THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Intégration d'un outil de transformation de modèles dans un outil de méta-modélisation (MetaDONE)

Cocu, Catherine

Award date:
2015

Awarding institution:
Universite de Namur

Link to publication

Université de Namur
Faculté d'informatique
Année académique 2014–2015

Intégration d'un outil de transformation de modèles dans un outil de méta-modélisation (MetaDONE)

Catherine Cocu

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Vincent Englebert

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

L'ingénierie dirigée par les modèles (IDM) est une réponse à la complexité des logiciels constamment croissante. Les transformations de modèles en sont le cœur. Plusieurs langages de transformation sont disponibles et peuvent être catégorisés par différentes approches. En fonction des situations, une approche peut être plus appropriée pour accomplir une tâche donnée. Il est également important de sélectionner un outil de transformation de modèles avec des caractéristiques adéquates afin de garantir la réussite du projet.

Dans ce domaine, les outils de méta-modélisation jouent également un rôle fondamental car ils permettent la mise en œuvre des différents concepts de l'IDM.

Ce mémoire se propose de dresser un état de l'art des techniques et outils disponibles en termes de transformation de modèles et d'offrir une méthode d'évaluation de différents langages afin de trouver le plus adéquat pour des exigences données. L'objectif final est de sélectionner un outil de transformation de modèles à intégrer dans un outil de méta-modélisation. Dans cette optique, une synthèse des architectures d'intégration possibles est établie. La solution globale la plus adaptée est proposée afin de permettre la mise en place d'un prototype intégré dans l'outil de méta-modélisation *MetaDONE*.

Mots clés

Ingénierie dirigée par les modèles (IDM), transformation de modèles, outils de méta-modélisation, architectures d'intégrations.

Abstract

The model driven engineering (MDE) is a response to the constantly growing complexity of software. Model transformations are at the heart of the MDE. Several transformation languages are available and may be categorized by different approaches. Depending on the situation, one approach may be more appropriate to perform a given task. It is also important to select a model transformation tool with adequate characteristics to ensure the success of a project.

In this area, meta-modeling tools also play a key role because they allow the implementation of various concepts of MDE.

This master's thesis proposes a state of the art about techniques and tools available in terms of transformation models and offers a method for evaluating transformation tools to find the most suitable for a given requirements. The final goal is to select a model transformation tool to integrate it into a meta-modeling tool. In this context, a summary of possible integration architectures is established. The most suitable overall solution is proposed to allow the establishment of an integrated prototype in the meta-modeling tool *MetaDONE*.

Keyword

MDE, model transformation, MetaCASE, tools integration.

Avant-propos

Je tiens à remercier toutes les personnes ayant participé directement ou indirectement à la réalisation de ce mémoire.

En premier lieu, le Professeur Vincent Englebert, pour son suivi et écoute accordés tout au long de ce travail ainsi qu'Amanuel Koshima, assistant à la faculté d'informatique, pour ses conseils.

Je remercie également l'ensemble des professeurs et du personnel de l'Université de Namur pour leur professionnalisme et la transmission de leur savoir. Merci également aux autres étudiants pour avoir contribué à une bonne ambiance et rendre ces études riches humainement.

Merci à tous mes amis pour leur soutien et motivation, m'ayant permis de surmonter les moments difficiles.

Je tiens également à remercier Marie-Françoise pour ses relectures et enthousiasme ainsi que Kenny pour sa patience et son soutien.

Enfin, mes pensées vont à ma grand-mère maternelle.

Table des matières

Glossaire

ATL Langage de transformation de modèles de type hybride.

CASE ou atelier de génie logiciel : aide à la conception et construction de systèmes d'informations.
CORBA Standard définit par l'OMG permettant la communication entre des systèmes déployés sur des plateformes différentes.

DSL ou langage dédié : langage possédant des spécifications adaptées à un domaine spécifique.
DSLM DSL permettant de spécifier un programme directement à partir de concepts du domaine cible.

EMF Framework de modélisation et standard pour les modèles de données (core) incluant le méta-modèle Ecore.
ETL Langage de transformation de modèles de type hybride appartenant à la famille Epsilon.

GraSyLa Langage fournissant un mapping entre un DSML définit par un méta-modèle et une syntaxe concrète.
GrGen.NET Système de réécriture de graphes permettant également la transformation de modèles.
GXL Dialecte XML, standard pour les outils de transformations de graphes.

IDM Discipline du domaine du génie logiciel prônant un travail sur un niveau d'abstraction plus élevé, centré sur les modèles.

MDA Démarche de réalisation de logiciels, à l'aide entre autres des PIM et PSM, proposée et soutenue par l'OMG.
MetaCASE ou outil de méta-modélisation : permet la spécification et génération d'ateliers logiciels (CASE).
MOF Standard de l'OMG utilisé pour la représentation des méta-modèles et leur manipulation.

OCL Langage formel utilisé pour exprimer des contraintes sur les modèles UML.
OMG Association à but non lucratif dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.
OSGi Organisation spécifiant une plate-forme de services fondée sur le langage Java pouvant être gérée de manière distante.

XML langage de balisage générique qui permet de structurer des données.

Acronymes

API *Application Programming Interface*.
ATL *ATLAS Transformation Language*.

CASE Computer Aided Software Engineering.
CORBA *Common Object Request Broker Architecture*.

DSL Domain Specific Language.
DSLM Domain Specific Modeling Languages.

EMF Eclipse Modeling Framework.
ETL *Epsilon Transformation Language*.

GraSyLa Graphical Symbolic Language.
GXL Graph eXchange Language.

IDM Ingénierie Dirigée par les Modèles.

MDA Model Driven Architecture.
MOF Meta-Object Facility.

OCL Object Constraint Language.
OMG Object Management Group.
OSGi Open Service Gateway initiative.

PIM Platform Independant Model.
PSM Platform Specific Model.
PTL Langage de transformation de modèles basé sur la programmation logique.

UML Unified Modeling Language.

XMI XML Metadata Interchange.
XML eXtensible Markup Language.

Chapitre 1

Introduction

La complexité des systèmes d'informations est en accroissement constant, de même que le coût et le temps liés à leur développement [Wikipédia, 2014]. L'IDM (Ingénierie Dirigée par les Modèles) ou en anglais MDE (Model Driven Engineering), une discipline du domaine du génie logiciel, fournit une réponse à un besoin d'automatisation du cycle de vie des développements, devant permettre une économie d'échelle dans le domaine industriel. Cette évolution prône un travail sur un niveau d'abstraction plus élevé que la programmation classique et centré sur les modèles.

La spécification du système s'effectue *via* des modèles permettant de se focaliser sur ses caractéristiques en omettant les détails de son implémentation. De manière simplifiée, ces modèles sont des entités contenant un ensemble de données décrivant le système en question. Toutefois, ceux-ci doivent être suffisamment précis pour pouvoir être traités par des machines. Ils sont définis dans des langages particuliers, des standards ont été conçus à cet effet tels que les méta-modèles UML (*Unified Modeling Language*).

Avec l'avènement de l'IDM, centrée sur les modèles, la question des transformations de modèles est devenue une problématique capitale car elles jouent un rôle fondamental dans cette approche. Elles en constituent même le cœur. En effet, elles permettent de passer d'un niveau d'abstraction à un autre, plus ou moins automatiquement. En outre, les transformations peuvent passer d'un modèle stocké dans un *repository* à un modèle représenté sous une forme textuelle et *vice versa*. Le code d'un système peut donc être généré par ce moyen. De la même manière, l'évolutivité d'un système sera plus aisée puisqu'il pourrait suffire de le régénérer lors de chaque modification. Ainsi, l'IDM peut être vue comme une forme d'ingénierie générative et les processus du cycle de vie des développements des systèmes comme une suite de transformations de modèles.

