aux méthodes des langages à objets.

Nous pouvons considérer VIATRA2 comme un langage hybride. Le langage de règles de transformation est déclaratif mais les règles ne peuvent pas être exécutées sans une stratégie d'exécution spécifiée de manière impérative. Le langage permet aussi la génération de code à l'aide de patrons de génération.

Il est possible de définir des patrons génériques de règles en VIATRA2. Dans ces règles, les classes sont des paramètres qui peuvent être substitués lors de l'instanciation du patron. Cette instanciation est réalisée par une méta-transformation selon les termes de VIATRA2. Il s'agit d'une transformation d'ordre supérieure qui traduit la transformation générique en une transformation simple.

L'environnement d'exécution fournit à la fois un interprète et un compilateur. Les transformations peuvent être interprétées pour des besoins de test ou de simulation. Ceci repose sur le mécanisme d'exécution des ASMs. De plus, les transformations peuvent être compilées vers un environnement cible donné. Il est possible d'invoquer des fonctions natives implémentées en Java. Ce mécanisme est similaire au mécanisme boîte noire de QVT.

3.5.3 GReAT

GReAT [45] permet la définition de transformations unidirectionnelles de plusieurs modèles source vers plusieurs modèles cible. Ces modèles sont conformes à des métamodèles spécifiés dans une notation proche de UML et peuvent être créés avec GME [33]. Chaque métamodèle est appelé un domaine. Les informations de traçabilité sont conformes à un métamodèle inter-domaines défini par l'utilisateur. Ce métamodèle peut aussi être utilisé pour étendre les domaines source et cible avec des éléments spécifiques à la transformation (e.g. des classes additionnelles). Le graphe courant est appelé graphe hôte.

GReAT utilise principalement une notation graphique. Cependant, certaines parties sont spécifiées textuellement : les expressions d'initialisation des attributs et les gardes. GReAT est composé des trois sous-langages suivants :

- **Langage de spécification de motifs.** Un motif est un graphe qui est reconnu dans le graphe source. Le détecteur de motifs de GReAT ne fonctionne pas sur la totalité du graphe hôte mais commence à travailler à partir d'un ensemble d'éléments déjà reconnus, ce qui est plus efficace. L'algorithme correspondant ne fonctionne cependant que sur des graphes connectés. La langage de spécification de motifs supporte des cardinalités (i.e. répétition d'éléments) et le groupement de sous-motifs.
- **Langage de transformation de graphes.** Les règles de transformation sont spécifiées dans le langage de transformation de graphes, qui est une extension du langage de spécification de motifs. Chaque élément d'un motif est associé à un rôle en fonction de son existence avant et après l'application de la règle. Le rôle *bind* correspond aux éléments existant avant et après. Le rôle *delete* est utilisé pour les éléments qui existent avant mais pas après (i.e. ils sont supprimés par la règle). Le rôle *new* spécifie des éléments qui n'existent qu'après (i.e. ils sont créés par la règles). Des gardes peuvent être associées aux règles sous la forme d'expressions booléennes en C++. L'initialisation des attributs des nouveaux éléments fait aussi usage d'expressions en C++.
- **Langage de flux de contrôle.** L'ordre d'application des règles est spécifié à l'aide d'un langage impératif dédié, qui fournit des constructions de séquencement, d'itération et d'application conditionnelle. De plus, des groupes d'associations (appelés paquets) entre éléments du graphe de la

règle et éléments du graphe hôte sont passés entre les règles afin de fournir l'ensemble d'éléments déjà reconnus nécessaire au détecteur de motifs. Ceci implique aussi un séquencement de l'exécution.

En fonction de la manière dont les règles et le flux de contrôle sont spécifiés, une transformation donnée peut être non-déterministe. Ceci signifie qu'elle peut fournir des résultats différents lorsqu'elle est exécutée plusieurs fois avec la même source.

3.5.4 AGG

AGG [69, 71] est un environnement de développement pour les systèmes de transformation de graphes supportant une approche algébrique de la transformation de graphes. Cette approche est basée sur des fondations formelles [21]. Les métamodèles source et cible et le métamodèle commun sont représentés par des graphes typés. Les graphes peuvent aussi être attribués à l'aide de code Java : leurs éléments peuvent être décorés par des morceaux de code. Des gardes peuvent être définies sous la forme de conditions d'application négatives qui spécifient des motifs dont la présence dans le graphe courant empêche l'application de certaines règles. Ceci peut, par exemple, être utilisé pour empêcher de multiples applications d'une règle donnée. Dans certains cas, les transformations de graphes peuvent être automatiquement inversées [70].

Les transformations peuvent être exécutées dans deux modes différents. Le premier est appelé "mode interprétation" et correspond à l'application automatique des règles. Cette application n'est pas déterministe et se poursuit jusqu'à ce qu'aucune règle ne puisse être appliquée. Si un ordre précis d'application est requis, les règles peuvent être groupées en couches ordonnées. Les règles contenues dans une couche donnée ne sont appliquées qu'une fois qu'un point fixe a été atteint dans la couche précédente. Dans le second mode, un utilisateur lance explicitement l'application de chaque règle.

Cette approche repose fortement sur la reconnaissance de motifs qui se fait sur la totalité du graphe courant. De plus, puisqu'il n'y a pas de déterminisme dans le choix des règles à appliquer lorsque plusieurs sont possibles, il y a un fort besoin pour la vérification de confluence. Un ensemble de règles est confluent si le résultat est déterministe quel que soit l'ordre d'application des règles. AGG peut détecter des conflits entre règles qui pourraient introduire de l'indéterminisme dans le résultat. Le processus correspondant s'appelle "analyse des paires critiques" (*critical pair analysis* en anglais) car il détermine un ensemble de paires de règles en conflit. Une fois tous les conflits résolus (e.g. en utilisant des couches), l'application des règles reste non-déterministe mais le résultat final est toujours le même. AGG peut aussi vérifier des critères de terminaison.

3.6 Comparaison

Les descriptions des langages données ci-dessus sont utilisées pour dériver la table comparative 3.2 qui résume les caractéristiques principales des langages. Les lignes de la table sont les catégories et caractéristiques utilisées pour analyser les langages. Les colonnes représentent les différents langages. Il y a notamment trois colonnes pour QVT : une pour chacun de ses sous-langages.

Dans une cellule donnée, nous indiquons si la caractéristique de la ligne est supportée par le langage de la colonne. En général, le contenu des cellules est soit *Oui* (i.e. caractéristique supportée), soit *Non*. Quand le support ne peut pas être jugé d'après les descriptions disponibles des langages, nous mettons un point d'interrogation dans la cellule. Quand la caractéristique ne s'applique pas à un langage, nous mettons *N/A* (pour "non applicable") dans la cellule. Dans certains cas, les cellules contiennent des

informations plus détaillées. C'est, par exemple, le cas des constructions utilisées pour l'organisation des règles. Il y a plusieurs clarifications à propos de la table qui sont données sous forme de notes.

3.7 Conclusion

Ce chapitre a présenté différentes approches de transformations. Il s'agit de QVT, décomposé en *Relations*, *Core* et *Operational Mappings*, de Tefkat et de trois approches basées sur la transformation de graphes : VIATRA2, GReAT et AGG. Ces différents langages ont ensuite été comparés entre eux et à ATL. Les résultats de ce travail ont été présentés sous la forme du tableau 3.2. D'autres langages comme AToM^3 [71, 17], Kent Model Transformation Language [1, 2], MOLA [44] et UMLX [76] auraient pu être intégrés dans cette étude comparative.

Bien sûr, cet état de l'art sur les langages de transformation de modèles n'est pas exhaustif, ne serait-ce que parce que ce domaine est toujours en évolution très rapide. Les quelques remarques qui suivent portent sur certains aspects qui n'ont pas été développés ci-dessus.

À la lecture de cet état de l'art, il apparaît difficile d'envisager la convergence vers un langage unique (ou unifié, ou universel) de transformation de modèles. Il semble plutôt que l'on s'oriente actuellement vers un ensemble de langages dédiés de transformation de modèles. Cette tendance semble déjà s'inscrire dans la recommandation QVT elle-même.

Pour être plus général, il aurait fallu inclure aussi les langages généralistes (comme Java ou C#) dans la liste des langages de transformation de modèles. Munis de puissantes bibliothèques d'accès aux modèles et à leurs éléments, ces solutions offrent l'avantage de ne pas exiger des compétences spécifiques de la part des programmeurs de transformation. De même, il aurait été intéressant de considérer les langages de transformation disponibles dans d'autres espaces techniques. Par exemple, XSLT et XQuery sont deux langages de transformation de l'espace technique XML.

Si la liste des langages n'est pas exhaustive, la liste des critères d'évaluation ne l'est pas beaucoup plus. Il serait intéressant de comparer les propositions de langages suivant de nombreux autres critères, comme celui de la facilité d'apprentissage ou d'utilisation, celui de la disponibilité de bibliothèques, etc.

Un autre exemple de critère à prendre en compte est celui présenté dans [9] qui considère les programmes de transformation comme des modèles, ce qui est en effet important dans le cadre de la plateforme AMMA et de son modèle conceptuel. Ceci est à la base de nombreuses applications pratiques basées sur les transformations d'ordre supérieur (en anglais HOT pour *Higher Order Transformation*). ATL partage avec QVT ce besoin important de satisfaire, ici aussi, le principe de base de l'unification par les modèles.

Caractéristiques		ATL	ATL VM	QVT Relations	QVT Core	QVT Operational Mappings	Tefkat	VIATRA2	GREaT	AGG
Scénarios de transformation	Synchronisation de modèles	Non[1]	Non[1]	Oui	Oui	Non[1]	Non[1]	Non[1]	Non[1]	Non[1]
	Vérification de correspondance	Non[1]	Non[1]	Oui	Oui	Non[1]	Non[1]	Non[1]	Non[1]	Non[1]
	Transformation de modèles	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
	Modifications en place	Oui[2]	Oui	Oui	Oui	Oui	Non	Oui	Oui	Oui
	Transformation interactive	Non	Non	Non	Non	Non	Non	Non	Non	Oui
Paradigme	Déclaratif	Oui[3]	Non	Oui	Oui	Non	Oui	Non	Non	Oui
	Hybride	Oui	Non	Non	Non	Non	Non	Oui[4]	Oui[4]	Non
	Impératif	Oui[3]	Oui	Non	Non	Oui	Non	Non	Non	Non
Directivité	Unidirectionnel seulement	Oui	Oui	Non	Non	Oui	Oui	Oui	Oui	Oui
	Multidirectionnel	Non	Non	Oui	Oui	Non	Non	Non	Non	Non[5]
Cardinalité	M-à-N	Oui	Oui	Oui[6]	Oui[6]	Oui	Oui	Oui	Oui	Non
	1-à-1	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
Traçabilité	Automatique	Oui	Non	Oui	Non	Oui	Non	Non	Non	Non
	Définie par l'utilisateur uniquement	Non	Oui	Non	Oui	Non	Oui	Oui	Oui	Oui
Langage de navigation		basé sur OCL	à pile	basé sur OCL et motifs	basé sur OCL et motifs	basé sur OCL	motifs, avec récursion	motifs de graphes, avec récursion	motifs de graphes et C++	motifs de graphes
Ordonnancement des règles		implicite, explicite interne	N/A	implicite	implicite	explicite interne	implicite	explicite externe	explicite externe	explicite externe, implicite[7]
Organisation des règles		héritage	N/A	héritage	héritage	héritage	héritage	groupes	blocs hiérarchiques	couches
Réflexion	Accès aux éléments de modèles et métamodèles	Oui	Oui	Non	Non	Oui	Oui	?	Oui	Non
	Expressions de types et de propriétés	Non	Non	Non	Non	Non	Oui	Oui[8]	Non	Non

[1] Scénario implémentable mais non directement supporté.

[2] En mode raffinage (*refining*).

[3] Il est possible d'écrire des transformations uniquement déclaratives ou uniquement impératives en ATL.

[4] Règles déclaratives et ordonnancement impératif.

[5] Règles parfois inversibles.

[6] Seuls des exemples M-à-1 sont donnés dans la recommandation pour les scénarios transformation et synchronisation de modèles.

[7] Implicite à l'intérieur d'une couche et explicite externe entre les couches.

[8] Avec les patrons de règles.

Table 3.2 – Caractéristiques principales des langages de transformation de modèles

CHAPITRE 4

La plateforme AMMA[a]

[a] Les travaux présentés dans ce chapitre sont partiellement adaptés de [38] et [12].

4.1 Introduction

La plateforme AMMA (ATLAS *Model Management Architecture* signifiant architecture de gestion de modèles ATLAS) est une plateforme de modélisation. Elle est basée sur des définitions précises de la notion de modèle, d'une part, et d'un métamétamodèle appelé KM3, d'autre part. Ces définitions sont compatibles avec la vision MDA de l'OMG présentée à la section 2.2.1. Ainsi, l'implémentation de AMMA est notamment basée sur des systèmes de manipulation de modèles tels que EMF (proche de EMOF 2.0) et MDR (utilisant MOF 1.4).

AMMA permet la construction de systèmes logiciels basés sur les principes de l'ingénierie des modèles. Nos travaux sont focalisés sur la problématique de la transformation de modèles. Ceci a demandé la définition d'un langage de transformation de modèles appelé ATL. La plateforme AMMA est donc, pour l'instant, souvent utilisée pour la création d'outils de conversion entre formalismes de représentation de données ou de programmes. Afin d'implémenter de tels outils, il est aussi nécessaire d'avoir un mécanisme pour la représentation des métamodèles et des modèles impliqués dans les scénarios de transformation. Nous avons donc défini le langage KM3 pour la spécification de métamodèles et le langage TCS pour la spécification des syntaxes textuelles associées. D'autres possibilités d'utilisation de AMMA sont basées sur des fonctionnalités de la plateforme faisant l'objet de travaux en cours.

Une utilisation possible de AMMA est la construction de DSLs. Ainsi qu'il a été présenté au chapitre 2, l'ingénierie des modèles est en effet particulièrement adaptée à leur implémentation. Cette implémentation prend typiquement la forme d'un ensemble d'outils de conversion entre langages dédiés et vers des langages généralistes. L'utilisation de AMMA comprend généralement les quatre étapes suivantes :

- **Définition des métamodèles.** L'un des principes de l'ingénierie des modèles est la définition explicite des métamodèles auxquels les modèles manipulés sont conformes. Il est donc nécessaire de commencer par définir ces métamodèles. Chaque métamodèle capture un domaine donné. Il peut ainsi s'agir du métamodèle d'un outil pré-existant tel que Microsoft Excel (données tabulaires), Microsoft Project (gestion de projets), etc. Ces outils n'ont généralement pas de définition explicite de leur métamodèle, qui est généralement implémenté dans leur code source souvent non accessible. Le travail à faire consiste donc en la récupération par ingénierie inverse du métamodèle. Dans d'autres cas, il s'agit du métamodèle d'un langage dédié tel que SQL ou HTML, ou encore d'un langage généraliste tel que Java, Pascal ou C.
- **Définition des syntaxes concrètes.** Une fois le métamodèle défini, il manque encore un moyen de saisir des modèles s'y conformant. Plusieurs approches existent, dont l'utilisation d'un outil spécialisé comme l'éditeur de modèles EMF. Ce dernier est basé sur une représentation arborescente des modèles. Cependant, dans de nombreux cas, les modèles existent déjà sous d'autres formats basés sur des grammaires ou encore des schémas XML. De plus, disposer d'une syntaxe adaptée

au domaine, qu'elle soit visuelle ou textuelle, facilite généralement la saisie et la lecture. Il est donc souvent avantageux de définir des syntaxes concrètes pour les métamodèles définis à l'étape précédente.

- **Définition de transformations de modèles.** Une transformation de modèles peut jouer plusieurs rôles. Ainsi, elle peut servir à convertir des données d'un format à un autre. Mais elle peut aussi traduire des programmes écrits dans un langage donné en programmes écrits dans un autre langage. Il est aussi possible de considérer qu'une telle traduction correspond à la définition de la sémantique du langage source à l'aide des concepts du langage cible.
- **Définition de l'outil.** L'implémentation d'un outil donné nécessite au moins deux métamodèles (source et cible), deux syntaxes concrètes ainsi qu'une transformation. Ceci correspond donc à un chaînage de trois transformations : une injection, une transformation de modèle à modèle et enfin une extraction. Cependant, il est souvent nécessaire d'utiliser un ou des métamodèles intermédiaires. Ainsi, chaque transformation est généralement plus simple qu'une unique transformation traduisant directement de la source vers la cible. De plus, il devient possible de réutiliser chaque transformation séparément. En conséquence, la définition effective de l'outil nécessite aussi une spécification du chaînage des transformations qui le constituent.

En pratique, il est rare de respecter exactement l'ordre donné ci-dessus. La définition d'un outil consiste généralement en un processus incrémental. Ainsi, il est courant de définir tout d'abord une première version minimale réalisant un sous-ensemble des fonctionnalités de l'outil de bout en bout (i.e. du format source au format cible) en respectant cet ordre. Ensuite, les différentes étapes sont reprises afin d'étendre cette première version en ajoutant progressivement les différentes fonctionnalités.

Ce chapitre est organisé de la manière suivante. La section 4.2 donne les définitions sur lesquelles la plateforme AMMA est basée. La section 4.3 présente les langages dédiés de la plateforme AMMA pour la définition de métamodèles, syntaxes concrètes et transformations de modèles. La section 4.4 présente les différents blocs fonctionnels de AMMA. Enfin, la section 4.5 donne les conclusions.

4.2 Définitions

Dans le cadre de l'ingénierie des modèles, nous considérons la notion de modèle comme concept unifiant de l'ingénierie des systèmes d'information. Il existe différentes sortes de modèles. Un modèle UML, un programme Java, un document XML ou RDF, une base de données relationelle, un schéma entité-association, etc. sont tous des exemples de modèles. Nous les appelons des λ-modèles où λ identifie un espace technique [13] associé à un métamétamodèle précis donné. Une représentation simple des notions de modèle terminal, métamodèle et métamétamodèle est donnée dans la figure 4.1.

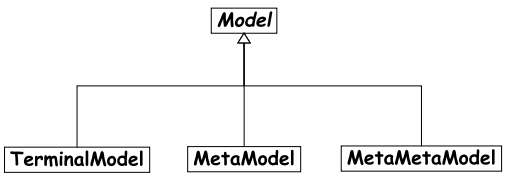

Figure 4.1 – Organisation générale d'une pile de métamodélisation

Nous pouvons considérer deux principales définitions pour un modèle correspondant à son *organisation* interne ou bien à ses *utilisations* potentielles. Nous choisissons de nous concentrer ici sur l'*organisation* des modèles. Ensuite, nous donnons une définition de DSL (Domain Specific Language

ou langage spécifique de domaine). Enfin, nous analysons les relations entre DSLs et modèles.

4.2.1 Définitions organisationnelles de la notion de modèle

D'un point de vue organisation, nous proposons les définitions suivantes :

Définition 9. Un *multi-graphe orienté* $G = (N_G, E_G, \Gamma_G)$ est composé d'un ensemble fini de noeuds N_G, d'un ensemble fini d'arcs E_G et d'une fonction $\Gamma_G : E_G \rightarrow N_G \times N_G$ reliant les arcs à leurs source et cible.

Définition 10. Un *modèle* $M = (G, \omega, \mu)$ est un triplet où :
- $G = (N_G, E_G, \Gamma_G)$ est un multi-graphe orienté,
- ω est un modèle (appelé le *modèle de référence* de M) associé à un graphe $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$,
- $\mu : N_G \cup E_G \rightarrow N_\omega$ est une fonction associant les éléments (noeuds et arcs) de G aux noeuds de G_ω.

Certains modèles sont leurs propres modèles de référence ($\omega = M$) et permettent de stopper la récursion introduite dans cette définition.

Remarques. La relation entre un modèle et son modèle de référence est appelée conformité et est notée *conformsTo* (pour conforme à) ou encore abrégée en *c2*. Les éléments de ω sont appelés métaéléments. μ n'est ni injective (plusieurs éléments de modèles peuvent être associés au même métaélément) ni surjective (tous les métaéléments ne sont pas nécessairement associés à un élément de modèle).

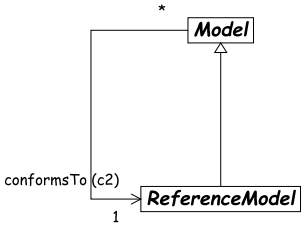

Figure 4.2 – Définition de *modèle* et de *modèle de référence*

La figure 4.2 illustre la définition 10. La définition de *modèle* donnée ci-dessus permet un nombre indéfini de niveaux de modélisations supérieurs. En pratique, il est commun de s'arrêter à un certain niveau. Nous observons que seulement trois niveaux sont utilisés dans plusieurs espaces techniques :
- En *XML* : documents, schémas et le schéma de XML Schema,
- En *EBNF* : programmes, grammaires et la grammaire de EBNF.

Nous appelons ces trois niveaux M1, M2 et M3. M1 comprend tous les modèles qui ne sont pas des métamodèles. M2 comprend tous les métamodèles qui ne sont pas le métamétamodèle. M3 comprend un unique métamétamodèle par espace technique. Nous donnons maintenant des définitions correspondant à ces contextes classiques.

Définition 11. Un *métamétamodèle* est un modèle qui est son propre modèle de référence : il est conforme à lui-même.

Définition 12. Un *métamodèle* est un modèle tel que son modèle de référence est un métamétamodèle.

Définition 13. Un *modèle terminal* est un modèle tel que son modèle de référence est un métamodèle.

La figure 4.3 montre comment adapter la définition de modèle à cette pile de modélisation à trois niveaux. La structure définie dans cette section est compatible avec la vue de l'OMG telle qu'illustrée dans le guide du MDA (ou MDA guide) [53].

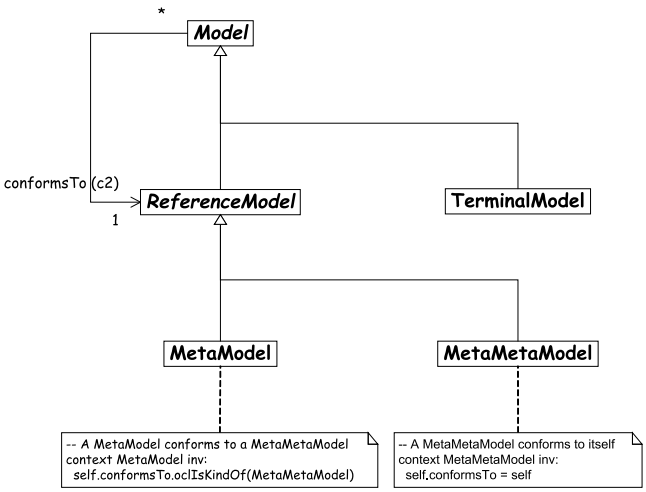

Figure 4.3 – Pile de métamodélisation représentée avec la définition de modèle

4.2.2 Langages dédiés

L'ingénierie des langages est au coeur de l'informatique. Il y a une grande variété de catégories de langages. Nous ne discutons ici que d'un aspect limité de l'ingénierie des langages. Une différence est souvent faite entre les langages de programmation et les langages de modélisation. Des exemples typiques sont le langage de programmation PL/1 et le langage de modélisation UML. La distinction entre ces catégories est principalement liée à l'exécutabilité canonique qui est actuellement un sujet encore mal défini. Une autre différence est faite entre les langages à utilisation générale (ou GPLs pour General Purpose Languages) et les langages dédiés (ou DSLs pour Domain Specific Languages). PL/1 et UML sont deux exemples de GPLs. R [6], SQL [49] ou encore Excel sont des exemples de DSLs. Java et C# sont des exemples de GPLs.

Nous observons aussi que la distinction entre GPLs et DSLs est orthogonale à beaucoup d'autres classifications des langages. Par exemple, il y a indifféremment des GPLs ou DSLs à syntaxe visuelle ou textuelle. De même, les DSLs et les GPLs peuvent appartenir à différentes catégories telles que : langage à objects, langage à événements, langage à règles, langage à fonctions, etc. Il y a aussi des exemples de GPLs et DSLs impératifs et déclaratifs.

Un DSL est un langage conçu pour être utile pour un ensemble limité de tâches. Un GPL, en revanche, est supposé être utile pour des tâches beaucoup plus génériques dans de nombreux domaines d'application. Un exemple typique de DSL est GraphViz [30], un langage utilisé pour définir des graphes orientés. Des outils peuvent ensuite être utilisés pour créer une représentation visuelle de ces graphes. Certains GPLs ont d'abord été des DSLs qui ont évolué vers la généralité en devenant des GPLs. Le processus inverse n'a pas été observé dans l'histoire des langages de programmation.

Les DSLs ont des propriétés communes avec la plupart des langages [36]. Nous en citons trois ici :

- Ils ont généralement une syntaxe concrète.
- Ils peuvent aussi avoir une syntaxe abstraite.

- Ils ont une sémantique définie implicitement ou explicitement.

Il y a bien entendu plusieurs manières de définir ces syntaxes et sémantiques. Les plus connues sont celles basées sur les grammaires.

4.2.3 DSLs et modèles

Il y a de fortes relations entre DSLs et modèles. Nous discutons ici de la possibilité d'utiliser des solutions basées sur les modèles pour définir les syntaxes et sémantiques des DSLs.

Définition 14. Un *DSL* est un ensemble coordonné de modèles.

Chaque modèle de cet ensemble contribue à une partie de la définition du DSL. Un modèle donné peut, par exemple, spécifier l'un des aspects suivants :

- **Métamodèle de définition du domaine.** Une des entitées principales de la définition d'un DSL est son métamodèle de définition de domaine (ou DDMM pour Domain Definition MetaModel). Il introduit les concepts élémentaires du domaine ainsi que leurs relations. Cette *ontologie de base* joue un rôle central dans la définition du DSL. Par exemple, un DSL pour la manipulation de graphes orientés contiendra les concepts de noeud et d'arc. Il définira aussi qu'un arc relie un noeud source à un noeud cible. Un tel DDMM joue le rôle de la syntaxe abstraite du DSL.
- **Syntaxes Concrètes.** Un DSL peut avoir plusieurs syntaxes concrètes. Chacune d'entre elles peut être définie par une transformation reliant le DDMM à un métamodèle de *surface d'affichage*. Des exemples de métamodèles de surface d'affichage sont SVG ou DOT (un des outils de GraphViz [30]) mais aussi XML. Un exemple de transformation pour un DSL de réseau de Petri est la mise en correpondance des places avec des cercles, des transitions avec des rectangles et des arcs en flèches. Le métamodèle de surface d'affichage doit alors définir les concepts de cercle, rectangle et flèche.
- **Sémantique d'exécution.** Un DSL peut avoir une définition de sa sémantique d'exécution. Cette définition peut aussi être spécifiée par une transformation mettant en correspondance le DDMM avec un autre DSL dont la sémantique est déja définie ou encore avec un GPL. Les règles de tir d'un réseau de Petri peuvent, ainsi, être mises en correspondance avec un modèle de code Java.
- **Autres opérations sur les DSLs.** En plus de leur exécution canonique, il y a beaucoup d'autres opérations possibles sur les programmes basés sur un DSL donné. Chacune peut être définie par une correpondance représentée par une transformation de modèles. Par exemple, afin de faire des requêtes sur des programmes, une correspondance standard de leur DDMM vers Prolog peut être utile. L'étude de ces autres opérations sur les DSLs est un sujet de recherche ouvert.

4.3 Langages dédiés noyau de AMMA

La figure 4.4 représente les langages dédiés composant AMMA et la manière dont ils sont utilisés pour définir d'autres langages dédiés. D'après la définition 14, un DSL est un ensemble coordonné de modèles. AMMA fournit plusieurs DSLs utilisés pour définir les modèles constitutifs des DSLs à implémenter. Ils forment le noyau de AMMA. Nous nous concentrons ici sur un sous-ensemble simplifié de AMMA composé des trois DSLs ayant fait l'objet de nos travaux. D'autres travaux en cours visent à étendre AMMA avec d'autres DSLs et donc des capacités plus larges.

Ces composants sont :

- **KM3** qui est un métamétamodèle et qui sert donc à la définition de métamodèles. Ce langage est présenté au chapitre 5.

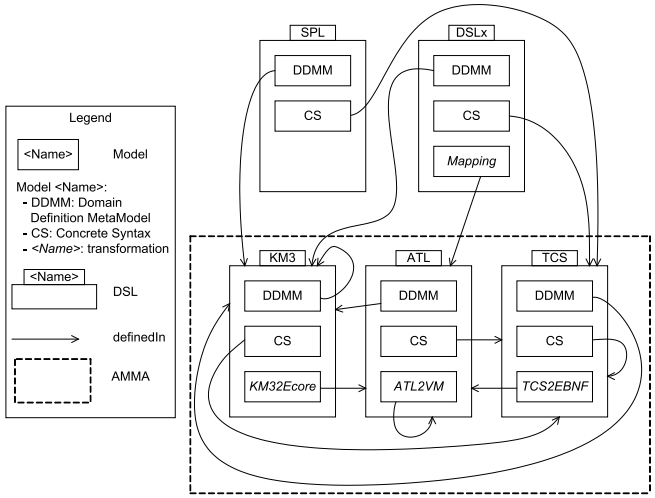

Figure 4.4 – Les langages noyau de AMMA (KM3, TCS et ATL) avec leur utilisation possible

- **TCS** pour la définition de syntaxes concrètes textuelles et leurs liens avec les métamodèles correspondants. Ce langage est présenté au chapitre 6.
- **ATL** pour la définition de transformations de modèles. Ce langage est présenté au chapitre 7.

Chacun de ces trois DSLs est constitué de modèles exprimés à l'aide des autres DSLs du noyau. Par exemple, le DDMM de KM3 est défini en KM3. La syntaxe concrète de KM3 est définie en TCS. De plus, il est possible de traduire KM3 en Ecore à l'aide d'une transformation ATL (la boîte *KM32Ecore*). La sémantique d'ATL est définie par une transformation vers le langage de la machine virtuelle ATL (boîte *ATL2VM*). Cette transformation est exprimable en ATL.

D'autres langages dédiés peuvent être définis en utilisant ces langages noyau. Par exemple, nous avons expérimenté la définition du langage de téléphonie SPL [15] (*Session Processing Language*) (voir section 8.4). Le métamodèle de définition de domaine de SPL est spécifié en KM3 et sa syntaxe concrète en TCS. Pour ce cas d'étude, nous n'avons pas représenté la sémantique du langage par une transformation. Au contraire, nous avons défini celle d'un autre langage du même domaine : CPL en transformant les programmes écrits dans ce langage vers SPL.

Un langage donné (appelé DSL_x sur la figure 4.4) peut être défini d'une manière similaire. Dans le contexte de DSL_x, la boîte *Mapping* dénote une transformation possible vers un autre langage dédié ou encore vers un langage généraliste tel que Java. Nous avons expérimenté l'utilisation du formalisme des machines d'état abstraites (*Abstract State Machines* abrégé en ASMs) pour définir la sémantique de SPL [66] et d'ATL [67]. Bien que les résultats obtenus soient prometteurs (simulation complète d'une transformation ATL simple en ASM), la généralisation de cette approche doit encore être étudiée. Nous considérons donc ici que, dans son état actuel, AMMA ne permet de définir la sémantique d'un langage (par exemle DSL_1) que par transformation de son DDMM vers celui d'un autre langage (par exemple DSL_2). La sémantique de ce langage de référence (DSL_2) peut être elle-même définie par une transformation vers un autre langage ou être définie formellement en dehors de AMMA. C'est, par exemple, le cas des réseaux de Petri.

Chacun des langages dédiés noyau de AMMA est détaillé dans les chapitres suivants : (5 pour KM3, 6 pour TCS et 7 pour ATL.

4.4 Fonctions offertes par AMMA

AMMA offre quatre fonctions principales représentées sur la figure 4.5 :

- **Transformation de modèles** avec ATL.
- **Tissage de modèles** avec AMW (ATLAS *Model Weaver* ou tisseur de modèles ATLAS).
- **Gestion globale de modèles** avec AM3 (ATLAS *MegaModel Management* ou gestion de méga-modèle ATLAS).
- **Projections techniques** avec ATP (ATLAS *Technical Projectors* ou projecteurs techniques ATLAS).

La figure représente ces quatre blocs fonctionnels connectés ensemble par un bus d'échange de modèles. Deux outils appelés *Tool X* et *Tool Y* sont connectés à la plateforme AMMA et peuvent communiquer grâce à elle :

- Les concepts de chaque outil peuvent être capturés par un métamodèle.
- Des projections peuvent être utilisées pour convertir les formats utilisés par les outils en modèles et inversement.
- Un tissage peut capturer les relations entre les deux métamodèles.
- Une transformation de modèles peut traduire les données d'un des outils vers un modèle compatible avec l'autre.
- La gestion globale de modèles peut capturer les relations entre les différents modèles évoqués ci-dessus.

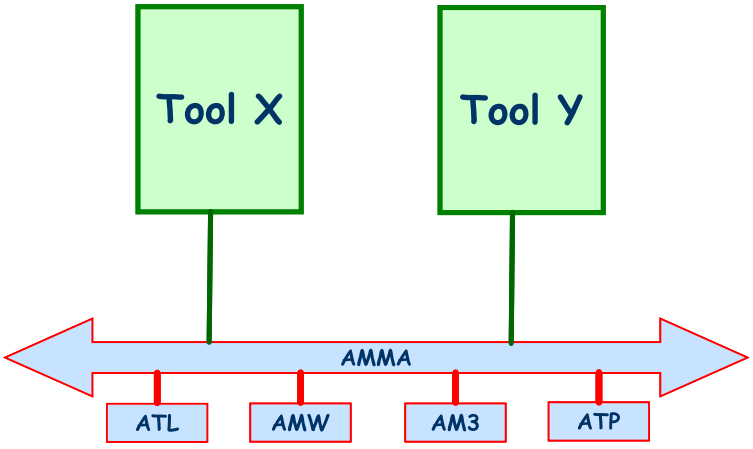

Figure 4.5 – Blocs fonctionnels de AMMA

Nos travaux se sont concentrés sur la transformation de modèles et le langage ATL en est un résultat. Nous avons aussi travaillé sur quelques projecteurs techniques dont TCS. Les travaux sur le tissage de modèles et la gestion globale de modèles font l'objet de recherches en cours.

4.5 Conclusion

Nous venons de voir la plateforme AMMA de gestion de modèles. Dans sa version finale, elle offrira quatre fonctions principales : la transformation de modèles (ATL), le tissage de modèles (AMW), la gestion globale de modèles (AM3) et la projection entre espaces techniques (ATP). Elle est basée sur trois langages dédiés noyau qui sont : KM3, TCS et ATL.

Ces langages sont nécessaires pour capturer les aspects principaux des systèmes. Mais, ils ne sont pas suffisants. Ainsi, TCS est limité aux syntaxes textuelles basées sur des grammaires. Bien que le langage ATL soit dédié à la transformation de modèles, ce domaine est relativement large et il serait possible d'introduire des langages de transformation dédiés à des sous-domaines plus limités. De plus, de nombreux aspects ne sont pas pris en compte, tels que : la synchronisation, la sécurité, la qualité de service, etc. Cependant, la plateforme AMMA elle-même peut-être utilisée pour la définition de DSLs couvrant ces domaines.

CHAPITRE 5

Le langage KM3[a]

[a] Le travail présenté dans ce chapitre est partiellement adapté de [38].

5.1 Introduction

Il existe de nombreux langages de métamodélisation tels que MOF 1.4, EMOF 2.0, Ecore, etc. La plupart de ces langages utilisent les concepts de classe, attribut et association ou référence. Cependant, ils sont incompatibles puisqu'un métamodèle conforme à l'un d'entre eux ne peut généralement pas être conforme à un autre. Ils n'ont pas non plus de définition formelle. De plus, certains de ces langages possèdent des concepts dépassant le domaine de la métamodélisation. C'est notamment le cas de Ecore qui permet par exemple d'annoter un métamodèle avec des directives pour la génération de code Java. Par ailleurs, il n'existe que peu d'outils pour la création ou la manipulation des métamodèles définis dans ces langages.

Ce chapitre présente le langage de métamodélisation KM3 (*Kernel MetaMetaModel* ou métamétamodèle noyau) et sa définition formelle basée sur la logique du premier ordre. Ce langage est une simplification des langages existants dans ce domaine et plus particulièrement de MOF 1.4, EMOF 2.0 et Ecore. Il est donc possible de traduire n'importe quel métamodèle KM3 vers un de ces langages. Les métamodèles définis en KM3 peuvent ainsi être utilisés avec différents systèmes de manipulation de modèles basés sur ces métamétamodèles (e.g. Netbeans/MDR et Eclipse/EMF). Par ailleurs, afin d'offrir une alternative à l'utilisation des quelques éditeurs graphiques de métamodèles, une syntaxe concrête textuelle a été définie pour KM3.

Nous allons d'abord présenter une vue d'ensemble de KM3 à la section 5.2. Puis nous présenterons la sémantique formelle de KM3 à la section 5.3. La section 5.4 compare KM3 à quelques travaux similaires et enfin, la section 5.5 conclut.

5.2 Vue d'ensemble de KM3

5.2.1 Description

L'objectif de KM3 est de fournir une solution relativement simple à la définition du métamodèle de définition du domaine (ou DDMM pour Domain Definition MetaModel) d'un DSL. KM3 est donc un DSL pour la définition des métamodèles tout comme EBNF est un DSL de définition de grammaires :

- **Métamodèle de définition du domaine.** Le DDMM de KM3 est un métamétamodèle auquel les autres DDMMs sont conformes. Ce DDMM est défini en KM3 (voir annexe A.2), de la même manière que la syntaxe EBNF peut être décrite en EBNF en quelques lignes. KM3 utilise les concepts de classe (*Class*), attribut (*Attribute*), référence (*Reference*), etc. Sa structure est proche de EMOF 2.0 [62] et de Ecore [14].

- **Syntaxe concrète.** La syntaxe par défaut de KM3 est textuelle (voir annexe A.1). Ceci permet la définition simple de métamodèles avec n'importe quel éditeur textuel.
- **Sémantique.** La sémantique de KM3 permet la définition de métamodèles et de modèles conformément aux définitions données précédemment (voir 4.2). Une définition conceptuelle précise de KM3 est présentée dans la section 5.3. Des transformations vers et depuis MOF 1.4 [54] et Ecore ont notamment été définies en ATL. KM3 est de ce fait utilisable avec des outils tels qu'Eclipse EMF [14] et Netbeans MDR [51].

En tant que métamétamodèle, KM3 est plus simple que MOF 1.4, MOF 2.0 ou encore Ecore. Il ne contient que 14 classes là où Ecore en a 18 et MOF 1.4 en a 28. Seuls les concepts essentiels de ces autres métamétamodèles ont été retenus dans KM3.

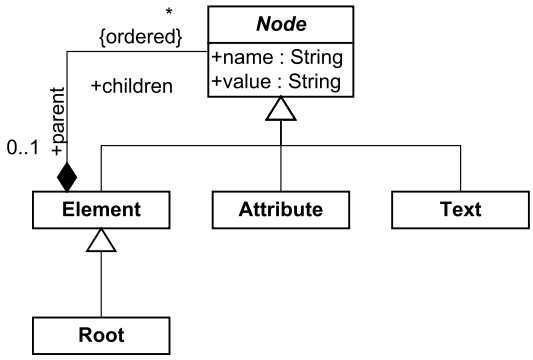

Figure 5.1 – Représentation visuelle d'un métamodèle XML

La figure 5.1 représente un métamodèle XML à l'aide de la notation standard des diagrammes de classes. Un noeud (classe *Node*) a un nom (attribut *name*) et une valeur (attribut *value*) tous deux de type chaîne de caractères (*String*). Un noeud peut être un fragment de texte (classe *Text*), un attribut (classe *Attribute*) ou un élément, qui peut être quelconque (classe *Element*) ou bien être un élément racine (classe *Root*). Un élément possède des enfants (référence *children*) qui sont des noeuds dont il est le parent (référence *parent*).

Le listing 5.1 donne la définition de ce même métamodèle XML en KM3. La classe *Node* est définie aux lignes 2 à 6 à l'aide du mot-clé `class`. Le mot-clé `abstract` spécifie que la classe est abtraite, ce qui est représenté sur la figure 5.1 par la mise en italique de son nom. Les attributs *name* et *value* sont respectivement définis aux lignes 3 et 4 à l'aide du mot-clé `attribute`. Leur type est donné après les deux points (`:`) et leur multiplicité (`1-1`) est implicite, comme sur la figure 5.1. La référence *parent* est définie à la ligne 5 à l'aide du mot-clé `reference` et sa multiplicité (`0-1`) est spécifiée entre crochets après son nom. La classe *Element*, vers laquelle *parent* pointe (i.e. son type) est déclarée après les deux points. Sa référence opposée `children`, dont la définition est donnée de manière similaire à la ligne 13 dans la classe *Element*, est déclarée après le mot-clé `oppositeOf`. Les classes *Attribute* (ligne 8), *Text* (ligne 10) et *Element* (lignes 12 à 14) sont définies de manière similaire mais sans le mot-clé `abstract`, puisqu'elles ne sont pas abstraites. De plus, chacune déclare sa classe parente (*Node*) à l'aide du mot-clé `extends`. La classe *Root* a *Element* pour classe parente (ligne 16). Les types primitifs *Boolean*, *Integer* et *String* sont respectivement définis aux lignes 20, 21 et 22 à l'aide du mot-clé `datatype`. Puisque seul *String* est utilisé, les définitions de *Boolean* et *Integer* auraient pu être omises. Enfin, les classes, d'une part, et les types primitifs, d'autre part, sont respectivement regroupés dans les paquetages *XML* (lignes 1 à 17) et *PrimitiveTypes* (lignes 19 à 23), définis à l'aide du mot-clé `package`. Ce découpage

en paquetage ainsi que les types primitifs n'étaient pas représentés sur la figure 5.1, qui ne donne qu'une vue des classes du métamodèle.

Listing 5.1 – Représentation du métamodèle XML de la figure 5.1 en KM3

```
1  package XML {
2    abstract class Node {
3      attribute name : String;
4      attribute value : String;
5      reference parent[0-1] : Element oppositeOf children;
6    }
7
8    class Attribute extends Node {}
9
10   class Text extends Node {}
11
12   class Element extends Node {
13     reference children[*] ordered container : Node oppositeOf parent;
14   }
15
16   class Root extends Element {}
17 }
18
19 package PrimitiveTypes {
20   datatype Boolean;
21   datatype Integer;
22   datatype String;
23 }
```

5.2.2 Applications

KM3 a été défini en réponse aux demandes fréquentes d'utilisateurs qui avaient besoin de définir des métamodèles pour leurs transformations en ATL. En principe, les métamodèles source et cible des langages de transformation tels que QVT devraient être écrits en XMI. Quand la transformation utilise des métamodèles standards comme UML, la représentation XMI de ces métamodèles est en principe fournie par l'OMG et disponible sur son site. Il n'y a alors pas besoin d'un autre formalisme, excepté pour la documentation dans laquelle des diagrammes de classes sont utilisés.

La pratique de la transformation de modèles avec une communauté croissante d'utilisateurs ATL nous a cependant obligés à revoir cette vision des choses. Pendant le développement de transformations, il est apparu que les métamodèles standards étaient souvent insuffisants. Beaucoup de transformations nécessitent en effet des métamodèles spécifiques. De plus, la définition de ces métamodèles est souvent un processus complexe et itératif impliquant une élaboration progressive.

Afin d'illustrer ceci, nous donnons ci-dessous quelques exemples de transformations écrites en ATL. Le code complet et la documentation de ces transformations sont disponibles dans la bibliothèque *open source* de transformations du site GMT [20].

- *Ant2Maven* et *Make2Ant* sont des transformations partielles entre les outils de construction de logiciel make, ant et maven.
- *BibTeX2DocBook* est une transformation d'un modèle BibTeXML en un document DocBook.
- L'exemple *JavaSource2Table* calcule le graphe d'appel statique d'un programme Java et présente le résultat sous la forme d'une table. À partir de là, il est possible d'utiliser les métamodèles XHTML ou Excel pour projeter cette table vers d'autres surfaces d'affichage en chaînant les transformations.
- *KM32DOT* permet l'affichage graphique des métamodèles définis en KM3. DOT est un outil de placement automatique de graphes faisant partie des outils GraphViz [30]. L'objectif de cette trans-

formation est de générer automatiquement une représentation visuelle de n'importe quel métamodèle KM3, sous la forme de diagrammes de classes.

- L'exemple *UMLActivityDiagram2MSProject* définit une transformation depuis un diagramme d'activité UML sans boucle, décrivant une séquence de tâches, vers MS Project. Cette transformation est basée sur un sous-ensemble simplifié des machines d'état du métamodèle UML. Le résultat est un projet conforme à un sous-ensemble limité du métamodèle de MS Project.

La table 5.1 donne une autre vue de la même bibliothèque de transformations. Les nombres de classes des métamodèles source et cible d'une douzaine de transformations sont donnés. Sans décrire en détail toutes ces transformations, il apparaît clairement que la majorité des métamodèles source et cible doivent être définis. Ceci reste vrai même lorsqu'il s'agit de métamodèles standards (comme les diagrammes d'activité UML) car souvent, seul un petit sous-ensemble du métamodèle est utilisé.

Nom	Nombre de classes source	Nombre de classes cible
BibTeXML vers Docbook	21	6
Class vers Relational	5	4
Java source vers Table	5	3
KM3 vers DOT	16	26
KM3 vers Problem	16	2
PathExp vers PetriNet	5	7
Table vers Microsoft Excel	3	15
UML vers Amble	11	14
UML vers Java	11	8
UML Activity Diagram vers MS Project	6	3
UMLDI vers SVG	26	38
XSLT vers XQuery	13	18

Table 5.1 – Exemples de transformations de la bibliothèque ATL

En conséquence, la définition des métamodèles source et cible d'une transformation représente une part importante de son développement. Nous avons besoin d'une notation qui permette la définition et la modification simple et précise de ces métamodèles. Même si cela peut sembler contraire à l'intuition, les utilisateurs ont demandé des langages textuels plutôt que des langages visuels pour réaliser cette tâche.

Le langage KM3 est très utile pour la définition rapide et précise de métamodèles dans des situations variées. Par exemple nous avons étudié l'interopérabilité entre différents outils tels que Bugzilla, Make, Ms Project, Mantis, etc. Le modèle de donnée de ces outils est généralement capturé dans un métamodèle. Les ponts peuvent alors être définis comme des transformations utilisant directement ces métamodèles.

Nous avons déjà mentionné la bibliothèque de transformations ATL disponible sur le site GMT. Ce qui est aussi intéressant est qu'il existe une autre bibliothèque, sur ce même site, dédiée aux métamodèles définis en KM3 [25]. De plus, d'autres vues de cette même bibliothèque en sont automatiquement dérivées via des transformations écrites en ATL (voir table 5.2). De nombreux problèmes peuvent être étudiés sur la base de cette bibliothèque initiale. L'un d'entre eux est, par exemple, la réutilisabilité des métamodèles. D'autres questions peuvent aussi apparaître sur les différentes relations entre ces métamodèles et aussi sur les métadonnées qui leur sont associées.

Métamodèle	Format	Outils compatibles
Ecore	XMI 2.0	Eclipse EMF[a]
MOF 1.4	XMI 1.2	Netbeans MDR[b]
UML 1.4	XMI 1.2	Netbeans MDR[b]
non applicable	PNG	visualiseurs d'images
SQL DDL	syntaxe SQL	bases de données
Microsoft DSL Tools	DSLDM, basé sur XML	outils Microsoft
Visual Basic (VB)	syntaxe VB	compilateur VB
XASM	syntaxe XASM	compilateur XASM[c]

[a] Les outils basés sur Eclipse EMF sont aussi compatibles. Par exemple : ATL.
[b] Les outils basés sur Netbeans MDR sont aussi compatibles. Par exemple : ATL, Poseidon.
[c] XASM est un langage basé sur la théorie des machines d'état abstraites (ou ASMs pour Abstract State Machines).

Table 5.2 – Exemples de formats vers lesquels les métamodèles KM3 peuvent être projetés

5.3 Définition conceptuelle de KM3

Définition 15. Un *modèle KM3* est un modèle défini avec le métamétamodèle KM3.

Cette section ne traite que de modèles KM3. En conséquence, nous utilisons ici le terme modèle dans le sens modèle KM3. Nous présentons ici une spécification formelle de KM3 basée sur la logique du premier ordre. Seuls les métamodèles et non les modèles terminaux peuvent être conformes à KM3. Cependant, la sémantique de KM3 influence aussi les modèles terminaux en leur imposant des contraintes liées à la sémantique des éléments de leurs modèles de référence.

Essentiellement deux prédicats sont utilisés pour définir les modèles, y compris le métamétamodèle lui-même. Pour un modèle M (voir définition 10), nous définissons :

- $Node(x, y)$**.** Ce prédicat définit le noeud $x \in N_G$ comme étant associé au noeud $y \in N_\omega$ par la fonction μ.
- $Edge(x, y, z)$**.** Ce prédicat définit un arc entre les noeuds $x \in N_G$ et $y \in N_G$ associé au noeud $z \in N_\omega$ par la fonction μ. En KM3, plusieurs arcs ne peuvent exister entre deux noeuds donnés qu'à la condition que leurs métaéléments associés soient distincts. Donc, le triplet (x, y, z) identifie un arc de manière unique.

Ces prédicats peuvent être utilisés pour définir des graphes qui ne sont pas des modèles KM3. Des contraintes doivent donc être définies afin de spécifier les conditions à satisfaire par un graphe pour être un modèle KM3. Il est typiquement nécessaire de contraindre le graphe correspondant à un modèle en fonction du graphe correspondant à son modèle de référence. Des formules sont utilisées tout au long de cette section pour exprimer ces contraintes de manière formelle. La définition d'un métamétamodèle tel que MOF 1.4 [54] contient aussi de telles contraintes, mais souvent exprimées en langage naturel ou en OCL.

Il n'est pas nécessaire pour un utilisateur de KM3 de comprendre les formules présentées ici. Ces contraintes sont en effet définies en langage naturel dans le manuel de KM3 [22]. De plus, la majorité de ces contraintes a aussi été implémentée en ATL, donc avec des expressions OCL, afin de pouvoir vérifier automatiquement les métamodèles. Ceci fait appel à la technique présentée à la section 8.3.

Nous commençons par définir une version simplifiée de KM3 appelé *SimpleKM3* (pour KM3 simple) avec uniquement des classes et des références. Ensuite, nous introduisons des concepts supplémentaires : référence opposée puis héritage.

5.3.1 Définition de SimpleKM3

SimpleKM3 est une version simplifiée de KM3 n'utilisant que des classes et des références. Une représentation visuelle de *SimpleKM3* sous la forme d'un diagramme de classes est donnée à la figure 5.2. La figure 5.3 donne la définition formelle de *SimpleKM3*. Il n'y a que deux classes : *class* (ligne 1) représentant le concept de classe et *reference* (ligne 2) représentant le concept de référence. Il y a deux références : *features* (ligne 3) et *type* (ligne 4). La référence *features* relie une classe à ses références (lignes 5-6). La référence *type* relie une référence à son type (lignes 7-8).

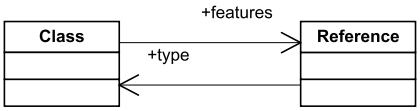

Figure 5.2 – Représentation de *SimpleKM3* sous la forme d'un diagramme de classes

1. $Node(\text{class}, \text{class})$
2. $Node(\text{reference}, \text{class})$
3. $Node(\text{features}, \text{reference})$
4. $Node(\text{type}, \text{reference})$
5. $Edge(\text{class}, \text{features}, \text{features})$
6. $Edge(\text{features}, \text{reference}, \text{type})$
7. $Edge(\text{reference}, \text{type}, \text{features})$
8. $Edge(\text{type}, \text{class}, \text{type})$

Figure 5.3 – Définition formelle de *SimpleKM3*

Les contraintes données ci-dessous pourraient être exprimées sans autre prédicat que $Node(x, y)$ et $Edge(x, y, z)$ définis précédemment. Cependant, certaines utilisations du prédicat $Node(x, y)$ devraient alors être remplacées plus tard avec l'introduction de l'héritage de classe. C'est le cas des formules (5.5) et (5.6) ci-dessous. Afin de ne pas avoir à réexprimer ces formules plus tard, nous définissons donc un nouveau prédicat : $IsKindOf(x, y)$. Pour l'instant, ce prédicat est équivalent au prédicat $Node(x, y)$:

$$\forall xy IsKindOf(x, y) \Leftrightarrow Node(x, y) \tag{5.1}$$

Il sera redéfini plus tard dans la section 5.3.3 quand nous introduirons l'héritage de classe. Nous utilisons toujours le prédicat $Node(x, y)$ pour définir des noeuds, mais nous utilisons ce nouveau prédicat dans les formules qui seront aussi valides lorsque x est lié par μ à une sous-classe de y.

Un modèle *SimpleKM3* (donc un modèle, un métamodèle ou le métamétamodèle) est valide si les formules suivantes sont vérifiées :

- **Unicité du métaélément.** μ étant une fonction, elle ne peut associer qu'un seul métaélément à un noeud donné du modèle.

  $$\forall xyz Node(x, y) \wedge Node(x, z) \Rightarrow y = z \tag{5.2}$$

  Il n'y a pas de formule similaire pour les arcs puisqu'il peut y avoir plusieurs arcs de différents types entre deux noeuds donnés. De plus, nous avons défini le prédicat $Edge(x, y, z)$ de telle sorte que deux arcs du même type ne puissent pas être définis.
- **Les métaéléments de noeud sont des classes.** Tout noeud utilisé comme métaélément d'un autre noeud doit avoir le noeud $class$ comme métaélément.

  $$\forall x \exists y Node(x, y) \wedge Node(y, \text{class}) \tag{5.3}$$

- **Les métaéléments des arcs sont des références.** Un arc ne peut exister qu'entre noeuds et doit avoir le noeud $reference$ comme type.

$$\forall xyzEdge(x,y,z) \Rightarrow (\exists x_t Node(x,x_t)) \wedge (\exists y_t Node(y,y_t)) \wedge Node(z,\text{reference}) \tag{5.4}$$

- **Cible d'un arc.** Un arc typé par la référence z ne peut avoir pour cible un noeud typé par la classe y_t que si la valeur de la référence *type* de z est y_t.

$$\forall xyzEdge(x,y,z) \Rightarrow (\exists y_t IsKindOf(y,y_t) \wedge Edge(z,y_t,\text{type})) \tag{5.5}$$

- **Source d'un arc.** Un arc typé par la référence z ne peut avoir pour source un noeud typé par la classe x_t que si z est une référence de x_t.

$$\forall xyzEdge(x,y,z) \Rightarrow (\exists x_t IsKindOf(x,x_t) \wedge Edge(x_t,z,\text{features})) \tag{5.6}$$

- **Unicité du type d'une référence.** Une référence n'a qu'un type.

$$\forall xyzEdge(x,y,\text{type}) \wedge Edge(x,z,\text{type}) \Rightarrow y = z \tag{5.7}$$

  Cette contrainte est nécessaire en *SimpleKM3* puisque le concept de multiplicité n'est pas défini. Si cela était le cas, la propriété *type* serait de multiplicité `[1-1]`.

5.3.2 Ajout des références opposées

Les références opposées fonctionnent par paire et permettent la navigation bidirectionnelle dans les modèles. Par exemple, dans notre première version de *SimpleKM3*, bien qu'il soit possible de récupérer les références d'une classe, il n'est pas possible de récupérer directement la classe possédant une référence donnée. La figure 5.4 définit la référence *opposite* appartenant et ciblant la classe *reference*.

9. $Node(\text{opposite},\text{reference})$
10. $Edge(\text{reference},\text{opposite},\text{features})$
11. $Edge(\text{opposite},\text{reference},\text{type})$

Figure 5.4 – Ajout des références opposées à *SimpleKM3*

Un modèle *SimpleKM3* (modèle terminal, métamodèle ou métamétamodèle) avec des références opposées est valide si les formules suivantes, en plus des formules présentées dans la section précédente, sont vérifiées :

- **Unicité de l'opposée.** Une référence a au plus une référence opposée.

$$\forall xyzEdge(x,y,\text{opposite}) \wedge Edge(x,z,\text{opposite}) \Rightarrow y = z \tag{5.8}$$

  Cette contrainte est nécessaire en *SimpleKM3* puisque le concept de multiplicité n'est pas défini. Si cela était le cas, la propriété *opposite* serait de multiplicité `[0-1]`.
- **Les références opposées fonctionnent par paire.**

$$\forall xyEdge(x,y,\text{opposite}) \Rightarrow Edge(y,x,\text{opposite}) \tag{5.9}$$

- **Les références opposées sont des extrémités inversées.**

$$\forall xyzEdge(x, y, \text{opposite}) \wedge Edge(z, x, \text{features}) \Rightarrow Edge(y, z, \text{type}) \tag{5.10}$$

Tous les modèles *SimpleKM3* ont maintenant la possibilité d'avoir des paires de références opposées. Ceci s'applique en particulier au métamétamodèle lui-même. Nous pouvons donc maintenant étendre *SimpleKM3* avec une référence *owner* opposée à la référence *features* comme présenté sur la figure 5.5. La définition résultante de *SimpleKM3* correspond au diagramme de classes donné à la figure 5.6. Il est maintenant possible de naviguer d'une référence (*reference*) à sa classe (*class*).

12. $Node(\text{owner}, \text{reference})$
13. $Edge(\text{reference}, \text{owner}, \text{features})$
14. $Edge(\text{owner}, \text{class}, \text{type})$
15. $Edge(\text{owner}, \text{features}, \text{opposite})$
16. $Edge(\text{features}, \text{owner}, \text{opposite})$

Figure 5.5 – Ajout de la référence *owner* opposée à la référence *features* dans *SimpleKM3*

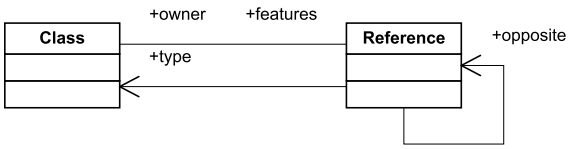

Figure 5.6 – Diagramme de classes de *SimpleKM3* avec références opposées

5.3.3 Ajout de l'héritage

En KM3, l'héritage permet la réutilisation des références définies dans les classes parentes, mais aussi le polymorphisme. La redéfinition ou le masquage d'une référence héritée ne sont pas autorisés. La figure 5.7 introduit la référence *supertypes* de *class* vers *class*. La figure 5.8 donne le diagramme de classes correspondant à *SimpleKM3* avec héritage. Afin de pouvoir utiliser les références héritées ou de définir des arcs ciblant des classes filles d'une classe donnée, nous redéfinissons le prédicat $IsKindOf(x)$ (voir la formule 5.1) :

$$\forall xyIsKindOf(x, y) \Leftrightarrow Node(x, y) \vee (\exists zNode(x, z) \wedge ConformsTo(z, y)) \tag{5.11}$$

Cette nouvelle définition utilise le prédicat $ConformsTo(x, y)$ défini récursivement comme suit :

$$\begin{aligned} \forall xyConformsTo(x, y) \Leftrightarrow (x = y) \vee \\ (\exists zEdge(x, z, \text{supertypes}) \wedge ConformsTo(z, y)) \end{aligned} \tag{5.12}$$

Les cycles d'héritage sont donc interdits. Le prédicat $ConformsTo(x, y)$ ne serait pas défini sinon. Avec ces nouvelles définitions, les formules (5.6) et (5.5) restent valides.

5.3.4 Autres concepts de KM3

Nous avons défini la sémantique formelle des concepts KM3 restants : paquetage, classes abstraites, types de données, attributs, énumérations, composition, multiplicité, etc. Ceci n'est cependant pas détaillé ici. Une définition complète de KM3 en Prolog est disponible en annexe A.3. Ce programme Prolog

17. $Node$(supertypes, reference)
18. $Edge$(class, supertypes, features)
19. $Edge$(supertypes, class, type)

Figure 5.7 – Ajout de l'héritage à *SimpleKM3*

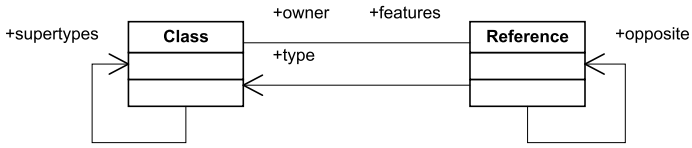

Figure 5.8 – Diagramme de classes de *SimpleKM3* avec références opposées et héritage

utilise les mêmes prédicats que ceux définis ici plus le prédicat $Prop(x, y, z)$ où $x \in N_G$, $y \in N_\omega$ est un attribut et z sa valeur. Nous ne détaillons pas plus ce prédicat, qui est un raccourci pour éviter de représenter explicitement les valeurs primitives comme des noeuds du graphe. L'ensemble des contraintes implémentées est une illustration des possibilités de KM3.

5.4 Travaux similaires

D'autres plateformes de modélisation offrent des capacités similaires à celles de KM3 :

- **OMG MOF.** MOF est le métamétamodèle standard de l'OMG. Il en existe différentes versions, dont MOF 1.4 [54] et MOF 2.0 [62]). Toutes sont plus complexes que KM3 : elles contiennent plus de classes (voir section 5.2.1). Aucune n'a de sémantique formelle. Leur syntaxe concrète standard est XMI, basé sur XML, et donc plus verbeux que KM3. Ainsi qu'il a été remarqué dans la section 5.2.1, nous avons défini des transformations ATL de MOF 1.4 vers KM3 et de KM3 vers MOF 1.4.
- **HUTN.** Human Usable Textual Notation [58] (HUTN) est un standard de l'OMG pour donner une notation textuelle par défaut à tout métamodèle. Lorsque HUTN est appliqué à MOF, le résultat est plus verbeux que KM3. Ceci est dû au fait que HUTN définit des règles de correspondances pour n'importe quel métamodèle vers une grammaire. Il n'y a donc pas d'adaptation particulière au métamétamodèle.
- **Eclipse EMF Ecore.** Ecore [14] est un métamétamodèle proche de MOF 2.0 mais avec une notation textuelle en plus de XMI : emfatic. Une différence avec KM3 est que emfatic fournit des constructions spécifiques à EMF, par exemple pour diriger la génération de code Java. Nous avons expérimenté l'encodage de certaines de ces constructions dans des commentaires KM3 avec succès. Une autre différence est que Ecore n'a pas de sémantique formelle. Ainsi qu'il a été remarqué dans la section 5.2.1, nous avons défini des transformations ATL de Ecore vers KM3 et de KM3 vers Ecore.
- **Graphes typés.** Les graphes typés [21] forment la théorie sur laquelle la transformation de graphe est basée. Ils ont une sémantique formelle précise. Contrairement à KM3 et aux définitions données à la section 4.2, il n'offrent pas de métamétamodèle explicite. Les graphes typés ne sont pas eux-mêmes typés.

5.5 Conclusion

Ce chapitre propose le DSL de définition de métamodèles KM3 avec sa notation textuelle. Nous avons décrit quelques autres approches similaires telles que XMI ou emfatic. Chaque approche à ses particularités, avantages et inconvénients. Pour emfatic, par exemple, la projection vers Java est une fonctionnalité importante. Pour XMI, la possibilité de prendre en compte les modèles terminaux en plus des métamodèles est essentielle.

KM3 est un langage textuel simple pour la définition de métamodèles permettant la création et la modification de métamodèles. Les métamodèles exprimés en KM3 ont de bonnes propriétés de lisibilité. Le formalisme est suffisamment riche pour supporter les informations essentielles. Des informations supplémentaires peuvent être exprimées sous la forme de métadonnées encapsulées dans des commentaires. Les métamodèles exprimés en KM3 peuvent être facilement convertis vers et depuis d'autres notations telles que emfatic et XMI.

L'une des propriétés de KM3 est la possibilité de définir des métamodèles non basés sur MOF. KM3 a aussi été conçu pour franchir les espaces techniques.

La contribution essentielle de ce chapitre est une sémantique propre pour un langage de définition de métamodèles. À notre connaissance, une telle définition n'a pas encore été proposée pour un tel langage. Tous les outils disponibles dans la plateforme AMMA sont complètement basés sur cette définition opérationnelle.

CHAPITRE 6

Le langage TCS[a]

[a] Le contenu de ce chapitre est partiellement adapté de [40].

6.1 Introduction

Beaucoup de problèmes relatifs aux syntaxes concrètes textuelles sont déjà résolus dans l'espace technique des grammaires (ou *grammarware*). Il n'y a aucune raison de redévelopper de tels outils dans l'espace technique de l'ingénierie des modèles. Ce dont nous avons besoin est un projecteur entre ces espaces. TCS (*Textual Concrete Syntax* ou syntaxe concrète textuelle) est un langage qui permet la spécification et la génération automatique de projecteurs entre les espaces techniques de l'ingénierie des modèles et des grammaires pour chaque langage dédié.

Ce chapitre est organisé de la manière suivante. La section 6.2 présente une vue d'ensemble de l'approche TCS. La section 6.3 introduit l'exemple qui sera utilisé tout au long du chapitre pour illustrer TCS. Le langage TCS est ensuite présenté en plusieurs étapes : constructions basiques à la section 6.4, génération de grammaire pour ces constructions basiques à la section 6.5, constructions additionnelles à la section 6.6, gestion de la table des symboles à la section 6.7 et constructions spécifiques à la génération de texte à partir d'un modèle à la section 6.8. La section 6.9 détaille des problèmes liés à l'implémentation de TCS. Quelques travaux similaires sont présentés à la section 6.10. Enfin, la section 6.11 donne les conclusions.

6.2 Vue d'ensemble

La vue d'ensemble de l'utilisation du langage TCS est représentée par la figure 6.1. Supposons que nous voulions construire un langage dédié appelé L. Dans l'espace technique de l'ingénierie des modèles, nous fournissons un métamodèle de L appelé MM_L exprimé en KM3. La définition de la syntaxe concrète est exprimée en TCS et est appelée CS_L. Le pont requis entre les deux espaces techniques comprend un *injecteur* et un *extracteur*. L'injecteur prend un modèle exprimé dans la syntaxe concrète textuelle de L et génère un modèle conforme à MM_L dans l'espace technique de l'ingénierie des modèles. Un exemple de modèle est appelé SM_L et est conforme à la grammaire de L appelée G_L. G_L est exprimé en *ANTLR*. L'extracteur génère la représentation textuelle des modèles de l'espace technique de l'ingénierie des modèles conformes à MM_L. La figure 6.1 montre un exemple dans lequel un modèle M_L est extrait en SM_L.

Notre approche commence avec le métamodèle et la spécification de la syntaxe concrète textuelle d'un langage L. Notre objectif est d'obtenir trois entités pour L : sa grammaire annotée G_L exprimée en ANTLR ainsi que le couple formé par l'injecteur et l'extracteur. G_L est générée par une transformation écrite en ATL appelée *TCS2ANTLR.atl*. Elle prend MM_L et CS_L en entrée, ce qui est représenté par des lignes en pointillés, et génère les règles et annotations de G_L. Cette grammaire est utilisée pour générer l'injecteur. L'injecteur est un analyseur syntaxique généré par les outils fournis par la technologie

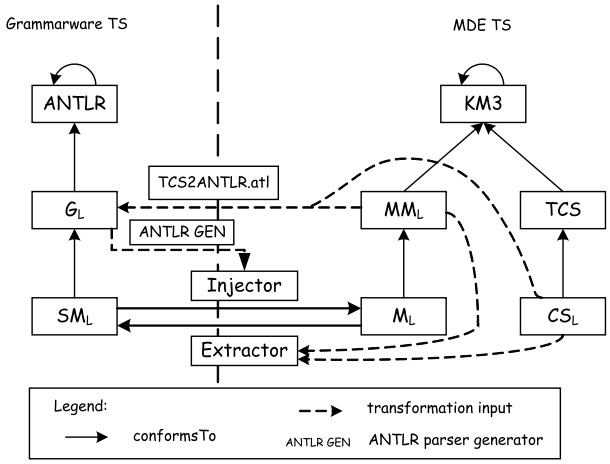

Figure 6.1 – Vue d'ensemble de l'utilisation de TCS

ANTLR. La génération est réalisée par le générateur d'analyseur syntaxique ANTLR (appelé *ANTLR GEN*).

L'extracteur travaille sur la représentation interne des modèles exprimés en L et crée leur représentation textuelle. Il est possible de générer un extracteur pour chaque langage L. Cependant, nous avons choisi une autre approche. Un unique extracteur est implémenté sous la forme d'un interprète qui fonctionne avec tous les langages. L'extracteur prend un modèle M_L écrit en L, son métamodèle MM_L et sa syntaxe décrite en TCS et génère la représentation textuelle SM_L de M_L.

Utiliser TCS est typiquement plus simple que de développer des injecteurs et extracteurs ad-hoc. Une seule spécification est suffisante pour les deux directions. De plus, la redondance entre un modèle TCS et le métamodèle correspondant est réduite (par exemple les multiplicités et types des propriétés sont omis en TCS). Avec un outil idéal, les syntaxes abstraites et concrètes devraient être spécifiées indépendamment sans que la structure de l'une ait une incidence sur celle de l'autre. Cependant, la puissance de simplification de TCS a un certain prix. L'écart structurel entre un métamodèle et un modèle TCS est limité. Ceci signifie que des compromis doivent être faits : soit la syntaxe est adaptée aux possibilités de TCS, soit le métamodèle est simplifié.

Une contrainte importante imposée par TCS sur les métamodèles est qu'ils doivent avoir un élément racine. Ceci correspond grossièrement au symbole initial de la grammaire correspondante. D'autres limites seront présentées dans la section 6.9.

6.3 Exemple courant : SPL

SPL est utilisé comme exemple tout au long de ce chapitre. Nous commençons par montrer à quoi ressemble la syntaxe concrète de SPL. Le listing 6.1 donne le code d'un programme SPL simple qui fait suivre les appels entrants vers l'adresse `sip:phoenix@barbade.enseirb.fr`. Le service `SimpleForward` (lignes 1-11) déclare l'adresse cible (ligne 3) et une session d'enregistrement (lignes 6-10). Cette session contient une méthode `INVITE` (lignes 6-8) qui redirige les appels entrants vers l'adresse déclarée (ligne 7).

Listing 6.1 – Programme SPL simple

```
1 service SimpleForward {
```

```
2   processing {
3     uri us = 'sip:phoenix@barbade.enseirb.fr';
4
5     registration {
6       response incoming INVITE() {
7         return forward us;
8       }
9     }
10  }
11 }
```

Les explications données dans ce chapitre sur le fonctionnement de TCS sont illustrées par leur application à SPL. Nous donnons donc des extraits du métamodèle de SPL en KM3 ainsi que des extraits correspondants de la spécification de la syntaxe concrète en TCS. Les extraits de métamodèles sont nécessaires car TCS fonctionne en annotant cette syntaxe abstraite. Seuls des sous-ensembles du métamodèle SPL et de sa syntaxe sont donnés dans ce chapitre. Les spécifications complètes du métamodèle et du modèle TCS de SPL sont données en annexe B. Il s'agit des listings B.1 pour le métamodèle et B.2 pour le modèle TCS.

Considérons tout d'abord l'extrait de métamodèle donné par le listing 6.2. Celui-ci commence par la déclaration du type de donnée *String* correspondant aux chaînes de caractères. Il spécifie ensuite qu'un programme (*Program* lignes 3-5) SPL contient (ligne 4) exactement un service (*Service* lignes 7-11). Ce dernier a un nom (`name`) de type *String* (ligne 8), des déclarations de type *Declaration* (ligne 9) et des sessions de type *Session* (ligne 10).

Listing 6.2 – Extrait du métamodèle SPL : *Program* et *Service*

```
1  datatype String;
2
3  class Program extends LocatedElement {
4    reference service container : Service;
5  }
6
7  class Service extends LocatedElement {
8    attribute name : String;
9    reference declarations[*] ordered container : Declaration;
10   reference sessions[*] ordered container : Session;
11 }
```

Le listing 6.3 donne un extrait du modèle TCS spécifiant la syntaxe concrète des éléments telle qu'elle a été présentée dans le listing 6.1. En voici une description informelle :

- **String.** Le type de donnée *String* est représenté par un identificateur (`identifier`) correspondant au non-terminal `NAME` de l'analyseur lexical (ligne 1).
- **Program.** La classe *Program* est représentée en tant que son service (lignes 3-5).
- **Service.** La classe *Service* est représentée par : le mot-clé `service`, le nom (`name`) du service, le symbole `{`, le mot-clé `processing`, le symbole `{`, les déclarations (`declarations`) du service, ses sessions (`sessions`) et deux symboles `}` (lignes 7-14).

Les éléments TCS sont associés à leurs éléments correspondants du métamodèle par leur nom. Par exemple, le patron (`template`) *Program* correspond à la classe KM3 *Program*. La propriété *service* correspond à la propriété KM3 du même nom. Cet exemple montre qu'il est relativement simple de coder une syntaxe textuelle peu complexe en TCS : les éléments syntaxiques apparaissent dans l'ordre de la syntaxe.

Listing 6.3 – Extrait du modèle TCS de SPL : *Program* et *Service*

```
1  primitiveTemplate identifier for String default using NAME;
2
```

```
3  template Program main
4    : service
5    ;
6
7  template Service
8    : "service" name "{"
9        "processing" "{"
10         declarations
11         sessions
12       "}"
13     "}"
14   ;
```

Une description détaillée des constructions basiques de TCS utilisées ici ainsi que de leur sémantique est donnée dans la section 6.4. La section 6.5 détaille la génération d'une grammaire à partir de ces constructions. Les sections 6.6, 6.7 et 6.8 présentent des constructions plus complexes de TCS.

6.4 Constructions basiques

Cette section présente les constructions TCS basiques, qui sont illustrées dans le listing 6.3 pour la plupart. Par défaut, les numéros de lignes donnés dans cette section font donc référence à ce listing.

Chaque *Classifier* du métamodèle est associé à un patron (*Template*) TCS. Ce patron spécifie comment représenter textuellement les valeurs (primitives ou éléments de modèles) typées par ce *Classifier*. Il y a deux principales sortes de patrons TCS :

- **PrimitiveTemplates.** Les patrons primitifs spécifient le terminal de l'analyseur lexical correspondant à un certain type de donnée (*DataType*) du métamodèle, identifié par son nom. Plusieurs patrons primitifs peuvent être définis pour un seul type de donnée. C'est typiquement le cas des chaînes de caractères : un patron les représente comme identificateur et un autre comme liltéral. Un seul patron primitif peut être déclaré par défaut (mot-clé `default`) pour chaque type de donnée. La ligne 1 définit le patron primitif par défaut `identifier` (i.e. identificateur) pour le type de donnée *String* qui correspond au terminal `NAME`. Ce dernier doit être défini dans l'analyseur lexical.
- **ClassTemplates.** Les patrons de classe définissent comment les classes sont représentées. Cette définition consiste en une séquence d'éléments syntaxiques tels que : mots-clés (par exemple `if`), symboles spéciaux (par exemple `+`), etc. Plus d'informations sur les éléments syntaxiques sont données ci-dessous. Un patron de classe a le même nom que la classe dont il définit la représentation. Un seul patron doit être déclaré comme étant la racine à l'aide du mot-clé `main` (voir ligne 3 pour le patron *Program*). Contrairement aux patrons primitifs, seul un patron de classe peut être défini pour chaque classe du métamodèle. Ce choix permet de simplifier les spécifications écrites en TCS. Nos expérimentations ont montré que ce choix n'imposait pas de restriction significative.

Les éléments syntaxiques sont utilisés pour représenter le contenu des classes. Ils peuvent être de différentes sortes :

- **Mots-clés.** Un mot-clé est un mot réservé porteur d'une signification particulière. En SPL, les mots-clés `service` (ligne 8) et `processing` (ligne 9) introduisent des parties bien déterminées du programme. Un mot-clé est défini entre guillemets (`"`) en TCS.
- **Symboles Spéciaux.** Un symbole spécial est une séquence de caractères utilisée comme séparateur ou opérateur (par exemple `{` aux lignes 8 et 9). Il est lui aussi défini entre guillemets. Chaque symbole doit de plus être listé dans la section correspondante (`symbols`) du modèle TCS.
- **Propriétés.** Une propriété TCS correspond à une propriété KM3 (attribut ou référence) de la classe

associée au patron dans lequel la propriété TCS est utilisée ou bien dans l'une de ses classes mères. Une propriété est définie par un identificateur dont la valeur est le nom de sa propriété KM3 associée. La représentation textuelle d'une propriété dépend de sa propriété KM3 associée et plus particulièrement de son type et de sa multiplicité. Pour simplifier, nous ferons désormais référence au type et à la multiplicité de la propriété TCS puisqu'elle est associée à une unique propriété KM3. Des arguments optionels peuvent être spécifiés entre accolades (`{` et `}`) après la propriété. Les arguments les plus utilisés sont détaillés ci-dessous. L'identificateur `service` à la ligne 4 est une propriété TCS correspondant à la propriété KM3 du même nom dans la classe *Program* (listing 6.2, ligne 2).

Ainsi qu'il est écrit ci-dessus, la représentation textuelle d'une propriété dépend de son type T. Il y a deux possibilités correspondant aux deux principales sortes de patrons présentées précédemment :

- **Type Primitif.** Quand T est un type de donnée (*DataType*) un patron primitif est utilisé. Ce patron est choisi parmi ceux associés à T. Un patron précis peut être spécifié par son nom en utilisant l'argument `as = <name>` entre accolades après l'identificateur de la propriété. Si aucun patron n'est explicitement défini, alors il doit exister un patron par défaut correspondant à T. Ce patron par défaut sera alors utilisé. La propriété `name` à la ligne 8 est associée au type de donnée *String* (voir listing 6.2, ligne 8). Le patron primitif `identifier` défini à la ligne 1 est donc utilisé pour représenter cette valeur.
- **Classe.** Quand T est une classe, le patron de classe correspondant à T est utilisé. Le patron `Service` défini aux lignes 7-14 est donc utilisé pour représenter la propriété `service` à la ligne 4.

La multiplicité de la propriété est utilisée pour connaître le nombre de fois qu'il faut appliquer le patron. Un séparateur à placer entre chaque application du patron peut être défini en utilisant l'argument `separator = <separator>`.

6.5 Génération d'une grammaire pour les constructions basiques

D'après la sémantique de chaque construction TCS informellement présentée précédemment, une grammaire peut être générée à partir d'un métamodèle KM3 et d'un modèle TCS. Nous avons implémenté cette traduction sous la forme de la transformation *TCS2ANTLR.atl*. Le listing 6.4 donne l'extrait de grammaire correspondant aux extraits KM3 et TCS des listings 6.2 et 6.3. La grammaire est exprimée dans le syntaxe d'ANTLR version 2 (ANTLRv2). Les annotations automatiquement générées pour la construction du modèle pendant l'analyse syntaxique en ont été retirées. Ces annotations rendraient l'exemple plus verbeux et moins lisible. Un analyseur lexical approprié, définissant les terminaux `NAME`, `LCURLY` et `RCURLY` existe. Nous nous concentrons ici sur la syntaxe.

Listing 6.4 – Extrait de la grammaire SPL sans les annotations : *Program* et *Service*

```
1  identifier
2    : NAME
3    ;
4
5  program
6    : service
7    ;
8
9  service
10   : "service" identifier LCURLY
11     "processing" LCURLY
12       (declaration (declaration)*)?
13       (session (session)*)?
```

```
14          RCURLY
15        RCURLY
16    ;
```

La transformation *TCS2ANTLR.atl* implémente un ensemble de règles déclaratives de traduction. Le code complet est fourni en annexe. Voici une brève description des règles utilisées pour la génération du listing 6.4 :

- **Patron Primitif vers Règle de Production.** Chaque patron primitif est traduit en une règle de production contenant le terminal correspondant (par exemple lignes 1-3). Ce détour est utilisé pour simplifier la génération des annotations. Les conversions de valeur (par exemple chaîne de caractères vers entier) peuvent ainsi être centralisées dans la règle correspondant au patron primitif plutôt que d'être répétées à chaque utilisation du terminal.
- **Patron de Classe vers Règle de Production.** Une règle de production est créée pour chaque patron de classe (par exemple lignes 5-7 et 9-16). Le nom de la règle est le nom du patron avec une miniscule en première position (imposé par ANTLRv2 pour les non-terminaux). Le contenu de la règle est dérivé du contenu du patron (voir règles ci-dessous). Les éléments syntaxiques apparaissent dans le même ordre.
- **Mots-clés et Symboles Spéciaux vers Terminaux.** Les mots-clés sont transformés en terminaux littéraux (par exemple `"service"`, ligne 10). Les symboles spéciaux sont transformés en terminaux non littéraux (par exemple `LCURLY`, ligne 10). Ces terminaux non littéraux doivent être définis dans l'analyseur lexical (par exemple : `LCURLY: "{";`).

Les propriétés sont gérées différemment suivant leur multiplicité. La table 6.1 résume la gestion des multiplicités de TCS. Les correspondances ne sont pas immédiates car les multiplicités imposées par TCS sont en réalité une simplification de celles offertes par KM3. La raison en est la suivante. ANTLRv2 n'offre pas la possibilité de gérer les multiplicités complexes telles qu'avec une borne inférieure $n > 1$ ou encore une borne supérieure fixe $m > 1$. Plutôt que d'implémenter une traduction complexe en développant les multiplicités complexes, nous avons choisi de simplifier la grammaire cible. Rien n'empêche d'améliorer cela dans le futur, mais nous avons observé que cela n'était pas vraiment gênant pour les syntaxes que nous avons traitées. En effet, les multiplicités les plus courantes sont : `[1-1]` pour les éléments requis, `[0-1]` pour les éléments optionnels, `[0-*]` pour un nombre quelconque d'éléments et `[1-*]` pour au moins un élément. Parmi ces multiplicités, le seul cas qui n'est pas directement géré par TCS est celui des éléments optionnels (`[0-1]`). La raison en est que nous avons préféré exiger une optionalité explicite. Une propriété ne peut donc être déclarée optionnelle en TCS qu'en la plaçant dans une construction conditionnelle (voir la section 6.6). Voici la description des règles traitant les propriétés en fonction de leur multiplicité :

- **Propriété Mono-valuée vers Non-terminal.** Une propriété mono-valuée est simplement transformée en un non-terminal, même quand elle est en fait optionnelle (voir ci-dessus) dans le métamodèle. Le non-terminal dérivé de la propriété a le nom de la règle correspondant au type de la propriété (classe ou type de donnée).
- **Propriété Multi-valuée vers Non-terminaux.** Chaque propriété multi-valuée est traduite en une séquence de deux non-terminaux du même nom. Le nom de ces non-terminaux est celui de la règle correspondant au type de la propriété. Le second non-terminal est suivi d'une construction de répétition ($\star$ en ANTLRv2). Les propriétés avec borne supérieure fixe (représentée par m dans la table, $m > 1$) sont gérées comme non bornées. Les séparateurs entre éléments, si il y en a, sont placés juste avant le non-terminal et à l'intérieur du bloc répété. Quand la borne inférieure (appelée n dans la table) est égale à un, rien de plus n'est nécessaire. Quand elle est supérieure à un, elle est gérée comme si elle était égale à un. Quand la borne inférieure est zéro, une construction option-

nelle est ajoutée (`?` en ANTLRv2). C'est par exemple le cas pour `declaration` et `session` aux lignes 12 et 13. Des cas particuliers tels que ceux du listing 6.4 pourraient être simplifiés (par exemple `(declaration)*` au lieu de `(declaration (declaration)*)?`) car aucun séparateur n'est utilisé. Cependant, cela n'est pas nécessaire en pratique. Nous avons donc décidé de ne pas introduire de complexité additionnelle dans la règle de transformation.