L'intérêt pour les transformations de modèle est très élevé, ainsi différentes approches ont été développées indépendamment tout comme un grand nombre de langages de transformations de modèles les mettant en place. Il n'est par conséquent pas aisé d'en avoir une vue globale. Leurs caractéristiques diffèrent considérablement. Sélectionner un langage de transformation de modèles pour une situation donnée peut donc s'avérer compliqué. À cet effet, une étape cruciale est d'établir des exigences claires dans le but de d'évaluer les outils et pouvoir prendre une décision.

Dans le domaine de l'IDM, les outils de méta-modélisation sont d'une aide fondamentale. Ils sont employés pour générer des ateliers de génie logiciel spécifiques à un contexte. Ces

outils assistent l'ingénieur dans la spécification et conception d'un système.

L'objet du mémoire consiste à effectuer un inventaire des outils et techniques de transformations de modèles actuellement disponibles, de dresser les exigences du problème pour finalement proposer un moyen de permettre l'intégration dans l'outil de méta-modélisation *MetaDONE* [Englebert, 2015] d'un langage de transformation de modèles en adéquation avec les besoins formulés.

De ce fait, la question de l'interopérabilité jouera ici un rôle primordial. En effet, différents outils spécialisés seront amenés à communiquer ensemble dans un environnement hétérogène. Différentes possibilités seront envisagées et évaluées d'un point de vue architectural afin de permettre une intégration adéquate de l'outil sélectionné dans MetaDONE.

Ce mémoire est réparti en différents chapitres avec premièrement, un état de l'art exposant les principes fondamentaux du domaine de l'IDM et des transformations de modèles. La classification de langages de transformation y est effectuée afin d'offrir une meilleure compréhension de ce qu'il est possible de réaliser avec un langage. Elle est basée sur des travaux existants tels que [Czarnecki et Helsen, 2003], proposant une taxonomie des caractéristiques générales qu'un langage peut posséder. Aussi, les différentes approches de transformation sont évaluées avec la mise en lumière de leur avantages et inconvénients afin de définir celles qui sont les plus matures et adéquates.

Deuxièmement, les exigences du problème sont présentées. Dans cette partie, nous présentons la liste des caractéristiques recommandées pour un outil de transformation de modèles extraite de [Mens et Van Gorp, 2006], les exigences architecturales quant à l'intégration d'un outil dans MetaDONE ainsi qu'un cas d'étude servant de fil conducteur pour la suite du travail. Ce cas d'étude concerne une transformation se voulant réaliste et exprimée dans le domaine des langages dédiés ou DSL (Domain Specific Langage). Sur cette base, des outils de transformation de modèles vont être pré-sélectionnés, jugés entre autre sur leur maturité, interopérabilité, fonctionnalité, etc. Par la suite, un benchmark est organisé autour du cas d'étude afin d'évaluer les langages de manière complémentaire et objective sur la conception d'une transformation réaliste pour trouver l'outil le plus adéquat. Cette évaluation se base sur la norme standard ISO-9126 reposant sur des critères d'évaluation tels que l'utilisabilité, la maintenabilité ou encore l'efficacité du langage.

Troisièmement, le sujet de l'interopérabilité est abordé et des perspectives d'intégration pour un outil dans un environnement de méta-modélisation sont listées et évaluées, sur base des travaux tel que [Wasserman, 1990], [Wicks, 2004] et [Brownsword *et al.*, 2004]. Enfin, des solutions pour permettre une intégration adéquate dans MetaDONE pour un langage de transformation de modèles jugé approprié sont proposées.

Pour terminer, le dernier chapitre vient conclure ce travail et énonce les différentes perspectives d'extension de ce dernier.

En annexe, les différentes implémentations de la transformation mis en œuvre pour résoudre le cas d'étude avec les outils de transformations de modèles retenus sont fournis.

Chapitre 2

Ingénierie dirigée par les modèles (IDM)

Ce chapitre fournit les fondations théoriques utiles à la compréhension du domaine de l'ingénierie dirigée par les modèles. Une présentation générale de l'IDM est effectuée dans la section 2.1. La section 2.2 introduit tous les concepts importants de ce domaine tel que le standard MOF (*Meta-Object Facility*), la transformation de modèles et son fonctionnement. Ensuite, une brève introduction est donnée sur le principe MDA (*Model Driven Architecture*) ou Architecture Dirigée par les Modèles en français, les concepts de DSL (*Domain Specific Language*) et DSLM (*Domain Specific Modeling Languages*) sont définis. Pour terminer, les outils de méta-modélisation sont présentés.

2.1 Introduction

L'IDM, ou *Model Driven Engineering* (MDE) en anglais, est une discipline récente dans le milieu de l'ingénierie logicielle permettant de maîtriser la complexité des développements des systèmes informatiques en promouvant les modèles en entités de première classe.

Selon la définition de Jean Bézivin, un modèle est « une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé » [Bézivin, 2004].

L'IDM est donc soumise à un niveau d'abstraction plus élevé. Le processus de développement repose sur ces modèles, ils sont exploités *via* une approche générative pour obtenir le résultat escompté. Afin de rendre leur exploitation possible, la représentation de ces modèles doit être formalisée. Un modèle est donc lui-même soumis à un méta-modèle qui est « un modèle qui définit le langage d'expression d'un modèle » [OMG, 2014a].

Il existe différents standards permettant de mettre en place l'IDM. Citons par exemple le standard MOF de l'OMG (*Object Management Group*) définissant des directives quant à la représentation des méta-modèles et leur manipulation ou encore EMF (*Eclipse Modeling Framework*) [1], standard définit par la fondation Eclipse. L'OMG propose également d'autres standards tel que le MDA, s'appuyant sur le MOF et les transformations de modèles.

L'exploitation des modèles se fait *via* des transformations. Elles sont le cœur de l'ingénierie des modèles. Celles-ci offrent la possibilité d'automatiser différentes tâches du processus

1. http://eclipse.org/modeling/emf/. Date : 15/01/2015

du développement. L'IDM repose donc sur la méta-modélisation et sur les transformations.

Les langages dédiés ont également un rôle important, ils offrent plusieurs avantages tel qu'un gain de productivité en répondant aux exigences propres d'un domaine particulier. Pour terminer, les outils de méta-modélisation sont des outils indispensables dans le domaine, ils supportent complètement des DSLM et la génération d'ateliers logiciels. Ces derniers sont nécessaires pour permettre aux utilisateurs de mettre en œuvre des tâches de l'IDM, reposant sur l'exploitation des modèles.

2.2 Concepts et principes de bases

2.2.1 Meta-Object Facility (MOF)

Le standard MOF possède une architecture composée de quatre niveaux de modélisation, allant du niveau le plus bas M0 au plus élevé M3 :

Le niveau M0 représente le système réel, une représentation simplifiée d'un système.

Le niveau M1, correspondant à celui des **modèles**, est une représentation abstraite d'une partie ou de l'ensemble d'un système (M0). Il permet de le décrire en se focalisant sur ses fonctionnalités importantes en ignorant les détails non-pertinents quant à son utilisation afin de permettre une réflexion sur ce système. Ainsi, les détails d'implémentation ou de choix technologiques y sont omis.

Le niveau M2 constitue celui des **méta-modèles**, les modèles sont définis par ces derniers, ils en définissent la sémantique et la structure.

Le dernier niveau M3, formant le **méta-méta-modèle**, est à la base de la définition de tous les méta-modèles. Ce niveau est méta-circulaire, c'est-à-dire qu'il peut se définir par lui-même. Il est issu du standard UML.

Les différents niveaux sont soumis à une relation de conformité dite « *conformTo* ». Un modèle est dit conforme à un méta-modèle si tous ses éléments sont définis par son méta-modèle. La figure 2.2 illustre les différents niveaux ainsi que leur relation de conformité.

Pour terminer, nous nous limiterons au « strict metamodeling » dans le cadre de ce travail, la figure 2.1 illustre ce concept. Cela signifie que « pour une architecture à n-niveaux, de M0 à M_n, chaque élément d'un niveau M_{n-1} doit être une instance d'exactement un élément du niveau M_n », [Atkinson et Kühne, 2002]. Cependant, l'outil de méta-modélisation sur lequel se concentre le mémoire s'affranchit de ce concept [Englebert et Heymans, 2007].

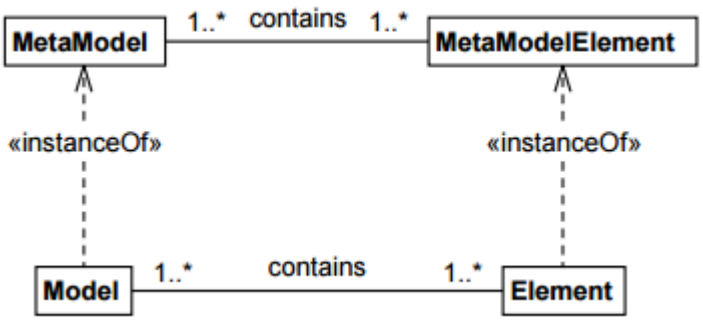

FIGURE 2.1 – Strict modeling, tiré de [Atkinson et Kühne, 2002]

2.2.2 Transformation de modèles

Une transformation de modèles consiste en « une génération automatique de un ou plusieurs modèle(s) de type cible à partir d'un ou plusieurs modèle(s) de type source, selon une spécification de transformation » [Mens et Van Gorp, 2006].

Une spécification ou description de transformation est écrite dans un langage de transformation, elle « exprime la façon dont le ou les modèles sources sont transformés en un ou plusieurs modèles cibles » [Biehl, 2010].

Cette description de transformation peut être composée de plusieurs règles de transformation, ce sont les plus petites unités de transformation, décrivant la façon dont un fragment du modèle source peut être transformé en un fragment du modèle cible [Kleppe *et al.*, 2003].

La figure 2.2 illustre le *pattern* d'un processus de transformation modèle vers modèle (M2M). Il représente la génération d'un ou plusieurs modèle(s) cible(s) à partir d'un ou plusieurs modèle(s) source(s) grâce à l'exécution de transformations. Ces trois types d'entités sont conformes à leur méta-modèle respectif, étant eux-mêmes conformes au méta-méta-modèle.

FIGURE 2.2 – Pattern de transformation de modèles, issu de [Syriani *et al.*, 2013a]

2.2.3 Model Driven Architecture (MDA)

Le MDA est un standard défini par l'OMG, s'appuyant sur la norme MOF. C'est une incarnation spécifique de l'IDM. Son objectif poursuivi est de séparer les diverses phases de développement d'un système et d'y attribuer des modèles distincts ayant des niveaux d'abstraction différents. Les modèles utilisés doivent être exprimés dans un langage basé sur le MOF [2] tel qu'UML. Les transformations de modèles sont impliquées lorsque les PIM (*Platform Independant Model*) sont transformés en PSM (*Platform Specific Model*) [Combemale,

2. http://www.omg.org/mda/specs.htm. Date 03/01/2015

2008].

Le PIM ne contient aucune information sur la plateforme qui sera utilisée. Le PSM contient des informations spécifiques quant à la plateforme qui sera employée, son niveau d'abstraction est moins élevé. Il est généré à partir du PIM en se basant sur le PDM (*Plateform Description Model*) [Diaw *et al.*, 2010].

Les principes du processus MDA peuvent être présentés comme suit (figure 2.3) :

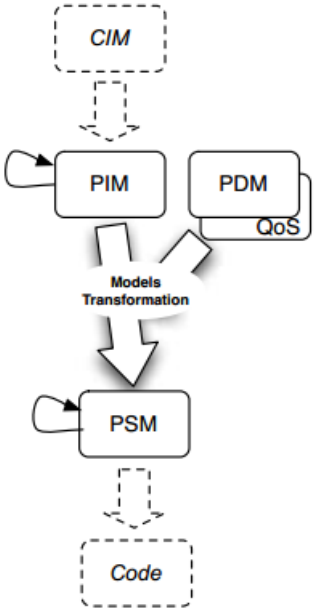

FIGURE 2.3 – Principes du processus MDA, tiré de [Diaw *et al.*, 2010]

Le langage QVT (Query/Views/Transformation) [OMG, 2011] est le standard proposé par l'OMG pour spécifier les transformations de modèles dans ce contexte [Kurtev, 2008].

Contrairement à cette approche, « l'IDM est une approche intégrative et ouverte englobant plusieurs espaces technologiques de manière uniforme » [Favre, 2004].

2.2.4 Langage dédié

Les langages dédiés, dont l'acronyme anglais est DSL, sont des langages possédant des spécifications adaptées à un domaine spécifique. Ils sont basés sur les concepts et les fonctionnalités de ce dernier. Les DSL répondent aux exigences propres du domaine et offrent une meilleure expressivité en fournissant des constructions et notations spécifiques pour le domaine en question. Ils sont en outre plus simples à utiliser que des langages de programmation plus généraux pour ce domaine. Les préoccupations sont séparées grâce à des constructions capitalisant l'expérience du domaine. Ils permettent également un gain de productivité et réduisent les coûts de maintenance [Mernik *et al.*, 2005].

2.2.5 Langage de modélisation dédié

Un langage de modélisation dédié à un domaine particulier (DSLM), permet de définir un programme en utilisant directement des concepts du domaine ciblé. Il permet d'élever le niveau d'abstraction et de générer le code *via* cette spécification de plus haut niveau.

« La méta-modélisation est l'activité consistant à définir le méta-modèle d'un langage de modélisation. Elle vise donc à bien modéliser un langage, qui joue alors le rôle de système à modéliser » [Combemale, 2008].

2.2.6 Outils de méta-modélisation

Cette section est inspirée des travaux de [Ebert *et al.*, 1997].

Les outils CASE (*Computer Aided Software Engineering*), ou ateliers de génie logiciel, ont pour but d'assister l'ingénieur dans les différentes phases de vie d'un système d'informations telles que la capture des besoins, la conception, le prototypage, le *refactoring*, etc. L'exploitation des modèles est au cœur du fonctionnement de ces outils et plusieurs fonctionnalités sont proposées comme la génération de code à partir de modèles, la transformation de modèles, etc.

Néanmoins, ce type d'outil manque de flexibilité, ne permettant pas la génération de systèmes adéquats à chaque situation particulière. Ce qui s'avère problématique dans un monde où les besoins changent constamment, avec des environnements hétérogènes où divers dialectes coexistent, etc. Il n'est généralement pas possible de modifier les outils CASE afin qu'ils soient parfaitement appropriés à un contexte spécifique ou qu'ils supportent complètement des DSLM.

Les outils de méta-modélisation ou MetaCASE sont une solution à ce besoin. Ils rendent possible la spécification et la génération automatique d'ateliers logiciels en vue de répondre de façon adéquate à un processus d'ingénierie spécifique. Ils permettent entre autres la définition de méta-modèles, de notations concrètes ou encore la gestion de modèles. MetaDONE et MetaEdit+ [Kelly *et al.*, 1996] sont des exemples de réalisations. Ce type d'outil permet de couvrir les besoins d'évolutions car l'outil CASE pourra être facilement et rapidement régénéré, en mettant simplement à jour sa spécification, pour répondre aux changements souhaités.