Type de propriété	Multiplicité		Traitement par TCS
	borne inférieure (lower)	borne supérieure (upper)	
Mono-valuée	0 1	1 1	Exactement une occurrence
Multi-valuée	$0 \leq n \leq m$	$m > 1$	Comme si $m = *$ (voir ci-dessous)
	0	*	Aucun ou plus
	1	*	Un ou plus
	$1 < n$	*	Comme si $n = 1$

Table 6.1 – Gestion des multiplicités par TCS

6.6 Constructions additionnelles

Dans les sections précédentes nous avons vu comment les constructions TCS basiques peuvent être utilisées pour définir une syntaxe simple. Ces constructions basiques ne sont cependant pas toujours suffisantes ou pratiques pour gérer des syntaxes plus complexes. Nous décrivons ici de nouvelles constructions TCS qui aident à dépasser certaines limites des constructions basiques. Leur sémantique est brièvement décrite. Les règles pour générer une grammaire à partir de ces constructions ne sont pas détaillées.

- **Patron de Classe Abstrait.** Les patrons de classe abstraits permettent d'utiliser l'héritage dans le métamodèle. Pour chaque patron abstrait, une règle de production est générée. Elle a la forme d'une alternative de non-terminaux correspondant aux sous-classes de la classe associée au patron abstrait. Cette construction est typiquement utilisée avec des classes abstraites.
- **Construction Conditionnelle.** Ce type de construction est utilisé lorsque la présence d'une séquence d'éléments syntaxiques dans la syntaxe concrète est soumise à condition. Une construction conditionnelle spécifie une condition, une séquence S_1 d'éléments syntaxiques à utiliser quand la condition est vrai et une séquence optionnelle S_2 à utiliser sinon. Il est toujours possible d'évaluer la condition pendant la sérialisation (modèle vers texte). La condition est de plus limitée à des expressions réversibles qui peuvent être utilisées pour assigner une valeur précise aux propriétés testées pendant l'analyse syntaxique. La condition est une conjonction d'expressions simples parmi :
  - Une propriété booléenne qui est positionnée à vrai si S_1 est reconnue et à faux si c'est S_2.
  - Une comparaison entre une propriété de type entier ou énuméré et un littéral qui peut être utilisé pour positionner la propriété à cette valeur si S_1 est reconnue. Si S_2 est reconnue, elle doit définir une valeur pour la propriété si celle-ci n'est pas optionnelle.
  - Un test de définition pour une propriété mono-valuée ou multi-valuée utilisant la syntaxe : `isDefined(<property>)`. La propriété doit être utilisée/initialisée dans S_1 et pas dans

S_2.

Une construction conditionnelle est utilisée dans le listing 6.10 à la ligne 2. Une déclaration de variable est représentée par son type (`type`), suivi de son nom (`name`), puis d'une expression d'initialisation optionnelle (`initExp`) après un symbole d'égalité (=) et enfin d'un point-virgule. L'expression d'initialisation est optionelle et le symbole d'égalité ne doit être présent que si elle est définie. Une construction conditionnelle est utilisée pour tester si une expression d'initialisation est définie et pour seulement représenter le symbole d'égalité et l'expression si c'est bien le cas. Nous pouvons voir que la décision décrite dans la section 6.5 d'imposer une optionalité explicite ne change rien dans le cas présent. Puisque l'expression d'initialisation est précédée d'un symbole d'égalité lui aussi optionnel, une construction conditionnelle doit impérativement être utilisée. En pratique, nous avons observé que cette situation est relativement courante.

6.7 Table des symboles

Les constructions syntaxiques de TCS présentées jusqu'à maintenant permettent la définition de syntaxes relativement complexes. Par exemple, les syntaxes concrètes de SPL, KM3 et TCS pourraient être définies avec ces constructions seulement. Il y a cependant une limite importante : nous avons uniquement vu comment représenter les compositions, pas les références simples. En se limitant aux compositions, les modèles sont limités à des arbres. En utilisant des références qui traversent la hiérarchie de composition les modèles deviennent des graphes. Dans le reste de cette section, nous appelons ces références des références simples.

Une fonctionnalité de TCS appelée table des symboles permet l'utilisation de références simples. Le terme "table des symboles" est un emprunt du terme équivalent de la théorie de la compilation. Cette fonctionnalité va maintenant être illustrée sur la déclaration et l'utilisation des variables en SPL. Nous décrivons tout d'abord un problème relatif aux références simples dans la section 6.7.1. Puis nous donnons deux solutions différentes. Nous montrons dans la section 6.7.2 comment contourner le problème en faisant un compromis sur le métamodèle. Enfin, nous montrons dans la section 6.7.3 comment la table des symboles TCS peut être utilisée pour obtenir une meilleure alternative.

6.7.1 Description du problème

Considérons tout d'abord l'extrait du métamodèle SPL donné dans le listing 6.5. Il correspond à la déclaration et à l'utilisation des variables en SPL. Une déclaration (*Declaration*, lignes 1-3) a un nom (`name`, ligne 2). Une déclaration de variable (*VariableDeclaration*, lignes 5-8) est une déclaration avec un type (`type`, ligne 6) et une expression d'initialisation (`initExp`, ligne 7) optionnelle. Une variable (*Variable*, lignes 10-12) fait référence à sa déclaration via la référence `source` (ligne 11). Cette référence n'est pas une composition car sa définition en KM3 n'inclut pas le mot-clé `container` : c'est une référence simple.

Listing 6.5 – Extrait du métamodèle SPL : déclaration et utilisation de variables

```
1 abstract class Declaration {
2   attribute name : String;
3 }
4
5 class VariableDeclaration extends Declaration {
6   reference type container : TypeExpression;
7   reference initExp[0-1] container : Expression;
8 }
9
```

```
10 class Variable {
11   reference source : Declaration;
12 }
```

Le listing 6.6 donne un premier codage naïf de la syntaxe correspondante. La propriété `source` (ligne 6) est spécifiée comme si elle était une composition (i.e. comme `type` et `initExp`, ligne 2), ce qui ne fonctionne pas. La grammaire générée pour cet extrait montre le problème.

Listing 6.6 – Extrait erroné du modèle TCS : déclaration et utilisation de variables

```
1 template VariableDeclaration
2   : type name (isDefined(initExp) ? "=" initExp) ";"
3   ;
4
5 template Variable
6   : source
7   ;
```

Le listing 6.7 donne l'extrait de la grammaire SPL générée à partir de la définition erronée donnée dans le listing 6.6. La propriété `source` a été transformée en un non-terminal correspondant à *VariableDeclaration*. Avec une telle grammaire, la ligne 7 du listing 6.1 ressemblerait à : `return forward uri us = ↪'sip:phoenix@barbade.enseirb.fr';`. Ceci n'est pas correct puisqu'elle devrait être : `return forward us;`.

Listing 6.7 – Extrait erroné de la grammaire ANTLR pour *Variable*

```
1 variable
2   : variableDeclaration
3   ;
```

En fait, la représentation textuelle d'une variable n'est pas la représentation de sa déclaration mais tout simplement un identificateur. Le listing 6.8 donne un extrait de la grammaire correcte. La propriété `source` est maintenant représentée par le non-terminal `identifier` correspondant à un identificateur. Cette nouvelle grammaire est bien une représentation de la syntaxe utilisée dans le listing 6.1.

Listing 6.8 – Extrait correct de la grammaire ANTLR pour *Variable*

```
1 variable
2   : identifier
3   ;
```

Il y a deux manières principales d'obtenir ce résultat. La première est de faire un compromis sur le métamodèle en interdisant l'utilisation de références simples. La seconde est d'utiliser la table des symboles TCS. Nous considérons successivement ces deux approches ci-dessous.

6.7.2 En faisant un compromis sur le métamodèle

Puisque les références simples sont un problème, essayons dans un premier temps de ne pas les utiliser. En conséquence, nous remplaçons la référence simple `source` de la classe *Variable* vers la classe *VariableDeclaration* par un attribut implémentant une référence par le nom. Le listing 6.9 donne l'extrait correspondant du métamodèle SPL. Le seul changement par rapport au listing 6.5 est qu'une variable fait maintenant référence à sa déclaration par son nom (attribut `referredVariableName`, line 11). Ce type de référence n'attache pas directement une variable à sa déclaration et est typique des arbres syntaxiques abstraits (notés ASTs pour Abstract Syntax Tree).

Listing 6.9 – Extrait du métamodèle SPL : *VariableDeclaration*, version arborescente

```
1 abstract class Declaration {
2   attribute name : String;
```

```
}

class VariableDeclaration extends Declaration {
  reference type container : TypeExpression;
  reference initExp[0-1] container : Expression;
}

class Variable {
  attribute referredVariableName : String;
}
```

L'extrait correspondant du modèle TCS est donné par le listing 6.10. Il utilise seulement des constructions présentées dans les sections précédentes. La grammaire correspondante est la version correcte qui a été donnée dans le listing 6.8. La propriété `referredVariableName` est en effet transformée en le non-terminal `identifier` puisque son type est le type de donnée *String*.

Listing 6.10 – Extrait du modèle SPL : *VariableDeclaration*, version arborescente

```
template VariableDeclaration
  : type name (isDefined(initExp) ? "=" initExp) ";"
  ;

template Variable
  : referredVariableName
  ;
```

La figure 6.2 donne une représentation simplifiée du modèle correspondant au listing 6.1. Nous ne représentons pas les types de la déclaration et de la méthode, la direction de la méthode ainsi que d'autres détails non pertinents ici. Cependant, avec la solution qui vient d'être présentée, la flèche en pointillés n'existe pas. Le modèle est donc limité à un arbre.

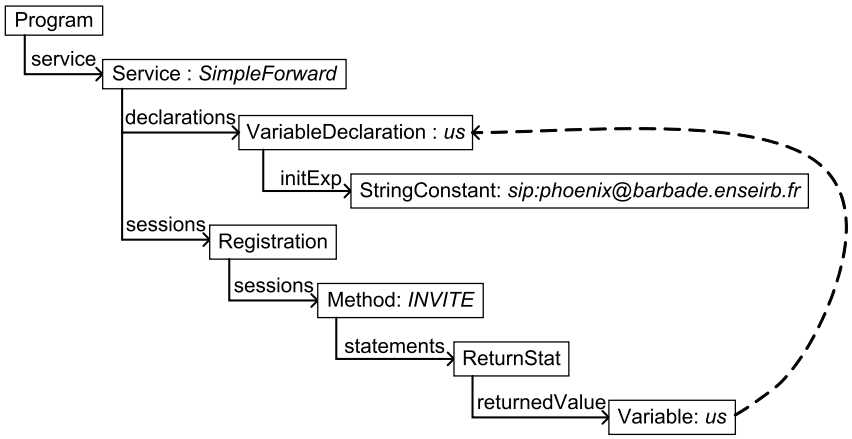

Figure 6.2 – Représentation simplifiée du modèle correspondant à l'exemple SPL

6.7.3 En utilisant la table des symboles TCS

Nous allons montrer comment la table des symboles de TCS peut être utilisée pour représenter les références simples. L'objectif est, d'un côté, de générer la même grammaire, donnée dans le listing 6.8, qu'avec la solution précédente. De l'autre côté, le modèle obtenu après utilisation de l'injecteur doit avoir des références simples et ne plus être limité à un arbre. La flèche en pointillés de la figure 6.2 devrait donc être directement représentée dans le modèle.

Nous allons utiliser le métamodèle donné dans le listing 6.5, dans lequel une variable pointe sur sa déclaration via la référence simple `source` (ligne 11). Le listing 6.11 donne l'extrait du modèle TCS correspondant. Premièrement, *VariableDeclaration* doit être ajouté à la table des symboles (mot-clé `addToContext`, ligne 1). Ceci signifie que chaque fois qu'une déclaration de variable est créée, elle est ajoutée à la table des symboles. Deuxièmement, la représentation de la propriété `source` est changée par rapport à l'approche naïve. L'argument `refersTo = name` (ligne 6) y est ajouté. Il signifie que chaque fois qu'une variable est reconnue, sa propriété `source` recevra pour valeur la déclaration (type connu d'après le métamodèle) ayant le nom (`name`) correspondant. Cette déclaration sera recherchée dans la table des symboles. La propriété ciblée par `refersTo` (`name` ici) doit être de type *String*.

Listing 6.11 – Extrait du modèle SPL : *VariableDeclaration*, version avec références simples

```
1 template VariableDeclaration addToContext
2   : type name (isDefined(initExp) ? "=" initExp) ";"
3   ;
4
5 template Variable
6   : source{refersTo = name}
7   ;
```

Les grammaires générées à partir des listings 6.10 et 6.11 sont identiques à l'exception de leurs annotations. Ce résultat était attendu car dans les deux cas on obtient la grammaire correcte de SPL. La différence est dans la structure définie par le métamodèle : un arbre pour le listing 6.10 et un graphe pour le listing 6.11. Des annotations adéquates sont générées à partir du modèle TCS donné par le listing 6.11 pour :

- **Patron VariableDeclaration** (ligne 1) : un morceau de code plaçant la déclaration dans la table des symboles est ajouté à la règle de production générée.
- **Propriété Source** (ligne 6) : un morceau de code recherchant la déclaration dans la table des symboles est ajouté après le non-terminal correspondant à la propriété `source`. La recherche est effectuée en cherchant une déclaration dont le nom (`name`) correspond à l'identificateur de la variable.

L'analyseur syntaxique généré résoud les références simples avec la table des symboles après avoir lu la totalité du texte source. Ceci signifie que les références en avant sont autorisées. Les références qui ne peuvent pas être résolues (par exemple l'utilisation d'une variable non définie) ou qui peuvent être résolues en plusieurs cibles (par exemple lors d'une double déclaration de la même variable) sont rapportées comme erreurs dans un modèle de diagnostique. Certains DSLs peuvent interdire les références en avant qui devraient donc aussi être rapportées comme erreurs. Dans ce cas, une vérification appropriée doit être effectuée sur le modèle injecté. Ceci peut être réalisé en utilisant la technique présentée à la section 8.3 du chapitre 8 ou encore dans [11].

La gestion de la table des symboles de TCS est en réalité un peu plus complexe que ce qui a été présenté ici. Il est notamment possible d'avoir plusieurs tables des symboles imbriquées. Chaque patron de classe peut en effet spécifier la création d'une nouvelle table des symboles dont la portée est limitée au contenu de ce patron. Ceci se déclare à l'aide du mot-clé `context` dans la déclaration d'un patron. Cette fonctionnalité est par exemple utilisée pour empêcher qu'une variable déclarée dans une méthode soit utilisée depuis une autre. Dans ce but, le patron *Method* est déclaré avec le mot-clé `context`.

6.8 Constructions spécifiques à la sérialisation d'un modèle en texte

Un modèle TCS définit la syntaxe concrète d'un DSL et peut être appliqué dans les deux directions : texte vers modèle et modèle vers texte. Il y a, cependant, des aspects spécifiques à la direction modèle

vers texte : le style de codage et l'indentation. Pour générer un texte lisible, ces aspects doivent aussi être pris en compte dans le modèle TCS. Le style de codage n'a aucune influence sur la grammaire, seulement sur la sérialisation des caractères blancs (ou tout autre caractère ignoré). Des éléments syntaxiques additionnels sont fournis pour la sérialisation :

- **Bloc.** Un bloc TCS fournit des informations sur l'indentation. Un bloc est délimité par des crochets (`[` et `]`). Par défaut, chaque élément directement contenu dans le bloc est placé sur une ligne séparée avec un niveau d'indentation de plus que le niveau précédent. Chaque bloc peut aussi avoir des arguments spécifiques permettant de régler plus finement l'indentation. Nous en citons deux ici :
  - `nbNL` est utilisé pour définir le nombre de sauts de ligne à insérer entre les éléments (`nbNL = 1` par défaut).
  - `indentIncr` est utilisé pour définir le nombre de niveaux d'indentation ajouté au niveau courant par le bloc (`indentIncr = 1` par défaut).

  Le listing 6.12 montre comment les informations d'indentation peuvent être ajoutées au patron *Service* originellement défini dans le listing 6.3. Le bloc autour des propriétés `declarations` et `sessions` aux lignes 3-6 définit que le contenu de `"processing""{""}"` doit être indenté. De plus, deux sauts de ligne doivent être insérés entre les éléments. Le bloc extérieur aux lignes 2-7 définit que le contenu de `"service"name"{""}"` doit être indenté. Le bloc intérieur à ces même lignes définit que son contenu doit être traité comme un élément simple, sans saut de ligne (`nbNL = 0`) ni incrément d'indentation (`indentIncr = 0`). Ce bloc sert à s'assurer que `processing` et `{` ne seront pas placés sur deux lignes séparées. Avec les informations ajoutées au patron *Service*, une indentation correcte telle que présentée dans le listing 6.1 est réalisée.
- **Espacement des Symboles Spéciaux.** La déclaration de chaque symbole spécial peut spécifier comment les espaces doivent être placés autour du symbole. Par défaut, les symboles ne sont ni précédés ni suivis d'espace car cela n'est généralement pas nécessaire pour obtenir une grammaire non ambiguë. `leftSpace` (resp. `rightSpace`) déclare que le symbole doit être précédé (resp. suivi) d'un espace. `leftNone` (resp. `rightNone`) déclare que le symbole ne doit pas être précédé (resp. suivi) d'un espace, même si le symbole précédent (resp. suivant) déclare `rightSpace` (resp. `leftSpace`).
- **Séparateurs Spéciaux.** Quand les constructions présentées ci-dessus ne sont pas suffisantes, des séparateurs spéciaux peuvent être utilisés. Par exemple : `<space>` force l'écriture d'un espace et `<newline>` force l'écriture d'un saut de ligne.

Listing 6.12 – Extrait du modèle TCS de SPL : *Service* avec indentation

```
1 template Service context
2   : "service" name "{" [ [
3       "processing" "{" [
4         declarations
5         sessions
6       ] {nbNL = 2} "}"
7     ] {nbNL = 0, indentIncr = 0} ] "}"
8   ;
```

Bien qu'aucune expérience n'ait encore été menée dans cette direction, nous pensons que les informations d'indentation définies en TCS pourrait aussi être utilisées par un éditeur de texte pour fournir une indentation automatique.

6.9 Problèmes relatifs à l'implémentation de TCS

Dans un premier temps, nous mentionnons deux fonctionnalités de TCS qui ne sont pas directement liées aux constructions du langage :

- **Traçabilité.** L'implémentation actuelle de TCS fournit une traçabilité texte vers modèle en gardant les informations de numéro de ligne et de colonne dans les modèles. Par exemple, le débogueur ATL fait usage de cette information pour indiquer l'emplacement dans le code source où une erreur a lieu.
- **Éditeur Générique.** L'éditeur textuel générique (TGE pour Textual Generic Editor) est un outil qui est partiellement construit sur les possibilités de TCS. Il fait partie du projet AM3 [26]. TGE fournit un éditeur de texte paramétré pour chaque langage par des informations dérivées du modèle TCS correspondant : un modèle conforme à un métamodèle *Editor* (pour éditeur) créé par transformation (en ATL) à partir du modèle TCS. Une vue d'ensemble arborescente (outline en anglais) du texte est générée en utilisant les capacités d'injection texte vers modèle de TCS. Des liens hyper-texte et des informations sous forme de bulles présentant un extrait du texte pointé par le lien sont aussi fournis automatiquement en utilisant les possibilités de traçabilité de TCS.

Bien que TCS permette la définition de syntaxes relativement complexes, certaines limites apparaissent. Dans un second temps, nous présentons ces limites des outils TCS actuels. Nous donnons aussi des indications sur des solutions potentielles :

- **Rapports d'Erreurs.** Les erreurs possibles dans les modèles TCS peuvent être à deux niveaux différents. Premièrement, certaines erreurs dans les modèles source TCS et KM3 peuvent empêcher la génération de la grammaire cible. Ces erreurs peuvent typiquement être exprimées sous la forme de contraintes OCL sur ces modèles source. En conséquence, la vérification d'erreur est implémentée en ATL à l'aide de la solution présentée à la section 8.3 et dans [11]. Deuxièmement, même quand la grammaire cible est syntaxiquement correcte, elle peut être ambiguë. Les non-déterminismes rapportés par le générateur d'analyseur syntaxique (ANTLRv2 dans notre cas) ne sont pas exprimés en fonction des éléments TCS. Une solution possible à ce problème serait d'implémenter une certaine traçabilité entre les modèles TCS et KM3 d'une part et la grammaire d'autre part. La discussion à propos des classes de grammaire ci-dessous présente une solution complémentaire. Le nombre d'ambiguïtés peut être réduit en utilisant un générateur d'analyseur syntaxique plus puissant.
- **Classes de Grammaires.** La classe de grammaire supportée par un ensemble donné d'outils TCS dépend du générateur d'analyseur syntaxique utilisé. Par exemple, avec ANTLRv2, il s'agit d'une approximation linéaire de LL(k). La nouvelle version d'ANTLR (version 3, ou ANTLRv3) est LL(*) [64]. Porter TCS vers ANTLRv3 nécessite d'adapter la grammaire générée à la syntaxe ANTLRv3 mais aussi à son interface de programmation. Cette dernière est en effet utilisée par les annotations de la grammaire. Cette solution fournirait un outil plus puissant : moins de grammaire sont ambiguës en LL(*) qu'en LL(k). De la même manière, TCS pourrait aussi être porté vers d'autres générateurs d'analyseur syntaxique tels que yacc, qui est LALR(1).
- **Analyse Lexicale.** Les questions relatives aux analyseurs lexicaux ne seront pas détaillées. Bien que TCS ait un support préliminaire pour définir les analyseurs lexicaux, ils doivent toujours être partiellement écrits dans la syntaxe ANTLRv2. L'étude de l'utilisation habituelle des analyseurs lexicaux devrait permettre d'étendre TCS avec des constructions appropriées.
- **Langages Insensibles à la Casse.** Les langages pour lesquels les mots-clés et identificateurs sont équivalents quelle que soit la combinaison de majuscules et minuscules utilisées ne sont pas encore correctement supportés. Des expérimentations préliminaires dans la grammaire et l'interprète TCS

ont montré que ce problème ne devrait pas être très difficile à résoudre.

- **Langages Utilisants des Espaces comme Délimiteurs.** Les langages pour lesquels l'indentation a une signification sont un autre défi pour TCS puisque ceci exige une coopération étroite entre analyseur lexical et analyseur syntaxique. Les blocs TCS qui ne servent pour l'instant qu'à améliorer la présentation pourraient probablement être étendus dans cet objectif. Des séparateurs spéciaux pourraient aussi être utilisés pour représenter les caractères d'espacements obligatoires. Cependant, nous ne pensons pas que ce problème soit facile à résoudre dans le cas général.
- **Références Complexes.** La version actuelle de TCS ne supporte que les références basées sur des chaînes de caractères. Un exemple en est le lien entre une variable et sa déclaration qui a été présenté dans la section 6.7. Il y a cependant des scénarios plus complexes, tels que l'attachement d'un appel de méthode à sa déclaration correspondante (par exemple en Java). Ce genre de cas ne peut pas être géré par la version actuelle de TCS car il est nécessaire de supporter des références beaucoup plus complexes que celles basées sur des identifiants simples (par exemple inférence de type en Java). Une solution possible est d'avoir un métamodèle pivot entre la grammaire et le métamodèle désiré. Dans ce pivot, que l'on peut appeler métamodèle syntaxique, tous les compromis nécessaires à l'utilisation de TCS sont faits. Ensuite, des transformations entre ces deux métamodèles peuvent être écrites pour résoudre les références complexes. Cette technique du pivot peut aussi être utilisée pour contourner d'autres limites de TCS.

6.10 Travaux similaires

Il y a plusieurs approches permettant de donner des syntaxes concrètes textuelles aux DSLs. Dans cette section, nous nous concentrons sur les DSLs dont la syntaxe abstraite est définie par un métamodèle et pour lesquels une syntaxe textuelle est définie. Nous commentons ci-dessous quelques approches permettant de donner des syntaxes concrètes aux langages de modélisation dans le contexte du IDM :

- **XMI.** L'OMG (Object Management Group) a défini un standard de sérialisation par défaut des modèles appelé XML Model Interchange [60] (XMI). Ce standard est basé sur XML, qui peut être considéré comme un cas très particulier de syntaxe textuelle. Un avantage d'XML est que les documents XML peuvent être lus très efficacement sans connaître leur schéma (i.e. métamodèle). Un autre avantage de XMI comparé à TCS est qu'il n'y a rien à définir en dehors du métamodèle. XMI définit des règles de traduction automatique d'un métamodèle en schéma XML. Cependant, la syntaxe XMI est relativement verbeuse. L'objectif de XMI est la sérialisation et l'échange de modèles entre les outils de modélisation. Il est difficile pour des développeurs de l'utiliser directement pour exprimer des modèles.
- **HUTN.** L'OMG a aussi standardisé Human Usable Textual Notation [58] (HUTN) qui fournit une sérialisation standard des modèles à l'aide d'une syntaxe textuelle plus concise que XML. HUTN est similaire à TCS et une implémentation nécessite généralement un générateur d'analyseur syntaxique, ce qui n'est pas le cas pour XMI. En revanche, contrairement à TCS, la grammaire utilisée par HUTN est automatiquement générée. Un avantage évident de cette approche est que n'importe quel modèle peut être représenté textuellement à un coup quasiment nul. Cependant, HUTN impose des contraintes très strictes sur la notation utilisée. Les utilisateurs ne peuvent pas définir leur propre syntaxe. TCS permet aux utilisateurs de définir la syntaxe qu'ils souhaitent avec une flexibilité beaucoup plus grande que HUTN.
- **Patrons de Génération de Code.** Des outils comme EMF JET [14] (Java Emitter Templates) permettent la génération flexible de code. Ces solutions sont globalement unidirectionnelles (modèle

vers texte) mais offrent une indépendance quasi totale entre le métamodèle source et la grammaire cible. Il n'est même pas nécessaire d'avoir une grammaire. Il est de ce fait relativement commun de voir des générateurs de code écrits avec des patrons qui réalisent une transformation de modèles en même temps. Par exemple, la génération de code Java à partir d'un modèle UML peut être réalisée en une étape avec cette solution. Ceci peut être intéressant dans certains cas, mais nous pensons que séparer la transformation de modèles de la génération de code est en général plus intéressante. Pour l'exemple UML vers Java cela implique d'avoir un métamodèle Java explicite. Un modèle UML est d'abord transformé en un modèle conforme au métamodèle Java puis ce dernier modèle est sérialisé en code. Nous voyons au moins deux avantages à cette approche. Premièrement, le métamodèle cible (par exemple Java) peut être réutilisé dans d'autre buts : calcul de métriques, réarrangement de code, transformation vers ou depuis d'autres langages, etc. Deuxièmement, la correspondance conceptuelle entre les langages source et cible (UML et Java ici) est explicite alors que dans la génération directe de code elle est cachée dans du code orienté syntaxe.

- **MOF Model to Text.** XMI et HUTN ne sont pas adaptés à la génération de code car il n'y a aucun contrôle de la syntaxe cible. Un autre standard de l'OMG est donc en développement pour gérer ce problème. Il s'appelle MOF Model to Text [59]. L'appel à proposition demande un langage de traduction unidirectionnel : modèle vers texte. Les commentaires et exemples donnés ci-dessus à propos des patrons de génération de code s'appliquent aussi à cette solution. De plus, nous nous attendons à ce qu'il y ait bientôt un autre standard OMG en chantier : Text to MOF Model.
- **Règles de dérivation.** Un ensemble de règles permettant de dériver une grammaire à partir d'un métamodèle sont données dans [32]. Ces règles sont similaires à celles de TCS, mais le processus ne semble pas avoir été automatisé. Le développeur doit donc lui-même appliquer ces règles afin d'obtenir une grammaire. Des indications sont fournies afin de permettre à l'analyseur syntaxique généré à partir de la grammaire de créer le modèle dans le système de manipulation de modèles MDR [51].
- **Génération de métamodèles à partir de grammaires.** Les approches présentées ci-dessus (y compris TCS) supposent l'existence d'un métamodèle pour lequel une syntaxe concrète doit être définie. Certaines autres approches prennent le problème dans l'autre sens et supposent l'existence d'une grammaire. Ainsi, un algorithme de génération de métamodèles à partir de grammaires est proposé dans [3]. La lisibilité des métamodèles est limitée par la présence d'éléments syntaxiques conservés afin de pouvoir regénérer la grammaire initiale. Cependant, il ne semble pas qu'il soit possible de générer une grammaire à partir d'un métamodèle quelconque (i.e. non obtenu par l'algorithme proposé). De plus, puisque l'approche repose uniquement sur la grammaire, sans information complémentaire, il n'est pas possible de guider la génération du métamodèle. Par ailleurs, seuls des algorithmes en pseudo-code sont proposés et l'approche ne semble pas validée en pratique. Une approche similaire est proposée dans [77] avec les capacités supplémentaires suivantes : un algorithme de simplification automatique des métamodèles générés et un mécanisme semi-automatique d'amélioration (e.g. pour la définition de références simples). Un prototype de cette approche a été implémenté sur la plateforme Eclipse. Cependant, seule la traduction des programmes en modèles est définie et non la traduction des modèles en programmes.
- **Définition d'une Syntaxe Concrète Visuelle.** Le travail présenté dans [29] propose une approche pour la définition de syntaxes visuelles pour les langages de modélisation. Elle est basée sur la définition d'un ensemble de classes médiatrices qui lient les éléments du métamodèle du langage aux classes représentant les éléments visuels (boîtes, flèches, etc.). TCS diffère principalement de cette approche sur deux points. TCS est ciblé sur les syntaxes concrètes textuelles et non visuelles. Au lieu d'utiliser un système basé sur des classes médiatrices nous utilisons un DSL pour spécifier

les relations entre métamodèle et grammaire.

6.11 Conclusion

Dans ce chapitre, nous avons présenté TCS : un DSL pour l'implémentation des syntaxes concrètes textuelles des DSLs définis dans ou avec AMMA. Les constructions du langage TCS permettent au développeur de DSL d'établir des correspondances entre les éléments du métamodèle du langage et leur représentation syntaxique.

Notre approche a plusieurs avantages. Premièrement, le développeur est libéré de la nécessité de définir une grammaire et ses annotations dans le but de générer un analyseur syntaxique. À la place, il peut se concentrer sur les patrons syntaxiques pour les constructions de son langage et obtenir la grammaire annotée automatiquement. Deuxièmement, l'utilisation d'un langage tel que TCS entraîne une meilleure séparation des préoccupations. Les détails du générateur d'analyseur syntaxique sont cachés, ce qui facilite le remplacement d'un générateur par un autre. Ainsi, l'implémentation actuelle repose sur ANTLR version 2 qui utilise des grammaires LL(k). Changer pour une autre technologie ne devrait principalement nécessiter qu'une nouvelle transformation ATL générant une grammaire annotée pour le nouvel outil. TCS pourrait ainsi bénéficier d'un générateur d'analyseur syntaxique plus puissant tel que ANTLR version 3 qui utilise des grammaires LL(*) ou d'autres outils tels que ceux utilisant des grammaires LALR(1). Troisièmement, les définitions écrites en TCS permettent la génération automatique de ponts bidirectionnels qui réalisent les tâches de conversion texte vers modèle et modèle vers texte.

L'automatisation que nous poursuivons a un certain prix : il faut faire certains compromis sur les syntaxes abstraites et concrètes. L'utilisation de TCS entraîne une restriction de la liberté d'adaptation de la syntaxe par rapport aux approches dans lesquelles la grammaire est définie à la main et un analyseur syntaxique dédié est développé pour un unique langage. Cependant, notre objectif est de fournir une solution pour le développement rapide de la syntaxe concrète des DSLs. Si le problème est de développer un unique langage, potentiellement à usage général, alors les efforts de développement d'un analyseur syntaxique dédié valent probablement le coût. Si, au contraire, un grand nombre de DSLs doivent être développés rapidement, alors une solution automatisée est une meilleure option.

En dehors de l'exemple présenté tout au long de ce chapitre, sur le langage SPL, nous avons expérimenté TCS sur d'autres langages d'AMMA : KM3, ATL et TCS lui-même. Tous ces langages ont une syntaxe concrète spécifiée en TCS. Le résultat de ces expérimentations est encourageant puisqu'il montre que TCS peut gérer des syntaxes non-triviales telles que la syntaxe d'ATL, qui utilise OCL sans faire de compromis critique à part le sacrifice de la plupart des raccourcis syntaxiques.

CHAPITRE 7

Le langage ATL[a]

[a]Le travail présenté dans ce chapitre est partiellement adapté de [43] et [41].

7.1 Introduction

Ce chapitre présente le langage ATL (ATLAS *Transformation Language* pour langage de transformation ATLAS) de transformation pour l'ingénierie des modèles. Le domaine de la transformation de modèles a été défini au chapitre 2. Les transformations spécifiées en ATL correspondent à la définition 3 et opèrent dans le contexte présenté à la figure 2.5. Un état de l'art des approches de transformation de l'ingénierie des modèles à été présenté au chapitre 3.

Des définitions et descriptions de différentes versions du langage ATL ont été publiées au cours de nos travaux, par exemple dans [10], [41], [43] et [42]. Le moteur d'exécution (compilateur et machine virtuelle) a de même évolué en parallèle. La description d'ATL donnée dans ce chapitre correspond à la version *ATL'2006*. Le moteur d'exécution ainsi que les outils de développement correspondant à cette version ont été publiés en 2006.

Ce chapitre est organisé de la manière suivante. La section 7.2 donne une vue d'ensemble d'ATL en rappelant son contexte opérationnel. La section 7.3 présente la structure d'ATL, ses règles, ses constructions impératives et son algorithme d'exécution. La section 7.4 décrit le prototype développé au cours de nos travaux. Enfin, la section 7.5 donne les conclusions.

7.2 Vue d'ensemble de l'approche de transformation ATL

ATL s'utilise dans le contexte de transformation présenté dans la figure 7.1. Dans ce schéma, un modèle source M_a est transformé en un modèle cible M_b. La transformation est dirigée par un programme de transformation *mma2mmb.atl* écrit en ATL. Ce programme est un modèle. Les modèles source et cible ainsi que le programme de transformation sont conformes à leurs métamodèles respectifs : MM_a, MM_b et ATL. Ces métamodèles sont conformes au métamétamodèle KM3.

ATL est un langage de transformation hybride. Il contient un mélange de constructions déclaratives et impératives. Nous encourageons l'utilisation du style déclaratif. Cependant, il est parfois difficile de développer une solution complètement déclarative à certains problèmes. Dans un tel cas, le développeur peut utiliser les possibilités impératives du langage.

Les transformations ATL sont unidirectionnelles. Une transformation bidirectionnelle est typiquement implémentée par un couple de transformations : une pour chaque direction.

Les transformations ATL opèrent sur des modèles source en lecture seule et produisent des modèles cible en écriture seule. Pendant l'exécution d'une transformation, les modèles source peuvent être navigués mais pas modifiés. Les modèles cible ne peuvent pas être navigués. Ces restrictions permettent de simplifier la sémantique d'exécution et notamment de garantir un résultat déterministe sans deman-

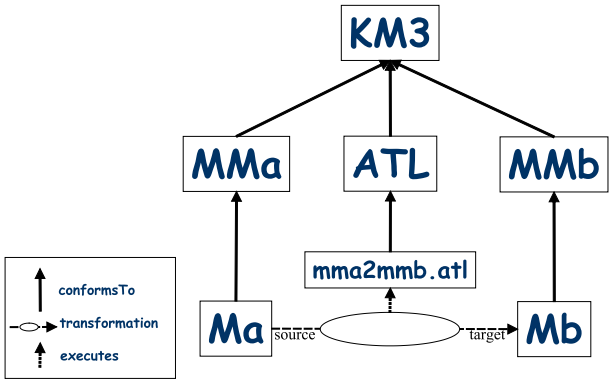

Figure 7.1 – Vue d'ensemble de l'approche de transformation ATL

der au développeur de définir explicitement un ordre d'exécution des règles (voir discussion à la section 7.3.5.3).

7.3 Présentation d'ATL

Dans cette section, nous présentons les fonctionnalités du langage ATL. La syntaxe du langage est présentée sur la base d'exemples (sections 7.3.1 à 7.3.4). Ensuite, dans la section 7.3.5, nous décrivons la sémantique d'exécution d'ATL.

7.3.1 Structure globale des programmes de transformation

En ATL, une transformation s'appelle un module. Un module contient un en-tête, un ensemble d'importation de bibliothèques de fonctions et un ensemble de fonctions et de règles de transformation. Les fonctions sont appelées *helper* en ATL.

L'en-tête donne le nom du module de transformation et déclare les modèles source et cible. Le listing 7.1 donne un exemple d'en-tête.

Listing 7.1 – En-tête d'un programme ATL

```
1 module SimpleClass2SimpleRDBMS;
2 create OUT : SimpleRDBMS from IN : SimpleClass;
```

L'en-tête commence par le mot-clé `module` suivi du nom du module. Ensuite, les modèles source et cible sont déclarés comme des variables typées par leurs métamodèles. Le mot-clé `create` indique les modèles cible. Le mot-clé `from` indique les modèles source. Dans notre exemple, le modèle cible est représenté par le variable `OUT` à partir du modèle source représenté par `IN`. Les modèles source et cible sont respectivement conformes aux métamodèles `SimpleClass` et `SimpleRDBMS`. En général, plus d'un modèle source et d'un modèle cible peuvent être listés dans l'en-tête.

Les fonctions et règles de transformation sont les constructions utilisées pour définir une transformation. Elles sont expliquées dans les deux sections suivantes.

7.3.2 Helpers

Les fonctions ATL sont appelées *helpers* d'après le standard OCL ([57], section 7.4.4, p11) sur lequel ATL se base. OCL définit deux sortes de *helpers* : opération et attribut.

En ATL, un *helper* peut être spécifié dans le contexte d'un type OCL (par exemple *String* ou *Integer*) ou d'un type source (venant de l'un des métamodèles source). Les modèles cible ne sont en effet pas navigables. Les *helpers* opération peuvent être utilisés pour définir des opérations dans le contexte d'un élément de modèle ou du module de transformation. Le rôle principal des *helpers* opération est de réaliser la navigation des modèles source. Ils peuvent avoir des paramètres et peuvent utiliser la récursivité. Les *helpers* opération définis dans le contexte d'éléments de modèles permettent les appels polymorphiques. Puisque la navigation n'est autorisée que sur les modèles source en lecture seule, une opération retourne toujours la même valeur pour un contexte et un ensemble d'arguments donnés.

Les *helpers* attribut sont utilisés pour associer des valeurs nommées en lecture seule sur les éléments de modèles source. Comme les opérations, ils ont un nom, un contexte et un type. La différence est qu'ils ne peuvent pas avoir de paramètre. Leur valeur est définie par une expression OCL. Comme les opérations, les attributs peuvent être définis récursivement avec les mêmes contraintes de terminaison et de cycles.

Les *helpers* attribut sont pratiquement comme les propriétés dérivées de MOF 1.4 [54] ou de Ecore [14] mais peuvent être associés à une transformation. Ils ne sont pas nécessairement liés à un métamodèle donné. Alors que dans EMF [14] et MDR [51] ils sont implémentés en Java, ils sont définis en OCL avec ATL.

Les *helpers* attribut peuvent être considérés comme un moyen de décorer les modèles source avant l'exécution de la transformation. La décoration d'un élément de modèle peut dépendre de la déclaration d'autres éléments. Pour illustrer la syntaxe des *helpers* attribut nous considérons un exemple.

Listing 7.2 – *Helper* attribut

```
1 helper context SimpleClass!Class def: allAttributes : Sequence(SimpleClass!Attribute) =
2   self.attrs->union(
3     if not self.parent.oclIsUndefined() then
4       self.parent.allAttributes->select(attr |
5         not self.attrs->exists(at | at.name = attr.name)
6       )
7     else Sequence {}
8     endif
9   )->flatten();
```

Le *helper* `allAttributes` est utilisé pour déterminer l'ensemble des attributs d'une classe donnée, y compris les attributs définis et hérités. Ce *helper* est associé aux classes du modèle source. Ceci est indiqué par le mot-clé `context` suivi d'une référence au type `SimpleClass!Class` dénotant la class *Class* du métamodèle *SimpleClass* (ligne 1). La valeur retournée par cet attribut est une séquence d'attributs (ligne 2). L'expression OCL utilisée pour calculer la valeur est donnée après le symbole = (lignes 3-10).

Il s'agit d'un exemple utilisant la récursivité en faisant référence à la valeur du même attribut sur la classe parente de la classe courante (ligne 5). Si la classe n'a pas de parent alors une séquence vide est utilisée (ligne 8). Il s'agit du cas de terminaison de la récursion.

Les *helpers* attribut peuvent aussi être utilisés pour établir des liens entre éléments de différents modèles source. Le type d'un attribut peut en effet être une classe provenant d'un métamodèle différent du métamodèle de son contexte. Ceci correspond à une forme élémentaire de composition de modèle.

7.3.3 Règles de transformation

La règle de transformation est la construction élémentaire en ATL pour exprimer la logique de transformation. Les règles ATL peuvent être soit déclaratives soit impératives. Nous considérons ici les premières, appelées *matched rules*. La section 7.3.4 décrit les constructions impératives d'ATL.

7.3.3.1 Règles déclaratives : matched rules

Une *matched rule* est composée d'un motif source et d'un motif cible. Le motif source d'une règle définit un ensemble de types source (venant des métamodèles source) et une garde sous la forme d'une expression OCL booléenne. Un motif source est évalué en un ensemble de tuples dans les modèles source.

Le motif cible est composé d'un ensemble d'éléments. Chacun de ces éléments définit un type cible (venant d'un métamodèle cible) et un ensemble d'affectations appelées *bindings*. Un *binding* fait référence à une propriété (attribut ou référence) du type et spécifie une expression dont la valeur est utilisée pour initialiser la propriété.

Le morceau de code suivant montre une règle déclarative simple.

Listing 7.3 – Exemple de règle déclarative

```
1  rule PersistentClass2Table{
2    from
3      c : SimpleClass!Class (
4        c.is_persistent and c.parent.oclIsUndefined()
5      )
6    to
7      t : SimpleRDBMS!Table (
8            name <- c.name
9      )
10 }
```

Le nom de la règle (*PersistentClass2Table*) est donné après le mot-clé `rule` (ligne 1). Le motif d'entrée définit une variable de type `SimpleClass!Class` (ligne 3). La garde (ligne 4) spécifie que seules les classes persistantes sans class mère sont retenues.

Le motif cible contient un élément de type `SimpleRDBMS!Table` (lignes 7-9). Cet élément a un *binding* (ligne 8) qui définit une expression utilisée pour initialiser l'attribut `name` (le nom de la table). Le symbole `<-` est utilisé pour délimiter la propriété à initialiser à gauche de l'expression servant à l'initialiser à droite.

7.3.3.2 Sémantique d'exécution des règles déclaratives

Les règles déclaratives sont exécutées pour chaque tuple reconnu par leur motif source. Dans un premier temps, nous considérons l'exécution d'une seule règle sur un unique tuple. La reconnaissance du motif source et le déclenchement d'une règle sont expliqués par la suite.

Pour un tuple donné, les éléments cible dont les types sont définis dans le motif cible sont créés dans les modèles cible et initialisés à l'aide des *bindings*. L'exécution d'une règle sur un tuple entraîne de plus la création d'un lien de traçabilité dans les structures internes du moteur de transformation. Ce lien met trois composants en relation : la règle, le tuple d'entrée et l'ensemble des éléments cible nouvellement créés. Les liens de traçabilités peuvent être considérés comme un modèle et sérialisés par un moteur ATL en tant que produit supplémentaire de l'exécution d'une transformation. Cependant, nous montrons à la section 8.2 du chapitre 8 ainsi que dans [37] qu'il existe d'autres méthodes plus flexibles pour implémenter la traçabilité en ATL.

L'initialisation des propriétés utilise un algorithme spécifique de résolution des valeurs appelé algorithme de résolution ATL. Après l'évaluation de l'expression d'un *binding*, la valeur obtenue est d'abord résolue avant d'être assignée à la propriété cible correspondante. La résolution dépend du type de la valeur. Si son type est primitif, alors la valeur est simplement affectée à la propriété. Si son type est défini dans un métamodèle il y a deux possibilités :

- Quand la valeur est un élément cible, elle est simplement affectée à la propriété ;
- Quand la valeur est un élément source, elle est d'abord résolue en un élément cible à l'aide des liens de traçabilité. Le résultat de la résolution est un élément d'un modèle cible, qui est affecté à la propriété. Cet algorithme utilise les liens de traçabilité pour identifier les éléments cible créés pour un élément source donné par l'exécution d'une règle de transformation.

Deux cas se présentent lorsqu'il n'existe aucun lien de traçabilité pour un élément source donné, en fonction du mode d'exécution de la transformation :

- En mode **normal**, aucune valeur n'est affectée à la propriété et l'élément est simplement oublié.
- En mode **raffinage** (*refining* en anglais), l'élément source est copié dans le modèle cible. L'initialisation des propriétés de ce nouvel élément fait appel à l'algorithme de résolution. Ceci peut entraîner un appel récursif de cet algorithme. La récursion s'arrête lorsqu'il n'y a plus d'élément source à copier : soit par épuisement de la partie du modèle connectée à l'élément initial, soit par la résolution d'un élément transformé par une règle qui n'a donc pas besoin d'être initialisé automatiquement.

Le mode raffinage est spécifié par l'emploi du mot-clé *refining* à la place de *from* dans l'en-tête de la transformation. Ce mode a une utilité particulière en ATL. En effet, nous avons vu que les modèles source sont en lecture seule et les modèles cible en écriture seule. Ceci empêche la modification en place des modèles. Or, certains scénarios de transformation ne requièrent que des modifications limitées des modèles. Dans ce genre de cas, la partie non modifiée du modèle reste inchangée. Le mode raffinage permet de prendre en compte ces scénarios en effectuant une copie automatique des éléments source qui ne sont pas explicitement transformés. Sans ce mode d'exécution, il serait nécessaire d'écrire des règles de copie pour tous les éléments laissés inchangés.

Grâce à cet algorithme, les éléments cible peuvent être liés les uns aux autres en n'utilisant que la navigation dans les modèles source. Trouver les éléments cible correspondants relève du moteur d'exécution ATL.

7.3.3.3 Types de règles déclaratives

Il y a plusieurs types de règles déclaratives qui diffèrent par la manière dont elles sont déclenchées :

- **Les règles standard** sont appliquées une seule fois pour chaque tuple correspondant à leur motif d'entrée trouvé dans les modèles source.
- **Les règles paresseuses** (*lazy rules*) sont déclenchées par les autres règles. Elles sont appliquées sur chaque tuple autant de fois qu'il est référencé par les autres règles. Ceci signifie qu'une règle paresseuse peut être appliquée plusieurs fois sur un même tuple en produisant à chaque fois un nouvel ensemble d'éléments cible.
- **Les règles paresseuses uniques** (*unique lazy rules*) sont aussi déclenchées par les autres règles. Elles sont en revanche appliquées une unique fois pour chaque tuple. Si une règle paresseuse unique est déclenchée plus tard sur le même tuple, les mêmes éléments cible, créés la première fois, sont utilisés.

L'algorithme de résolution ATL décrit ci-dessus est aussi responsable du déclenchement des règles paresseuses et paresseuses uniques quand un élément source est référencé depuis une expression d'initialisation.

7.3.3.4 Héritage de règles

L'héritage de règles en ATL peut être utilisé comme moyen de réutilisation de code et aussi pour définir des règles polymorphiques.

Une règle (appelée fille) peut hériter d'une autre règle (appelée mère). Une règle fille reconnaît un sous-ensemble de ce que sa règle mère reconnaît. Ceci implique un ensemble de contraintes sur le motif source des règles filles que l'on peut résumer en "une règle fille ne peut pas changer la structure du motif source". Les types du motif source de la règle fille doivent soit être les mêmes que ceux de sa mère soit être remplacés par des sous-types de ceux de sa mère. La garde d'une règle fille n'est appliquée qu'aux éléments déjà reconnus par la règle mère. En pratique, la garde d'une règle fille est donc la conjonction des deux gardes.

Le motif cible d'une règle fille étend celui de sa mère en utilisant une combinaison des changements suivants : remplacer un type par un sous-type, ajouter des *bindings*, remplacer des *bindings* et ajouter des nouveaux éléments cible. Il est à noter qu'un *binding* ne peut pas être simplement étendu mais doit être totalement remplacé.

7.3.4 Constructions impératives d'ATL

Le style déclaratif a de nombreux avantages. Il est généralement basé sur la spécification des relations entre motifs source et cible et a ainsi tendance à être plus proche de la manière dont les développeurs perçoivent intuitivement une transformation. Ce style met l'accent sur le codage de ces relations et cache les détails liés à la sélection des éléments source, au déclenchement et à l'ordre d'exécution des règles, à la gestion de la traçabilité, etc. En conséquence, il peut cacher des algorithmes de transformation de modèles complexes derrière une syntaxe simple.

Cependant, dans certains cas, des algorithmes liés aux domaines source ou cible sont nécessaires (par exemple la diagonalisation d'une matrice). Il peut alors être difficile de définir une solution purement déclarative. Il y a plusieurs approches pour la résolution de ce problème :

- **Autoriser l'appel d'opérations natives.** Il devient ainsi possible d'utiliser le langage le plus adapté pour chaque algorithme nécessaire. Cette solution présente l'inconvénient de déplacer le flux de contrôle en dehors de la sémantique du langage de transformation.
- **Offrir une partie impérative dans le langage de transformation.** De cette manière, le flux de contrôle reste dans les limites de la sémantique du langage de transformation. En revanche, le développeur doit coder ce flux de contrôle explicitement. De plus, il y est plus difficile de garantir l'efficacité du code écrit par le développeur ou encore de l'optimiser. Néanmoins, le développeur peut réaliser lui-même les optimisations nécessaires, même en l'absence d'un optimiseur automatique.
- **Extraire les données à traiter.** Les données nécessitant un traitement spécifique peuvent être extraites vers un outil spécifique au domaine du traitement puis réinjectées en tant que modèles. Un inconvénient de cette approche est qu'elle nécessite un mécanisme relativement lourd comparé aux autres. Cependant, elle offre une très grande flexibilité pour le traitement, qui peut par exemple réutiliser des outils existants.

La troisième option s'applique sans problème à ATL puisqu'elle ne requiert pas de support explicite dans le langage de transformation. De plus, la plateforme AMMA fournit nativement certains outils facilitant cette approche : des injecteurs et extracteurs génériques tels que TCS. Les deux premières options ont un inconvénient en commun. Le traitement est exécuté en dehors du contrôle du moteur d'exécution : soit totalement dans le cas du code natif, soit partiellement avec des constructions impératives. Le résul-

tat est la réduction des avantages apportés par l'approche déclarative. L'avantage de ces deux premières solutions est qu'elles sont généralement moins coûteuses à mettre en œuvre.

Pour ces raisons, ATL contient une partie impérative. Les deux principales constructions impératives d'ATL sont :

- **Les règles appelées** (*called rules*) qui sont similaires à des procédures. Elles sont invoquées par leur nom et peuvent prendre des arguments. Leur implémentation peut être native ou spécifiée en ATL. Dans ce dernier cas, la définition ressemble à une règle déclarative sans motif d'entrée. En effet, la règle n'est pas exécutée sur des tuples reconnus dans les modèles d'entrée mais appelée.
- **Le bloc impératif** (*action block*) qui contient une séquence d'instructions impératives. Il peut être utilisé à la place de ou en combinaison avec un motif cible, dans des règles déclaratives ou impératives. Les instructions disponibles en ATL sont celles communément offertes par les langages impératifs. Ainsi, il y a, pour le contrôle de flux, une instruction conditionnelle, des boucles, mais aussi des affectations, etc.

Si soit une règle appelée soit un bloc impératif est utilisé dans un programme ATL, ce programme n'est plus totalement déclaratif. Dans certains cas, il peut être utile d'interdire l'utilisation de ces constructions.

7.3.5 Exécution des programmes de transformation

Dans cette section, nous discutons différents points liés à l'exécution des transformations ATL. Premièrement, nous présentons un algorithme en pseudo-code pour l'exécution des transformations. Ensuite, nous discutons quelques questions d'optimisation et donnons des informations sur la terminaison et le déterminisme des transformations.

7.3.5.1 Algorithme d'exécution des transformations ATL

L'algorithme suivant (listing 7.4) présente grossièrement la sémantique opérationnelle d'ATL. Il s'agit d'une simplification de celui implémenté dans le moteur de transformation *ATL'2006*. Il ne prend en compte que l'exécution des règles déclaratives standard sans héritage de règle. Le traitement des expressions OCL ainsi que des *helpers* n'est pas précisé car il relève de la sémantique du langage OCL, pas de celle de ATL. Le traitement des instructions impératives est classique et n'est pas donné non plus. De plus, l'algorithme de résolution a déjà été donné à la section 7.3.3.2 et n'est donc pas détaillé ici.

Listing 7.4 – Algorithme d'exécution des transformations ATL

```
exécuter la règle impérative marquée entrypoint
-- Ceci résulte en un contrôle de flux impératif classique.

-- Calcule des tuples reconnus par les règles déclaratives :
PourChaque règle standard R {
  PourChaque tuple candidat C de R {
    -- Un tuple candidat est un ensemble d'éléments ayant les types
    -- du motif source de la règle.

    évaluer la garde de R sur C
    Si la garde est vraie Alors
      créer les éléments cible du motif cible de R
      créer le lien de traçabilité pour (R, C et les éléments cible)
    Sinon
      ignorer C
    FinSi
  }
}

-- Application des règles déclaratives :
```

```
21 PourChaque lien de traçabilité T {
22   R <- la règle associée à T
23   C <- le tuple reconnu par R associé à T
24   P <- les éléments cible associés à T
25
26   -- Initialiser les éléments cible :
27   PourChaque élément cible E dans P {
28     -- Initialiser chaque propriété de E :
29     PourChaque binding B déclaré dans R pour E {
30       expression =  expression d'initialisation de B
31       valeur = évaluer expression dans le contexte de C
32       valeurFinale = résolution de valeur
33       positionner la propriété correspondant à B à valeurFinale
34     }
35   }
36   exécuter le bloc impératif de R dans le contexte de C et P
37   -- Le bloc impératif peut réaliser n'importe quelle navigation sur les éléments
38   -- source (C) et cible (P) et n'importe quelle action sur les éléments cible (P).
39   -- Le programmeur est responsable de la validité de ces opérations.
40 }
41
42 exécuter la règle impérative marquée endpoint
43 -- Nous avons à nouveau un flux impératif.
```

Cet algorithme commence par l'exécution de la règle impérative marquée comme point d'entrée (verb'entrypoint'), si une telle règle existe. Cette règle peut alors appeler d'autres règles impératives. Ensuite, l'algorithme calcule les tuples reconnus par les règles déclaratives standard. Pour chaque tuple reconnu, les éléments cible et le lien de traçabilité correspondants sont créés. Puis, les règles déclaratives sont appliquées en initialisant tous les éléments créés lors de l'étape précédente. Enfin, la règle impérative marquée comme point de sortie (`endpoint`) est appelée si elle existe.

L'algorithme d'exécution donné ici a été précisément défini à l'aide du formalisme des machines d'état abstraites (*Abstract State Machines* abrégé en ASMs) au cours d'une expérience présentée dans [67].

Remarques. L'application des règles ne se fait qu'une fois que tous les tuples reconnus par toutes les règles standard sont connus. Ce n'est pas absolument nécessaire pour l'exécution correcte d'une transformation ATL. Cependant, il est plus simple de décrire et d'implémenter un algorithme en deux étapes car tous les liens de traçabilité sont disponibles au moment de l'application des règles. Il est donc relativement simple de résoudre les éléments cible dans la deuxième étape. Les éléments cible pourraient aussi être initialisés au moment de leur création. Ceci aurait en revanche pour conséquence de rendre l'algorithme de résolution plus complexe car certaines initialisations devraient être retardées juqu'à la fin de l'exécution de la transformation.

L'algorithme présenté ne suppose pas un ordre particulier pour : la reconnaissance des tuples d'une règle donnée, la création des éléments cible correspondant à un tuple donné, l'initialisation des éléments cible pour un lien de traçabilité donné et l'initialisation des propriétés d'un élément cible donné. Le bloc impératif d'une règle, si il existe, ne doit cependant être exécuté pour un tuple donné qu'après l'application de la partie déclarative de cette règle pour ce tuple. Cette contrainte simplifie la tâche du programmeur puisque le motif cible est dans un état relativement prévisible. En fait, les contraintes liées à l'ordre d'exécution pourraient même être moins restrictives. Par exemple, toutes les propriétés des éléments cible pourraient être initialisées dans n'importe quel ordre sans changer le résultat.

7.3.5.2 Aperçu de quelques optimisations possibles

Nous avons vu qu'ATL impose les contraintes de lecture seule pour les modèles d'entrée et d'écriture seule pour les modèles de sortie. Ceci entraîne une grande flexibilité d'exécution des transformations ATL. Par exemple :

- L'évaluation de chaque expression de navigation produit le même résultat quel que soit le moment où elle se produit. Ceci est aussi vrai pour les sous-expressions identiques évaluées dans le même contexte.
- La reconnaissance de chaque règle déclarative standard produit les mêmes tuples quel que soit le moment où elle se produit.
- Lorsque les règles paresseuses uniques ne sont pas utilisées, la résolution d'un élément source produit le même élément cible quel que soit le moment où elle se produit. Lorsqu'elles sont utilisées, chacune de leurs applications produit la même structure d'éléments cible dans les modèles cible.

La simplicité conceptuelle d'ATL laisse donc de nombreuses possibilités d'exécution effective des transformations. Bien entendu, l'utilisation d'éléments impératifs réduit les possibilités en rajoutant des contraintes. Ainsi, les différentes instructions doivent en général être exécutées dans l'ordre spécifié. Nous n'avons pas étudié l'optimisation des transformations en général, mais nous pensons que la flexibilité de la sémantique d'ATL devrait permettre l'utilisation de différentes approches d'optimisation.

Par exemple, une seule itération pourrait être faite pour les motifs d'entrée avec la même structure mais des gardes différentes. Le moteur *ATL'2006* le fait déjà en présence d'héritage de règles mais ceci pourrait être généralisé. De plus, il n'est pas toujours nécessaire de tester tous les tuples candidats si certains peuvent être rejetés par une analyse statique de la garde. Considérons par exemple le métamodèle source donné dans le listing 7.5 dans lequel les éléments de type A (lignes 2-4) peuvent contenir (ligne 3) des éléments de type B (ligne 6). Le listing 7.6 présente une règle simplifiée (sans motif cible) qui reconnaît toutes les paires (a : Test !A, b : Test !B) d'éléments de type A et B pour lesquelles l'élément B est contenu dans l'élément A.

Listing 7.5 – Exemple de métamodèle source en KM3

```
1 package Test {
2   class A {
3     reference bs[*] container : B;
4   }
5
6   class B {}
7 }
```

Listing 7.6 – Exemple simplifié de règle sans motif cible

```
1 rule AAndBs {
2   from
3     a : Test!A,
4     b : Test!B (
5       a.bs->includes(b)
6     )
7   -- [...]
8 }
```

Le pseudo code correspondant à une implémentation naïve du calcul des tuples reconnus par la règle du listing 7.6 est donné par le listing 7.7. Il y a une boucle pour chaque élément source et ces boucles sont imbriquées. La garde est testée sur le produit cartésien des ensembles d'éléments de type A et de type B.

Listing 7.7 – Construction naïve des tuples reconnus

```
1 PourChaque a de type A {
2   PourChaque b de type B {
3     évaluer la garde
4     Si la garde est vraie Alors
5       créer les éléments cible et le lien de traçabilité
6     FinSi
7   }
8 }
```

Dans notre cas la garde teste si l'élément de type B est contenu dans l'élément de type A. En effectuant une analyse statique du métamodèle et de la transformation il est possible d'optimiser l'implémentation du calcul des tuples. Le listing 7.8 donne le pseudo code du résultat possible d'une telle optimisation. Il y a toujours un parcours de tous les éléments de type A, mais seuls les éléments de type B contenus dans chaque élément de type A sont effectivement parcourus. Les éléments de type B non contenus dans un élément de type A donné ne sont pas parcourus.

Listing 7.8 – Construction optimisée des tuples reconnus

```
1 PourChaque a de type A {
2   PourChaque b de a.bs {
3     créer les éléments cible et le lien de traçabilité
4   }
5 }
```

Une autre possibilité d'optimisation consiste en la mise en cache des valeurs retournées par les *helpers*. Ceux-ci retournent en effet toujours le même résultat pour un contexte et un ensemble d'arguments donnés puisqu'ils ne peuvent naviguer que les modèles source non modifiables. Nous avons implémenté, à titre expérimental, la mise en cache des résultats donnés par les *helpers* attribut dans notre machine virtuelle ATL. Nous avons ensuite mesuré le gain en nombre d'instructions élémentaires (*bytecodes* en anglais) sur la transformation présentée dans [41]. Sans cache, 40853 instructions sont exécutées contre 15543 avec cache. Ceci correspond à une division par 2,6 du nombre d'instructions. L'implémentation de cette optimisation est relativement simple et ne nécessite aucun changement dans la transformation elle-même : il suffit d'activer le cache au niveau de la machine virtuelle. Bien entendu, le gain réel dépend de la sémantique de la transformation et de la manière dont elle est implémentée. Il n'y a ainsi aucun gain si aucun *helper* n'est utilisé et si les expressions sont simplement copiées et collées à chaque endroit où elles sont utilisées.

7.3.5.3 Déterminisme et terminaison

Lorsque ni les règles paresseuses ni les règles impératives et blocs impératifs ni les *helpers* ne sont utilisés, l'algorithme d'exécution se termine et est déterministe. Bien que l'ordre d'exécution des règles soit non-déterministe, différents ordres d'exécution produisent le même résultat pour un modèle source donné pour les raisons suivantes :

- Puisque les modèles source ne sont pas modifiables, l'exécution d'une règle ne change pas le résultat du calcul des tuples reconnus par chaque règle.
- Puisque les modèles cible ne sont pas navigables, l'initialisation d'un élément cible n'a aucun effet sur l'initialisation des autres.

L'utilisation des règles paresseuses ne conduit pas à un non-déterminisme pour les même raisons. Une transformation utilisant des règles paresseuses peut, en revanche, ne pas se terminer. Il est en effet possible de définir des règles récursives ou mutuellement récursives comme dans le listing 7.9. Dans cet exemple, deux règles paresseuses R1 et R2 font référence l'une à l'autre (aux lignes 6 et 15). Il est

ainsi possible d'avoir une récursion infinie. Dans ce cas, la transformation peut ne pas se terminer si la condition d'arrêt n'est pas précisée ou bien est erronée.

Listing 7.9 – Exemple de règles paresseuses mutuellement récursives

```
1  lazy rule R1 {
2    from
3      s : Element
4    to
5      t : Element (
6        value <- [R2.t]s
7      )
8  }
9
10 lazy rule R2 {
11   from
12     s : Element
13   to
14     t : Element (
15       value <- [R1.t]s
16     )
17 }
```

L'utilisation des *helpers* ne conduit pas non plus à un non-déterminisme car il s'agit de navigation sur des modèles source non modifiables. Ils peuvent en revanche être définis récursivement et donc leur évaluation peut ne pas se terminer. La transformation ne se termine alors pas non plus. Enfin, lorsque des règles impératives ou des blocs impératifs sont utilisés, il n'est pas possible, dans le cas général, de décider du déterminisme ou de la terminaison des transformations.

7.4 Description du prototype

L'utilisation d'ATL requiert un moteur d'exécution. Un prototype architecturé autour d'une machine virtuelle de transformation a été développé au cours de nos travaux. De plus, afin de faciliter la mise au point des transformations, un environnement de développement intégré à la plateforme Eclipse a été créé. Ces deux éléments (moteur d'exécution et environnement de développement) sont développés en *open source* dans le projet Eclipse GMT [20].

La section 7.4.1 décrit le moteur d'exécution et la section 7.4.2 présente l'environnement de développement ATL.

7.4.1 Moteur d'exécution

Le moteur d'exécution ATL est responsable de l'exécution des transformations exprimées en ATL. Cette responsabilité est partagée entre un compilateur et une machine virtuelle. Les transformations sont donc d'abord compilées vers un code élémentaire de manipulation de modèles. Ce code est ensuite interprété par la machine virtuelle. L'architecture du moteur d'exécution ATL est présentée à la figure 7.2.

La machine virtuelle peut fonctionner sur différents systèmes de manipulation de modèles (appelés *model handlers* en anglais) grâce à sa couche d'abstraction (*model handler abstraction layer*). Les systèmes de manipulation de modèles sont des composants qui fournissent une interface de programmation pour la manipulation de modèles : lecture et écriture dans des fichiers, création et suppression d'éléments, modification des liens entre éléments, etc. La couche d'abstraction traduit les instructions de manipulation de modèles de la machine virtuelle en instructions spécifiques à chaque système. Deux exemples de tels systèmes sont représentés sur la figure 7.2 : EMF [14] (pour *Eclipse Modeling Framework*) et MDR

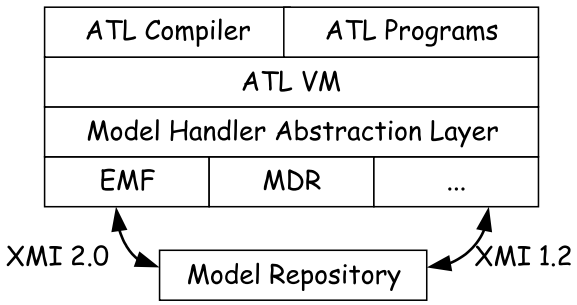

Figure 7.2 – Architecture du moteur d'exécution ATL

[51] (pour *MetaData Repository*). Ces systèmes nécessitent typiquement des systèmes de stockage des modèles (appelés *model repositories*).

Il y a plusieurs avantages à cette approche basée sur une machine virtuelle. Nous en citons trois :

- Étendre ATL ne requiert généralement que des changements dans le compilateur, et pas dans la machine virtuelle.
- Ajouter la possibilité d'utiliser de nouveaux systèmes de manipulation de modèles ne nécessite que des changements dans la machine virtuelle.
- Il est possible de compiler d'autres langages de manipulation de modèles vers la machine virtuelle ATL. Ces langages bénéficient alors de ses capacités quant au support des systèmes de manipulation de modèles et à certains outils de développement, tels que le débogueur présenté à la section 7.4.2.

La machine virtuelle ATL est spécifiée dans [23][1]. Nous en donnons ici un bref apperçu. Les valeurs gérées par la machine virtuelle et les opérations qui leurs sont applicables sont celles définies dans la recommandation OCL [57]. Il s'agit entre autres des types de données suivants : *Boolean* pour les booléens, *Integer* pour les entiers, *Real* pour les réels, *Collection* et ses sous-classes pour les collections, etc. Le langage de la machine virtuelle ATL est un petit jeu d'instructions impératif composé d'une vingtaine d'instructions réparties en quatre catégories :

- **Gestion de la pile.** La machine virtuelle ATL est une machine à pile. Son jeu d'instructions comporte donc un ensemble d'instructions permettant de placer des valeurs sur la pile (e.g. `pushi` pour les entiers) et de les manipuler (e.g. `dup` pour la duplication, `swap` pour l'inversion).
- **Gestion des variables.** Deux instructions : `store` et `load` permettent de transférer des valeurs entre la pile et les variables locales.
- **Contrôle de flux.** Le contrôle de flux est assuré par : une instruction de saut inconditionnel (`goto`), une instruction de saut conditionnel (`if`), une instruction d'appel d'opération (`call`) et des instructions d'itération sur les collections (`iterate` et `enditerate`).
- **Manipulation de modèles.** La machine virtuelle ATL possède des instructions pour référencer les classes des métamodèles source et cible (`findme`), pour créer des nouveaux éléments (`new`) ainsi que pour accéder à leurs propriétés (`get` et `set`). Dans la version actuelle, il n'y a pas d'instruction de suppression d'élément car ce n'est pas nécessaire pour ATL. Cependant, une telle instruction pourrait être ajoutée afin de supporter un langage en ayant besoin.

Le compilateur ATL est chargé de traduire les transformations en code exécutable par la machine

[1] Cette spécification a été rendue publique sous licence EPL (*Eclipse Public License*). Ceci signifie que d'autres mises en œuvre de la machine ATL pourraient être réalisées par des tierces parties. Il est par exemple envisageable de voir apparaître des versions industrielles plus optimisées que le prototype actuel.

virtuelle. Chaque transformation est injectée en un modèle à l'aide de TCS. Un traducteur génère ensuite le code utilisant les instructions présentées ci-dessus. La version actuelle de ce traducteur n'est pas écrite en ATL mais utilise un langage dédié à la génération de code pour la machine virtuelle ATL (ACG pour *ATL Code Generation*) qui représente chaque instruction à générer par un mot-clé. Ce traducteur est lui-même compilé pour s'exécuter sur la machine virtuelle ATL. L'utilisation de ce langage a permis la définition du compilateur ATL alors qu'aucun moteur n'existait encore.

Néanmoins, il serait possible d'implémenter la génération du code dans un traducteur écrit en ATL. Un tel traducteur aurait cependant l'inconvénient d'être moins concis que la solution actuelle. En effet, la génération de chaque instruction demanderait l'utilisation d'un élément cible de règle, ce qui est plus complexe qu'un simple mot-clé. Ceci est dû au fait que le langage ATL est indépendant du métamodèle cible alors que ce dernier est cablé dans le langage ACG.

7.4.2 Environnement de développement

L'environnement de développement ATL s'appelle ADT (pour *ATL Development Tools*), ce qui signifie outils de développement ATL. La figure 7.3 présente son architecture. ADT est composé de quatre briques principales : le moteur d'exécution (ou *engine*), l'éditeur (ou *editor*), le compilateur (ou *builder*) et le débogueur (ou *debug*). Chacun de ces éléments utilise des fonctionnalités offertes par Eclipse. De plus, l'éditeur, le compilateur de le débogueur utilisent le moteur d'exécution, qui fournit les primitives de gestion des transformations ATL. L'utilisation de ADT est décrite dans le manuel utilisateur d'ATL [24]. Nous présentons brièvement ici l'édition, la compilation, l'exécution et le débogage des transformations.

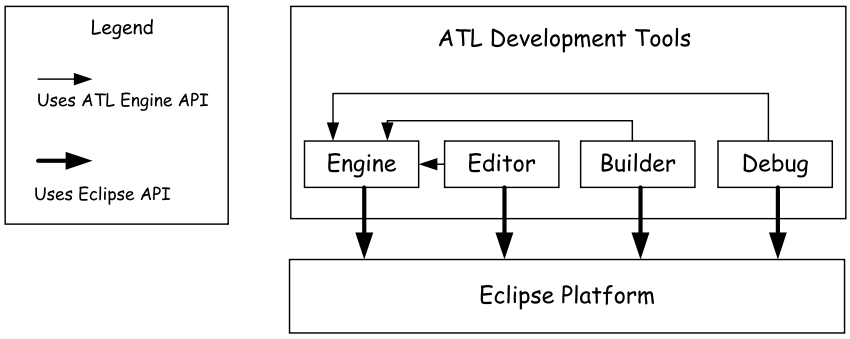

Figure 7.3 – Architecture des outils de développement ATL

7.4.2.1 Édition des transformations

L'éditeur ATL supporte la coloration syntaxique, le rapport d'erreurs et une vue arborescente de la transformation. La partie en bas à gauche de la figure 7.4 montre comment une transformation ATL est représentée dans l'éditeur. La partie en bas à droite de cette figure montre la vue arborescente correspondante. Cette vue est générée par injection du code ATL en un modèle à l'aide de TCS. Ce modèle est ensuite représenté sous la forme d'un arbre de composition.

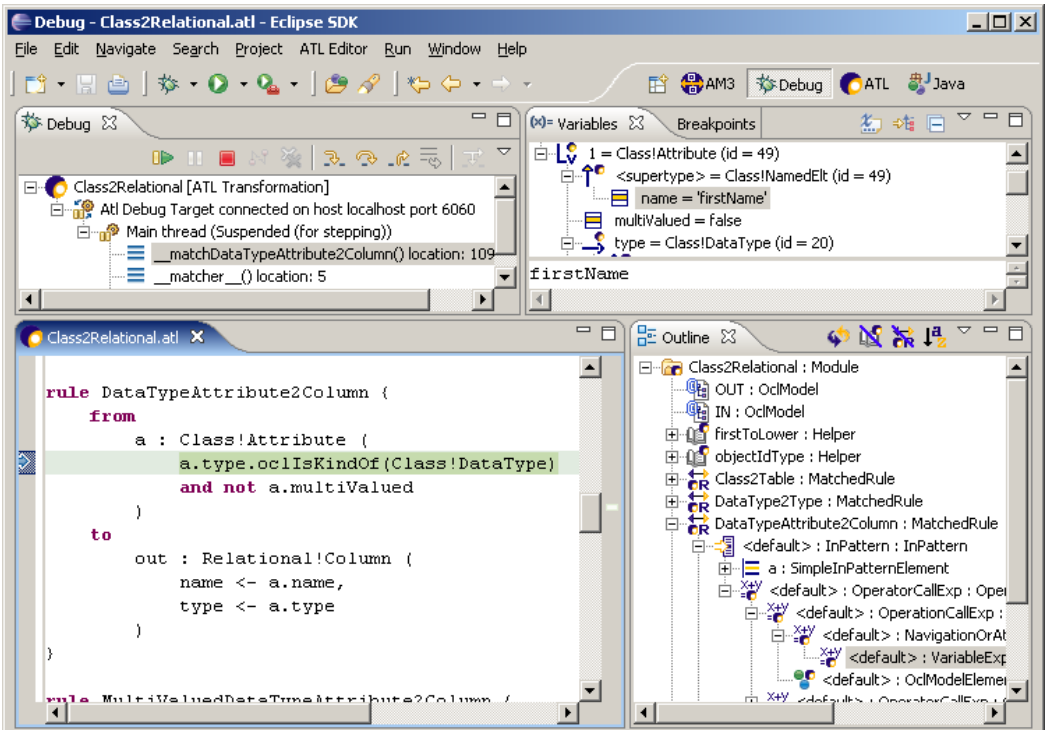

Figure 7.4 – Capture d'écran de l'éditeur et du débogueur ATL

7.4.2.2 Compilation et exécution des transformations

Le compilateur ATL est automatiquement invoqué sur chaque fichier ATL (portant l'extension `.atl`) dans chaque projet ATL au cours du processus de compilation Eclipse. Par défaut, ce processus est déclenché dès qu'un fichier est modifié. Un fichier portant l'extension `.asm` et contenant les instructions pour la machine virtuelle ATL est généré pour chaque fichier ATL.

L'exécution d'une transformation nécessite la mise en relation des modèles et métamodèles source et cible déclarés dans l'en-tête avec des modèles concrets. Il s'agit typiquement de fichier XMI [60] portant les extensions `.xmi` ou `.ecore`. La fenêtre de configuration de lancement permet d'effectuer cette mise en relation. La figure 7.5 présente une capture d'écran de cette fenêtre et montre la correspondance entre l'interface utilisateur et le contexte opérationnel d'ATL donné à la figure 7.1. Les quatre flèches du haut font correspondre les modèles et métamodèles à leurs déclarations alors que les quatre flèches du bas font correspondre ces déclarations à des fichiers XMI portant une extension `.ecore`.

Le moteur ATL délègue la lecture et l'écriture des modèles à l'un des systèmes de manipulation de modèles supportés. Ainsi, lorsque EMF est utilisé, les modèles et métamodèles doivent utiliser le format XMI 2.0. L'utilisation d'injecteurs et d'extracteurs peut être nécessaire pour gérer d'autres formats.

7.4.2.3 Débogage des transformations

Les transformations ATL peuvent être déboguées à l'aide de la même configuration de lancement utilisée pour l'exécution simple. Les transformations peuvent être exécutées pas à pas ou normalement. Dans ce cas, l'exécution s'interrompt lorsqu'une erreur apparaît ou lorsqu'un point d'arrêt est atteint. Le contexte courant (i.e. les valeurs des variables) peut être analysé à l'aide de la vue Eclipse représentant

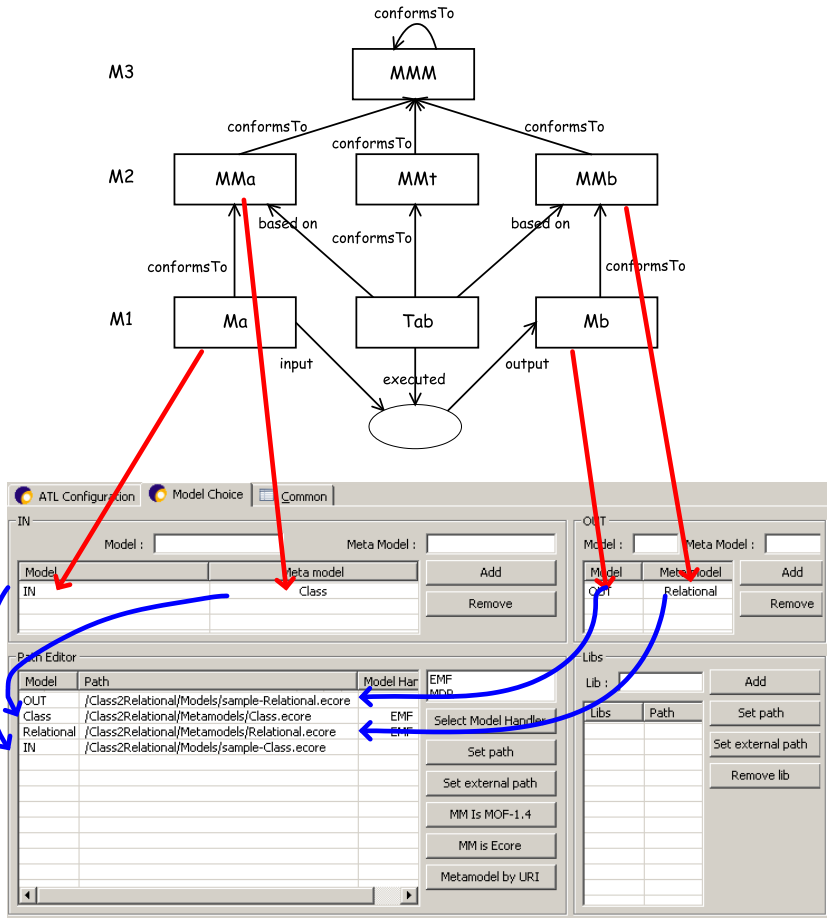

Figure 7.5 – Configuration de lancement d'une transformation

les variables. Cette vue figure en haut à droite de la figure 7.4. Ceci permet la navigation dans les modèles source et cible à partir du contexte courant.

7.5 Conclusion

Ce chapitre a présenté le langage de transformation de modèles ATL. Il s'agit d'un langage de transformation pour l'ingénierie des modèles basé sur un paradigme hybride : avec des constructions impératives et des constructions déclaratives. Il utilise le langage OCL pour la navigation dans les modèles source. Les modèles cible ne sont pas navigables.

Il existe deux types principaux de règles : les règles appelées, qui sont impératives, et les règles déclaratives (*matched rules*). L'exécution d'une règle appelée est déclenchée par son appel explicite depuis une autre règle. Ces règles impératives peuvent être considérées comme des procédures. Au contraire, l'exécution des règles déclaratives est contrôlée par le moteur de transformation. Celui-ci recherche dans les modèles source des ensembles d'éléments correspondant aux motifs source de ces règles. Pour chaque ensemble reconnu, la règle correspondante est appliquée.

L'exécution d'une règle impérative ou déclarative peut soit entraîner la création d'un ensemble d'éléments correspondant à son motif cible, soit entraîner l'exécution des instructions spécifiées dans son bloc impératif, soit les deux, en commençant par le motif cible. Ceci permet notamment de compléter l'initialisation des éléments cible à l'aide d'algorithmes implémentés impérativement. De plus, le fait de pouvoir définir un motif cible déclaratif dans une règle impérative ou bien un bloc impératif dans une règle déclarative permet de combiner très étroitement les aspects impératifs et déclaratifs.

Des exemples d'utilisation d'ATL sont donnés au chapitre 8 et en annexe B. L'état de l'art sur la transformation dans le contexte de l'ingénierie des modèles présenté au chapitre 3 propose une comparaison d'ATL et du langage de sa machine virtuelle avec sept autres langages dont les trois définis dans la recommandation QVT. Cette comparaison est donnée à la table 3.2.

Un prototype de moteur d'exécution pour le langage ATL a été développé au cours de nos travaux. L'une des décisions architecturales de ce travail a été de définir de façon précise et ouverte une machine virtuelle de transformation. Rétrospectivement, ce choix nous apparaît important car il permet d'offrir une forte adaptabilité à notre prototype. Non seulement des versions industrialisées, plus performantes, pourraient être produites, mais il est également possible d'envisager une extension du jeu d'instruction. La machine actuelle, ou une version étendue, pourrait devenir la cible d'autres langages dédiés de transformation.

CHAPITRE 8
Cas d'étude

8.1 Introduction

Ce chapitre présente trois cas d'étude de la plateforme AMMA et plus particulièrement de ses langages dédiés noyau : KM3, TCS et ATL. Ces trois cas d'étude ont été choisis parce qu'ils permettent d'illustrer quelques caractéristiques importantes de nos propositions. Ils ne sont cependant pas représentatifs de toutes les possibilités offertes par la plateforme AMMA. Une vision plus complète peut être obtenue à partir de la bibliothèque composée de plus d'une centaine de transformations disponibles dans le projet AM3 [26]. La communauté des utilisateurs d'ATL, forte actuellement de plus de cent cinquante personnes, illustre également la grande variété des champs d'application de la plateforme AMMA.

Le premier cas d'étude concerne la gestion de la traçabilité par les transformations ATL et est présenté à la section 8.2. La section 8.3 présente le deuxième, qui consiste en l'utilisation de la transformation de modèles pour la vérification de modèles. Le troisième cas d'étude est l'objet de la section 8.4 et correspond à l'implémentation de deux langages dédiés du domaine de la téléphonie par internet. Enfin, la section 8.5 présente les conclusions de ce chapitre.

8.2 Ajout de la traçabilité aux transformations ATL[1]

Dans le contexte de la transformation de modèles, les informations de traçabilité peuvent être utilisées dans des scénarios variés. Chacun de ces scénarios requiert potentiellement un format ou un niveau de complexité différent. Nous utilisons ici la définition suivante de la notion de traçabilité. Un langage, ou un moteur d'exécution, qui supporte la traçabilité doit pouvoir maintenir un ensemble de relations entre éléments de modèles source et cible correspondants. Des liens vers les éléments de transformation (par exemple des règles) responsables de l'existence de ces relations peuvent aussi être conservés.