2.3 Résumé

L'IDM est un domaine vaste, reposant sur les modèles et leurs transformations, requierant l'utilisation d'outils tel que des MetaCASE et pour lequel plusieurs standards sont mis en place tel que le MOF ou MDA. Sa généralisation à l'ensemble de l'industrie n'en est qu'à ses débuts. Il y a encore beaucoup à en dire mais le but est ici de donner une vue globale des concepts principaux.

Un des concepts central est celui des transformations de modèles, il est introduit dans ce chapitre et présenté en détails dans le suivant.

Chapitre 3

État de l'art

Dans ce chapitre, nous tentons d'élaborer un état de l'art s'inspirant de travaux existants dans le domaine. Nous présentons la façon dont les transformations peuvent être caractérisées par leurs méta-modèles et niveau d'abstraction (section 3.2). Ensuite nous abordons le sujet du nombre de modèles pouvant être impliqués dans une transformation (section 3.3) ou encore définir ce qu'est un espace technique (section 3.4). Après, les caractéristiques générales d'une transformation sont exposées (section 3.5). Enfin, les différents types d'approches existantes sont présentées et évaluées afin de choisir les plus favorables pour nos besoins. (section 3.6).

3.1 Introduction

Différents travaux tels que [Czarnecki et Helsen, 2003], [Huber, 2008] ou [Diaw *et al.*, 2010] ont déjà dressés des états de l'art concernant le domaine des transformations de modèles. Ils présentent principalement les différentes approches existantes et recensent les caractéristiques possibles pour un langage de transformations. Certains, comme Huber, offrent des pistes envisageables quant au choix d'un outil de transformations en évaluant les différentes approches et caractéristiques offertes en fonction d'un besoin.

Il existe plusieurs approches techniques permettant la mise en place de transformations de modèles. Elles sont divisées en deux principales catégories (modèles vers modèles et modèles vers texte), subdivisées elles-mêmes en plusieurs sous-catégories possédant chacune leurs forces et faiblesses. Les transformations de modèles peuvent être classifiées par plusieurs dimensions tel que leur méta-modèle, leur niveau d'abstraction ou encore l'approche mise en œuvre. Aussi, le nombre de modèles impliqués dans une transformation peut varier tout comme son espace technique. Une large gamme de langages de transformations existe, possédant tous leurs propres caractéristiques. Par conséquent, il n'est pas aisé de trouver un outil pour un besoin précis.

3.2 Méta-modèles et niveau d'abstraction

Les transformations de modèles peuvent être catégorisées en fonction de leur méta-modèle et de leur niveau d'abstraction.

D'une part, les transformations peuvent être de type endogène ou exogène selon leur méta-modèle :

les transformations de type *endogène* : une transformation est définie comme telle si les modèles impliqués dans la transformation sont issus du même méta-modèle comme le raffinement ;

les transformations de type *exogène* : une transformation est définie comme telle si les modèles source et cible sont issus de méta-modèles distincts tel que la migration de logiciels et fusion de modèles.

Dautre part, elles peuvent être classifiées de type horizontal ou vertical en fonction de leur niveau d'abstraction :

les transformations de type *horizontal* : une transformation est définie comme telle si les modèles impliqués dans la transformation ont un niveau d'abstraction identique. Autrement dit, une modification de la structure interne qui compose le modèle et non son niveau d'abstraction, telle qu'une restructuration ;

les transformations de type *vertical* : une transformation est définie comme telle si les modèles sources et cibles sont de niveaux d'abstraction différents. Par exemple, la génération qui rend le niveau d'abstraction moins élevé en ajoutant des informations dans le modèle afin de le rendre plus précis.

La figure 3.1 reprend les différentes combinaisons de types de transformations et expose des scénarios d'utilisation possibles. Elle illustre le fait que les dimensions sont orthogonales. Dès lors, on s'aperçoit que la restructuration, la normalisation et intégration de patrons sont des transformations endogènes et horizontales. Le raffinement est un exemple endogène et vertical. Tandis que la génération et la rétro-conception sont des transformations exogènes et verticales. La migration de logiciels et la fusion de modèles sont des transformations de types exogènes et horizontales.

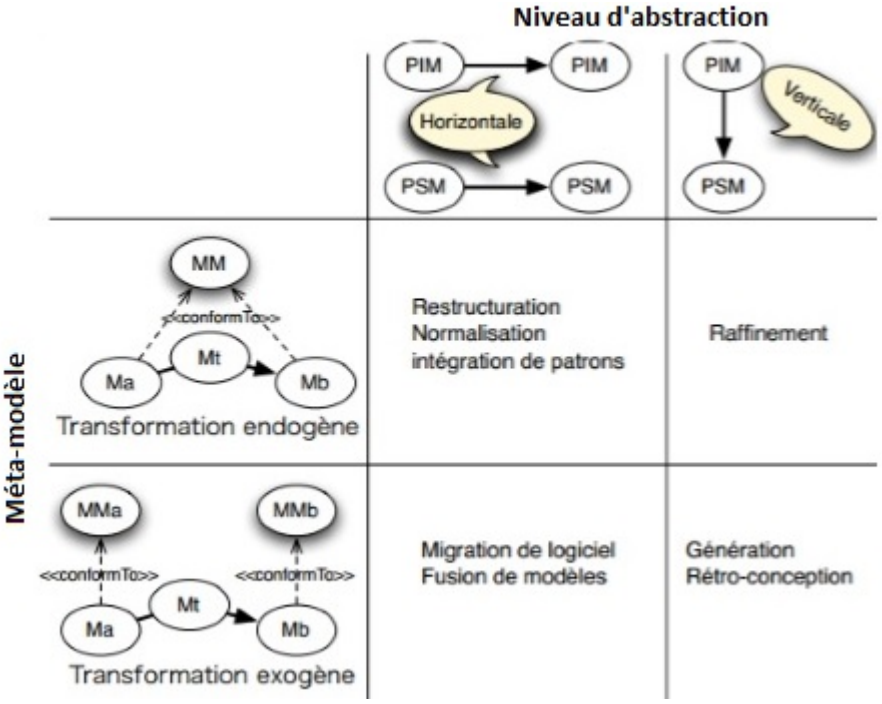

FIGURE 3.1 – Types de transformation et leurs principales utilisations [Diaw *et al.*, 2010]

3.3 Nombre de modèles impliqués

Les transformations peuvent impliquer plusieurs modèles de type source et cible. En général, une transformation porte sur deux modèles distincts, un de type source et l'autre de type cible. Si le modèle cible existe déjà, il sera écrasé et réécrit sinon il sera généré *from scratch* à partir du modèle source. Toutefois, une transformation peut impliquer un seul modèle, il sera simultanément la source et la cible. C'est une transformation *in-place*, ou littéralement « sur place », le modèle cible sera généré en modifiant le modèle source. Ou encore, des transformations peuvent également impliquer plusieurs modèles de type source permettant la génération d'un ou de plusieurs modèles de type cible.

3.4 Espace technique

Un espace technique représente la technologie par laquelle est représentée les modèles. « Cette technologie inclut le format des fichiers, les structures de données, les parseurs et les moyens pour gérer les données »[Biehl, 2010]. L'espace technique est déterminé par le méta-modèle utilisé (niveau M3). Il limite l'ensemble des moteurs de transformations pouvant être employés.

Nous pouvons citer XMI (*XML Metadata Interchange*), XML (*eXtensible Markup Language*), MOF, GXL (*Graph eXchange Language*), *Kernel Meta-Meta-Model*(KM3) ou EMF. KM3 est considéré comme un DSL neutre, disponible sur la plateforme Eclipse, permettant la création des méta-modèles et la définition de DSL [Jouault et Bézivin, 2006]. Tandis que « GXL est un dialecte XML, *de facto* le standard pour les outils de transformations de graphes. » [Winter *et al.*, 2002]
Quant au MOF, il a déjà été présenté précédemment. Le MOF 2 est composé de deux composants principaux : *Essential MOF* (EMOF) et *Complete MOF* (CMOF) [OMG, 2014a].

Le projet EMF est fortement adopté au sein des projets de la fondation Eclipse, permettant une interopérabilité entre ses projets. EMF permet de définir les concepts de l'utilisateur sous forme de méta-modèles (Ecore) afin de les rendre manipulables. Ces modèles sont ensuite instanciables en format XMI. « XMI est *de facto* le standard pour les outils de transformations de modèles. » Il offre une représentation concrète des modèles sous forme de documents XML.

Il est à noter qu'il existe quelques différences mineures entre le XMI d'Ecore et celui du MOF, principalement liées à des noms d'éléments [1], par exemple le nom des classes est « Class » pour MOF et « Eclass » pour Ecore ou encore la correspondance de l'attribut « isReadOnly » de MOF est « changeable » en Ecore. Toutefois, EMF peut lire et écrire de manière transparentes le format des fichiers EMOF.

3.5 Caractéristiques générales d'une transformation

La figure 3.2 présente les différentes catégories générales de caractéristiques et propriétés qu'une transformation peut posséder :

spécification : « En général, la spécification décrit les relations entre le modèle source et cible » [Czarnecki et Helsen, 2006]. Cependant, certaines approches proposent de spécifier des pré et post-conditions à l'aide d'OCL (*Object Constraint Language*) [OMG, 2014b] ;

1. http://www.eclipse.org/gmt/umlx/doc/EclipseAndOMG08/EMOFAdapters.pdf. Date 21/05/2015

règles de transformation : ce sont les unités de base des transformations. Elles possèdent un sous-ensemble de caractéristiques. Ainsi, elles sont spécifiées dans un ou plusieurs domaines, peuvent employer des structures intermédiaires, instaurer une séparation syntaxique des règles en fonction du domaine où elles opèrent, sont paramétrables pour simplifier la réutilisation, peuvent être déterministes ou non, etc ;

contrôle de l'application des règles : stratégie de détermination de l'ordre des règles. Une règle ne peut être appliquée que sous certaines conditions ou un contexte spécifique ;

organisation d'une règle : traite de la structure de la règle, de sa modularité ainsi que de sa réutilisation ;

relation entre le modèle source et cible : gère l'identité des modèles source et cible, les modèles impliqués dans la transformation peuvent être deux modèles distincts ou un même modèle ;

incrémentalité : concerne la capacité d'un modèle cible à s'adapter suite aux modifications effectuées sur son ou ses modèle(s) source(s) correspondant. Cette caractéristique se révèle utile lorsqu'il doit y avoir une certaine consistance entre les modèles ;

directionnalité : détermine si une transformation peut être appliquée dans une seule direction (unidirectionnelle) de la source vers la cible, ou plusieurs (multi-directionnelle), de la source vers la cible et *vice-versa* ;

traçabilité : c'est un mécanisme permettant de conserver un historique des liens entre les éléments sources et cibles lors de l'exécution de la transformation. La traçabilité peut être implicite ou explicite. C'est une notion importante pour aider au debugage des transformations.

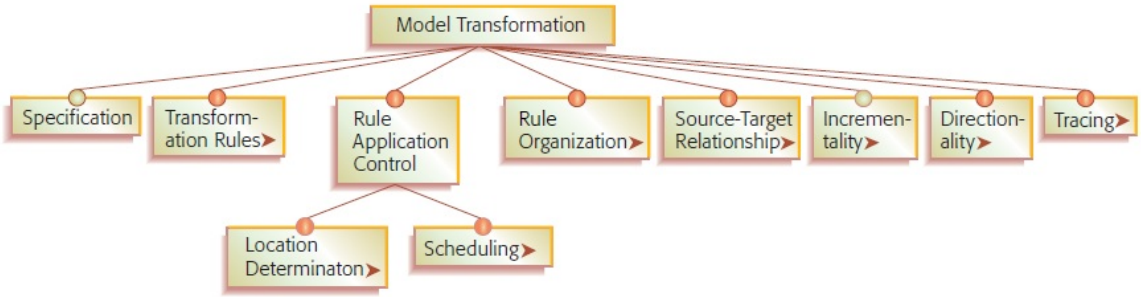

FIGURE 3.2 – Principales catégories de caractéristiques des transformations, tiré de [Czarnecki et Helsen, 2006]

Pour une description plus détaillée et complémentaires de ces caractéristiques, nous vous renvoyons vers l'article de [Syriani et Vangheluwe, 2009].

3.6 Types d'approches existantes

Deux approches principales ont été identifiées par Czarnecki [Czarnecki et Helsen, 2006]. Il s'agit des transformations de modèle vers texte et modèle vers modèle ou respectivement « model-to-text » et « model-to-model » en anglais.
Ces approches sont elles-mêmes subdivisées en différentes approches comme l'illustre la figure 3.3.

3.6.1 Model-to-text (M2T)

Le terme « model-to-text » est ici employé car cette catégorie de transformation permet la génération de texte tels que des fichiers de documentation, du XML ou du code (ayant pour synonyme « model-to-code »). Il existe deux manières différentes de mettre en œuvre cette approche :

template-based : les templates contiennent des fragments du texte cible et un méta-programme pouvant accéder et sélectionner le code du modèle source. La majorité des outils supportent cette approche, nous pouvons citer : Codagen [Codagen, 2002], AndroMDA [AndroMDA, 2014], OptimalJ [OptimalJ, 2002], XDE [IBM, 2015], XDoclet [Walls *et al.*, 2004] ou encore ANGIE [Technology, 2010]. L'extrait [2] d'un *template* d'AndroMDA est fourni ci-dessous, il permet de modifier et récupérer le nom d'un élément correspondant. Toutes les variables commençant par « $ » sont des références au méta-modèle ;

```
1  public $target.getterSetterTypeName ${target.getterName}()
2  {
3      return this.${target.name};
4  }
5
6  public void ${target.setterName}
7   ($target.getterSetterTypeName $target.name)
8  {
9      this.${target.name} = ${target.name};
10 }
```

Listing 3.1 – Extrait d'un template d'AndroMDA

visitor-based : cette approche se base sur l'idée du *pattern visitor* [3] pour parcourir la structure interne d'un modèle afin d'en écrire le texte. Le *framework* orienté objet Jamda [Boocock, 2003] met en place cette approche.

3.6.2 Model-to-model (M2M)

Cette approche effectue des transformations à partir d'un modèle source vers un modèle cible. Elle est subdivisée en plusieurs approches distinctes :

manipulation directe : la plupart des outils appartenant à cette section sont implémentés comme des *frameworks* orientés objet. Il s'agit de langages de programmation à usage général mettant à disposition des librairies pour manipuler la représentation interne des modèles et fournissant une infrastructure afin de gérer les transformations, *e.g.* Jamda, JMI (Java Metadata Interface) [Pivl, 2005] et Sitra [Bordbar *et al.*, 2007] ;

impérative/opérationnelle : cette approche est similaire à la précédente, les concepts des langages de transformation de modèles impératifs sont semblables à ceux des langages de programmation classiques. Cependant, sa mise en œuvre est réalisée *via* des langages dédiés et des librairies ou *frameworks* offrant un support à l'utilisateur plus adapté pour effectuer ses transformations.