Nous anticipons que les besoins en information de traçabilité vont varier selon au moins deux dimensions que nous définissons ainsi :

- **Portée.** Il n'est pas toujours nécessaire ou souhaitable de conserver tous les liens de traçabilité correspondant à l'exécution d'une transformation. La portée de la traçabilité correspond à la proportion des liens qui sont conservés.
- **Format.** La manière dont les liens de traçabilité sont codés est appelée format de la traçabilité. Certaines applications n'ont besoin que de liens simples entre éléments de modèles. Mais d'autres ont besoin de codages plus complexes tels que : des clés, des identificateurs, des expressions de chemins, etc.

De plus, une transformation donnée peut être exécutée dans différents contextes requérant des portées ou formats différents parfois inconnus au moment de l'écriture de la transformation.

Le cas d'étude présenté dans cette section montre comment la gestion de la traçabilité peut être ajoutée aux transformations ATL tout en limitant les dépendances avec la logique de transformation. Cette

[1] Ce cas d'étude a été présenté dans [37].

approche est elle-même mise en œuvre à l'aide de transformations de modèles grâce aux considérations suivantes :

- Il est possible de représenter les informations de traçabilité sous la forme de modèles.
- Une transformation ATL est un modèle conforme au métamodèle ATL.

Le code de génération des informations de traçabilité est clairement séparé de la logique de transformation. En conséquence, il est possible d'ajouter le support de nouvelles portées ou de nouveaux formats.

La section 8.2.1 présente l'approche qui est discutée à la section 8.2.2. La section 8.2.3 conclut ce cas d'étude.

8.2.1 Présentation de l'approche

La section 8.2.1.1 présente un exemple très simple de génération d'informations de traçabilité. La section 8.2.1.2 montre comment le fait de considérer les informations de traçabilité comme un modèle permet un codage simple de la génération de ces informations. Enfin, la section 8.2.1.3 présente l'intérêt de considérer les transformations comme des modèles dans ce contexte.

8.2.1.1 Exemple de traçabilité

Considérons les métamodèles présentés à la figure 8.1 : *Src* (pour source) et *Dst* (pour destination). *Src* contient une unique classe appelée *A* possédant un unique attribut *name* (pour nom). *Dst* est identique à *Src* à l'exception du nom de la classe qui est *B*.

1.a. *Src* 1.b. *Dst*

Figure 8.1 – Les métamodèles *Src* et *Dst*

Considérons à présent la transformation de *Src* vers *Dst* qui traduit chaque *A* en un *B* ayant le même nom. Le code ATL implémentant cette transformation est donné au listing 8.1. Un modèle de transformation, appelé *Src2Dst* (ligne 1), est déclaré comme transformant les modèles Src en modèles Dst (ligne 2). Une unique règle, appelée *A2B* (lignes 4 à 11), implémente la traduction des éléments *A* (ligne 6) en éléments *B* (lignes 8 à 10).

Listing 8.1 – Transformation Src vers Dst, écrite en ATL

```
1  module Src2Dst;
2  create OUT : Dst from IN : Src;
3
4  rule A2B {
5    from
6      s : Src!A
7    to
8      t : Dst!B (
9        name <- s.name
10     )
11 }
```

8.2.1.2 Modèle de traçabilité

Le principe "tout est modèle" (voir section 2.2) est applicable aux informations de traçabilité. Il est en effet possible de considérer ces informations comme un modèle cible supplémentaire d'une transformation donnée. Ceci signifie que les éléments de traçabilité peuvent être créés de la même manière que les autres éléments cible le sont. Il n'est pas nécessaire de disposer de constructions additionnelles dans le langage ou même de modifier le moteur d'exécution, à la condition, toutefois, qu'il soit possible d'utiliser plusieurs modèles cible. Or, c'est le cas avec ATL. De plus, une fois un modèle de traçabilité obtenu, il est possible d'appliquer toute opération de manipulation de modèle telle que la transformation.

Puisque les informations de traçabilité sont un modèle, elles ont besoin d'un métamodèle. La figure 8.2 présente le métamodèle *Trace* qui peut jouer ce rôle. Il contient une classe *TraceLink* (pour lien de traçabilité) possédant un attribut *ruleName* (pour nom de règle) qui sert à stocker le nom de la règle de transformation. *TraceLink* référence deux ensembles d'éléments de modèles : *sourceElements* pour les éléments source et *targetElements* pour les élément cible. Ces deux références ont pour type *AnyModelElement*. Cette classe représente n'importe quel élément de n'importe quel modèle. Il ne s'agit donc pas d'une simple classe. Elle est nécessaire car les liens de traçabilité doivent pouvoir relier des éléments de modèles conformes à n'importe quel métamodèle (par exemple *Src* et *Dst*. Il existe plusieurs implémentations concrètes de ce mécanisme. Nous en citons deux ici :

- **EMF/EObject.** En utilisant EMF [14], la classe *AnyModelElement* peut être remplacée par la classe *EObject* faisant partie du métamétamodèle Ecore.
- **AMW.** En utilisant la brique AMW (voir chapitre 4) de AMMA, le métamodèle de traçabilité peut être défini comme un métamodèle de tissage. Le problème du référencement des éléments définis dans d'autres modèles est alors sous la responsabilité de AMW.

Dans la suite de la présentation de ce cas d'étude, nous considérons qu'un de ces mécanismes ou qu'un mécanisme équivalent est utilisé. Nous n'utilisons donc pas les références *sourceElements* et *targetElements* différemment des autres références.

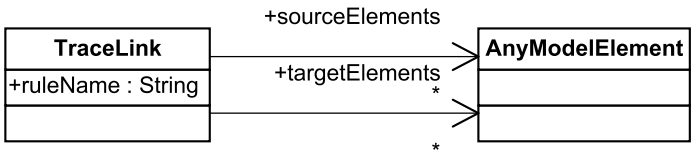

Figure 8.2 – Un métamodèle simple de traçabilité : *Trace*

Bien que nous pensions que toute forme de traçabilité peut en réalité être implémentée à l'aide de l'approche présentée (voir discussion à la section 8.2.2), nous ne présentons qu'un scénario simple. Chaque lien *TraceLink* porte le nom de la règle dont l'application a entraîné la création et référence les éléments source et cible correspondants.

Afin d'intégrer la gestion de la traçabilité dans la transformation *Src2Dst* présentée au listing 8.1, nous ajoutons un élément cible générant un élément *TraceLink*. Le résultat est la transformation *Src2DstPlusTrace* dont le code est donné au listing 8.2.

Listing 8.2 – Transformation Src vers Dst avec traçabilité

```
1 module Src2DstPlusTrace;
2 create OUT : Dst, trace : Trace from IN : Src;
3
4 rule A2BPlusTrace {
5   from
```

```
6     s : Src!A
7   to
8     t : Dst!B (
9       name <- s.name
10    ),
11    traceLink : Trace!TraceLink (
12      ruleName <- 'A2BPlusTrace',
13      targetElements <- Sequence {t}
14    )
15  do {
16    traceLink.refSetValue('sourceElements', Sequence {s});
17  }
18 }
```

Un modèle cible supplémentaire est spécifié dans l'en-tête de transformation (ligne 2) pour stocker les liens de traçabilité. Un élément de motif cible (lignes 11 à 14) est aussi ajouté à la règle *A2B* (renommée en *A2BPlusTrace*, ligne 4) pour créer un *TraceLink*. Un bloc impératif contenant une unique instruction est aussi ajouté afin d'initialiser la propriété *sourceElements* du *TraceLink* (lignes 15 à 18).

Cette instruction impérative est nécessaire pour contourner l'algorithme de résolution ATL (voir chapitre 7). En effet, si *sourceElements* était initialisée à l'aide de l'opérateur <−, les éléments source seraient résolus en éléments cible, ce qui empêcherait de les référencer.

La transformation *Src2DstPlusTrace* peut générer les informations de traçabilité correspondant à son exécution sans nécessité de support spécifique au niveau langage ou moteur d'exécution. Il a cependant été nécessaire qu'un développeur ajoute les fragments de code décrits ci-dessus.

8.2.1.3 Modèle de transformation

Nous venons de voir que les transformations peuvent être relativement facilement étendues pour supporter la traçabilité si nous considérons les informations de traçabilité comme un modèle. Nous allons maintenant analyser ce que le fait que les transformations soient des modèles apporte dans ce contexte.

Puisque les transformations sont des modèles, nous pouvons transformer un programme ATL en un autre programme ATL par transformation de modèle. Il s'agit alors d'une transformation d'ordre supérieur. De cette manière, une transformation ATL peut être écrite afin d'insérer automatiquement le code de génération des informations de traçabilité décrit à la section précédente. Nous avons développé deux versions de cette transformation : *ATL2Tracer* utilisant le mécanisme EMF/EObject pour référencer les éléments source et cible et *ATL2WTracer* utilisant AMW dans ce but. Le code complet de ces deux transformations est disponible dans le projet AM3 [26].

Ces transformations opèrent dans le mode de raffinage car elles ne font qu'ajouter quelques éléments à leur source sans changer l'existant. Une première règle est chargée d'insérer le modèle cible additionnel pour les informations de traçabilité. Une deuxième règle ajoute les éléments de motifs cible nécessaires pour créer un lien de traçabilité par règle. Tous les autres éléments sont copiés automatiquement.

Ces transformations peuvent être insérées dans la chaîne de compilation ATL, après l'analyse syntaxique et avant la génération du code pour la machine virtuelle. Elles peuvent donc être considérées comme des précompilateurs. Cependant, elles opèrent sur la syntaxe abstraite (i.e. les modèles) et non pas sur la syntaxe concrète comme le préprocesseur C ou encore les outils d'instrumentation du code binaire Java.

8.2.2 Discussion

Nous venons de présenter une technique pour étendre automatiquement les transformations ATL de telle sorte qu'elles génèrent des informations de traçabilité. Nous allons maintenant discuter cette

solution en commençant, à la section 8.2.2.1, par expliquer en quoi le système de génération de traçabilité obtenu est faiblement couplé à la logique de transformation. Puis, nous citerons, à la section 8.2.2.2, quelques exemples d'extension du scénario simple présenté ci-dessus. Enfin, nous présenterons quelques autres options d'implémentation de la traçabilité à la section 8.2.2.3.

8.2.2.1 Traçabilité faiblement couplée

Un des avantages principaux de la solution présentée ici est que le Code de Génération des Informations de Traçabilité (ou CGIT) n'est pas fortement couplé à la logique de transformation. En effet, la logique de transformation ne dépend pas du CGIT qui peut même être ajouté à une transformation non conçue pour l'utiliser. Ceci a été expérimenté sur un scénario de transformation écrit avant la conception de cette approche. Le CGIT ne dépend pas fortement de la logique de transformation non plus.

De plus, différents CGIT peuvent être attachés à la même transformation en fonction des besoins spécifiques d'applications données. Ceci signifie qu'une transformation peut être automatiquement adaptée à une portée ou à un format de traçabilité appropriés.

La solution actuelle inclut le CGIT dans le code de la transformation d'ordre supérieur (e.g. *ATL2Tracer* ou *ATL2WTracer*). Des versions plus complexes de ces transformations pourraient prendre un second modèle en entrée. Ce modèle supplémentaire définirait le CGIT à injecter.

Des bibliothèques de transformations d'ajout de CGIT telles que *ATL2Tracer* et *ATL2WTracer* pourraient être développées.

8.2.2.2 Exemples d'extensions

L'exemple utilisé ici est très simple. Cependant, les transformations *ATL2Tracer* et *ATL2WTracer* ont aussi été testées sur des exemples plus complexes avec succès. Il n'y a pas de raison particulière pour laquelle cette approche ne passerait pas à l'échelle sur des scénarios plus complexes, à condition qu'ils soient purement déclaratifs. En effet, bien que le code impératif puisse aussi être instrumenté, il est plus difficile de reconnaître les relations source-cible (i.e. ce que nous voulons représenter par des liens de traçabilité) dans ce type de code, où elles sont implicites.

Le métamodèle de traçabilité n'est pas fixé et les modèles de traçabilité peuvent toujours être transformés vers n'importe quel format après leur création. En ce qui concerne la portée de la traçabilité, des transformations telles que *ATL2Tracer* et *ATL2WTracer* pourraient relativement facilement être modifiées pour n'ajouter le CGIT qu'à certaines règles. Ceci aurait pour résultat de réduire la taille des modèles de trace.

À la section 8.2.1.3 nous avons écrit que notre approche opère sur la syntaxe abstraite. Elle peut naturellement aussi fonctionner sur la syntaxe concrète en utilisant les projecteurs adéquats.

8.2.2.3 Autres options d'implémentation

Comme beaucoup de langages déclaratifs, ATL a un support intégré de la traçabilité. Ceci est nécessaire pour pouvoir relier les éléments générés par différentes règles. Un élément cible peut ainsi être référencé en utilisant son élément source correspondant comme clé. C'est le principe de l'algorithme de résolution ATL (voir chapitre 7). Cependant, une telle forme de traçabilité ne persiste pas nécessairement après l'exécution d'une transformation. Pour cette raison, nous l'appelons traçabilité interne (i.e. interne au moteur d'exécution).

Au contraire, la traçabilité externe correspond à la capacité pour un langage ou moteur d'exécution de supporter la persistence des relations de traçabilité au-delà de l'exécution d'une transformation. Quand

la traçabilité interne est supportée par un outil, une forme simple de traçabilité externe peut souvent être implémentée à bas coût. Il suffit en effet de sérialiser les liens utilisés en interne. Dans le cas d'ATL, ceci permettrait d'obtenir l'ensemble de tous les liens entre éléments source et cible. Cependant, cette solution ne présente pas autant d'avantages que l'approche présentée ici, notamment en ce qui concerne l'adaptabilité à différentes portées et différents formats. De plus, cette solution nécessite la modification du moteur d'exécution et est limitée au moteur modifié.

Mémoriser les événements survenant aux modèles cible pendant l'exécution de la transformation est une autre solution opérant au niveau du moteur d'exécution. Cependant, elle souffre des mêmes inconvénients que la sérialisation de la traçabilité interne.

8.2.3 Conclusion

Nous avons montré dans ce cas d'étude comment du code de génération des informations de traçabilité peut être facilement ajouté aux transformations ATL. Nous avons de plus montré que ce processus peut être entièrement automatisé. Nous avons aussi discuté en quoi cette méthode permet une traçabilité faiblement couplée à la logique de transformation et en quoi elle peut être adaptée à n'importe quelle portée ou à n'importe quel format.

Cependant, l'approche présentée ici est limitée aux transformations déclaratives. De plus, elle ne considère qu'une transformation à la fois. Or, plusieurs étapes sont souvent nécessaires, ainsi que le cas d'étude sur les langages dédiés de téléphonie par internet présenté à la section 8.4 le montre.

Nous nous sommes limités ici à la gestion de la traçabilité. Cependant, nous pensons que l'approche utilisée peut être adaptée à d'autres formes d'instrumentation de transformations écrites en ATL ou dans d'autres langages. Il s'agit en effet essentiellement d'utiliser la transformation de modèle pour réaliser des transformations sur la syntaxe abstraite des programmes, ce qui peut s'appliquer à d'autres domaines.

8.3 Vérification de modèles[2]

Dans ce cas d'étude, nous utilisons la transformation pour la vérification des modèles. Ceci est rendu possible en appliquant ici aussi le principe "tout est modèle" (voir section 2.2) et en considérant le résultat de cette opération comme un modèle.

Traditionnellement, une vérification est une fonction retournant une valeur booléenne (notée $Boolean$) à partir d'un modèle (noté $Model$). Si nous appelons f la fonction de vérification, nous pouvons alors écrire $f : Model \rightarrow Boolean$. Les contraintes à vérifier peuvent être extraites de f sous la forme d'un type spécifique de modèle de contraintes (noté $Constraints$). Nous obtenons alors $f' : Model \times Constraints \rightarrow Boolean$. Pour indiquer un niveau de criticité, le résultat d'une vérification peut être représenté par un entier (noté $Integer$), ce qui donne $f'' : Model \times Constraints \rightarrow Integer$. Nous proposons une autre extension qui consiste à considérer ce résultat comme un type spécifique de modèle : un diagnostic (noté $Diagnostic$). Nous obtenons finalement $f''' : Model \times Constraints \rightarrow Diagnostic$. Nous allons illustrer ici l'application de ces principes à un type particulier de modèles : les métamodèles KM3.

La section 8.3.1 présente des extensions de OCL dont le but est de pouvoir vérifier des modèles et de présenter le résultat de cette vérification à l'utilisateur. Une discussion sur différentes représentations d'un diagnostic ainsi qu'une proposition de métamodèle sont données à la section 8.3.2. La section 8.3.3

[2] Ce cas d'étude a été présenté dans [11].

montre comment le moteur ATL peut être utilisé pour vérifier des modèles. Enfin, la section 8.3.4 donne les conclusions.

8.3.1 Contraintes sur les modèles

Afin d'être automatiquement vérifiées, les contraintes doivent être écrites dans un langage exécutable. Le langage OCL [57] est celui que nous utilisons ici. Dans cette section, nous montrons comment il est possible d'étendre la notion de contrainte OCL en y ajoutant des sévérités et des descriptions. Le résultat peut être utilisé pour définir des contraintes qui peuvent être vérifiées automatiquement sur des modèles. De plus, le résultat de cette vérification peut être présenté à l'utilisateur.

8.3.1.1 Utilisation de OCL pour exprimer des contraintes

OCL définit trois types de contraintes. Les invariants doivent être satisfaits à tout moment auquel un modèle est supposé dans un état valide. Ceci exclut d'éventuels états intermédiaires, par exemple au cours de transformations. Les pré- et post-conditions sont spécifiées sur des opérations et doivent respectivement être satisfaites avant et après l'exécution de leurs corps. Or, OCL a été conçu pour UML [56], mais il n'y a pas d'opération en KM3. Nous ne considérons donc ici que les invariants, qui peuvent s'appliquer aux modèles conformes à des métamodèles KM3.

Un invariant est défini dans le contexte d'une classe d'un métamodèle. Il est composé d'une expression booléenne qui doit s'évaluer en `vrai` pour chaque élément de modèle de ce type. Considérons les deux contraintes suivantes : le nom d'un classifieur (*Classifier*) doit être unique dans le paquetage (*Package*) le contenant (C1) et le nom d'une propriété (*StructuralFeature*) doit être unique dans la classe (*Class*) la contenant et dans ses classes parentes (C2). La figure 8.3 donne la spécification de (C1) et (C2) sous la forme d'invariants OCL. L'opération `allStructuralFeatures()` retourne pour chaque classe la collection de toutes ses *StructuralFeatures* (i.e. attributs et références), y compris celles héritées des classes parentes. Cette opération peut aussi s'exprimer en OCL.

```
-- (C1) Error: the name of a Classifier must
-- be unique within its package.
context Classifier inv:
  not self.package.contents->exists(e |
    (e <> self) and (e.name = self.name))

-- (C2) Error: the name of a StructuralFeature must
-- be unique within its Class and its supertypes.
context StructuralFeature inv:
  not self.owner.allStructuralFeatures()->exists(e |
    (e <> self) and (e.name = self.name))
```

Figure 8.3 – Expression de (C1) et (C2) par des invariants OCL

8.3.1.2 Ajout d'une sévérité aux contraintes

Les deux invariants (C1) et (C2) *doivent* être satisfaits pour tout métamodèle KM3. Il est cependant parfois utile de spécifier des contraintes qui *devraient* être satisfaites. Quand de telles contraintes

ne le sont pas, la validité du modèle n'est pas remise en question. Ceci est très similaire à la distinction classique entre *erreur* (ou *error* en anglais) et *avertissement* (ou *warning* en anglais) généralement faite par les compilateurs. Une erreur doit être corrigée alors qu'un avertissement n'indique qu'un problème potentiel. Des degrés additionnels peuvent être définis dans cette échelle ainsi que nous le verrons au paragraphe suivant. Nous ferons désormais référence à cette notion comme étant la *sévérité* d'une contrainte. D'après cette définition, (C1) et (C2) sont des erreurs, car un métamodèle qui ne les satisfait pas n'est pas un métamodèle KM3 valide.

Considérons à présent les deux contraintes suivantes : une classe abstraite *devrait* avoir des enfants (C3) et le nom d'un classifieur (classe ou type de données) *devrait* commencer par une majuscule (C4). Un métamodèle ne satisfaisant pas ces contraintes reste cependant valide. Il ne s'agit donc pas d'erreurs. La sévérité précise associée à une contrainte qui n'est pas une erreur peut varier. Dans notre exemple, nous considérons que (C3) est un avertissement au même titre que le code non atteignable pour certains langages de programmation. La contrainte (C4) est en revanche appelée critique (ou *critic* en anglais) car elle n'impacte pas du tout la structure du métamodèle. Il s'agit simplemement d'un problème de style. La figure 8.4 donne les expressions OCL correspondant à ces deux contraintes.

```
-- (C3) Warning: an abstract class should have children.
context Class inv:
  not (self.isAbstract and
       (Class.allInstances()->select(e |
         e.supertypes->includes(self)
       )->size() = 0))

-- (C4) Critic: the name of a Classifier should
-- begin with an upper case letter.
context Classifier inv:
  not (let firstChar : String =
       self.name.substring(1, 1) in
  firstChar <> firstChar.toUpper())
```

Figure 8.4 – Expression de (C3) et (C4) par des invariants OCL

8.3.1.3 Ajout d'une description aux contraintes

Jusqu'ici, le fait qu'une contrainte ne soit pas satisfaite ne peut être associé qu'à cette contrainte, sa sévérité et l'élément de modèle incriminé. Il n'est cependant pas toujours facile de comprendre le problème en lisant l'expression booléenne associée à la contrainte. Une description devrait donc être associée à chaque contrainte. C'est ce que nous avons fait pour (C1) et (C2) à la figure 8.3 et pour (C3) et (C4) à la figure 8.4 sous la forme de commentaires. Mais ce n'est pas suffisant car un outil peut difficilement utiliser une telle définition.

En UML, les contraintes ont un nom qui peut être utilisé pour indiquer sommairement leur sémantique sous la forme d'une chaîne de caractères. C'est un premier pas. Mais dans certains cas, le texte du message doit être adapté au contexte. Par exemple, il pourrait contenir des références aux valeurs de certaines propriétés des éléments (e.g. leurs noms). La solution que nous préconisons ici est de spécifier la description sous la forme d'une expression OCL s'évaluant en une chaîne de caractères. Un exemple

est donné à la figure 8.5 pour (C2). Cependant, puisqu'un diagnostic est un modèle, il serait possible de générer un représentation structurée éventuellement plus précise du problème si nécessaire.

```
'The Class ' + self.owner.name + ' contains several ' +
'properties having the same name: ' + self.name + '.'
```

Figure 8.5 – Une expression OCL s'évaluant en un message d'erreur adapté au contexte pour (C2)

8.3.1.4 Extension de OCL

Ainsi que nous l'avons précédemment observé, les outils de vérification de contraintes doivent avoir accès à leurs sévérités et à leurs descriptions. Autrement, des définitions telles que celles de la figure 8.4 ne peuvent pas être distinguées de celles de la figure 8.3. Or la non-satisfaction des premières ne remet pas en cause la validité du métamodèle. OCL doit donc être étendu pour supporter ces éléments supplémentaires.

La figure 8.6 donne une version étendue de (C1) définissant, en plus de son contexte et de son expression booléenne :

- **Une sévérité** indiquant la criticité du problème. Il s'agit ici d'une erreur, mais pourrait aussi être un avertissement ou une critique, par exemple.
- **Une description** permettant à un opérateur humain de comprendre le problème. Cette chaîne de caractères utilise ici le nom de l'élément ne satisfaisant pas la contrainte.
- **La localisation** du problème sous la forme d'une chaîne de caractères interprétable par ordinateur. Le choix du format à utiliser est dépendant de l'outil et peut être implémenté par une opération `getLocation()`.

```
context KM3!Classifier report error
    'a Classifier of the same name already ' +
    'exists in the same package: ' + self.name
  at self.getLocation()
  if self.package.contents->exists(e |
     (e <> self) and (e.name = self.name));
```

Figure 8.6 – Specification de (C1) avec les extensions faites à OCL

8.3.2 Représentation du résultat d'une vérification par un modèle

Nous avons vu que le résultat de l'évaluation d'un ensemble de contraintes sur un modèle résulte en un diagnostic. Dans cette section, nous considérons plusieurs représentations des diagnostics et nous donnons un exemple de métamodèle.

8.3.2.1 Différentes représentations des diagnostics

La forme la plus simple d'un diagnostic est une valeur booléenne. La valeur `vrai` signifie que le modèle vérifié satisfait toutes les contraintes alors que la valeur `faux` signifie qu'il ne les satisfait pas toutes. Si le diagnostic est représenté par un entier, il est par exemple possible de représenter la criticité

du problème de manière globale. Sa valeur pourrait, par exemple, être le nombre de contraintes non satisfaites optionnellement pondéré par la sévérité de chacune d'entre elles. Ceci est déjà plus intéressant qu'une simple valeur booléenne mais ne permet pas de représenter la nature et la localisation de chaque problème.

Des représentations plus complexes sont souhaitables. Certains outils présentent les diagnostics à l'utilisateur sous la forme de texte (e.g. les compilateurs), intégré dans une interface utilisateur, etc. Nous proposons de représenter les diagnostics par des modèles. La structure des diagnostics est donc explicitement spécifiée par un métamodèle.

Puisqu'un diagnostic est un modèle, des transformations peuvent lui être appliquées. Il peut ainsi s'agir d'une extraction vers une représentation textuelle ou XML. Par exemple, en écrivant chaque problème sur une ligne avec sa localisation, puis sa sévérité et enfin sa description, nous obtenons une notation similaire à celle utilisée par de nombreux compilateurs. Dans un environnement de développement intégré, le modèle de diagnostic peut être traduit en une représentation native (e.g. des *IMarkers* sous Eclipse). Les problèmes sont alors directement représentés à leur localisation propre dans l'éditeur et dans la vue *Problems* ainsi qu'illustré par la figure 8.7.

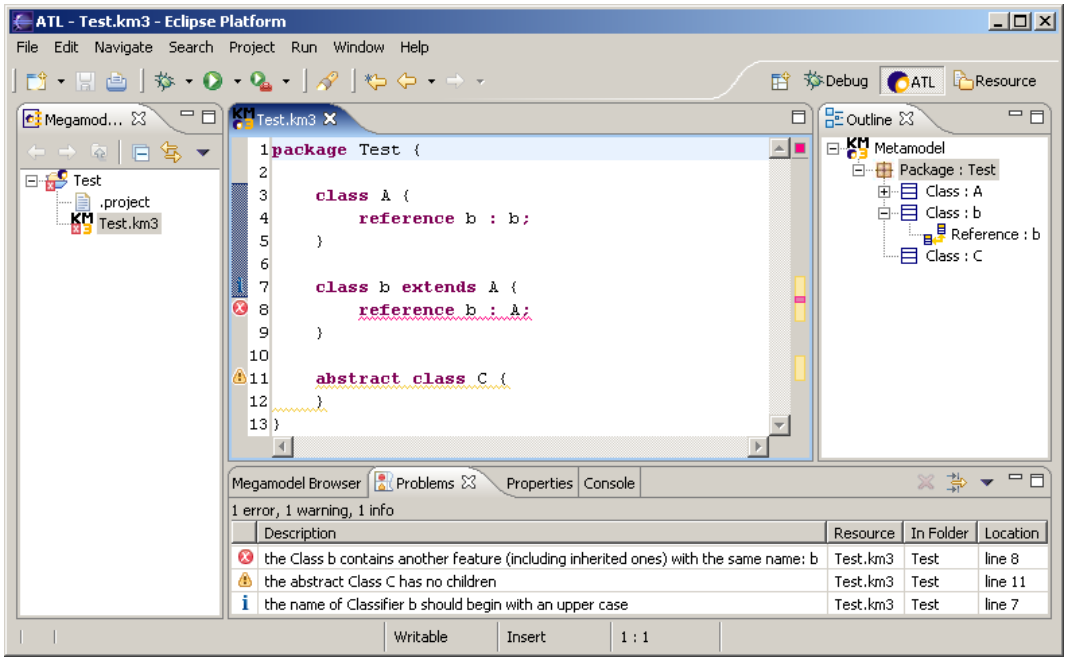

Figure 8.7 – Une capture d'écran de l'implémentation sous Eclipse

8.3.2.2 Un métamodèle de diagnostic

Un élément donné peut satisfaire ou non une contrainte donnée. Le premier cas est considéré comme normal et n'a pas besoin d'être mémorisé. Dans le second cas, un problème a été identifié. Le métamodèle de la figure 8.8 définit la classe *Problem* pour représenter un tel problème.

Un modèle satisfait un ensemble de contraintes lorsque leur évaluation produit un modèle *Problem* vide. Chaque élément *Problem* porte la sévérité (`severity`), typée par une énumération (`Severity`), associée à la contrainte. Il contient aussi une description (`description`) et une localisation (`location`) du problème, toutes deux représentées par des chaînes de caractères. Le nombre de niveaux de

sévérité et la représentation des localisation et description de problème peuvent être adaptés à des besoins spécifiques.

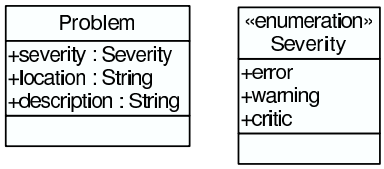

Figure 8.8 – Un métamodèle de diagnostic : *Problem*

8.3.3 Utiliser le moteur ATL pour vérifier des contraintes

Le moteur d'exécution des transformations ATL peut être utilisé pour vérifier des contraintes sur des modèles. Nous commençons par présenter comment ceci peut être mis en œuvre, puis nous analysons la solution proposée.

8.3.3.1 Représentation des contraintes en ATL

Afin de vérifier des contraintes avec ATL, il est nécessaire de les représenter sous la forme d'une transformation de modèles. L'algorithme qui permet de générer une transformation de vérification à partir d'un ensemble de contraintes est le suivant. Pour chaque contrainte, il faut créer une règle de transformation telle que :

- Le type du motif source de la règle soit le contexte de la contrainte.
- La garde de la règle corresponde à la négation de l'expression booléenne associée à la contrainte. En effet, nous voulons reconnaître les éléments posant problème et ignorer les autres. Nous avons exprimé (C1) et (C2) à la figure 8.3 et (C3) et (C4) à la figure 8.4 sous la forme de négations (i.e. de la forme `not <expr>`). Dans ce cas, la double négation pourra disparaître.
- Le motif cible de la règle spécifie un seul type : *Problem*, qui est créé pour chaque élément ne satisfaisant pas la contrainte.
- L'élément cible sera initialisé à l'aide de trois *bindings* : un pour la sévérité, un pour la description et un pour la localisation du problème.

Cet algorithme peut être appliqué par un développeur pour traduire des contraintes en code ATL ou encore automatiquement à partir d'une notation telle que celle présentée à la figure 8.6.

Les contraintes (C1), (C2), (C4) et (C4) ainsi que d'autres contraintes sont implémentées dans le scénario de transformation *KM32Problem* disponible dans le projet AM3 [26]. Le listing 8.3 donne le code ATL correspondant à (C1) à titre d'illustration de l'algorithme présenté ci-dessus.

Listing 8.3 – Transformation de vérification de (C1) sur les métamodèles KM3

```
1  -- ATL transformation that verifies (C1) on KM3 metamodels.
2  module KM32Problem;
3  create OUT : Problem from IN : KM3;
4
5  -- (C1) Error: the name of a Classifier must
6  -- be unique within its package.
7  rule ClassifierNameUniqueInPackage {
8    from
9      i : KM3!Classifier (
10       i.package.contents->exists(e |
11         (e <> i) and (e.name = i.name)
```

```
12        )
13      )
14    to
15      o : Problem!Problem (
16        location <- i.location,
17        severity <- #error,
18        description <- 'a Classifier of the same name ' +
19          'already exists in the same package: ' + i.name
20      )
21  }
```

L'environnement de développement de métamodèle KM3 utilise la version complète de cette transformation afin de vérifier les métamodèles KM3 injectés depuis la syntaxe textuelle avant de générer des métamodèles MOF 1.4 ou Ecore correspondants. Dans ce cas, la localisation d'un problème est composée d'un numéro de ligne et d'un numéro de colonne. La figure 8.7 montre comment un diagnostic est présenté à l'utilisateur.

8.3.3.2 Analyse de la solution proposée

La figure 8.9 présente une structure générique pour la vérification de contraintes. Un vérificateur (*Verifier*) vérifie un ensemble de contraintes (*Constraints*) sur un modèle (*Model*), ce qui résulte en un diagnostic (*Diagnostic*). Notons V l'opération implémentée par le vérificateur, M le modèle à vérifier, C l'ensemble de contraintes et D le diagnostic. Nous pouvons alors écrire : $D = V(M, C)$.

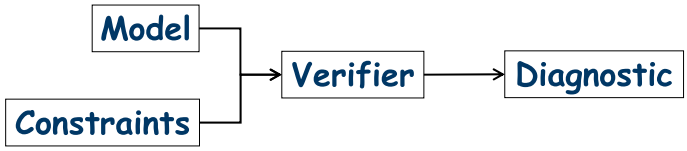

Figure 8.9 – Le vérificateur

Nous avons présenté au listing 8.3 une solution intégrant les contraintes et le vérificateur dans la même entité : une transformation ATL. Si nous notons T cette transformation, alors nous pouvons écrire : $D = T(M)$. En comparant les deux expressions de D, nous observons que T correspond à V et C. Il s'agit d'une évaluation partielle des contraintes, ce qui est similaire à la currification dans les langages fonctionnels. En effet, une version currifiée de V pourrait être notée $V_C(M)$ et on pourrait alors écrire : $D = V_C(M)$, ce qui correspond bien à la formule donnée ci-dessus avec la transformation T remplacée par V_C.

Un ensemble donné de contraintes C est souvent utilisé pour vérifier plusieurs modèles. Il est alors intéressant de ne pas avoir à traduire les contraintes à chaque fois. C'est ce que nous faisons en pratique lorsque nous utilisons le compilateur ATL pour compiler les contraintes.

8.3.4 Conclusion

Ce cas d'étude a montré comment la vérification de modèles peut être réalisée à l'aide d'ATL grâce aux trois considérations suivantes :

- Les contraintes, qui sont composées d'un contexte et d'une expression booléenne en OCL, sont associées à des informations supplémentaires. Il peut s'agir, par exemple, d'une sévérité, d'une description, d'une localisation, etc.

- Le diagnostic résultant de la vérification d'un ensemble de contraintes sur un modèle est considéré comme un modèle. Ce modèle peut ensuite être transformé vers une représentation spécifique : textuelle, visuelle, intégrée dans un environnement de développement tel que Eclipse, etc.
- Le langage ATL peut être utilisé pour implémenter des contraintes sur des modèles sous la forme de transformations ciblant un métamodèle de diagnostic. Le moteur d'exécution ATL est ensuite utilisé pour réaliser la vérification.

Cette approche peut être utilisée pour différents métamodèles et dans différents contextes. Nous avons montré qu'elle pouvait être appliquée à la vérification de métamodèles KM3. Une autre possibilité est de protéger l'exécution d'une transformation de modèles en définissant des pré-conditions. Les développeurs de transformations font en effet souvent des suppositions sur les modèles source qui ne sont pas intégrées dans leurs métamodèles. Une vérification de ces contraintes avant d'exécuter la transformation peut permettre de détecter des utilisations incorrectes de cette transformation[3].

Les mêmes principes peuvent être adaptés au calcul de métriques sur les modèles. Le métamodèle cible définit alors un système de représentation des résultats de mesure. Des expressions OCL sont utilisées pour implémenter le calcul des métriques. Un travail préliminaire dans ce domaine a été présenté dans [75]. Ce travail est actuellement en cours d'extension.

8.4 Langages de téléphonie SPL et CPL[4]

Nous présentons, dans cette section, une expérience consistant en l'implémentation de deux langages dédiés de téléphonie par internet. Le premier est SPL [15] (Session Processing Language), issu des travaux de l'équipe INRIA Phoenix, et le second CPL [48] (Call Processing Language). Les différents modèles (métamodèles, modèle TCS, transformations ATL) qui ont été construits pour cette application sont fournis en annexe B. Des extraits ou versions simplifiées de ces modèles sont donnés dans cette section afin que sa compréhension ne requiert pas la lecture des annexes. De plus, tous les éléments nécessaires à la reproduction de cette expérience, y compris un jeu de tests, sont disponibles dans le projet AM3 [26] dans la transformation appelée *CPL2SPL* (signifiant *CPL vers SPL*).

Le résultat de cette expérience fournit un exemple intéressant de construction de langages dédiés. Trois aspects de chaque langage dédié sont pris en compte : syntaxe abstraite, syntaxe concrète et sémantique dynamique. De plus, notre cas d'étude permet d'illustrer plusieurs approches : SPL a une syntaxe concrète textuelle alors que celle de CPL est basée sur XML. Enfin, les deux langages étant dans le même domaine, l'un peut être défini en termes de l'autre.

La figure 8.10 montre comment ces deux langages dédiés sont implémentés avec AMMA. Il s'agit d'une adaptation de la figure 4.4 présentée au chapitre 4. Chaque langage dédié est représenté par un ensemble de modèles. Ainsi, les métamodèles de définition des domaines (DDMMs pour *Domain Definition MetaModels*) des différents langages sont définis en KM3. Le langage des machines d'état abstraites (noté ASM) est représenté ici comme faisant partie de AMMA car il peut être utilisé pour définir les sémantiques dynamiques des langages dédiés ainsi que nous l'avons montré dans [66].

[3] Cette réflexion pourrait se généraliser à l'étude des possibilités de typage par les métamodèles.
[4] Ce cas d'étude a été présenté dans [39].

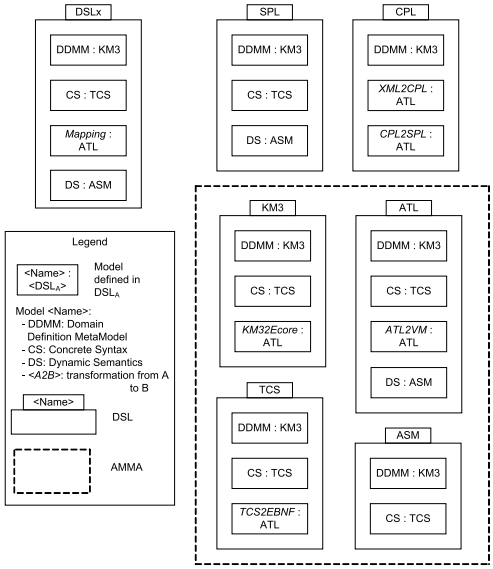

Figure 8.10 – Langages dédiés noyau de AMMA

8.4.1 Session Processing Language

8.4.1.1 Vue d'ensemble

Les programmes SPL sont utilisés pour contrôler des agents de téléphonie implémentant le protocole SIP [65] (Session Initiation Protocol). Ces agents peuvent, par exemple, être des clients utilisés par les utilisateurs pour passer des appels, ou encore des serveurs mandataires (en anglais *proxy*) relayant les communications. Les concepts SIP sont directement disponibles dans le langage. En conséquence, SPL permet l'écriture de programmes concis et simples pour exprimer les services de téléphonie. De plus, SPL peut garantir certaines propriétés qui ne pourraient pas être vérifiées avec un GPL. Les programmes SPL tournent sur un environnement d'exécution de la logique de service (ou Service Logic Execution Environment) pour SIP.

Le listing 8.4 (déjà donné dans le listing 6.1 du chapitre 6) donne un exemple simple de service SPL. Chaque appel entrant est redirigé vers l'adresse SIP `sip:phoenix@barbade.enseirb.fr`. L'adresse cible est déclarée à la ligne 3. Les lignes 6-8 correspondent à la définition de l'action à exécuter pour chaque appel entrant. L'instruction `return` à la ligne 7 fait suivre l'appel.

Listing 8.4 – Programme SPL simple : faire suivre un appel

```
1  service SimpleForward {
2    processing {
3      uri us = 'sip:phoenix@barbade.enseirb.fr';
4
5      registration {
6        response incoming INVITE() {
7          return forward us;
8        }
9      }
10   }
11 }
```

8.4.1.2 Syntaxe Abstraite

La syntaxe abstraite de SPL (ou DDMM) est définie en KM3. Ceci correspond à la boîte *DDMM : KM3* du langage SPL sur la figure 8.10. Le listing 8.5 donne un extrait du métamodèle SPL. Les lignes 1-5 définissent la classe *Service* qui a un nom et peut contenir des déclarations et des sessions. Une déclaration (*Declaration*) (lignes 7-9) a un nom et peut être une déclaration de variable (*VariableDeclaration*, lignes 11-14), qui a un type et une expression d'initialisation optionnelle (`initExp`, ligne 13). Une session (*Session*, ligne 16) peut être un enregistrement (*Registration*, lignes 18-20) contenant d'autres sessions ou une méthode (*Method*, lignes 22-27). Une méthode (*Method*) a un type de retour, une direction (voir l'énumération *Direction* aux lignes 35-39), un nom et des instructions. *Expression*, *TypeExpression* et *Statement* (lignes 29-33) sont des classes abstraites étendues pour définir la totalité du langage SPL qui n'est pas donnée ici. La version complète de ce métamodèle est donnée dans le listing B.1 de l'annexe B.