Ces langages sont séquentiels et mettent en avant le **comment**, la manière dont les transformations doivent être exécutées. Un exemple est Kermeta [Jézéquel *et al.*, 2011] ;

2. http://sourceforge.net/p/andromdaplugins/mailman/message/5085432/. Date 25/05/2015
3. http://en.wikipedia.org/wiki/Visitor_pattern. Date 25/05/2015

relationnelle : le principe de base de cette approche est d'établir une relation mathématique entre le modèle source et cible. La spécification des relations se fait *via* des contraintes, à l'aide d'OCL par exemple.
Elle n'offre pas de contrôle explicite sur le flux de la transformation car elle se concentre sur le **quoi**, sur ce qui doit être transformé lors de la transformation. À titre d'exemple, nous pouvons trouver les langages QVT, metatools [Lepper *et al.*, 2011], UML-RSDS [Lano et Kolahdouz-Rahimi, 2010], MOLA Kalnins *et al.* [2005] et Tefkat [Lawley et Steel, 2006] (basé sur F-Logic [Kifer et Lausen, 1989]).

De plus, les langages de programmation logique comme Mercury [Somogyi *et al.*, 1995], F-Logic, Prolog ou encore Maude [Clavel *et al.*, 2002] appartiennent à cette catégorie et sont également des candidats adéquats. En effet, les prédicats peuvent être utilisés pour mettre en place les relations. De plus, leur système de backtracking et de recherche, leur principe d'unification et leur bidirectionnalité sont des atouts majeurs.
Les deux premiers langages cités ci-dessus sont testés pour la mise en place de transformations dans [Gerber *et al.*, 2002]. Une autre explication quant à la possibilité de mettre en places de transformations *via* Prolog est disponible dans [Almendros-Jiménez et Iribarne, 2008].
En outre, EPromote(*Eclipse Prolog MOdel Transformation Engine*) [Sadilek et Wachsmuth, 2008] permet également la définition de transformation de modèles (basés sur EMF) à l'aide de Prolog. Cependant, peu de documentation existe à son sujet.
Aussi, MoMaT (*Model Manipulation Toolkit*)[Störrle, 2007] est un framework basé sur Prolog, représentant et exploitant les modèles à l'aide de Prolog. Dans cette optique, nous pouvons également citer PETE (*Prolog EMF Transformation Engine*)[Schätz, 2010].

Des systèmes à base de règles comme Jess [Friedman-Hill, 2003] peuvent aussi entrer en considération. Jess met à disposition des listes de faits gérant les données, une base de connaissance contenant toutes les règles et un moteur d'inférence contrôlant l'exécution des règles. Cet outil a été mis en œuvre dans une large gamme d'applications commerciales comme des systèmes experts, des serveurs exécutant des règles business, etc. Enfin, il est utilisé dans [Saada *et al.*, 2012] pour effectuer des transformations de modèles ;

basée sur les transformations de graphes : les langages appartenant à cette catégorie se basent sur les fondations théoriques des transformations de graphes. Les modèles sont considérés comme des graphes. Pour résumer en quelques mots, « les éléments des modèles et leurs relations sont vus comme des arêtes et des sommets. Les règles de transformations de graphes possèdent une application gauche et droite, correspondant aux modèles source et cible de la transformation » [Huber, 2008]. Des notations graphiques et textuelles sont souvent combinées pour exprimer les transformations. Cette approche est une sous catégorie de l'approche de type déclarative.
Citons par exemple : AGG [Taentzer, 2004], $ATOM^3$ (*A Tool for Multi-formalism and Meta-Modelling*) [De Lara et Vangheluwe, 2002], BOTL [Braun et Marschall, 2003], DSLTrans [Barroca *et al.*, 2011], eMoflon [Anjorin *et al.*, 2011], EMorF [Klassen et Wagner, 2012], Fujaba [Wagner, 2006], GReAT (*Graph Rewriting and Transformation*) [Balasubramanian *et al.*, 2007], GrGen.net [Geiß et Kroll, 2008], Groove [Ghamarian *et al.*, 2012], Henshin [Arendt *et al.*, 2010], MDELab SDI [Hasso Plattner

Institute , HPI], MoTif [Syriani et Vangheluwe, 2013], MoTMoT (*Model driven, Template based, Model Transformer*) [Pieter Van Gorp et Muliawan, 2015], PROGRES [Schürr *et al.*, 1999], T-Core [Syriani *et al.*, 2015], TGG (*Triple Graph Grammars*) [Königs, 2005], VMTS (*Visual Modeling and Transformation System*)[Levendovszky *et al.*, 2005], UMLX [Willink, 2003] et VIATRA (*VIsual Automated model TRAnsformations*) [Varró et Balogh, 2007] ;

dirigée par la structure : cette approche découpe les transformations par phases. Dans une première phase, la structure hiérarchique du modèle cible est créée. Ensuite, dans une seconde phase, les valeurs des attributs sont définies et les références sont assignées. Des outils mettant en place cette approche sont OptimalJ [OptimalJ, 2002] et IOPT (Interactive Objects and Project Technology) [Voudouris *et al.*, 2001] ;

hybride : elle résulte de la combinaison de plusieurs des approches citées précédemment. Des outils comme ATL (*ATLAS Transformation Language*) [Jouault *et al.*, 2008], ETL [Kolovos *et al.*, 2008], YATL [Patrascoiu, 2004] et XDE [IBM, 2015] sont des langages de transformation de règles combinant l'approche impérative et déclarative. Ces langages laissent le choix à l'utilisateur d'utiliser les structures impératives ou déclaratives ;

autres : il existe des outils appartenant à la famille XML tels que XSLT [Consortium, 2007] ou XQuery [Consortium, 2010], effectuant principalement des transformations syntaxiques.
En outre, dans [Syriani et Vangheluwe, 2009], l'approche de transformation de modèles dans une base de données relationnelles est abordée et résumée. Ces transformations consistent en des requêtes et vues sur un modèle stocké dans un SGBDR (Système de Gestion de Base de Données Relationnelle). Cela convient particulièrement pour la manipulation de modèles très larges.

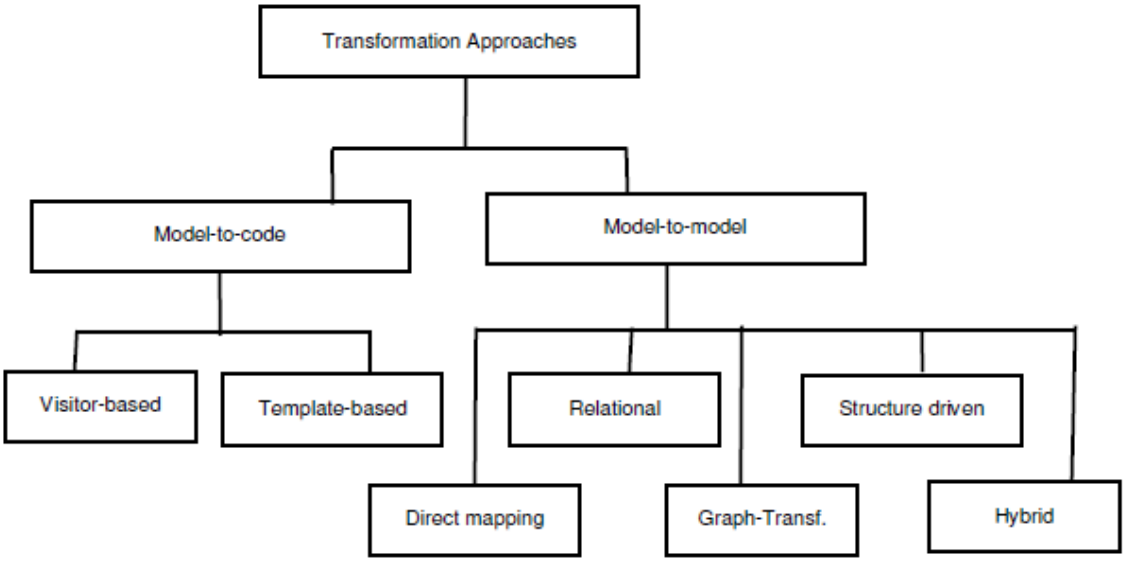

FIGURE 3.3 – Approches existantes de transformations de modèles, tiré de [Bondé, 2006]
.

3.6.3 Évaluation des différentes approches M2M

Chaque type de langage a ses avantages et inconvénients. Le choix doit être effectué en fonction d'un contexte donné ([Huber, 2008]). Dans le cadre de ce travail, nous allons principalement nous intéresser aux approches de type « model-to-model ».

l'approche par manipulation directe requiert l'utilisation de langages de programmation généraux, l'avantage est que l'utilisateur est probablement déjà familier avec ce type d'outil. Cependant, la plupart des caractéristiques typiques utiles pour les transformations sont manquantes car ces langages ne sont pas initialement conçus à cet effet. Ils opèrent à un niveau d'abstraction plus bas que ceux de l'approche déclarative. En effet, les utilisateurs doivent gérer eux-mêmes des propriétés importantes telles que l'orchestration de l'exécution des transformations, la traçabilité, etc.
À terme, cela peut se révéler difficilement maintenable et fastidieux, [Sendall et Kozaczynski, 2003], [Huber, 2008] et [Czarnecki et Helsen, 2006] ;

les langages de type impératif sont capables de résoudre bon nombre de scénarios mais en général le code requis est plus abondant que pour les autres approches, rendant la solution moins lisible. Toutefois, ils permettent de résoudre les transformations complexes plus facilement. En outre, leur implémentation est flexible et l'utilisateur possède un grand contrôle, amenant à des transformations efficaces. Pour terminer, le contrôle explicite sur l'ordonnancement des règles de transformations peut se révéler être un avantage pour résoudre certains problèmes que les approches déclaratives ne peuvent solutionner que difficilement [Huber, 2008] ;

les langages de type relationnel sont plus simples à comprendre et à implémenter car les spécifications des transformations sont plus concises et compactes car ils opèrent à un niveau d'abstraction plus élevé. Dû à ce dernier fait, l'utilisateur perd du contrôle, contrairement aux langages impératifs. Cependant, ils offrent souvent divers services tels qu'un support de traçabilité implicite et la bidirectionnalité [Mens, 2010]. De plus, les outils appartenant à cette catégorie sont sans effets de bord.

Concernant la mise en œuvre de cette approche *via* des langages de programmation logique, leur nature fait d'eux de bons candidats potentiels et leur utilisation a déjà fait l'objet de quelques réalisations telle qu'avec la mise en place d'un langage de transformation de modèles basé sur le langage Prolog, PTL (*Prolog based Transformation Language*), présenté dans [Almendros-Jiménez et Iribarne, 2013]. D'autres avantages sont leur connaissance par bon nombre de développeurs ainsi que leur interopérabilité, ils peuvent coopérer aisément avec CORBA (*Common Object Request Broker Architecture*) [Group, 2012] comme c'est le cas pour Mercury, [Gerber *et al.*, 2002] ou encore Prolog ;

l'approche de type transformation de graphes se base sur une fondation formelle solide. Elle est puissante mais très complexe dû principalement à leur non-déterminisme.

« Les langages de transformation de graphes sont nativement récursifs » [Tisi *et al.*, 2011]. De ce fait, cette approche se révèle être un candidat de choix pour résoudre de nombreux problèmes nécessitant la récursivité. Toutefois, pour les autres types de problème demandant un plus grand contrôle de la part du développeur, la mise en place d'une solution s'avèrera laborieuse car ce dernier devra explicitement désactiver la récursivité par des moyens techniques. Cette solution alternative peut conduire à

une source d'erreur supplémentaire [Tisi *et al.*, 2011].

Cependant, selon [Mens et Van Gorp, 2006], il existait une incertitude quant à l'évolutivité de ce type de langage et sa mise en pratique était encore limitée. Les langages de transformation de graphes peuvent être également confrontés à un manque de maturité et se révéler incompatibles avec une utilisation industrielle. Actuellement, ce doute est dissipé, « cette approche est la plus utilisée et celle avec le plus grand impact sur la littérature et l'industrie » selon Syriani [Syriani et Vangheluwe, 2009]. Ce constat se pose dans d'autres ouvrages tel que [Jakumeit *et al.*, 2014], où plusieurs de ces outils sont évalués. Pour conclure, un fait souligné par Syriani est que la plupart des outils actuels ne permettent qu'une transformation unidirectionnelle ;

dirigé par la structure, l'ordonnancement des règles est ici gérée par le *framework*. Cette approche est plus appropriée à la génération d'EJB ou de schéma de base de données à partir d'UML ;

les langages de type hybride offrent un compromis puisqu'ils proposent une combinaison de différentes approches. Chaque type d'approche offre des avantages et des inconvénients. Cette combinaison peut permettre la mise en place d'un large panel de scénarios. Selon [Syriani et Vangheluwe, 2009], une combinaison entre le type déclaratif et opérationnel permet de résoudre des problèmes plus complexes et sur des modèles plus grands que l'approche basée sur les graphes. De plus, l'utilisateur a le choix entre l'utilisation des structures d'un type ou de l'autre, afin d'en retirer un maximum d'avantages ;

les autres langages, tel que XSLT, ont eu des retours d'expériences soulignant leur inadéquation pour effectuer des transformations complexes, [Diaw *et al.*, 2010]. De plus, leur mise en place est fastidieuse car il n'est pas possible de travailler au niveau de la sémantique des modèles faisant l'objet de la transformations.

3.7 Conclusion

Comme nous avons pu le constater, les transformations de modèles permettent de réaliser une multitude d'opérations utiles liées à l'exploitation des modèles. Il y a beaucoup de dimensions entrant en compte et les techniques disponibles sont nombreuses et possèdent chacune leurs forces et faiblesses.

Les différentes approches existantes pour la mise en place de transformations sont présentées et évaluées, permettant de mettre en avant plusieurs avantages et inconvénients pour chacune. Les caractéristiques pouvant être offertes pour une transformation sont très variées et complètes également. Nous pouvons ainsi nous rendre compte que c'est un domaine riche où une approche adéquate peut être trouvée pour chaque problème particulier.

À la lumière de ces faits, nous allons tenter d'écarter certaines approches afin d'en présélectionner pour la suite de ce travail.

Tout d'abord, nous allons écarter l'approche par manipulation directe car elle n'offre pas de support adéquat aux transformations de modèles. De même que celle dirigée par la structure, pour cette même raison. C'est une approche peu répandue et mise en place

dans peu d'outils. Mais aussi, la catégorie des « autres » tel que XSLT, à cause de leurs inconvénients importants et manque d'adéquation pour la transformation de modèles.

De plus, nous voulons nous intéresser au problème, sur le « quoi » et non le « comment ». Cela d'une façon efficace, maintenable et concise, ce qui nous renvoie à l'approche déclarative.