Listing 8.5 – Extrait du métamodèle SPL en KM3

```
1  class Service {
2    attribute name : String;
3    reference declarations[*] ordered container : Declaration;
4    reference sessions[*] ordered container : Session;
5  }
6
7  abstract class Declaration {
8    attribute name : String;
9  }
10
11 class VariableDeclaration extends Declaration {
12   reference type container : TypeExpression;
13   reference initExp[0-1] container : Expression;
14 }
15
16 abstract class Session {}
17
18 class Registration extends Session {
19   reference sessions[*] ordered container : Session;
20 }
21
22 class Method extends Session {
23   reference type container : TypeExpression;
24   attribute direction : Direction;
25   attribute name : String;
26   reference statements[1-*] ordered container : Statement;
27 }
28
29 abstract class Expression {}
30
31 abstract class TypeExpression {}
32
33 abstract class Statement {}
34
35 enumeration Direction {
36   literal inout;
37   literal in;
38   literal out;
39 }
```

8.4.1.3 Syntaxe Concrète

La syntaxe concrète de SPL a été implémentée en TCS à partir de sa définition en EBNF disponible sur le site [28]. Ceci correspond à la boîte *CS : TCS* du langage SPL sur la figure 8.10. TCS génère automatiquement une grammaire annotée à partir du métamodèle SPL en KM3 et du modèle TCS de SPL. Cette grammaire peut ensuite être utilisée pour injecter les programmes SPL en modèles conformes au métamodèle SPL. Les modèles SPL peuvent aussi être extraits vers des programmes en utilisant l'extracteur TCS.

La définition de la syntaxe concrète de SPL en TCS est utilisée comme exemple dans le chapitre 6 sur TCS. De plus, le modèle TCS complet est aussi fourni en annexe B dans le listing B.2. Nous ne décrivons donc ici que très brièvement la syntaxe utilisée dans le listing 8.4. Le service (classe *Service*) `SimpleForward` est défini aux lignes 1 à 11. La ligne 3 contient une déclaration de variable (classe *VariableDeclaration*) de type `uri` initialisée avec l'adresse cible de la redirection. Les lignes 5 à 10 correspondent à un élément *Registration* qui contient une unique méthode (classe *Method*, lignes 6-8). Cette méthode retourne une réponse (`response`) et est appelée pour chaque invitation entrante (mot-clé `incoming` correspondant au littéral `Direction::in`).

8.4.1.4 Sémantique Dynamique

Plusieurs définitions de la sémantique de SPL peuvent être données. Par exemple, les sémantiques statiques et dynamiques (à l'aide de règles de transition) de SPL sont spécifiées dans [28]. Nous avons aussi développé une spécification exécutable de la sémantique dynamique de SPL en utilisant le formalisme des machines d'état abstraites (ou ASM). Ceci correspond à la boîte *DS : ASM* du langage SPL sur la figure 8.10. Des parties de ce modèle ASM ont été automatiquement dérivées depuis le métamodèle SPL en KM3 : la définition du modèle de donnée de SPL. D'autres parties comme les environnements et les machines d'état ne font pas partie de ce métamodèle. Elles ont donc dû être traduites à partir de la spécification. Ce travail est présenté dans [66].

8.4.2 Call Processing Language

8.4.2.1 Vue d'Ensemble

CPL est un langage de script standard [48] pour le protocole SIP. Il offre un ensemble limité de constructions. CPL est supposé suffisamment simple pour ne pas présenter de problème de sécurité. Il doit en effet être possible d'exécuter des scripts CPL sur un serveur public sans nécessairement avoir confiance en leurs auteurs. Le listing 8.6 donne un exemple en CPL qui est équivalent à l'exemple en SPL donné au listing 8.4.

Listing 8.6 – Script CPL simple : faire suivre un appel

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <cpl xmlns="urn:ietf:params:xml:ns:cpl"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd ">
5    <incoming>
6      <location url="sip:phoenix@barbade.enseirb.fr">
7        <proxy/>
8      </location>
9    </incoming>
10 </cpl>
```

Les lignes 2 à 4 déclarent l'espace de nommage par défaut comme étant celui de CPL ainsi que son emplacement. L'élément `incoming` aux lignes 5-9 déclare que les actions qu'il contient doivent être réalisées sur les appels entrants. L'élément `location` ajoute l'adresse cible à l'environnement courant. L'élément `proxy` fait suivre l'appel vers l'adresse trouvée dans l'environnement.

8.4.2.2 Syntaxe Abstraite

La syntaxe standard de CPL est définie par un schéma XML. Cependant, afin de pouvoir la manipuler dans notre espace technique, il est nécessaire de redéfinir la syntaxe abstraite de CPL en KM3. Ceci correspond à la boîte *DDMM : KM3* du langage CPL de la figure 8.10. Le listing 8.7 donne un extrait de ce métamodèle. La version complète de ce métamodèle est donnée dans le listing B.3 de l'annexe B.

Un script CPL a un élément *CPL* pour racine (lignes 1-3) qui contient un élément *Incoming* (line 9). Le fait que la classe *CPL* soit la racine n'est pas défini en KM3 mais dans la syntaxe concrète. Nous y faisons cependant référence ici afin de rendre les explications plus claires. La classe abstraite *NodeContainer* (lignes 5-7) représente tous les éléments qui peuvent contenir des noeuds (classe *Node*, ligne 11). Il peut s'agir ici des éléments *Incoming* et *Location* (lignes 13-15). *Proxy* (ligne 21) est une action de signalisation (classe *SignallingAction*, ligne 19), donc une action (classe *Action*, ligne 17).

Listing 8.7 – Extrait du métamodèle CPL en KM3

```
1  class CPL {
2    reference incoming[0-1] container : Incoming;
3  }
4
5  abstract class NodeContainer {
6    reference contents[0-1] container : Node;
7  }
8
9  class Incoming extends NodeContainer {}
10
11 abstract class Node {}
12
13 class Location extends Node, NodeContainer {
14   attribute url : String;
15 }
16
17 abstract class Action extends Node {}
18
19 abstract class SignallingAction extends Action {}
20
21 class Proxy extends SignallingAction {}
```

8.4.2.3 Syntaxe Concrète

La syntaxe concrète de CPL est basée sur XML. Un outil comme TCS n'est donc pas vraiment adapté. La solution que nous avons retenue consiste en deux étapes.

La première utilise un injecteur XML générique capable de créer un modèle XML conforme à un métamodèle XML générique à partir de n'importe quel document XML. Le métamodèle XML ne contient que les concepts indépendants de tout schéma tels que élément et attribut. Il a été présenté dans le chapitre 5 sous forme d'un diagramme de classes à la figure 5.1 et en KM3 au listing 5.1. Le coût de cette première étape est très faible puisque cet injecteur et ce métamodèle XML sont des composants génériques fournis par AMMA.

Dans un second temps, nous transformons le modèle XML en un modèle CPL à l'aide d'une transformation ATL. Ceci correspond à la boîte *XML2CPL : ATL* du langage CPL de la figure 8.10. Le listing

8.8 donne un extrait de cette transformation appelée *XML2CPL* (ligne 1). Elle traduit un modèle XML en un modèle CPL (ligne 2) en utilisant une bibliothèque de *helpers* pour la manipulation de modèles XML (ligne 4). Cette bibliothèque fournit le *helper* `getElemsByNames` (utilisé à la ligne 12) qui retourne tous les éléments fils de l'élément contexte dont le nom est inclus dans la liste passée en paramètre. Une seule règle est montrée ici : la règle *CPL* (lignes 6-17) qui transforme la racine du document XML (*XML!Root*) en un élément CPL (*CPL!CPL*). La notion de racine CPL est ainsi définie par correspondance avec la notion de racine en XML. L'élément imbriqué `incoming` est attaché à cette racine (lignes 13-15). La version complète de cette transformation est donnée dans le listing B.4 de l'annexe B.

Listing 8.8 – Extrait de la transformation XML vers CPL, écrite en ATL

```
1  module XML2CPL;
2  create OUT : CPL from IN : XML;
3
4  uses XMLHelpers;
5
6  rule CPL {
7    from
8      s : XML!Root (
9        s.name = 'cpl'
10     )
11   to
12     t : CPL!CPL (
13       incoming <- s.getElemsByNames(
14           Sequence {'incoming'}
15         )->first()
16     )
17 }
```

8.4.2.4 Sémantique Dynamique

Une seconde transformation ATL (appelée *CPL2SPL*) fournit une implémentation de la sémantique de CPL par traduction des concepts CPL en leurs équivalents SPL. Ceci correspond à la boîte *CPL2SPL : ATL* du langage CPL de la figure 8.10. Le listing 8.9 donne un extrait de cette transformation. La ligne 2 déclare les modèles source et cible respectivement conformes aux métamodèles CPL et SPL. La règle *CPL2Program* (lignes 4-19) transforme l'élément CPL racine (lignes 5-6) en un programme SPL (lignes 8-10), un service (lignes 11-15) et un dialogue (lignes 16-18). La version complète de cette transformation est donnée dans le listing B.5 de l'annexe B.

Listing 8.9 – Extrait de la transformation CPL vers SPL écrite en ATL

```
1  module CPL2SPL;
2  create OUT : SPL from IN : CPL;
3
4  rule CPL2Program {
5    from
6      s : CPL!CPL
7    to
8      t : SPL!Program (
9        service <- service
10     ),
11     service : SPL!Service (
12       name <- 'unnamed',
13       declarations <- s.subActions,
14       sessions <- dialog
15     ),
16     dialog : SPL!Dialog (
17       methods <- Sequence {s.incoming, s.outgoing}
18     )
```

```
19 }
```

La figure 8.11 montre le scénario complet de transformation. Un script CPL conforme au schéma CPL (`Sample.cpl`) est tout d'abord injecté en un modèle XML conforme au métamodèle XML. Ce modèle XML est ensuite transformé en un modèle CPL par la transformation *XML2CPL.atl*. La transformation noyau *CPL2SPL.atl* est ensuite appliquée pour générer un modèle SPL. Ce dernier est enfin sérialisé en un programme SPL à l'aide de TCS.

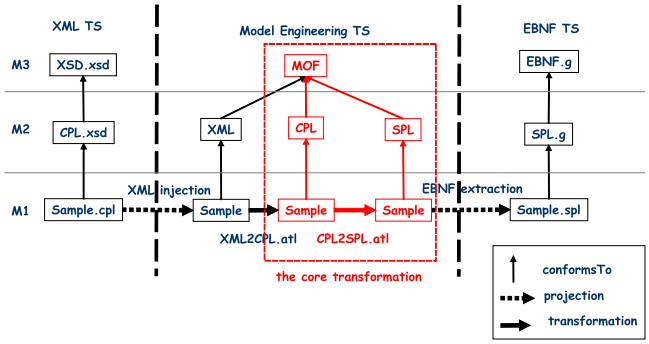

Figure 8.11 – Scénario complet de la transformation CPL vers SPL

8.4.3 Conclusion

Cette section a détaillé un cas d'étude pour ATL et plus généralement AMMA : l'implémentation de deux langages dédiés de téléphonie. Des détails ont été donnés sur l'implémentation de SPL et CPL. Ce cas d'étude illustre la manière dont les langages dédiés noyau de AMMA peuvent être utilisés pour capturer différents aspects d'un langage dédié. KM3 est utilisé pour exprimer les métamodèles de définition de domaine (DDMM), par exemple ceux de KM3, ATL, TCS, ASM, CPL et SPL. Certaines syntaxes concrètes sont définies en TCS : celles de KM3, ATL, TCS, ASM et SPL. D'autres, comme la syntaxe concrète de CPL, sont définies par des transformations ATL. La sémantique dynamique d'un langage dédié peut être définie formellement en ASM, ce que nous avons fait pour ATL et SPL.

De plus, des transformations depuis n'importe quel langage dédié DSL_A vers n'importe quel autre langage dédié DSL_B peuvent aussi être implémentées en ATL. Ceci peut, par exemple, être utilisé pour implémenter la sémantique de DSL_A en fonction de la sémantique de DSL_B. C'est notamment ce que nous avons fait pour la sémantique de CPL avec la transformation CPL vers SPL (voir la section 8.4.2.4). Une telle transformation peut ensuite être utilisée pour traduire les programmes exprimés en DSL_A vers des programmes exprimés en DSL_B. Une autre utilisation d'ATL est l'implémentation de syntaxes concrètes pour les langages dédiés, par exemple celle de CPL à l'aide de la transformation *XML2CPL* (voir la section 8.4.2.3).

8.5 Conclusion

Les travaux présentés dans ce chapitre montrent quelques utilisations possibles de AMMA et de ses langages dédiés noyau (KM3, TCS et ATL) dans trois domaines différents :

- pour la gestion de la traçabilité dans les transformations ATL,
- pour la vérification de modèles,

- pour l'implémentation de langages dédiés.

Les versions complètes des métamodèles, syntaxes concrètes et transformations présentés dans ce chapitre sont disponibles dans le projet AM3 [26]. De plus, des listings complets sont donnés en annexe B pour le troisième cas d'étude sur l'implémentation des langages dédiés de téléphonie par internet CPL et SPL. Plus d'une centaine de transformations utilisant AMMA et couvrant des domaines variés sont aussi disponibles dans le projet AM3. La plupart de ces scénarios sont fournis avec un ou plusieurs jeux de tests permettant de les évaluer. Mais le test le plus significatif de l'utilisabilité de notre approche réside dans la très grande diversité des applications actuellement réalisées par la communauté internationale des utilisateurs d'ATL [27].

CHAPITRE 9
Conclusion

9.1 Introduction

Ce chapitre tire les conclusions des travaux présentés. La section 9.2 rappelle les problèmes auxquels nos travaux contribuent à donner des solutions. La section 9.3 résume ces solutions en faisant référence aux chapitres dans lesquels elles ont été présentées. La section 9.4 présente quelques perspectives et extensions possibles de nos travaux qui permettraient d'en dépasser les limites. Enfin, la section 9.5 résume les contributions de nos travaux.

9.2 Rappel des problèmes considérés

Nous avons présenté la problématique de l'ingénierie des modèles (IdM) ainsi que ses relations avec la technologie des objets et l'ingénierie des langages dédiés au chapitre 2. L'applicabilité des techniques de l'IdM à la gestion de données hétérogènes a aussi été évoquée. Un ensemble de problèmes a été identifié que nous rappelons ici :

1. L'IdM a besoin de définitions précises des notions de modèle, métamodèle et métamétamodèle. Ceci nécessite aussi la définition formelle d'un métamétamodèle basé sur ces définitions. Bien qu'il existe de nombreuses approches basées sur les principes de l'IdM, il manque encore une formalisation de ces principes.
2. Un cadre conceptuel rigoureux et formel de l'IdM est indispensable. Cependant, ce cadre ne peut pas être défini *ex nihilo* et sans considérer l'évolution industrielle des pratiques, sous peine de devenir inutile et sans impact. La prise en compte, au moins partielle, des normes de droit et de fait, telles que celles de l'OMG, est indispensable. L'alignement d'un schéma minimal et régulier sur des normes complexes et en évolution a constitué un défi important de notre travail.
3. Le cycle de vie du logiciel peut s'interpréter comme un réseau de traitement de données hétérogènes. Pour que cette vision soit applicable et généralisable, il faut que le réseau soit le plus régulier possible. Ceci suppose non seulement que les données produites et consommées soient traitées de façon homogène, mais que les différents opérateurs le soient aussi.
4. Afin de pouvoir gérer des données hétérogènes, des systèmes de projection entre espaces techniques doivent être définis. En effet, des approches telles que le MDA de l'OMG ne définissent souvent que des formats de représentation des modèles pour leurs propres besoins, mais ne fournissent pas d'outils pour manipuler des modèles définis avec des technologies différentes.
5. L'ingénierie des modèles a besoin de langages de transformation de modèles. De tels langages permettent la définition de programmes pour la traductions de modèles exprimés à l'aide d'un métamodèle donné vers des modèles conformes à un autre métamodèle. Les objectifs de ces transformations sont très variés. Il peut s'agir de traduire des données au format d'un outil vers le format d'un autre outil afin de les rendre interopérables. Une transformation peut aussi être réalisée entre

deux langages de programmation afin de définir la sémantique du langage source en termes de celle du langage cible.

6. L'application de l'ingénierie des modèles à la construction de langages dédiés nécessite la création d'"établis à langages". Il s'agit de plateformes permettant la définition de langages dédiés à l'aide des techniques de l'ingénierie des modèles. De telles boîtes à outils de modélisation doivent, entre autres, fournir des systèmes de représentation des syntaxes abstraites et concrètes des langages. Il est aussi nécessaire qu'ils permettent l'implémentation de leurs sémantiques.

Au cours des chapitres suivant, nous avons donné des solutions à ces problèmes. Elles sont présentées à la section suivante.

9.3 Solutions apportées aux différents problèmes

Nous avons défini au chapitre 4 la notion de modèle en nous basant sur la notion de multi-graphe orienté. Cette définition fait intervenir la notion de modèle de référence qui sert à typer, via la relation μ, les éléments (noeuds et arcs) du modèle. Nous disons qu'un modèle est conforme à son modèle de référence. Ce modèle de référence est lui aussi un modèle conforme à son propre modèle de référence. Cette première définition autorise un nombre indéfini de niveaux de modélisations. Cependant, afin de pouvoir appliquer notre définition aux techniques existantes de l'ingénierie des modèles, nous avons ensuite spécifié une architecture basée sur trois niveaux de modélisation :

- le niveau *M1* composé des modèles terminaux, c'est-à-dire des modèles qui ne sont pas des modèles de référence pour d'autres modèles ;
- le niveau *M2* composé des métamodèles, qui sont les modèles de référence des modèles terminaux ;
- et enfin le niveau *M3* composé d'un unique métamétamodèle, qui est le modèle de référence de tous les métamodèles et qui est de plus son propre modèle de référence.

Cette architecture est compatible avec l'approche MDA [53] mais aussi, entre autres, avec les approches Eclipse EMF [14], Microsoft Software Factories et DSL Tools [34] et GME [33]. De plus, nous avons observé que plusieurs espaces techniques, tels que celui basé sur XML ou celui des grammaires, possèdent eux aussi des architectures similaires. Ensuite, nous avons spécifié au chapitre 5 un langage de métamodélisation (i.e. un métamétamodèle) appelé KM3. Ce langage est basé sur les définitions du chapitre 4 et possède une formalisation basée sur la logique du premier ordre. Une notation textuelle peut être utilisée afin de définir des métamodèles en KM3. Ces définitions et ce métamétamodèle contribuent à répondre au problème 1. Leur compatibilité avec certaines des recommandations MDA de l'OMG, dont MOF, prend en compte le problème 2.

Le chapitre 6 a présenté le langage TCS de définition de transformations bidirectionnelles entre les espaces techniques de l'ingénierie des modèles et des grammaires. Le principe de ce langage est de définir, dans un modèle conforme au métamodèle TCS, toutes les informations nécessaires à la génération d'une grammaire pour un métamodèle mais qui ne sont pas présentes dans la définition de ce métamodèle (par exemple en KM3). Il s'agit des mots-clés, des séparateurs et autres symboles tels que les opérateurs, de l'enchaînement syntaxique des différents éléments, etc. TCS a été appliqué à la définition de la syntaxe textuelle du langage KM3 mais aussi à la définition de sa propre syntaxe textuelle. Ce langage fournit une réponse au problème 4, limitée cependant à la projection entre deux espaces techniques particuliers.

Le langage ATL de transformation de modèles a été présenté au chapitre 7. Il s'agit d'un langage hybride possédant à la fois des aspects déclaratifs et impératifs. La navigation dans les modèles s'effectue à l'aide du langage OCL [57]. OCL permet la définition d'expressions effectuant des requêtes sur les modèles source, mais aussi d'opérations et d'attributs permettant d'effectuer des parcours complexes

pouvant utiliser la récursion. La navigation dans les modèles cible n'est pas autorisée et les modèles source ne peuvent pas être modifiés. La partie impérative d'ATL est constituée de règles appelées, de blocs impératifs et d'instructions de contrôle de flux. Une règle appelée composée d'un bloc impératif, c'est-à-dire d'une succession d'instructions, peut être considérée comme étant une procédure. La partie déclarative est constituée de règles dont l'exécution est conditionnée par la reconnaissance d'un motif source dans les modèles source. Un motif source définit un ensemble de types ainsi qu'une garde exprimée en OCL. L'exécution d'une règle déclarative entraîne la création, dans les modèles cible, des éléments définis dans son motif cible. Ces deux paradigmes peuvent être utilisés séparément, pour la définition de transformations totalement déclaratives ou totalement impératives, ou bien combinés. Une règle appelée peut avoir un motif cible définissant déclarativement des éléments à créer. De même, une règle déclarative peut posséder un bloc impératif complétant par exemple l'initialisation des éléments cible à l'aide d'un algorithme implémenté impérativement. Le langage ATL est une réponse au problème 5. Le problème 2 est pris en compte par l'utilisation de la recommandation OCL de l'OMG comme langage de navigation. Les concepts et relations du langage ATL sont capturés dans un métamodèle défini en KM3. Ceci permet de traiter les transformations comme des modèles, ce qui répond au problème 3

La plateforme de gestion de modèles AMMA est composée des trois langages principaux : KM3, TCS et ATL. Elle a été présentée au chapitre 4. AMMA permet la construction de langages dédiés à l'aide d'ensembles coordonnés de modèles. La syntaxe abstraite d'un langage, constituée des concepts et relations de son domaine, est prise en compte par un métamodèle défini en KM3. Une syntaxe concrète textuelle peut être définie à l'aide de TCS. Différentes transformations peuvent ensuite être définies entre ces langages. Les quatre blocs fonctionnels de AMMA ont été présentés : transformation (avec ATL), projections (avec ATP pour *A*TLAS *Technical Projectors*), tissage de modèles (avec AMW pour *A*TLAS *Model Weaver*) et gestion globale de modèles (avec AM3 pour *A*TLAS *MegaModel Management*). Les deux dernières briques (AMW et AM3) font l'objet de recherches en cours. La plateforme AMMA constitue un "établi à langages" contribuant à répondre au problème 6.

Des prototypes pour les trois langages KM3, ATL et TCS cités ci-dessus ont été implémentés au cours de nos travaux. Ces prototypes ont permis de tester l'applicabilité de l'ingénierie des modèles et plus particulièrement de la plateforme AMMA à différents domaines.

9.4 Perspectives et extensions

Dans son état actuel, la plateforme AMMA permet la définition de scénarios de transformation couvrant de nombreux domaines. Ceci est illustré par les cas d'étude présentés au chapitre 8 mais aussi par les bibliothèques de transformations et de métamodèles disponibles dans le projet AM3 [26]. Cependant, certains problèmes limitent encore l'utilisation de AMMA pour certains types d'applications. Nous en évoquons certains dans cette section et nous suggérons des axes de recherche possibles afin de tenter de les résoudre.

La plupart des scénarios de transformation disponibles ne contiennent qu'un nombre limité de modèles : métamodèles KM3, transformations ATL, syntaxes concrètes TCS, etc. Il n'est pas aujourd'hui possible de combiner automatiquement ces scénarios pour en former de plus complexes. En effet, il manque pour ce faire un ensemble de métadonnées permettant de déterminer quelles combinaisons sont possibles. Par exemple, le fait qu'une transformation donnée utilise tel métamodèle source et tel métamodèle cible n'est souvent défini qu'en langage naturel dans la documentation. Similairement, bien qu'il soit aisé de déterminer la conformité d'un modèle donné à son métamodèle dans le contexte d'un scénario, il est beaucoup plus difficile de le faire à l'échelle de la bibliothèque entière. Ce sont ces problèmes

que la brique fonctionnelle AM3 de la plateforme AMMA a pour objectif de résoudre en fournissant un système de gestion global de modèles.

Nous avons vu au chapitre 8 que la traduction entre deux langages du même domaine requiert plusieurs étapes de transformation. Ce découpage en injecteurs, transformations de modèles et extracteurs permet de simplifier chaque problème pris séparément. Cependant, la solution globale a un problème relativement complexe demande un grand nombre de ces étapes. Il sera donc probablement souvent nécessaire de pouvoir enchaîner de nombreuses transformations. La brique AM3 sera certainement utile à la gestion des nombreux métamodèles et transformations impliqués. Mais un problème de performance risque d'apparaître. Certaines possibilités d'optimisation des transformations ATL ont été présentées au chapitre 7 et plus particulièrement à la section 7.3.5.2. L'étude de ces possibilités reste encore à faire. De plus, l'application de l'ingénierie des modèles à certains contextes échangeant des données hétérogènes en temps réel, tels que la gestion de réseaux de capteurs, offrira probablement d'autres perspectives d'optimisation.

ATL est un langage dédié à la transformation de modèles, mais il n'est pas particulièrement adapté à un ou des scénarios de transformation particuliers. Par exemple, il est possible de définir des transformations de fusion de modèles ou encore de découverte de correspondances entre modèles en ATL. Cependant, ces problèmes particuliers requièrent entre autres des techniques de reconnaissance de motifs spécifiques. Plusieurs solutions existent afin d'adresser ces scénarios. Il serait ainsi possible d'étendre le langage ATL avec des constructions spécifiques. Mais nous pensons qu'il y a un trop grand nombre d'extensions possibles pour espérer obtenir un langage unique de transformation adapté à toutes les classes de problèmes. Une autre solution serait de définir des langages de transformation dédiés à des problèmes de transformation particuliers tels que ceux évoqués ci-dessus. Les capacités de définition de langages dédiés de la plateforme AMMA devraient pouvoir permettre la mise en œuvre de ces langages. La sémantique de ces nouveaux langages pourrait être définie par des transformations ciblant le langage ATL. Dans le cas ou les techniques de reconnaissance de motifs d'ATL ne seraient pas adaptée à un scénario donné, il serait toujours possible de générer du code impératif. Par ailleurs, la machine virtuelle ATL est une autre cible potentielle pour la compilation de ces langages. Il est aussi possible d'utiliser la brique AMW dans le but de définir des langages de transformation dédiés. Ceci est déjà illustré par différents scénarios de la bibliothèque de tissage du projet AM3.

Enfin, les possibilités de projections offertes par TCS sont limitées à la traduction de modèles entre les espaces techniques de l'ingénierie des modèles et des grammaires. Afin de pouvoir gérer une plus grande variété de données hétérogènes et ainsi de rendre AMMA utilisable pour un plus grand nombre de problèmes, il est nécessaire de définir de nouveaux projecteurs. La technique de projection depuis l'espace technique XML présentée au chapitre 8 sur l'exemple du langage CPL est un autre exemple de "pont" qu'il faudrait définir avec d'autres espaces techniques. Nous pensons avoir démontré concrètement l'intérêt d'utiliser les espaces techniques et les projecteurs pour l'organisation des solutions informatiques. Ces propositions restent évidemment à étendre. Des projecteurs depuis et vers différents types de fichiers binaires pourraient, par exemple, être construits.

9.5 Résumé des contributions

Notre travail à consisté à pousser plus avant la frontière d'applicabilité de l'ingénierie des modèles. De nombreuses propositions industrielles dans ce domaine ont successivement vu le jour récemment : le MDA de l'OMG, Eclipse EMF, Microsoft DSL Tools, GME, etc. Notre approche a consisté, tout en restant autant que possible compatibles avec ces standards industriels, à proposer un schéma conceptuel plus

abstrait, plus simple et plus homogène. Cette proposition va au delà d'une proposition théorique puisqu'elle a été validée non seulement par la réalisation de prototypes et par son application à un ensemble de cas d'étude, mais aussi par une interaction directe et permanente avec une communauté significative d'utilisateurs de toutes origines (académiques et industrielles).

Bibliographie

[1] David H. AKEHURST, Gareth HOWELLS et Klaus D. MCDONALD-MAIER.
Kent Model Transformation Language.
Dans *Proceedings of the Model Transformations in Practice (MTiP) Workshop at MoDELS 2005*, 2005.

[2] David H. AKEHURST et Stuart J. H. KENT.
A Relational Approach to Defining Transformations in a Metamodel.
Dans Jean-Marc JEZEQUEL et Heinrich HUSSMANN, réds., *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language, LNCS 2460*, pages 243–258. Springer, octobre 2002.

[3] Marcus ALANEN et Ivan PORRES.
A Relation Between Context-Free Grammars and Meta Object Facility Metamodels.
Rapport technique 606, Turku Centre for Computer Science, mars 2003.

[4] Ola ANDERSSON *& al.*
W3C Working Draft of Scalable Vector Graphics (SVG) 1.2.
http://www.w3.org/TR/SVG12/, 2005.

[5] András BALOGH et Dániel VARRÓ.
Advanced Model Transformation Language Constructs in the VIATRA2 Framework.
Dans *Proceedings of ACM Symposium on Applied Computing (SAC 06), model transformation track*, pages 1280–1287, Dijon, France, 2006.

[6] Douglas BATES *& al.*
R Language Definition, 2006.
Disponible à l'adresse
http://stat.ethz.ch/R-manual/R-patched/doc/manual/R-lang.html.

[7] Jean BÉZIVIN.
Sur les principes de base de l'ingénierie des modèles.
RSTI-L'Objet, 10(4):145–157, 2004.

[8] Jean BÉZIVIN.
sNets: A First Generation Model Engineering Platform.
Dans *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, pages 169–181. Springer Berlin / Heidelberg, janvier 2006.

[9] Jean BÉZIVIN, Fabian BÜTTNER, Martin GOGOLLA, Frédéric JOUAULT, Ivan KURTEV et Arne LINDOW.
Model Transformations? Transformation Models!
Dans *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, Proceedings*, 2006.
À paraître.

[10] Jean BÉZIVIN, Grégoire DUPÉ, Frédéric JOUAULT, Gilles PITETTE et Jamal E. ROUGUI.

First experiments with the ATL model transformation language: Transforming XSLT into XQuery.
Dans *OOPSLA 2003 Workshop*, Anaheim, California, 2003.

[11] Jean BÉZIVIN et Frédéric JOUAULT.
Using ATL for Checking Models.
Dans *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT)*, Tallinn, Estonia, 2005.

[12] Jean BÉZIVIN, Frédéric JOUAULT, Ivan KURTEV et Patrick VALDURIEZ.
Model-based DSL Frameworks.
Dans *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, OR, USA*. ACM, 2006.
À paraître.

[13] Jean BÉZIVIN et Ivan KURTEV.
Model-based Technology Integration with the Technical Space Concept.
Dans *Proceedings of the Metainformatics Symposium*. Springer-Verlag, 2005.

[14] Frank BUDINSKY, David STEINBERG, Raymond ELLERSICK, Ed MERKS, Stephen A. BRODSKY et Timothy J. GROSE.
Eclipse Modeling Framework.
Addison Wesley, 2003.

[15] Laurent BURGY, Charles CONSEL, Fabien LATRY, Julia LAWALL, Nicolas PALIX et Laurent RÉVEILLÈRE.
Language Technology for Internet-Telephony Service Creation.
Dans *IEEE International Conference on Communications*, 6 2006.

[16] Krzysztof CZARNECKI et Simon HELSEN.
Classification of Model Transformation Approaches.
Dans *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003.

[17] Juan de LARA et Hans VANGHELUWE.
AToM^3: A Tool for Multi-formalism and Meta-modelling.
Dans Ralf-Detlef KUTSCHE et Herbert WEBER, réds., *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, LNCS 2306*, pages 174–188. Springer, 2002.

[18] DSTC, IBM et CBOP.
DSTC et al Second Revised QVT Submission, OMG Document ad/2004-01-06, 2004.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ad/2004-01-06.

[19] Keith DUDDY, Anna GERBER, Michael LAWLEY, Kerry RAYMOND et Jim STEEL.
Model Transformation: A declarative, reusable patterns approach.
Dans *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 174, Washington, DC, USA, 2003. IEEE Computer Society.

[20] ECLIPSE FOUNDATION.
Generative Model Transformer (GMT) Home page.
http://www.eclipse.org/gmt/, 2006.

[21] Hartmut EHRIG, Ulrike PRANGE et Gabriele TAENTZER.

Fundamental Theory for Typed Attributed Graph Transformation.
Dans *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings, LNCS 3256*, pages 161–177. Springer-Verlag, octobre 2004.

[22] ÉQUIPE ATLAS.
KM3: Kernel MetaMetaModel, Manual, 2005.
Disponible à l'adresse
http://www.eclipse.org/gmt/atl/doc/KernelMetaMetaModel[v00.06].pdf.

[23] ÉQUIPE ATLAS.
Specification of the ATL Virtual Machine, 2005.
Disponible à l'adresse
http://www.eclipse.org/gmt/atl/doc/ATL_VMSpecification[v00.01].pdf.

[24] ÉQUIPE ATLAS.
ATL User Manual, 2006.
Disponible à l'adresse
http://www.eclipse.org/gmt/atl/doc/ATL_User_Manual[v0.7].pdf.

[25] ÉQUIPE ATLAS.
Atlantic Metamodel Zoo.
http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/, 2006.

[26] ÉQUIPE ATLAS.
ATLAS MegaModel Management (AM3) Home page.
http://www.eclipse.org/gmt/am3/, 2006.

[27] ÉQUIPE ATLAS.
Groupe de discussion ATL, 2006.
Disponible à l'adresse
http://groups.yahoo.com/group/atl_discussion/.

[28] ÉQUIPE PHOENIX.
The Session Processing Language (SPL), Reference site.
http://phoenix.labri.fr/software/spl/, 2006.

[29] Frédéric FONDEMENT et Thomas BAAR.
Making Metamodels Aware of Concrete Syntax.
Dans Alan HARTMAN et David KREISCHE, réds., *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings, LNCS 3748*, pages 190–204. Springer, 2005.

[30] Emden R. GANSNER et Stephen C. NORTH.
An open graph visualization system and its applications to software engineering.
Software — Practice and Experience, 30(11):1203–1233, 2000.

[31] Tracy GARDNER, Catherine GRIFFIN, Jana KOEHLER et Rainer HAUSER.
Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards final Standard, OMG Document ad/2003-08-02, 2003.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ad/2003-08-02.

[32] Angelo GARGANTINI, Elvinia RICCOBENE et Patrizia SCANDURRA.
Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware.

Dans *Proceedings of the European Workshop on Milestones, Models and Mappings for Model-Driven Architecture (3M4MDA)*, pages 33–47. Centre for Telematics and Information Technology, University of Twente, 2006.

[33] GME.
The Generic Modeling Environment, Reference site, 2006.
Disponible à l'adresse
http://www.isis.vanderbilt.edu/Projects/gme.

[34] Jack GREENFIELD, Keith SHORT, Steve COOK et Stuart KENT.
Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.
Wiley, août 2004.

[35] Vidar Bronken GUNDERSEN et Zeger W. HENDRIKSE.
BibTeX as XML markup, the BibTeXML homepage, 2005.
Disponible à l'adresse
http://bibtexml.sourceforge.net/.

[36] David HAREL et Bernhard RUMPE.
Meaningful Modeling: What's the Semantics of "Semantics"?
Computer, 37(10):64–72, 2004.

[37] Frédéric JOUAULT.
Loosely Coupled Traceability for ATL.
Dans *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, Nuremberg, Germany, 2005.

[38] Frédéric JOUAULT et Jean BÉZIVIN.
KM3: a DSL for Metamodel Specification.
Dans *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, pages 171–185, Bologna, Italy, 2006.

[39] Frédéric JOUAULT, Jean BÉZIVIN, Charles CONSEL, Ivan KURTEV et Fabien LATRY.
Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages.
Dans *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France*, 2006.

[40] Frédéric JOUAULT, Jean BÉZIVIN et Ivan KURTEV.
TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering.
Dans *GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering*, 2006.
À paraître.

[41] Frédéric JOUAULT et Ivan KURTEV.
Transforming Models with ATL.
Dans *Proceedings of the Model Transformations in Practice (MTiP) Workshop at MoDELS 2005*, 2005.

[42] Frédéric JOUAULT et Ivan KURTEV.
On the Architectural Alignment of ATL and QVT.
Dans *Proceedings of ACM Symposium on Applied Computing (SAC 06), model transformation track*, pages 1188–1195, Dijon, France, 2006.

[43] Frédéric JOUAULT et Ivan KURTEV.
Transforming Models with ATL.

Dans *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, pages 128–138. Springer Berlin / Heidelberg, janvier 2006.

[44] Audris KALNINS, Janis BARZDINS et Edgars CELMS.
Model Transformation Language MOLA.
Dans Uwe ASSMANN, Mehmet AKSIT et Arend RENSINK, réds., *Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers, LNCS 3599*, pages 62–76. Springer, 2004.

[45] Gabor KARSAI, Aditya AGRAWAL, Feng SHI et Jonathan SPRINKLE.
On the Use of Graph Transformation in the Formal Specification of Model Interpreters.
J. UCS, 9(11):1296–1321, 2003.

[46] Ivan KURTEV, Jean BÉZIVIN et Mehmet AKSIT.
Technological Spaces: An Initial Appraisal.
Dans *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[47] Michael LAWLEY et Jim STEEL.
Practical Declarative Model Transformation with Tefkat.
Dans *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, pages 139–150. Springer Berlin / Heidelberg, janvier 2006.

[48] Jonathan LENNOX, Xiaotao WU et Henning SCHULZRINNE.
Call Processing Language (CPL): A Language for User Control of Internet Telephony Services, RFC 3880.
http://www.ietf.org/rfc/rfc3880.txt, 2004.

[49] Paul R. MCJONES, réd.
The 1995 SQL Reunion: People, Project, and Politics, May 29, 1995, volume SRC1997-018, 1997.

[50] Marjan MERNIK, Jan HEERING et Anthony M. SLOANE.
When and how to develop domain-specific languages.
ACM Comput. Surv., 37(4):316–344, décembre 2005.

[51] NETBEANS.ORG.
Netbeans Meta Data Repository (MDR), 2006.
Disponible à l'adresse
http://mdr.netbeans.org/.

[52] OBJECT AND REFERENCE MODEL SUBCOMMITTEE (ORMSC) OF THE OMG ARCHITECTURE BOARD.
An ORMSC Definition of MDA, OMG Document ormsc/2004-08-02, 2004.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ormsc/2004-08-02.

[53] OBJECT AND REFERENCE MODEL SUBCOMMITTEE (ORMSC) OF THE OMG ARCHITECTURE BOARD.
A Proposal for an MDA Foundation Model, white paper, OMG Document ormsc/2005-08-01, juillet 2005.
Disponible à l'adresse

http://www.omg.org/cgi-bin/doc?ormsc/2005-08-01.

[54] OMG.
Meta Object Facility (MOF) Specification, version 1.4, OMG Document formal/2002-04-03, 2002.
Disponible à l'adresse
http://www.omg.org/technology/documents/formal/mof.htm.

[55] OMG.
MOF 2.0 Query / Views / Transformations RFP, OMG Document ad/2002-04-10, 2002.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ad/2002-04-10.

[56] OMG.
UML 2.0 Superstructure Specification, OMG Document ptc/2003-08-02, 2003.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ptc/2003-08-02.

[57] OMG.
UML OCL 2.0 Specification, OMG Document ptc/2003-10-14, 2003.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.

[58] OMG.
Human-Usable Textual Notation, v1.0, OMG Document formal/2004-08-01, 2004.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?formal/2004-08-01.

[59] OMG.
MOF Model to Text Transformation Language, OMG Document ad/2004-04-07, 2004.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ad/2004-04-07.

[60] OMG.
MOF 2.0 / XMI Mapping Specification, v2.1, OMG Document formal/2005-09-01, 2005.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[61] OMG.
MOF QVT Final Adopted Specification, OMG Document ptc/2005-11-01, 2005.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[62] OMG.
Meta Object Facility (MOF) 2.0 Core Specification, OMG Document formal/2006-01-01, 2006.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[63] OPENQVT.
OpenQVT revised submission to the MOF 2.0 Q/V/T RFP, OMG Document ad/2003-08-05, 2003.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?ad/2003-08-05.

[64] Terence PARR.
ANTLR v3.
http://antlr.org/v3/index.html, 2006.

[65] Jonathan ROSENBERG *& al.*
SIP: Session Initiation Protocol, RFC 3261.
http://www.ietf.org/rfc/rfc3261.txt, 2002.

[66] Davide Di RUSCIO, Frédéric JOUAULT, Ivan KURTEV, Jean BÉZIVIN et Alfonso PIERANTONIO.
A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development.
Rapport technique 06.03, LINA, 2006.

[67] Davide Di RUSCIO, Frédéric JOUAULT, Ivan KURTEV, Jean BÉZIVIN et Alfonso PIERANTONIO.
Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs.
Rapport technique 06.02, LINA, 2006.

[68] Sun.
Java Metadata Interface (JMI) Specification, 2002.
Java Specification Request number 40.
Disponible à l'adresse
http://www.jcp.org/en/jsr/detail?id=40.

[69] Gabriele TAENTZER.
AGG: A Graph Transformation Environment for Modeling and Validation of Software.
Dans John L. PFALTZ, Manfred NAGL et Boris BÖHLEN, réds., *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers, LNCS 3062*, pages 446–453. Springer, 2003.

[70] Gabriele TAENTZER et Giovanni Toffetti CARUGHI.
A Graph-Based Approach to Transform XML Documents.
Dans Luciano BARESI et Reiko HECKEL, réds., *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings, LNCS 3922*, pages 48–62. Springer, 2006.

[71] Gabriele TAENTZER, Karsten EHRIG, Esther GUERRA, Juan de LARA, Laszló LENGYEL, Tihamér LEVENDOVSZKY, Ulrike PRANGE, Dániel VARRÓ et Szilvia VARRÓ-GYAPAY.
Model Transformation by Graph Transformation: A Comparative Study.
Dans *Proceedings of the Model Transformations in Practice (MTiP) Workshop at MoDELS 2005*, 2005.

[72] Arie van DEURSEN, Paul KLINT et Joost VISSER.
Domain-Specific Languages: An Annotated Bibliography.
SIGPLAN Notices, 35(6):26–36, 2000.

[73] Dániel VARRÓ et András PATARICZA.
VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML.
Journal of Software and Systems Modeling, 2(3):187–210, octobre 2003.

[74] Dániel VARRÓ, Gergely VARRÓ et András PATARICZA.
Designing the Automatic Transformation of Visual Languages.
Science of Computer Programming, 44(2):205–227, août 2002.

[75] Éric VÉPA, Jean BÉZIVIN, Hugo BRUNELIÈRE et Frédéric JOUAULT.
Measuring Model Repositories.
Dans *Model Size Metrics Workshops of the MoDELS/UML 2006 conference*, 2006.
À paraître.

[76] Edward D. WILLINK.
UMLX: A graphical transformation language for MDA.
Dans Arend RENSINK, réd., *CTIT Technical Report TR-CTIT-03-27*, pages 13–24, Enschede, The Netherlands, juin 2003. University of Twente.

[77] Manuel WIMMER et Gerhard KRAMLER.
Bridging Grammarware and Modelware.
Dans *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, pages 159–168. Springer Berlin / Heidelberg, janvier 2005.

Liste des tableaux

Liste des figures

Liste des listings

Table des matières

Annexes

ANNEXE A

Définition de KM3

Le chapitre 5 a présenté le langage de métamodélisation KM3. Cette annexe contient des éléments complétant la définition de ce langage : la syntaxe textuelle de KM3 à la section A.1, le métamodèle de KM3 défini en KM3 à la section A.2 et la définition de KM3 en Prolog à la section A.3.

A.1 Syntaxe formelle de KM3

La syntaxe de KM3 est définie par la grammaire sans contexte donnée ci-dessous. Les non-terminaux sont représentés en *italique* et les terminaux en **gras**. Le symbole initial est *metamodel*.

Le non-terminal *name* correspond à la notion d'identificateur mais sa définition n'est pas détaillée. En KM3, un identificateur est une lettre ou un symbole de soulignement (_) suivi d'un nombre quelconque de lettres, de symboles de soulignement et de chiffres. Les mots-clés et identificateurs sont sensibles à la casse. Un identificateur ne peut pas prendre la valeur d'un des mots-clés de KM3 : **package**, **class**, **abstract**, **extends**, **attribute**, **reference**, **oppositeOf**, **ordered**, **container**, **datatype**, **enumeration** et **literal**. Un mécanisme d'échappement à l'aide de guillemets est disponible afin de permettre l'utilisation d'identificateurs qui soit sont des mots-clés soit contiennent d'autres caractères que ceux autorisés.

Voici une liste d'identificateurs valides :

- `identificateur`
- `_identificateur3`
- `Package`
- `"package"`
- `"identificateur(complexe)"`

Et voici une liste d'identificateurs non valides :

- `3identificateur`
- `package`
- `identificateur(complexe)`

metamodel → *packages*
packages → *package*
packages → *package packages*
package → **package** *name* **{** *classifiers* **}**
classifiers →
classifiers → *classifier classifiers*
classifier → *class*
classifier → *datatype*
classifier → *enumeration*
class → *isabstract* **class** *name supertypes* **{** *features* **}**
isabstract →
isabstract → **abstract**
supertypes →
supertypes → **extends** *typelist*
typelist → *typeref*
typelist → *typeref*, *typelist*
features →
features → *feature features*
feature → *attribute*
feature → *reference*
attribute → **attribute** *name multiplicity* **:** *typeref* **;**
reference → **reference** *name multiplicityr* **:** *typeref opposite* **;**
multiplicityr → *multiplicity iscontainer*
opposite →
opposite → **oppositeOf** *name*
multiplicity → *bounds*
multiplicity → *bounds* **ordered**
bounds →
bounds → **[** *integer* **-** *integer* **]**
bounds → **[** *integer* **- *** **]**
bounds → **[*** **]**
iscontainer →
iscontainer → **container**
datatype → **datatype** *name* **;**
enumeration → **enumeration** *name* **{** *literals* **}**
literals →
literals → *literal literals*
literal → **literal** *name* **;**
typeref → *name*

A.2 Définition de KM3 en KM3

Le listing A.1 contient la définition de KM3 en KM3 utilisant la syntaxe concrète présentée en A.1.

Listing A.1 – Définition de KM3 en KM3

```
1  package KM3 {
2    abstract class ModelElement {
3      attribute name : String;
4      reference "package" : Package oppositeOf contents;
5    }
6
7    class Classifier extends ModelElement {}
8
9    class DataType extends Classifier {}
10
11   class Enumeration extends Classifier {
12     reference literals[*] ordered container : EnumLiteral oppositeOf enum;
13   }
14
15   class EnumLiteral extends ModelElement {
16     reference enum : Enumeration oppositeOf literals;
17   }
18
```

```
19    class Class extends Classifier {
20      attribute isAbstract : Boolean;
21      reference supertypes[*] : Class;
22      reference structuralFeatures[*] ordered container : StructuralFeature oppositeOf owner;
23    }
24
25    class TypedElement extends ModelElement {
26      attribute lower : Integer;
27      attribute upper : Integer;
28      attribute isOrdered : Boolean;
29      attribute isUnique : Boolean;
30      reference type : Classifier;
31    }
32
33    class StructuralFeature extends TypedElement {
34      reference owner : Class oppositeOf structuralFeatures;
35      reference subsetOf[*] : StructuralFeature oppositeOf derivedFrom;
36      reference derivedFrom[*] : StructuralFeature oppositeOf subsetOf;
37    }
38
39    class Attribute extends StructuralFeature {}
40
41    class Reference extends StructuralFeature {
42      attribute isContainer : Boolean;
43      reference opposite[0-1] : Reference;
44    }
45
46    class Package extends ModelElement {
47      reference contents[*] ordered container : ModelElement oppositeOf "package";
48      reference metamodel : Metamodel oppositeOf contents;
49    }
50
51    class Metamodel {
52      reference contents[*] ordered container : Package oppositeOf metamodel;
53    }
54  }
55
56  package PrimitiveTypes {
57    datatype Boolean;
58    datatype Integer;
59    datatype String;
60  }
```

A.3 Définition de KM3 en Prolog

Le listing A.2 est une définition en Prolog de KM3. Elle utilise les prédicats `node`, `edge` et `prop` présentés au chapitre 5. La majuscule initiale du nom de chaque prédicat est ici remplacée par une minuscule à cause des contraintes du système Prolog utilisé : SWI-Prolog.

Les morceaux de texte commençant par `/*` et se terminant par `*/` sont des commentaires. La verbosité de cette définition en Prolog ne facillite pas sa lecture. Mais, des commentaires donnés dans une syntaxe proche de celle de KM3 permettent de suivre la définition des différents éléments.

Après la définition de KM3, des prédicats supplémentaires sont définis tel que `isKindOf`, qui a été présenté au chapitre 5. Ces prédicats sont ensuite utilisés pour définir le prédicat `hasErrors`. Ce dernier permet de vérifier qu'un modèle, tel que le métamétamodèle KM3 lui-même, satisfait les contraintes de KM3.

Listing A.2 – Définition de KM3 en Prolog

```
1 /* KM3 metametamodel definition */
```

```
2
3  node(km3,package).
4  prop(km3,name,"KM3").
5  edge(km3,modelElement,contents).
6  edge(modelElement,km3,me_package).
7  edge(km3,package,contents).
8  edge(package,km3,me_package).
9  edge(km3,classifier,contents).
10 edge(classifier,km3,me_package).
11 edge(km3,dataType,contents).
12 edge(dataType,km3,me_package).
13 edge(km3,class,contents).
14 edge(class,km3,me_package).
15 edge(km3,structuralFeature,contents).
16 edge(structuralFeature,km3,me_package).
17 edge(km3,reference,contents).
18 edge(reference,km3,me_package).
19 edge(km3,attribute,contents).
20 edge(attribute,km3,me_package).
21 edge(km3,boolean,contents).
22 edge(boolean,km3,me_package).
23 edge(km3,integer,contents).
24 edge(integer,km3,me_package).
25 edge(km3,string,contents).
26 edge(string,km3,me_package).
27
28   /* class ModelElement */
29   node(modelElement,class).
30   prop(modelElement,name,"ModelElement").
31   prop(modelElement,isAbstract,true).
32
33     /* attribute ModelElement.name : String */
34     node(name,attribute).
35     prop(name,name,"name").
36     prop(name,lower,1).
37     prop(name,upper,1).
38     prop(name,isOrdered,false).
39     prop(name,isUnique,false).
40     edge(modelElement,name,features).
41     edge(name,modelElement,owner).
42     edge(name,string,type).
43
44     /* reference ModelElement.package : Package oppositeOf contents */
45     node(me_package,reference).
46     prop(me_package,name,"package").
47     prop(me_package,isContainer,false).
48     prop(me_package,lower,0).
49     prop(me_package,upper,1).
50     prop(me_package,isOrdered,false).
51     prop(me_package,isUnique,false).
52     edge(modelElement,me_package,features).
53     edge(me_package,modelElement,owner).
54     edge(me_package,package,type).
55     edge(me_package,contents,opposite).
56
57   /* class Package extends ModelElement */
58   node(package,class).
59   prop(package,name,"Package").
60   prop(package,isAbstract,false).
61   edge(package,modelElement,supertypes).
62
63     /* reference Package.contents[*] ordered container : ModelElement oppositeOf package */
64     node(contents,reference).
65     prop(contents,name,"contents").
66     prop(contents,isContainer,true).
```

```
67      prop(contents,lower,0).
68      prop(contents,upper,-1).
69      prop(contents,isOrdered,false).
70      prop(contents,isUnique,false).
71      edge(package,contents,features).
72      edge(contents,package,owner).
73      edge(contents,modelElement,type).
74      edge(contents,me_package,opposite).
75
76    /* class Classifier extends ModelElement */
77    node(classifier,class).
78    prop(classifier,name,"Classifier").
79    prop(classifier,isAbstract,true).
80    edge(classifier,modelElement,supertypes).
81
82    /* class DataType extends Classifier */
83    node(dataType,class).
84    prop(dataType,name,"DataType").
85    prop(dataType,isAbstract,false).
86    edge(dataType,classifier,supertypes).
87
88    /* class Class extends Classifier */
89    node(class,class).
90    prop(class,name,"Class").
91    prop(class,isAbstract,false).
92    edge(class,classifier,supertypes).
93
94      /* attribute Class.isAbstract : Boolean */
95      node(isAbstract,attribute).
96      prop(isAbstract,name,"isAbstract").
97      prop(isAbstract,lower,1).
98      prop(isAbstract,upper,1).
99      prop(isAbstract,isOrdered,false).
100     prop(isAbstract,isUnique,false).
101     edge(class,isAbstract,features).
102     edge(isAbstract,class,owner).
103     edge(isAbstract,boolean,type).
104
105     /* reference Class.features[*] ordered container : StructuralFeature oppositeOf owner */
106     node(features,reference).
107     prop(features,name,"structuralFeatures").
108     prop(features,isContainer,true).
109     prop(features,lower,0).
110     prop(features,upper,-1).
111     prop(features,isOrdered,true).
112     prop(features,isUnique,true).
113     edge(class,features,features).
114     edge(features,class,owner).
115     edge(features,structuralFeature,type).
116     edge(features,owner,opposite).
117
118     /* reference Class.supertypes[*] : Class */
119     node(supertypes,reference).
120     prop(supertypes,name,"supertypes").
121     prop(supertypes,isContainer,false).
122     prop(supertypes,lower,0).
123     prop(supertypes,upper,-1).
124     prop(supertypes,isOrdered,false).
125     prop(supertypes,isUnique,true).
126     edge(class,supertypes,features).
127     edge(supertypes,class,owner).
128     edge(supertypes,class,type).
129
130   /* abstract class StructuralFeature extends ModelElement */
131   node(structuralFeature,class).
```

```
132  prop(structuralFeature,name,"StructuralFeature").
133  prop(structuralFeature,isAbstract,true).
134  edge(structuralFeature,modelElement,supertypes).
135
136    /* attribute StructuralFeature.lower : Integer */
137    node(lower,attribute).
138    prop(lower,name,"lower").
139    prop(lower,lower,1).
140    prop(lower,upper,1).
141    prop(lower,isOrdered,false).
142    prop(lower,isUnique,false).
143    edge(structuralFeature,lower,features).
144    edge(lower,structuralFeature,owner).
145    edge(lower,integer,type).
146
147    /* attribute StructuralFeature.upper : Integer */
148    node(upper,attribute).
149    prop(upper,name,"upper").
150    prop(upper,lower,1).
151    prop(upper,upper,1).
152    prop(upper,isOrdered,false).
153    prop(upper,isUnique,false).
154    edge(structuralFeature,upper,features).
155    edge(upper,structuralFeature,owner).
156    edge(upper,integer,type).
157
158    /* attribute StructuralFeature.isOrdered : Boolean */
159    node(isOrdered,attribute).
160    prop(isOrdered,name,"isOrdered").
161    prop(isOrdered,lower,1).
162    prop(isOrdered,upper,1).
163    prop(isOrdered,isOrdered,false).
164    prop(isOrdered,isUnique,false).
165    edge(structuralFeature,isOrdered,features).
166    edge(isOrdered,structuralFeature,owner).
167    edge(isOrdered,boolean,type).
168
169    /* attribute StructuralFeature.isUnique : Boolean */
170    node(isUnique,attribute).
171    prop(isUnique,name,"isUnique").
172    prop(isUnique,lower,1).
173    prop(isUnique,upper,1).
174    prop(isUnique,isOrdered,false).
175    prop(isUnique,isUnique,false).
176    edge(structuralFeature,isUnique,features).
177    edge(isUnique,structuralFeature,owner).
178    edge(isUnique,boolean,type).
179
180    /* reference StructuralFeature.owner : Class oppositeOf features */
181    node(owner,reference).
182    prop(owner,name,"owner").
183    prop(owner,isContainer,false).
184    prop(owner,lower,1).
185    prop(owner,upper,1).
186    prop(owner,isOrdered,false).
187    prop(owner,isUnique,false).
188    edge(structuralFeature,owner,features).
189    edge(owner,structuralFeature,owner).
190    edge(owner,class,type).
191    edge(owner,features,opposite).
192
193    /* reference StructuralFeature.type : Class */
194    node(type,reference).
195    prop(type,name,"type").
196    prop(type,isContainer,false).
```

```
197     prop(type,lower,1).
198     prop(type,upper,1).
199     prop(type,isOrdered,false).
200     prop(type,isUnique,false).
201     edge(structuralFeature,type,features).
202     edge(type,structuralFeature,owner).
203     edge(type,classifier,type).
204
205   /* class Reference extends StructuralFeature */
206   node(reference,class).
207   prop(reference,name,"Reference").
208   prop(reference,isAbstract,false).
209   edge(reference,structuralFeature,supertypes).
210
211     /* attribute Reference.isContainer : Boolean */
212     node(isContainer,attribute).
213     prop(isContainer,name,"isContainer").
214     prop(isContainer,lower,1).
215     prop(isContainer,upper,1).
216     prop(isContainer,isOrdered,false).
217     prop(isContainer,isUnique,false).
218     edge(reference,isContainer,features).
219     edge(isContainer,reference,owner).
220     edge(isContainer,boolean,type).
221
222     /* reference Reference.opposite : Reference */
223     node(opposite,reference).
224     prop(opposite,name,"opposite").
225     prop(opposite,isContainer,false).
226     prop(opposite,lower,0).
227     prop(opposite,upper,1).
228     prop(opposite,isOrdered,false).
229     prop(opposite,isUnique,false).
230     edge(reference,opposite,features).
231     edge(opposite,reference,owner).
232     edge(opposite,reference,type).
233
234   /* class Attribute extends StructuralFeature */
235   node(attribute,class).
236   prop(attribute,name,"Attribute").
237   prop(attribute,isAbstract,false).
238   edge(attribute,structuralFeature,supertypes).
239
240   /* datatype Boolean */
241   node(boolean,dataType).
242   prop(boolean,name,"Boolean").
243
244   /* datatype Integer */
245   node(integer,dataType).
246   prop(integer,name,"Integer").
247
248   /* datatype String */
249   node(string,dataType).
250   prop(string,name,"String").
251
252
253
254 /*
255   Helper predicates
256 */
257
258 is_boolean(true).
259 is_boolean(false).
260 is_string(X) :- is_list(X), member(C,X), integer(C).
261
```

```
262 structuralFeatures(Class,X) :- edge(Class,X,features).
263 allStructuralFeatures(Class,X) :- structuralFeatures(Class,X).
264 allStructuralFeatures(Class,X) :- supertypes(Class,SuperClass),
265     allStructuralFeatures(SuperClass,X).
266
267 conformsTo(Class1,Class2) :- supertypes(Class1,Class2).
268 conformsTo(Class1,Class2) :- supertypes(Class1,SuperClass), conformsTo(SuperClass,Class2).
269
270 isTypeOf(X,Type) :- node(X,Type).
271
272 isKindOf(X,Type) :- isTypeOf(X,Type).
273 isKindOf(X,Type) :- isTypeOf(X,SomeType), conformsTo(SomeType,Type).
274
275 supertypes(Class,X) :- edge(Class,X,supertypes).
276 allSupertypes(Class,X) :- supertypes(Class,X).
277 allSupertypes(Class,X) :- supertypes(Class,SuperClass), supertypes(SuperClass,X).
278
279 subtypes(Class,X) :- supertypes(X,Class).
280 allSubtypes(Class,X) :- subtypes(Class,X).
281 allSubtypes(Class,X) :- subtypes(Class,SubClass), allSubtypes(SubClass,X).
282
283 oppositeOf(X,Y) :- node(X,reference), node(Y,reference), edge(X,Y,opposite).
284
285 lookupElement(Class,ElementName,Element) :- structuralFeatures(Class,Element),
286     prop(Element,name,ElementName).
287 lookupElementExtended(Class,ElementName,Element) :- allStructuralFeatures(Class,Element),
288     prop(Element,name,ElementName).
289
290 className(Class,Name) :- isTypeOf(Class,class), prop(Class,name,Name).
291 packageName(Package,Name) :- node(Package,package), prop(Package,name,NameAsList),
292     string_to_list(Name,NameAsList).
293
294 /*
295   Some Well-Formedness Rules for KM3-models
296 */
297 hasErrors :- wfr(M,X,Y,Z), writef(M,[X,Y,Z]).
298 hasErrors :- wfr(M,X,Y), writef(M,[X,Y]).
299 hasErrors :- wfr(M,X), writef(M,[X]).
300
301 wfr('The type of node %w should be a class.',X) :-
302     node(X,Y), not(node(Y,class)).
303 wfr('Node %w is defined twice with different types %w and %w.',X,Y,Z) :-
304     node(X,Y), node(X,Z), \=(Y,Z).
305 wfr('The type of edge %w->%w should be a reference.',X,Y) :-
306     edge(X,Y,Z), not(node(Z, reference)).
307 wfr('The opposite of reference %w is reference %w, but the reverse is not true.',X,Y) :-
308     node(X,reference), edge(X,Y,opposite), not(edge(Y,X,opposite)).
309 wfr('The opposite of reference %w is reference %w, but types and owner do not match.',X,Y) :-
310     node(X,reference), edge(X,XOwner,owner), edge(X,Y,opposite), not(edge(Y,XOwner,type)).
311 wfr('The target of edge %w->%w is not a valid node.',X,Y) :-
312     edge(X,Y,_), not(node(Y,_)).
313 wfr('The source of edge %w->%w is not a valid node.',X,Y) :-
314     edge(X,Y,_), not(node(X,_)).
315 wfr('The source of edge %w->%w does not have a correct type.',X,Y) :-
316     edge(X,Y,Reference), edge(Reference,SourceType,owner), not(isKindOf(X,SourceType)).
317 wfr('The target of edge %w->%w does not have a correct type.',X,Y) :-
318     edge(X,Y,Reference), edge(Reference,TargetType,type), not(isKindOf(Y,TargetType)).
319 wfr('Edge %w->%w does not have an opposite with appropriate type.',X,Y) :-
320     edge(X,Y,Z), oppositeOf(Z,O), not(edge(Y,X,O)).
321 wfr('%w is an instance of abstract type %w, which is forbidden.',X,Y) :-
322     node(X,Y), prop(Y,isAbstract,true).
323 wfr('Attribute %w is not valid for %w.',X,Y) :-
324     prop(Y,X,_), isTypeOf(Y,YType), not(allStructuralFeatures(YType,X)).
325 wfr('Property %w of %w should be of type Integer.',X,Y) :-
326     prop(Y,X,V), edge(X,T,type), node(T,dataType), prop(T,name,"Integer"), not(integer(V)).
```

```
327 wfr('Property %w of %w should be of type String.',X,Y) :-
328     prop(Y,X,V), edge(X,T,type), node(T,dataType), prop(T,name,"String"), not(is_string(V)).
329 wfr('Property %w of %w should be of type Boolean.',X,Y) :-
330     prop(Y,X,V), edge(X,T,type), node(T,dataType), prop(T,name,"Boolean"),
331     not(is_boolean(V)).
332 wfr('Classifier %w should be in a package.',X) :-
333     isKindOf(X,classifier), not(edge(X,_,me_package)).
334 wfr('Element %w should have a value for attribute %w.',X,Y) :-
335     isTypeOf(X,XType), allStructuralFeatures(XType,Y), node(Y,attribute),
336     not(prop(Y,lower,0)), not(prop(X,Y,_)).
337 wfr('Element %w should have a value for reference %w.',X,Y) :-
338     isTypeOf(X,XType), allStructuralFeatures(XType,Y), node(Y,reference),
339     not(prop(Y,lower,0)), not(edge(X,_,Y)).
340 wfr('Element %w is contained in both %w and %w.',X,Y,Z) :-
341     edge(X,Y,F1), edge(X,Z,F2), oppositeOf(F1,OF1), oppositeOf(F2,OF2),
342     prop(OF1,isContainer,true), prop(OF2,isContainer,true), \=(F1,F2).
```

ANNEXE B

Le cas d'étude SPL et CPL

Le cas d'étude concernant les langages de téléphonie par internet SPL et CPL a été présenté à la section 8.4. Ce chapitre donne les listings complets des différents éléments qui y ont été décrits : le métamodèle de SPL en KM3 à la section B.1, la syntaxe concrète de SPL en TCS à la section B.2, le métamodèle de CPL en KM3 à la section B.3, la transformation XML vers CPL en ATL à la section B.4 et la transformation CPL vers SPL en ATL à la section B.5. Ces différents modèles sont aussi disponibles dans la bibliothèque de transformations du projet AM3 [26]. Ils peuvent être téléchargés et exécutés sur un ensemble de tests.

B.1 Métamodèle de SPL en KM3

Le listing B.1 donne la définition du métamodèle de SPL [15] en KM3. Ce métamodèle a été créé à partir des informations disponibles sur le site web de SPL [28] et notamment de la grammaire de SPL.

Listing B.1 – Métamodèle SPL en KM3

```
1  package SPL {
2
3    abstract class LocatedElement {
4      attribute location : String;
5      attribute commentsBefore[*] ordered : String;
6      attribute commentsAfter[*] ordered : String;
7    }
8
9    class Program extends LocatedElement {
10     reference service container : Service;
11   }
12
13   class Service extends LocatedElement {
14     attribute name : String;
15     reference declarations[*] ordered container : Declaration;
16     reference sessions[*] ordered container : Session;
17   }
18
19 -- @begin Sessions
20   abstract class Session extends LocatedElement {
21   }
22
23   class Registration extends Session {
24     reference declarations[*] ordered container : Declaration;
25     reference sessions[*] ordered container : Session;
26   }
27
28   class Dialog extends Session {
29     reference declarations[*] ordered container : Declaration;
```

```
30      reference methods[1-*] ordered container : Method;
31    }
32
33    class Event extends Session {
34      attribute eventId : String;
35      reference declarations[*] ordered container : Declaration;
36      reference methods[1-*] ordered container : Method;
37    }
38
39  -- @begin Methods
40    class Method extends Session {
41      reference type container : TypeExpression;
42      attribute direction : Direction;
43      reference methodName container : MethodName;
44      reference arguments[*] ordered container : Argument;
45      reference statements[1-*] ordered container : Statement;
46      reference branches[1-*] ordered container : Branch;
47    }
48
49    class Argument extends VariableDeclaration {
50    }
51
52  -- @begin Method Names
53    abstract class MethodName extends LocatedElement {
54    }
55
56    class SIPMethodName extends MethodName {
57      attribute name : SIPMethod;
58    }
59
60    class ControlMethodName extends MethodName {
61      attribute name : ControlMethod;
62    }
63  -- @end Method Names
64  -- @end Methods
65  -- @end Sessions
66
67  -- @begin Branches
68    class Branch extends LocatedElement {
69      reference statements[1-*] ordered container : Statement;
70    }
71
72    class DefaultBranch extends Branch {
73    }
74
75    class NamedBranch extends Branch {
76      attribute name[1-*] ordered : String;
77    }
78  -- @end Branches
79
80
81  -- @begin Types
82    abstract class TypeExpression extends LocatedElement {
83    }
84
85    class SimpleType extends TypeExpression {
86      attribute type : PrimitiveType;
87    }
88
89    class SequenceType extends TypeExpression {
90      attribute modifier[0-1] : Modifier;
91      attribute type : PrimitiveType;
92      attribute size[0-1] : Integer;
93    }
94
```

```
95    class DefinedType extends TypeExpression {
96      attribute typeName : String;
97    }
98  -- @end Types
99
100 -- @begin Declarations
101   abstract class Declaration extends LocatedElement {
102     attribute name : String;
103   }
104
105   class VariableDeclaration extends Declaration {
106     reference type container : TypeExpression;
107     reference initExp[0-1] container : Expression;
108   }
109
110 -- @begin FunctionDeclarations
111   abstract class FunctionDeclaration extends Declaration {
112     reference returnType container : TypeExpression;
113     reference arguments[*] ordered container : Argument;
114   }
115
116   class RemoteFunctionDeclaration extends FunctionDeclaration {
117     attribute functionLocation : FunctionLocation;
118   }
119
120   class LocalFunctionDeclaration extends FunctionDeclaration {
121     reference statements[1-*] ordered container : Statement;
122   }
123 -- @end FunctionDeclarations
124
125 -- @begin StructureDeclarations
126   class StructureDeclaration extends Declaration {
127     reference properties[1-*] ordered container : Argument;
128   }
129
130   class StructureProperty extends LocatedElement {
131     attribute name : String;
132     reference type container : TypeExpression;
133   }
134 -- @end StructureDeclarations
135 -- @end Declarations
136
137   class FunctionCall extends LocatedElement {
138     reference function : FunctionDeclaration;
139     reference parameters[*] ordered container : Expression;
140   }
141
142 -- @begin Statements
143   abstract class Statement extends LocatedElement {
144   }
145
146   class CompoundStat extends Statement {
147     reference statements[1-*] container : Statement;
148   }
149
150   class SetStat extends Statement {
151     reference target container : Place;
152     reference setValue container : Expression;
153   }
154
155   class DeclarationStat extends Statement {
156     reference declaration container : Declaration;
157   }
158
159   class ReturnStat extends Statement {
```

```
160     reference returnedValue[0-1] container : Expression;
161     reference branch[0-1] : NamedBranch;
162   }
163
164   class IfStat extends Statement {
165     reference condition container : Expression;
166     reference thenStatements[1-*] ordered container : Statement;
167     reference elseStatements[*] ordered container : Statement;
168   }
169
170   class WhenStat extends Statement {
171     reference idExp container : Variable;
172     reference whenHeaders[1-*] ordered container : WhenHeader;
173     reference statements[1-*] ordered container : Statement;
174     reference elseStatements[0-*] ordered container : Statement;
175   }
176
177   class ForeachStat extends Statement {
178     attribute iteratorName : String;
179     reference sequenceExp container : Expression;
180     reference statements[1-*] ordered container : Statement;
181   }
182
183   class SelectStat extends Statement {
184     reference matchedExp container : Expression;
185     reference selectCases[*] ordered container : SelectCase;
186     reference selectDefault[0-1] container : SelectDefault;
187   }
188
189   class FunctionCallStat extends Statement {
190     reference functionCall container : FunctionCall;
191   }
192
193   class ContinueStat extends Statement {
194   }
195
196   class BreakStat extends Statement {
197   }
198
199   class PushStat extends Statement {
200     reference target container : Place;
201     reference pushedValue container : Expression;
202   }
203 -- @end Statements
204
205   class WhenHeader extends VariableDeclaration {
206     attribute headerId : String;
207     reference value[0-1] container : Constant;
208   }
209
210   abstract class SelectMember extends LocatedElement {
211     reference statements[*] ordered container : Statement;
212   }
213
214   class SelectDefault extends SelectMember {
215   }
216
217   class SelectCase extends SelectMember {
218     reference values[1-*] ordered container : Constant;
219   }
220
221 -- @begin Expressions
222   abstract class Expression extends LocatedElement {
223   }
224
```

```
225   class ConstantExp extends Expression {
226     reference value container : Constant;
227   }
228
229   class OperatorExp extends Expression {
230     attribute opName : String;
231     reference leftExp container : Expression;
232     reference rightExp[0-1] container : Expression;
233   }
234
235   class ForwardExp extends Expression {
236     attribute isParallel : Boolean;
237     reference exp[0-1] container : Expression;
238   }
239
240   class WithExp extends Expression {
241     reference exp container : Expression;
242     reference msgFields[1-*] ordered container : MessageField;
243   }
244
245   class BlockExp extends Expression {
246     reference exp container : Expression;
247   }
248
249   class ReasonExp extends Expression {
250   }
251
252   class BODYExp extends Expression {
253   }
254
255   class RequestURIExp extends Expression {
256   }
257
258   class PopExp extends Expression {
259     reference source container : Place;
260   }
261
262   class FunctionCallExp extends Expression {
263     reference functionCall container : FunctionCall;
264   }
265
266 -- @begin Places
267   abstract class Place extends Expression {
268   }
269
270   class SIPHeaderPlace extends Place {
271     attribute header : SIPHeader;
272   }
273
274   abstract class VariablePlace extends Place {
275   }
276
277   class PropertyCallPlace extends VariablePlace {
278     attribute propName : String;
279     reference source container : VariablePlace;
280   }
281
282   class Variable extends VariablePlace {
283     reference source : Declaration;
284   }
285 -- @end Places
286
287   abstract class MessageField extends LocatedElement {
288     reference exp container : Expression;
289   }
```

```
290
291   class ReasonMessageField extends MessageField {
292   }
293
294   class HeadedMessageField extends MessageField {
295     attribute headerId : String;
296   }
297 -- @end Expressions
298
299 -- @begin Constants
300   abstract class Constant extends LocatedElement {
301   }
302
303   class BooleanConstant extends Constant {
304     attribute value : Boolean;
305   }
306
307   class IntegerConstant extends Constant {
308     attribute value : Integer;
309   }
310
311   class StringConstant extends Constant {
312     attribute value : String;
313   }
314
315   class URIConstant extends Constant {
316     attribute uri : String;
317   }
318
319   class SequenceConstant extends Constant {
320     reference values[*] ordered container : Constant;
321   }
322
323   class ResponseConstant extends Constant {
324     reference response container : Response;
325   }
326 -- @end Constants
327
328 -- @begin Responses
329   abstract class Response extends LocatedElement {
330   }
331
332   class SuccessResponse extends Response {
333     attribute successKind : SuccessKind;
334   }
335
336   class ErrorResponse extends Response {
337   }
338
339   class ClientErrorResponse extends ErrorResponse {
340     attribute errorKind[0-1] : ClientErrorKind;
341   }
342
343   class GlobalErrorResponse extends ErrorResponse {
344     attribute errorKind[0-1] : GlobalErrorKind;
345   }
346
347   class RedirectionErrorResponse extends ErrorResponse {
348     attribute errorKind[0-1] : RedirectionErrorKind;
349   }
350
351   class ServerErrorResponse extends ErrorResponse {
352     attribute errorKind[0-1] : ServerErrorKind;
353   }
354 -- @end Responses
```

```
355 }
356
357
358 package Enum {
359
360   enumeration Direction {
361     literal inout;
362     literal in;
363     literal out;
364   }
365
366   enumeration SIPMethod {
367     literal ACK;
368     literal BYE;
369     literal CANCEL;
370     literal INVITE;
371     literal NOTIFY;
372     literal OPTIONS;
373     literal REACK;
374     literal REGISTER;
375     literal REINVITE;
376     literal REREGISTER;
377     literal RESUBSCRIBE;
378     literal SUBSCRIBE;
379   }
380
381   enumeration ControlMethod {
382     literal deploy;
383     literal undeploy;
384     literal uninvite;
385     literal unregister;
386     literal unsubscribe;
387   }
388
389   enumeration PrimitiveType {
390     literal void;
391     literal bool;
392     literal int;
393     literal request;
394     literal response;
395     literal string;
396     literal time;
397     literal uri;
398   }
399
400   enumeration Modifier {
401     literal LIFO;
402     literal FIFO;
403   }
404
405   enumeration FunctionLocation {
406     literal remote;
407     literal local;
408   }
409
410   enumeration SIPHeader {
411     literal CALL_ID;
412     literal CONTACT;
413     literal CSEQ;
414     literal EVENT;
415     literal FROM;
416     literal MAX_FORWARDS;
417     literal SUBSCRIPTION_STATE;
418     literal TO;
419     literal VIA;
```

```
420   }
421
422   enumeration SuccessKind {
423     literal OK;
424     literal ACCEPTED;
425   }
426
427   enumeration ClientErrorKind {
428     literal ADDRESS_INCOMPLETE;
429     literal AMBIGUOUS;
430     literal BAD_EXTENSION;
431     literal BAD_REQUEST;
432     literal BUSY_HERE;
433     literal CALL_OR_TRANSACTION_DOES_NOT_EXIST;
434     literal EXTENSION_REQUIRED;
435     literal FORBIDDEN;
436     literal GONE;
437     literal INTERVAL_TOO_BRIEF;
438     literal LOOP_DETECTED;
439     literal METHOD_NOT_ALLOWED;
440     literal NOT_ACCEPTABLE_HERE;
441     literal NOT_ACCEPTABLE;
442     literal NOT_FOUND;
443     literal PAYMENT_REQUIRED;
444     literal PROXY_AUTHENTICATION_REQUIRED;
445     literal REQUESTURI_TOO_LONG;
446     literal REQUEST_ENTITY_TOO_LARGE;
447     literal REQUEST_PENDING;
448     literal REQUEST_TERMINATED;
449     literal REQUEST_TIMEOUT;
450     literal TEMPORARILY_UNAVAILABLE;
451     literal TOO_MANY_HOPS;
452     literal UNAUTHORIZED;
453     literal UNDECIPHERABLE;
454     literal UNSUPPORTED_MEDIA_TYPE;
455     literal UNSUPPORTED_URI_SCHEME;
456   }
457
458   enumeration GlobalErrorKind {
459     literal BUSY_EVERYWHERE;
460     literal DECLINE;
461     literal DOES_NOT_EXIST_ANYWHERE;
462     literal NOT_ACCEPTABLE;
463   }
464
465   enumeration RedirectionErrorKind {
466     literal ALTERNATIVE_SERVICE;
467     literal MOVED_PERMANENTLY;
468     literal MOVED_TEMPORARILY;
469     literal MULTIPLE_CHOICES;
470     literal USE_PROXY;
471   }
472
473   enumeration ServerErrorKind {
474     literal BAD_GATEWAY;
475     literal MESSAGE_TOO_LARGE;
476     literal NOT_IMPLEMENTED;
477     literal SERVER_INTERNAL_ERROR;
478     literal SERVER_TIMEOUT;
479     literal SERVICE_UNAVAILABLE;
480     literal VERSION_NOT_SUPPORTED;
481   }
482 }
483
484 package PrimitiveTypes {
```

```
485   datatype String;
486   datatype Integer;
487   datatype Boolean;
488 }
```

B.2 Syntaxe concrète de SPL en TCS

Le listing B.2 donne la définition de la syntaxe concrète de SPL [15] en KM3. Ce modèle a été créé à partir des informations disponibles sur le site web de SPL [28] et notamment de la grammaire de SPL. En utilisant la transformation *TCS2ANTLR.atl*, présentée au chapitre 6, sur le métamodèle (voir section B.1) et le modèle TCS de SPL, on obtient une grammaire de SPL définie en ANTLR. Certaines constructions avancées du langage telles que les séquences ne sont pas correctement gérées par la version présentée ici. Ces simplifications ont été faites afin de pouvoir utiliser ANTLRv2 mais pourront probablement être levées avec ANTLRv3 (voir la discussion sur les classes de grammaires à la section 6.9).

Listing B.2 – Modèle TCS de SPL

```
1  syntax SPL(k = 4) {
2
3    primitiveTemplate identifier for String default using NAME:
4      value = "%token%";
5
6    primitiveTemplate stringSymbol for String using STRING:
7      value = "%token%";
8
9    primitiveTemplate uriSymbol for String using URI:
10     value = "%token%";
11
12   primitiveTemplate headerIdSymbol for String using HEADERID:
13     value = "%token%";
14
15   primitiveTemplate integerSymbol for Integer default using INT:
16     value = "Integer.valueOf(%token%)";
17
18   primitiveTemplate booleanSymbol for Boolean default using BOOLEAN:
19     value = "Boolean.valueOf(%token%)";
20
21   template Program main
22     : service
23     ;
24
25   template Service context
26     : "service" name "{" [
27         [ "processing" "{" ] {indentIncr = 0, startNL = false, endNL = false, nbNL = 0} [
28           declarations
29           sessions
30         ] {nbNL = 2} "}"
31       ] "}"
32     ;
33
34 -- @begin Sessions
35   template Session abstract;
36
37   template Registration context
38     : "registration" "{" [
39         sessions
40       ] {nbNL = 2} "}"
41     ;
42
43   template Dialog context
```

```
44      : "dialog" "{" [
45          declarations
46          methods
47        ] {nbNL = 2} "}"
48      ;
49
50  -- @begin Methods
51    template Method context
52      : $methodHeader
53        "{" [
54          (isDefined(branches) ?
55              branches
56            :
57              statements
58          )
59        ] "}"
60      ;
61
62    function methodHeader(Method)
63      : type
64        (direction = #in ?
65            "incoming"
66          :
67            (direction = #out ?
68                "outgoing"
69              :
70                -- inout
71            )
72        )
73        methodName
74        "(" arguments{separator = ","} ")"
75      ;
76
77    template Argument addToContext
78      : type name
79      ;
80
81  -- @begin Method Names
82    template MethodName abstract;
83
84    template SIPMethodName
85      : name
86      ;
87
88    template ControlMethodName
89      : name
90      ;
91  -- @end Method Names
92  -- @end Methods
93  -- @end Sessions
94
95
96  -- @begin Branches
97    template Branch abstract;
98
99    template DefaultBranch
100     : "branch" "default"
101       "{" statements "}"
102     ;
103
104   template NamedBranch
105     : "branch" name{separator = "|"}
106       "{" statements "}"
107     ;
108 -- @end Branches
```

```
109
110 -- @begin Declarations
111   template Declaration abstract;
112
113   template VariableDeclaration addToContext
114     : type name (isDefined(initExp) ? "=" initExp) ";"
115     ;
116
117 -- @begin FunctionDeclarations
118   template FunctionDeclaration abstract;
119
120   template RemoteFunctionDeclaration addToContext
121     : functionLocation returnType name "(" arguments{separator = ","} ")" ";"
122     ;
123
124   template LocalFunctionDeclaration addToContext
125     : returnType name "(" arguments{separator = ","} ")" "{" [
126         statements
127       ] "}"
128     ;
129 -- @end FunctionDeclarations
130
131 -- @begin StructureDeclarations
132   template StructureDeclaration
133     : "type" name "{" [
134         properties
135       ] "}"
136     ;
137
138   template StructureProperty
139     : type name ";"
140     ;
141 -- @end StructureDeclarations
142 -- @end Declarations
143
144 -- @begin Statements
145   template Statement abstract;
146
147   template CompoundStat
148     : "{" [ statements ] "}"
149     ;
150
151   template SetStat
152     : target "=" setValue ";"
153     ;
154
155   template DeclarationStat
156     : declaration
157     ;
158
159   template ReturnStat
160     : "return"
161       (isDefined(returnedValue) ? returnedValue)
162       (isDefined(branch) ? "branch" branch{refersTo = name, autoCreate = ifmissing})
163       ";"
164     ;
165
166   template IfStat
167     : "if" "(" condition ")"
168       (one(thenStatements) ?
169           [ thenStatements ]
170         :
171           "{" [
172             thenStatements
173           ] "}"
```

```
174       )
175       (isDefined(elseStatements) ?
176         "else"
177         (one(elseStatements) ?
178             [ elseStatements ]
179           :
180             "{" [
181               elseStatements
182             ] "}"
183         )
184       )
185     ;
186
187   template WhenStat context
188     : "when" idExp "(" whenHeaders{separator = ","} ")" "{" [
189         statements
190       ] "}"
191       (isDefined(elseStatements) ?
192         "else"
193         (one(elseStatements) ?
194             [ elseStatements ]
195           :
196             "{" [
197               elseStatements
198             ] "}"
199         )
200       )
201     ;
202
203   template WhenHeader addToContext
204     : headerId{as = headerIdSymbol} type name (isDefined(value) ? value)
205     ;
206
207   template ForeachStat
208     : "foreach" "(" iteratorName "in" sequenceExp ")" "{" [
209         statements
210       ] "}"
211     ;
212
213   template SelectStat
214     : "select" "(" matchedExp ")" "{" [
215         selectCases
216         (isDefined(selectDefault) ?
217           selectDefault
218         )
219       ] "}"
220     ;
221
222   template SelectCase
223     : "case" values{separator = "|"} ":" [ statements ]
224     ;
225
226   template SelectDefault
227     : "default" ":" [ statements ]
228     ;
229
230   template FunctionCallStat
231     : functionCall ";"
232     ;
233
234   template ContinueStat
235     : "continue" ";"
236     ;
237
238   template BreakStat
```

```
239     : "break" ";"
240     ;
241
242   template PushStat
243     : "push" target pushedValue ";"
244     ;
245 -- @end Statements
246
247 -- @begin Expressions
248   template Expression abstract operatored;
249
250   template ConstantExp
251     : value
252     ;
253
254   operatorTemplate OperatorExp(operators =
255         opNot opMinus1
256         opStar opSlash
257         opPlus opMinus2
258         opEq opGt opLt opGe opLe opNe
259         opAnd opOr
260         opMatch opNoMatch
261       , source = leftExp, storeOpTo = opName, storeRightTo = rightExp);
262
263   template ForwardExp nonPrimary
264     : (isParallel ? "parallel") "forward" (isDefined(exp) ? exp)
265     ;
266
267   operatorTemplate WithExp(operators = opWith, source = exp)
268     : "{" msgFields{separator = ","} "}"
269     ;
270
271   template MessageField abstract;
272
273   template ReasonMessageField
274     : "reason" "=" exp
275     ;
276
277   template HeadedMessageField
278     : headerId{as = headerIdSymbol} exp
279     ;
280
281   template ReasonExp
282     : "reason"
283     ;
284
285   template BODYExp
286     : "BODY"
287     ;
288
289   template RequestURIExp
290     : "requestURI"
291     ;
292
293   template PopExp
294     : "pop" source
295     ;
296
297   template FunctionCallExp
298     : functionCall
299     ;
300
301   template FunctionCall
302     : function{refersTo = name} "(" parameters{separator = ","} ")"
303     ;
```

```
304
305   template Place abstract;
306
307   template SIPHeaderPlace
308     : header
309     ;
310
311   template VariablePlace abstract;
312
313   template Variable
314     : source{refersTo = name}
315     ;
316
317 -- @end Expressions
318
319 -- @begin Constants
320   template Constant abstract;
321
322   template BooleanConstant
323     : value
324     ;
325
326   template IntegerConstant
327     : value
328     ;
329
330   template StringConstant
331     : value{as = stringSymbol}
332     ;
333
334   template URIConstant
335     : uri{as = uriSymbol}
336     ;
337
338   template SequenceConstant
339     : "<" values{separator = ","} ">"
340     ;
341
342   template ResponseConstant
343     : response
344     ;
345 -- @end Constants
346
347 -- @begin Responses
348   template Response abstract;
349
350   template SuccessResponse abstract
351     : "/" <no_space> "SUCCESS"
352       (isDefined(successKind) ? <no_space> "/" <no_space> successKind)
353     ;
354
355   template ErrorResponse abstract
356     : "/" <no_space> "ERROR"
357     ;
358
359   template ClientErrorResponse
360     : "/" <no_space> "ERROR" <no_space> "/" <no_space> "CLIENT"
361       (isDefined(errorKind) ? <no_space> "/" <no_space> errorKind)
362     ;
363
364   template GlobalErrorResponse
365     : "/" "<no_space> ERROR" <no_space> "/" <no_space> "GLOBAL"
366       (isDefined(errorKind) ? <no_space> "/" <no_space> errorKind)
367     ;
368
```

```
369   template RedirectionErrorResponse
370     : "/" <no_space> "ERROR" <no_space> "/" <no_space> "REDIRECTION"
371       (isDefined(errorKind) ? <no_space> "/" <no_space> errorKind)
372     ;
373
374   template ServerErrorResponse
375     : "/" <no_space> "ERROR" <no_space> "/" <no_space> "SERVER"
376       (isDefined(errorKind) ? <no_space> "/" <no_space> errorKind)
377     ;
378 -- @end Responses
379
380
381 -- @begin Types
382   template TypeExpression abstract;
383
384   template SimpleType
385     : type
386     ;
387
388   enumerationTemplate Modifier auto
389     : #LIFO   = "LIFO",
390       #FIFO   = "FIFO"
391     ;
392 -- @end Types
393
394   enumerationTemplate SIPMethod auto
395     : #ACK    = "ACK",
396       #BYE    = "BYE",
397       #CANCEL   = "CANCEL",
398       #INVITE   = "INVITE",
399       #NOTIFY   = "NOTIFY",
400       #OPTIONS  = "OPTIONS",
401       #REACK    = "REACK",
402       #REGISTER = "REGISTER",
403       #REINVITE = "REINVITE",
404       #RESUBSCRIBE  = "RESUBSCRIBE",
405       #SUBSCRIBE  = "SUBSCRIBE"
406     ;
407
408   enumerationTemplate ControlMethod auto
409     : #deploy   = "deploy",
410       #undeploy = "undeploy",
411       #uninvite = "uninvite",
412       #unregister = "unregister",
413       #unsubscribe  = "unsubscribe"
414     ;
415
416   enumerationTemplate FunctionLocation auto
417     : #remote   = "remote",
418       #local    = "local"
419     ;
420
421   enumerationTemplate SuccessKind auto
422     : #OK   = "OK",
423       #ACCEPTED = "ACCEPTED"
424     ;
425
426   enumerationTemplate ClientErrorKind auto
427     : #ADDRESS_INCOMPLETE     = "ADDRESS_INCOMPLETE",
428       #AMBIGUOUS        = "AMBIGUOUS",
429       #BAD_EXTENSION        = "BAD_EXTENSION",
430       #BAD_REQUEST        = "BAD_REQUEST",
431       #BUSY_HERE        = "BUSY_HERE",
432       #CALL_OR_TRANSACTION_DOES_NOT_EXIST = "CALL_OR_TRANSACTION_DOES_NOT_EXIST",
433       #EXTENSION_REQUIRED     = "EXTENSION_REQUIRED",
```

```
434       #FORBIDDEN        = "FORBIDDEN",
435       #GONE         = "GONE",
436       #INTERVAL_TOO_BRIEF     = "INTERVAL_TOO_BRIEF",
437       #LOOP_DETECTED        = "LOOP_DETECTED",
438       #METHOD_NOT_ALLOWED     = "METHOD_NOT_ALLOWED",
439       #NOT_ACCEPTABLE_HERE      = "NOT_ACCEPTABLE_HERE",
440       #NOT_ACCEPTABLE       = "NOT_ACCEPTABLE",
441       #NOT_FOUND        = "NOT_FOUND",
442       #PAYMENT_REQUIRED     = "PAYMENT_REQUIRED",
443       #PROXY_AUTHENTICATION_REQUIRED    = "PROXY_AUTHENTICATION_REQUIRED",
444       #REQUESTURI_TOO_LONG      = "REQUESTURI_TOO_LONG",
445       #REQUEST_ENTITY_TOO_LARGE   = "REQUEST_ENTITY_TOO_LARGE",
446       #REQUEST_PENDING      = "REQUEST_PENDING",
447       #REQUEST_TERMINATED     = "REQUEST_TERMINATED",
448       #REQUEST_TIMEOUT      = "REQUEST_TIMEOUT",
449       #TEMPORARILY_UNAVAILABLE    = "TEMPORARILY_UNAVAILABLE",
450       #TOO_MANY_HOPS        = "TOO_MANY_HOPS",
451       #UNAUTHORIZED       = "UNAUTHORIZED",
452       #UNDECIPHERABLE       = "UNDECIPHERABLE",
453       #UNSUPPORTED_MEDIA_TYPE     = "UNSUPPORTED_MEDIA_TYPE",
454       #UNSUPPORTED_URI_SCHEME     = "UNSUPPORTED_URI_SCHEME"
455     ;
456
457   enumerationTemplate GlobalErrorKind auto
458     : #BUSY_EVERYWHERE    = "BUSY_EVERYWHERE",
459       #DECLINE      = "DECLINE",
460       #DOES_NOT_EXIST_ANYWHERE  = "DOES_NOT_EXIST_ANYWHERE",
461       #NOT_ACCEPTABLE     = "NOT_ACCEPTABLE"
462     ;
463
464   enumerationTemplate RedirectionErrorKind auto
465     : #ALTERNATIVE_SERVICE  = "ALTERNATIVE_SERVICE",
466       #MOVED_PERMANENTLY  = "MOVED_PERMANENTLY",
467       #MOVED_TEMPORARILY  = "MOVED_TEMPORARILY",
468       #MULTIPLE_CHOICES = "MULTIPLE_CHOICES",
469       #USE_PROXY    = "USE_PROXY"
470     ;
471
472   enumerationTemplate ServerErrorKind auto
473     : #BAD_GATEWAY    = "BAD_GATEWAY",
474       #MESSAGE_TOO_LARGE  = "MESSAGE_TOO_LARGE",
475       #NOT_IMPLEMENTED  = "NOT_IMPLEMENTED",
476       #SERVER_INTERNAL_ERROR  = "SERVER_INTERNAL_ERROR",
477       #SERVER_TIMEOUT   = "SERVER_TIMEOUT",
478       #SERVICE_UNAVAILABLE  = "SERVICE_UNAVAILABLE",
479       #VERSION_NOT_SUPPORTED  = "VERSION_NOT_SUPPORTED"
480     ;
481
482   enumerationTemplate SIPHeader auto
483     : #CALL_ID    = "CALL_ID",
484       #CONTACT    = "CONTACT",
485       #CSEQ     = "CSEQ",
486       #EVENT      = "EVENT",
487       #FROM     = "FROM",
488       #MAX_FORWARDS   = "MAX_FORWARDS",
489       #SUBSCRIPTION_STATE = "SUBSCRIPTION_STATE",
490       #TO     = "TO",
491       #VIA      = "VIA"
492     ;
493
494   enumerationTemplate PrimitiveType auto
495     : #void   = "void",
496       #bool   = "bool",
497       #int    = "int",
498       #request  = "request",
```

```
499       #response = "response",
500       #string   = "string",
501       #time     = "time",
502       #uri      = "uri"
503     ;
504
505   symbols {
506     lsquare   = "[";
507     rsquare   = "]" : rightSpace;
508     excl    = "!";
509     coma    = "," : leftNone, rightSpace;
510     lparen    = "(";
511     rparen    = ")" : leftNone, rightSpace;
512     lcurly    = "{" : leftSpace;
513     rcurly    = "}" : leftNone, rightSpace;
514     semi    = ";" : leftNone, rightSpace;
515     colon   = ":" : leftNone, rightSpace;
516     colons    = "::";
517     pipe    = "|" : leftSpace, rightSpace;
518     sharp   = "#" : leftSpace;
519     qmark   = "?";
520
521     -- operator symbols
522     point   = "." : leftNone;
523     rarrow    = "->"  : leftNone;
524     minus   = "-" : leftSpace, rightSpace;
525     star    = "*" : leftSpace, rightSpace;
526     slash   = "/" : leftSpace, rightSpace;
527     plus    = "+" : leftSpace, rightSpace;
528     eq    = "=" : leftSpace, rightSpace;
529     eqeq    = "==";
530     gt    = ">" : leftSpace, rightSpace;
531     lt    = "<" : leftSpace, rightSpace;
532     ge    = ">="  : leftSpace, rightSpace;
533     le    = "<="  : leftSpace, rightSpace;
534     ne    = "<>"  : leftSpace, rightSpace;
535     larrow    = "<-"  : leftSpace, rightSpace;
536     ampamp    = "&&";
537     pipepipe  = "||";
538   }
539
540   operators {
541     priority 0 {  -- 0 is highest
542       opWith = "with", 2;
543     }
544
545     priority 1 {
546       opNot = excl, 1;
547       opMinus1 = minus, 1;
548     }
549
550     priority 2 {
551       opStar = star, 2;
552       opSlash = slash, 2;
553     }
554
555     priority 3 {
556       opPlus = plus, 2;
557       opMinus2 = minus, 2;
558     }
559
560     priority 4 {
561       opEq = eqeq, 2;
562       opGt = gt, 2;
563       opLt = lt, 2;
```

```
564       opGe = ge, 2;
565       opLe = le, 2;
566       opNe = ne, 2;
567     }
568
569     priority 5 {
570       opAnd = ampamp, 2;
571       opOr = pipepipe, 2;
572     }
573
574     priority 6 {
575       opMatch = "match", 2;
576       opNoMatch = "nomatch", 2;
577     }
578   }
579
580   token NAME  : word(
581           start = [alpha] | "_",
582           part = [alnum] | "_"
583           , words = ("true" : BOOLEAN, "false" : BOOLEAN)
584         );
585
586   token COMMENT : endOfLine(start = "//") | multiLine(start = "/*", end = "*/");
587
588   token URI : multiLine(start = "\'", end = "\'");
589
590   token HEADERID  : word(
591           start = "#",
592           part = [alnum] | "_" | "!" | "%" | "*" | "-" | "+" | "'" | "`" | "~",
593           end = ":"
594         );
595
596   lexer = "
597 class SPLLexer extends Lexer;
598
599 options {
600   k = 2;
601   charVocabulary = '\\0' .. '\\u00FF';
602 }
603
604 {
605
606   protected Token makeToken(int t) {
607     org.atl.engine.injectors.ebnf.LocationToken ret = null;
608
609     tokenObjectClass = org.atl.engine.injectors.ebnf.LocationToken.class;
610     ret = (org.atl.engine.injectors.ebnf.LocationToken)super.makeToken(t);
611     ret.setEndLine(getLine());
612     ret.setEndColumn(getColumn());
613
614     return ret;
615   }
616
617 }
618
619 NL
620   : ( '\\r' '\\n'
621     | '\\n' '\\r' //Improbable
622     | '\\r'
623     | '\\n'
624     )
625   {newline();}
626   ;
627
628 WS
```

```
629   : ( ' '
630     | '\\t'
631     )
632   ;
633 protected
634 DIGIT
635   : '0'..'9'
636   ;
637
638 INT
639   : (DIGIT)+
640   ;
641
642 protected
643 ESC
644   : '\\\\'!
645     ( 'n' {$setText(\"\\n\");}
646     | 'r' {$setText(\"\\r\");}
647     | 't' {$setText(\"\\t\");}
648     | 'b' {$setText(\"\\b\");}
649     | 'f' {$setText(\"\\f\");}
650     | '\"' {$setText(\"\\\"\");}
651     | '\\'' {$setText(\"\\'\");}
652     | '\\\\' {$setText(\"\\\\\");}
653     | (
654         ('0'..'3')
655         (
656           options {
657             warnWhenFollowAmbig = false;
658           }
659         : ('0'..'7')
660           (
661             options {
662               warnWhenFollowAmbig = false;
663             }
664           : '0'..'7'
665           )?
666         )?
667       | ('4'..'7')
668         (
669           options {
670             warnWhenFollowAmbig = false;
671           }
672         : ('0'..'7')
673         )?
674       )
675         {
676           String s = $getText;
677           int i;
678           int ret = 0;
679           String ans;
680           for (i=0; i<s.length(); ++i)
681             ret = ret*8 + s.charAt(i) - '0';
682           ans = String.valueOf((char) ret);
683           $setText(ans);
684         }
685     )
686   ;
687
688 STRING
689   : '\\"'!
690     ( ESC
691     | '\\n' {newline();}
692     | ~('\\\\'|'\\'|'\\n')
693     )*
```

```
694       '\\"'!
695     ;
696     ";
697
698 }
```

B.3 Métamodèle de CPL en KM3

Le listing B.3 donne la définition du métamodèle de CPL en KM3. Il a été dérivé à partir de la présentation du schéma XML du langage CPL donnée dans [48].

Listing B.3 – Métamodèle CPL en KM3

```
1  package CPL {
2
3    abstract class Element {
4    }
5
6    class CPL extends Element {
7      reference subActions[*] ordered container : SubAction;
8      reference outgoing[0-1] container : Outgoing;
9      reference incoming[0-1] container : Incoming;
10   }
11
12 -- @begin Node Containers
13   abstract class NodeContainer extends Element {
14     reference contents[0-1] container : Node;
15   }
16
17   class SubAction extends NodeContainer {
18     attribute id : String;
19   }
20
21 -- @begin Top-level Actions
22   class Outgoing extends NodeContainer {
23   }
24
25   class Incoming extends NodeContainer {
26   }
27 -- @end Top-level Actions
28
29 -- @begin Switched Node Containers
30   class NotPresent extends NodeContainer {
31   }
32
33   class Otherwise extends NodeContainer {
34   }
35
36   class SwitchedAddress extends NodeContainer {
37     attribute is[0-1] : String;
38     attribute contains[0-1] : String;
39     attribute subDomainOf[0-1] : String;
40   }
41
42   class SwitchedString extends NodeContainer {
43     attribute is[0-1] : String;
44     attribute contains[0-1] : String;
45   }
46
47   class SwitchedLanguage extends NodeContainer {
48     attribute matches : String;
49   }
```

```
50
51    class SwitchedTime extends NodeContainer {
52      attribute dtstart : String;
53      attribute dtend[0-1] : String;
54      attribute duration[0-1] : String;
55      attribute freq[0-1] : String;
56      attribute until[0-1] : String;
57      attribute count[0-1] : String;
58      attribute interval : String;
59      attribute bySecond[0-1] : String;
60      attribute byMinute[0-1] : String;
61      attribute byHour[0-1] : String;
62      attribute byDay[0-1] : String;
63      attribute byMonthDay[0-1] : String;
64      attribute byYearDay[0-1] : String;
65      attribute byWeekNo[0-1] : String;
66      attribute byMonth[0-1] : String;
67      attribute wkst : String;
68      attribute bySetPos[0-1] : String;
69    }
70
71    class SwitchedPriority extends NodeContainer {
72      attribute less[0-1] : String;
73      attribute greater[0-1] : String;
74      attribute equal[0-1] : String;
75    }
76  -- @end Switched Node Containers
77
78  -- @begin Proxy Items
79    class Busy extends NodeContainer {
80    }
81
82    class NoAnswer extends NodeContainer {
83    }
84
85    class Redirection extends NodeContainer {
86    }
87
88    class Failure extends NodeContainer {
89    }
90
91    class Default extends NodeContainer {
92    }
93  -- @end Proxy Items
94  -- @end Node Containers
95
96  -- @begin Nodes
97    abstract class Node extends Element {
98
99    }
100
101 -- @begin Switches
102   abstract class Switch extends Node {
103     reference notPresent[0-1] container : NotPresent;
104     reference otherwise[0-1] container : Otherwise;
105   }
106
107   class AddressSwitch extends Switch {
108     attribute field : String;
109     attribute subField[0-1] : String;
110     reference addresses[*] ordered container : SwitchedAddress;
111   }
112
113   class StringSwitch extends Switch {
114     attribute field : String;
```

```
115     reference strings[*] ordered container : SwitchedString;
116   }
117
118   class LanguageSwitch extends Switch {
119     reference languages[*] ordered container : SwitchedLanguage;
120   }
121
122   class TimeSwitch extends Switch {
123     attribute tzid[0-1] : String;
124     attribute tzurl[0-1] : String;
125     reference times[*] ordered container : SwitchedTime;
126   }
127
128   class PrioritySwitch extends Switch {
129     reference priorities[*] ordered container : SwitchedPriority;
130   }
131 -- @end Switches
132
133   class Location extends Node, NodeContainer {
134     attribute url : String;
135     attribute priority[0-1] : String;
136     attribute clear : String;
137   }
138
139   class SubCall extends Node {
140     attribute ref : String;
141   }
142
143 -- @begin Actions
144   abstract class Action extends Node {
145   }
146
147   abstract class SignallingAction extends Action {
148   }
149
150   class Proxy extends SignallingAction {
151     attribute timeout[0-1] : String;
152     attribute recurse : String;
153     attribute ordering : String;
154
155     reference busy[0-1] container : Busy;
156     reference noAnswer[0-1] container : NoAnswer;
157     reference redirection[0-1] container : Redirection;
158     reference failure[0-1] container : Failure;
159     reference default[0-1] container : Default;
160   }
161
162   class Redirect extends SignallingAction {
163     attribute permanent : String;
164   }
165
166   class Reject extends SignallingAction {
167     attribute status : String;
168     attribute reason[0-1] : String;
169   }
170 -- @end Actions
171 -- @end Nodes
172 }
173
174 package PrimitiveTypes {
175   datatype Boolean;
176   datatype Integer;
177   datatype String;
178 }
```

B.4 Transformation XML vers CPL en ATL

Le listing B.4 donne la transformation XML vers CPL [48] écrite en ATL. Cette transformation implémente la projection de la syntaxe concrète de CPL, basée sur XML, vers des modèles conformes au métamodèle CPL (voir section B.3). Cette transformation a été automatiquement générée à partir d'une liste des éléments du schéma de CPL. Ceci explique notamment la verbosité des séquences de chaînes de caractères.

Listing B.4 – Transformation XML vers CPL en ATL

```
1  module XML2CPL;
2  create OUT : CPL from IN : XML;
3
4  uses XMLHelpers;
5
6  rule CPL {
7    from
8      s : XML!Root
9      (
10       s.name = 'cpl'
11     )
12   to
13     t : CPL!CPL (
14       subActions <- s.getElemsByNames(Sequence {'subaction'}),
15       outgoing <- s.getElemsByNames(Sequence {'outgoing'})->first(),
16       incoming <- s.getElemsByNames(Sequence {'incoming'})->first()
17     )
18 }
19
20 rule SubAction {
21   from
22     s : XML!Element
23     (
24       s.name = 'subaction'
25     )
26   to
27     t : CPL!SubAction (
28       id <- s.getAttrVal('id'),
29       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
             ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
             ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
             ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
30     )
31 }
32
33 rule Incoming {
34   from
35     s : XML!Element
36     (
37       s.name = 'incoming'
38     )
39   to
40     t : CPL!Incoming (
41       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
             ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
             ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
             ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
42     )
43 }
44
45 rule Outgoing {
46   from
47     s : XML!Element
48     (
```

```
49        s.name = 'outgoing'
50      )
51    to
52      t : CPL!Outgoing (
53        contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
54      )
55  }
56
57  rule NotPresent {
58    from
59      s : XML!Element
60      (
61        s.name = 'not-present'
62      )
63    to
64      t : CPL!NotPresent (
65        contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
66      )
67  }
68
69  rule Otherwise {
70    from
71      s : XML!Element
72      (
73        s.name = 'otherwise'
74      )
75    to
76      t : CPL!Otherwise (
77        contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
78      )
79  }
80
81  rule SwitchedAddress {
82    from
83      s : XML!Element
84      (
85        s.name = 'address'
86      )
87    to
88      t : CPL!SwitchedAddress (
89        is <- s.getAttrVal('is'),
90        contains <- s.getAttrVal('contains'),
91        subDomainOf <- s.getAttrVal('subdomain-of'),
92        contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
93      )
94  }
95
96  rule SwitchedString {
97    from
98      s : XML!Element
99      (
100       s.name = 'string'
101     )
```

```
102   to
103     t : CPL!SwitchedString (
104       is <- s.getAttrVal('is'),
105       contains <- s.getAttrVal('contains'),
106       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
107     )
108 }
109
110 rule SwitchedLanguage {
111   from
112     s : XML!Element
113     (
114       s.name = 'language'
115     )
116   to
117     t : CPL!SwitchedLanguage (
118       matches <- s.getAttrVal('matches'),
119       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
120     )
121 }
122
123 rule SwitchedTime {
124   from
125     s : XML!Element
126     (
127       s.name = 'time'
128     )
129   to
130     t : CPL!SwitchedTime (
131       dtstart <- s.getAttrVal('dtstart'),
132       dtend <- s.getAttrVal('dtend'),
133       duration <- s.getAttrVal('duration'),
134       freq <- s.getAttrVal('freq'),
135       until <- s.getAttrVal('until'),
136       count <- s.getAttrVal('count'),
137       interval <- let v : String = s.getAttrVal('interval') in
138       if v.oclIsUndefined() then
139         '1'
140       else
141         v
142       endif,
143       bySecond <- s.getAttrVal('bysecond'),
144       byMinute <- s.getAttrVal('byminute'),
145       byHour <- s.getAttrVal('byhour'),
146       byDay <- s.getAttrVal('byday'),
147       byMonthDay <- s.getAttrVal('bymonthday'),
148       byYearDay <- s.getAttrVal('byyearday'),
149       byWeekNo <- s.getAttrVal('byweekno'),
150       byMonth <- s.getAttrVal('bymonth'),
151       wkst <- let v : String = s.getAttrVal('WKST') in
152       if v.oclIsUndefined() then
153         'MO'
154       else
155         v
156       endif,
157       bySetPos <- s.getAttrVal('bysetpos'),
158       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
```

```
            ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
159     )
160 }
161
162 rule SwitchedPriority {
163   from
164     s : XML!Element
165     (
166       s.name = 'priority'
167     )
168   to
169     t : CPL!SwitchedPriority (
170       less <- s.getAttrVal('less'),
171       greater <- s.getAttrVal('greater'),
172       equal <- s.getAttrVal('equal'),
173       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
            ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
            ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
            ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
174     )
175 }
176
177 rule Busy {
178   from
179     s : XML!Element
180     (
181       s.name = 'busy'
182     )
183   to
184     t : CPL!Busy (
185       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
            ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
            ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
            ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
186     )
187 }
188
189 rule NoAnswer {
190   from
191     s : XML!Element
192     (
193       s.name = 'noanswer'
194     )
195   to
196     t : CPL!NoAnswer (
197       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
            ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
            ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
            ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
198     )
199 }
200
201 rule Redirection {
202   from
203     s : XML!Element
204     (
205       s.name = 'redirection'
206     )
207   to
208     t : CPL!Redirection (
209       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
            ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
            ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
            ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
210     )
```

```
211 }
212
213 rule Failure {
214   from
215     s : XML!Element
216     (
217       s.name = 'failure'
218     )
219   to
220     t : CPL!Failure (
221       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
222     )
223 }
224
225 rule Default {
226   from
227     s : XML!Element
228     (
229       s.name = 'default'
230     )
231   to
232     t : CPL!Default (
233       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
234     )
235 }
236
237 rule AddressSwitch {
238   from
239     s : XML!Element
240     (
241       s.name = 'address-switch'
242     )
243   to
244     t : CPL!AddressSwitch (
245       field <- s.getAttrVal('field'),
246       subField <- s.getAttrVal('subField'),
247       addresses <- s.getElemsByNames(Sequence {'address'}),
248       notPresent <- s.getElemsByNames(Sequence {'not-present'})->first(),
249       otherwise <- s.getElemsByNames(Sequence {'otherwise'})->first()
250     )
251 }
252
253 rule StringSwitch {
254   from
255     s : XML!Element
256     (
257       s.name = 'string-switch'
258     )
259   to
260     t : CPL!StringSwitch (
261       field <- s.getAttrVal('field'),
262       strings <- s.getElemsByNames(Sequence {'string'}),
263       notPresent <- s.getElemsByNames(Sequence {'not-present'})->first(),
264       otherwise <- s.getElemsByNames(Sequence {'otherwise'})->first()
265     )
266 }
267
268 rule LanguageSwitch {
269   from
```

```
270     s : XML!Element
271     (
272       s.name = 'language-switch'
273     )
274   to
275     t : CPL!LanguageSwitch (
276       languages <- s.getElemsByNames(Sequence {'language'}),
277       notPresent <- s.getElemsByNames(Sequence {'not-present'})->first(),
278       otherwise <- s.getElemsByNames(Sequence {'otherwise'})->first()
279     )
280 }
281
282 rule TimeSwitch {
283   from
284     s : XML!Element
285     (
286       s.name = 'time-switch'
287     )
288   to
289     t : CPL!TimeSwitch (
290       tzid <- s.getAttrVal('tzid'),
291       tzurl <- s.getAttrVal('tzurl'),
292       times <- s.getElemsByNames(Sequence {'time'}),
293       notPresent <- s.getElemsByNames(Sequence {'not-present'})->first(),
294       otherwise <- s.getElemsByNames(Sequence {'otherwise'})->first()
295     )
296 }
297
298 rule PrioritySwitch {
299   from
300     s : XML!Element
301     (
302       s.name = 'priority-switch'
303     )
304   to
305     t : CPL!PrioritySwitch (
306       priorities <- s.getElemsByNames(Sequence {'priority'}),
307       notPresent <- s.getElemsByNames(Sequence {'not-present'})->first(),
308       otherwise <- s.getElemsByNames(Sequence {'otherwise'})->first()
309     )
310 }
311
312 rule Location {
313   from
314     s : XML!Element
315     (
316       s.name = 'location'
317     )
318   to
319     t : CPL!Location (
320       url <- s.getAttrVal('url'),
321       clear <- let v : String = s.getAttrVal('clear') in
322       if v.oclIsUndefined() then
323         'no'
324       else
325         v
326       endif,
327       contents <- s.getElemsByNames(Sequence {'<no-element-for:Switch>', 'sub', 'address-
              ↪switch', '<no-element-for:Action>', 'priority-switch', 'location', 'proxy', '<no-
              ↪element-for:SignallingAction>', 'reject', 'time-switch', 'string-switch', '
              ↪redirect', 'language-switch', '<no-element-for:Node>'})->first()
328     )
329 }
330
331 rule Proxy {
```

```
332   from
333     s : XML!Element
334     (
335       s.name = 'proxy'
336     )
337   to
338     t : CPL!Proxy (
339       timeout <- s.getAttrVal('timeout'),
340       recurse <- let v : String = s.getAttrVal('recurse') in
341       if v.oclIsUndefined() then
342         'yes'
343       else
344         v
345       endif,
346       ordering <- let v : String = s.getAttrVal('ordering') in
347       if v.oclIsUndefined() then
348         'parallel'
349       else
350         v
351       endif,
352       busy <- s.getElemsByNames(Sequence {'busy'})->first(),
353       noAnswer <- s.getElemsByNames(Sequence {'noanswer'})->first(),
354       redirection <- s.getElemsByNames(Sequence {'redirection'})->first(),
355       failure <- s.getElemsByNames(Sequence {'failure'})->first(),
356       default <- s.getElemsByNames(Sequence {'default'})->first()
357     )
358 }
359
360 rule Redirect {
361   from
362     s : XML!Element
363     (
364       s.name = 'redirect'
365     )
366   to
367     t : CPL!Redirect (
368       permanent <- let v : String = s.getAttrVal('permanent') in
369       if v.oclIsUndefined() then
370         'no'
371       else
372         v
373       endif
374     )
375 }
376
377 rule Reject {
378   from
379     s : XML!Element
380     (
381       s.name = 'reject'
382     )
383   to
384     t : CPL!Reject (
385       status <- s.getAttrVal('status'),
386       reason <- s.getAttrVal('reason')
387     )
388 }
389
390 rule SubCall {
391   from
392     s : XML!Element
393     (
394       s.name = 'sub'
395     )
396   to
```

```
397     t : CPL!SubCall (
398       ref <- s.getAttrVal('ref')
399     )
400 }
```

B.5 Transformation CPL vers SPL en ATL

Le listing B.5 donne la transformation CPL vers SPL écrite en ATL. Cette transformation traduit des scripts définis en CPL vers des programmes SPL.

Listing B.5 – Transformation CPL vers SPL en ATL

```
1  module CPL2SPL;
2  create OUT : SPL from IN : CPL;
3
4  -- We consider nodes are statements, by default.
5  helper context CPL!Node def: statement : CPL!Node =
6    self;
7
8  -- The "location" node is not a statement.
9  helper context CPL!Location def: statement : CPL!Node =
10   self.contents.statement;
11
12 -- @begin Locations computation
13 -- By default, locations of an element are locations of its container.
14 helper context CPL!Element def: locations : Sequence(CPL!Location) =
15   self.refImmediateComposite().locations;
16
17 -- Locations of the CPL root is the empty sequence.
18 helper context CPL!CPL def: locations : Sequence(CPL!Location) =
19   Sequence {};
20
21 -- @see last paragraph of 6.1 just before 6.1.1 of CPL RFC
22 helper context CPL!Location def: locations : Sequence(CPL!Location) =
23   Sequence {self}->union(self.refImmediateComposite().locations);
24 -- @end Locations computation
25
26 rule CPL2Program {
27   from
28     s : CPL!CPL
29   to
30     t : SPL!Program (
31       service <- service
32     ),
33     service : SPL!Service (
34       name <- 'unnamed',
35       declarations <- s.subActions,
36       sessions <- dialog
37     ),
38     dialog : SPL!Dialog (
39       methods <- Sequence {s.incoming, s.outgoing}
40     )
41 }
42
43 rule SubAction2Function {
44   from
45     s : CPL!SubAction
46   to
47     t : SPL!LocalFunctionDeclaration (
48       name <- s.id,
49       returnType <- rt,
50       statements <- s.contents.statement
```

```
51      ),
52      rt : SPL!SimpleType (
53        type <- #response
54      )
55  }
56
57  rule Incoming2Method {
58    from
59      s : CPL!Incoming
60    to
61      t : SPL!Method (
62        type <- rt,
63        direction <- #"in",
64        methodName <- mn,
65        statements <-
66          if s.contents.oclIsUndefined() then
67            Sequence {}
68          else
69            Sequence {s.contents.statement}
70          endif
71      ),
72      rt : SPL!SimpleType (
73        type <- #response
74      ),
75      mn : SPL!SIPMethodName (
76        name <- #INVITE
77      )
78  }
79
80  helper context CPL!Proxy def: isSimple : Boolean =
81    self.busy.oclIsUndefined() and
82    self.noAnswer.oclIsUndefined() and
83    self.redirection.oclIsUndefined() and
84    self.failure.oclIsUndefined() and
85    self.default.oclIsUndefined();
86
87  rule Proxy2Return {
88    from
89      s : CPL!Proxy (
90        s.isSimple
91      )
92    to
93      t : SPL!ReturnStat (
94        returnedValue <- fwd
95      ),
96      fwd : SPL!ForwardExp (
97        isParallel <- s.ordering = 'parallel',
98        exp <- exp
99      ),
100     exp : SPL!ConstantExp (
101       value <- thisModule.Location2URIConstant(s.locations->first())
102     )
103 }
104
105 -- @begin Complex Proxy
106 rule Proxy2Select {
107   from
108     s : CPL!Proxy (
109       not s.isSimple
110     )
111   to
112     t : SPL!CompoundStat (
113       statements <- Sequence {declStat, select}
114     ),
115
```

```
116     -- response r = [parallel] forward <uri>;
117     declStat : SPL!DeclarationStat (
118       declaration <- decl
119     ),
120     decl : SPL!VariableDeclaration (
121       type <- rt,
122       name <- 'r',
123       initExp <- fwd
124     ),
125     rt : SPL!SimpleType (
126       type <- #response
127     ),
128     fwd : SPL!ForwardExp (
129       isParallel <- s.ordering = 'parallel',
130       exp <- exp
131     ),
132     exp : SPL!ConstantExp (
133       value <- thisModule.Location2URIConstant(s.locations->first())
134     ),
135
136     -- select(r)
137     select : SPL!SelectStat (
138       matchedExp <- v,
139       selectCases <- Sequence {s.busy, s.noAnswer, s.redirection, s.failure},
140       selectDefault <- s.default
141     ),
142     v : SPL!Variable (
143       source <- decl
144     )
145 }
146
147 rule Busy2SelectCase {
148   from
149     s : CPL!Busy
150   to
151     t : SPL!SelectCase (
152       commentsBefore <- Sequence {'// busy'},
153       values <- Sequence {v},
154       statements <- Sequence {s.contents.statement}
155     ),
156     v : SPL!ResponseConstant (
157       response <- r
158     ),
159     r : SPL!ClientErrorResponse (
160       errorKind <- #BUSY_HERE
161     )
162 }
163
164 rule NoAnswer2SelectCase {
165   from
166     s : CPL!NoAnswer
167   to
168     t : SPL!SelectCase (
169       commentsBefore <- Sequence {'// noanswer'},
170       values <- Sequence {v},
171       statements <- Sequence {s.contents.statement}
172     ),
173     v : SPL!ResponseConstant (
174       response <- r
175     ),
176     r : SPL!ClientErrorResponse (
177       errorKind <- #REQUEST_TIMEOUT
178     )
179 }
180
```

```
181 rule Redirection2SelectCase {
182   from
183     s : CPL!Redirection
184   to
185     t : SPL!SelectCase (
186       commentsBefore <- Sequence {'// redirection'},
187       values <- Sequence {v},
188       statements <- Sequence {s.contents.statement}
189     ),
190     v : SPL!ResponseConstant (
191       response <- r
192     ),
193     r : SPL!RedirectionErrorResponse (
194       errorKind <- OclUndefined
195     )
196 }
197
198 rule Default2SelectDefault {
199   from
200     s : CPL!Default
201   to
202     t : SPL!SelectDefault (
203       commentsBefore <- Sequence {'// default'},
204       statements <- Sequence {s.contents.statement}
205     )
206 }
207 -- @end Complex Proxy
208
209 rule SubCall2Return {
210   from
211     s : CPL!SubCall
212   to
213     t : SPL!ReturnStat (
214       returnedValue <- callExp
215     ),
216     callExp : SPL!FunctionCallExp (
217       functionCall <- call
218     ),
219     call : SPL!FunctionCall (
220       function <- CPL!SubAction.allInstancesFrom('IN')->select(e | e.id = s.ref)->first()
221     )
222 }
223
224 rule Redirect2Return {
225   from
226     s : CPL!Redirect
227   to
228     t : SPL!ReturnStat (
229       returnedValue <- withExp
230     ),
231     withExp : SPL!WithExp (
232       exp <- v,
233       msgFields <- Sequence {reason, contact}
234     ),
235     v : SPL!ConstantExp (
236       value <- rc
237     ),
238     rc : SPL!ResponseConstant (
239       response <- rer
240     ),
241     rer : SPL!RedirectionErrorResponse (
242       errorKind <- if s.permanent = 'true' then #MOVED_PERMANENTLY else #MOVED_TEMPORARILY
            ↪endif
243     ),
244
```

```
245     reason : SPL!ReasonMessageField (
246       exp <- reasonConstant
247     ),
248       reasonConstant : SPL!ConstantExp (
249         value <- reasonString
250       ),
251       reasonString : SPL!StringConstant (
252         value <- 'unspecified'
253       ),
254
255     contact : SPL!HeadedMessageField (
256       headerId <- '#contact:',
257       exp <- contactConstant
258     ),
259       contactConstant : SPL!ConstantExp (
260         value <- thisModule.Location2URIConstant(s.locations->first())
261       )
262 }
263
264 -- @begin Switches
265 rule AddressSwitch2SelectStat {
266   from
267     s : CPL!AddressSwitch
268   to
269     t : SPL!SelectStat (
270       matchedExp <- v,
271       selectCases <- s.addresses->including(s.notPresent),
272       selectDefault <- s.otherwise
273     ),
274     v : SPL!SIPHeaderPlace (
275       header <- if s.field = 'origin' then
276             #FROM
277           else -- s.field = 'destination' or s.field = 'original-destination'
278             #TO
279           endif
280     )
281 }
282
283 rule SwitchedAddress2SelectCase {
284   from
285     s : CPL!SwitchedAddress
286   to
287     t : SPL!SelectCase (
288       values <- Sequence {v},
289       statements <- Sequence {s.contents.statement}
290     ),
291     v : SPL!StringConstant (
292       value <- s.is
293     )
294 }
295
296 rule Otherwise2SelectDefault {
297   from
298     s : CPL!Otherwise
299   to
300     t : SPL!SelectDefault (
301       statements <- s.contents.statement
302     )
303 }
304 -- @end Switches
305
306 lazy rule Location2URIConstant {
307   from
308     s : CPL!Location
309   to
```

```
310     t : SPL!URIConstant (
311       uri <- s.url
312     )
313 }
```

Contribution à l'étude des langages de transformation de modèles

Frédéric JOUAULT

Résumé

Les techniques classiques de développement logiciel consistent généralement en l'écriture du code source d'un système par des programmeurs à partir d'une spécification comportant des modèles. Ces derniers sont souvent des dessins qui ne peuvent pas être traités automatiquement. On parle donc de modèles contemplatifs. L'ingénierie des modèles, dont le MDA (*Model Driven Architecture*) est une variante, est un nouveau paradigme de l'ingénierie du logiciel qui considère les modèles comme entités de première classe. Les modèles ne sont donc plus limités à la documentation d'un système mais peuvent faire partie de sa définition, au même titre que le code source. Ainsi, des techniques de transformations de modèles peuvent être mises en œuvre afin de générer automatiquement des parties du système à partir de modèles.
Cette thèse contribue à faire avancer les connaissances sur l'ingénierie des modèles et en particulier sur la transformation de modèles. Trois langages sont proposés : un langage de métamodélisation appelé KM3 (*Kernel MetaMetaModel*), un langage de transformation de programmes en modèles et modèles en programmes appelé TCS (*Textual Concrete Syntax*) et un langage de transformation de modèles appelé ATL (*ATLAS Transformation Language*). Une plateforme de modélisation appelée AMMA (*ATLAS Model Management Architecture*) basée sur ces trois langages est définie. Un ensemble de cas d'étude implémentés avec AMMA et couvrant différents domaines sont décrits.

Mots-clés : ingénierie des modèles, MDE, MDA, métamodélisation, transformation de modèles

Contribution to the study of model transformation languages

Abstract

Traditional software development techniques generally involve programmers writing source code of a system from a specification including models. These models are often drawings that cannot be automatically processed. They are therefore called contemplative models. Model engineering, of which MDA (Model Driven Architecture) is a variant, is a new paradigm of software engineering, which considers models as first-class entities. Thus, the models are no longer restricted to the documentation of a system but can be part of its definition like source code. Model transformation techniques can then be used in order to automatically generate parts of a system from models.
This thesis contributes to increasing the knowledge on model engineering, and more particularly on model transformation. Three languages are proposed: a metamodelisation language called KM3 (Kernel MetaMetaModel), a language for transformation of programs into models and of models into programs called TCS (Textual Concrete Syntax), and a model transformation language called ATL (ATLAS Transformation Language). A modeling platform called AMMA (ATLAS Model Management Architecture) based on these three languages is defined. A set of case studies implemented using AMMA and covering various domains are described.

Keywords: model engineering, MDE, MDA, metamodelisation, model transformation

Discipline : INFORMATIQUE

Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.