Ensuite, concernant les langages impératifs, la transformation pourrait être fastidieuse à concevoir et selon [Tamura *et al.*, 2010], une des recommandations du MOF est que la définition du langage de transformation doit être de type déclaratif dans le but de supporter l'exécution des transformations de manière incrémentale. Ou comme dit dans [Kolovos *et al.*, 2008], « les langages déclaratifs sont considérés comme plus adéquats pour les scénarios où le méta-modèle source et cible sont similaires », ce qui est ici un avantage car c'est la situation exposée par notre cas d'étude, présenté au chapitre suivant. Cependant, cette contrainte devra être reconsidérée lors de notre choix final car ce n'est pas une condition générale, l'outil servira à terme à transformer des modèles issus de méta-modèles différents.

En conclusion, les approches se révélant les intéressantes et prometteuses dans notre cas sont celles de types déclarative, hybride et basée sur les transformations de graphes. Soulignons toutefois que cette analyse devra être affinée au regard des critères énoncés dans le chapitre suivant.

Chapitre 4

Présentation de l'outil de méta-modélisation MetaDONE

L'outil de méta-modélisation au cœur de ce travail, pour lequel l'intégration d'un outil de transformation de modèle doit être effectué est MetaDONE. Il est développé dans le cadre des activités de l'Université de Namur, sous la responsabilité de Vincent Englebert. La section 4.1 aborde brièvement le type de *framework* sur lequel il a été développé. Ensuite, son langage de méta-modélisation $MetaL_2$ est présenté (section 4.2). Par après (section 4.3), les détails sur le DSLM GraSyla (*Graphical Symbolic Language*) sont exposés, c'est un composant central de MetaDONE. Pour terminer par une explication sur son méta-méta modèle et son fonctionnement (section 4.4).

4.1 OSGi

MetaDONE repose sur le framework d'OSGi (*Open Service Gateway initiative*). OSGi [Alliance, 2003] est une plate-forme de services fondée sur le langage Java pouvant être gérée de manière distante. Elle permet une gestion dynamique, simple et centralisée des composants. Ce *framework* est normalisé par l'OSGi alliance.

MetaDONE est développé avec la librairie OSGi Equinox d'Eclipse [1] mais il existe d'autres implémentations d'OSGi tel que *Felix* qui est l'implémentation de la fondation Apache [2], *knoplerfish* qui une autre implémentation mature et open source ou encore *Oscar* qui est une implémentation d'*Object Web* [3].

4.2 $MetaL_2$

$MetaL_2$ [Englebert, 2007] est le langage de méta-modélisation utilisé dans MetaDONE. Il permet d'effectuer de manière simple et efficace la description de langages de modélisation. Les quatre concepts centraux sont :

méta-modèle : il décrit un DSLM. Il est un sous-type d'un méta-objet et peut appartenir à d'autres méta-modèles ;

méta-objet : est une information représentant un objet, c'est-à-dire tout concept ayant de l'intérêt dans un DSLM ;

1. https://eclipse.org/equinox/. Date : 03/04/2015
2. http://felix.apache.org/. Date : 03/04/2015
3. http://oscar.objectweb.org. Date : 03/04/2015

méta-propriété : est un sous-type d'un méta-objet équipant d'attributs le méta-objet auquel il est associé, il possède un type et une cardinalité indiquant son occurrence ;

méta-rôle : est un sous-type d'un méta-objet servant de relation entre deux méta-objets (nommés *domain* et *range*) et possédant une cardinalité ;

Ces éléments sont tous des sous-types de méta-objet. Un méta-objet hérite toujours du *CommonMetaMojbect* d'une manière directe ou indirecte. Les méta-objets peuvent également hériter d'autres méta-objets.

L'ensemble de ces concepts permet de définir un méta-modèle. Ce méta-modèle peut ensuite être instancié afin de créer un *objet concret*, qui lui-même possèdera des *propriétés concrètes* et *rôles concrets* correspondant à leurs équivalents dans le méta-modèle. Le *modèle concret* est une composition d'*objets concrets*.

$MetaL_2$ est basé sur une couche $MetaL_1$, langage formel composant les fondations élémentaires. Il est utilisé dans le *back-end* ou par les utilisateurs expérimentés.

La figure 4.1 illustre l'articulation entre les différentes couches de l'architecture, $MetaL_1$ et $MetaL_2$, ainsi que des exemples de concepts dans $MetaL_2$.

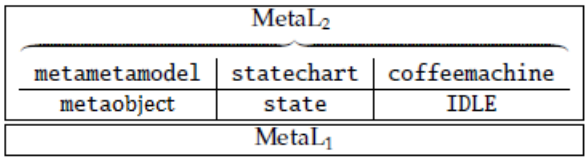

FIGURE 4.1 – Architecture/couches composant MetaDONE, tiré de [Englebert, 2007]

4.3 GraSyLa

GraSyLa est « un langage fournissant un mapping entre un DSLM défini par un méta-modèle et une syntaxe concrète » [Englebert et Magusiak, 2013]. GraSyLa est lui-même un DSLM, il est implémenté dans MetaDONE en définissant son méta-modèle à l'aide de $MetaL_2$. Ce DSLM permet de décrire la manière dont la visualisation d'un méta-modèle doit être construite. Les informations quant à cette construction sont décrite par l'intermédiaire du langage GraSyLa dans son modèle concret, qui est un script pouvant être interprété par MetaDONE en l'appliquant au modèle concret du méta-modèle pour lequel le script a été écrit. Les caractéristiques de GraSyLa, tirées de [Englebert et Magusiak, 2013], sont les suivantes :

polymorphique : le langage GraSyLa « peut être utilisé pour décrire différents types de notations concrètes avec une syntaxe unique et intégrée » ;

générique : son interpréteur est implémenté comme un *framework* générique pouvant être facilement étendu avec des plugins ;

déclaratif : par conséquent, il possède les avantages liés à ce paradigme telle que ses facilités de maintenance, d'analyse, *etc.* Les notations DSLM sont définies sous forme d'équations. GraSyLa est également ouvert à d'autres paradigmes et peut être enrichi à l'aide de scripts écrits à l'aide de Groovy [Koenig *et al.*, 2007] ;

modulaire : afin de faciliter sa factorisation, il peut importer d'autres spécifications ;

régulier : les expressions ont le même pattern syntaxique ;

extensible : des nouvelles constructions fournies par des plugins externes peuvent être employées.

4.4 Méta-méta-modèle

Le méta-méta-modèle de MetaDONE est plus expressif que celui des autres outils CASE. Un concept est défini par un seul littéral, par conséquent ce dernier référence plusieurs objets de type différents mais se rapportant à ce même concept. Par exemple, un objet « person » peut tout autant représenter une classe de ce type, qu'une table SQL, qu'un *use-case*, etc. Le méta-méta-modèle est utilisé pour créer des méta-modèles, sa représentation n'appartient pas aux domaines standards tel qu'EMF ou MOF.

La figure 4.2 illustre l'application de MetaL à un langage de machine à états. Trois niveaux sont représentés, le méta-méta-modèle ($M3$), le méta-modèle ($M2$) du diagramme d'états et son instance concrète ($M1$). Une portion du méta-modèle d'un diagramme d'état est représenté dans $MetaL_2$ dans le compartiment central de la figure. Le méta-modèle du diagramme d'état est défini comme l'agrégation des méta-objets *état* et *transition*. *Transition* est également un méta-rôle. Ce méta-modèle sert de *pattern* à la création des modèles concret, comme « coffeemachine » illustré sur la figure [Englebert et Heymans, 2007].

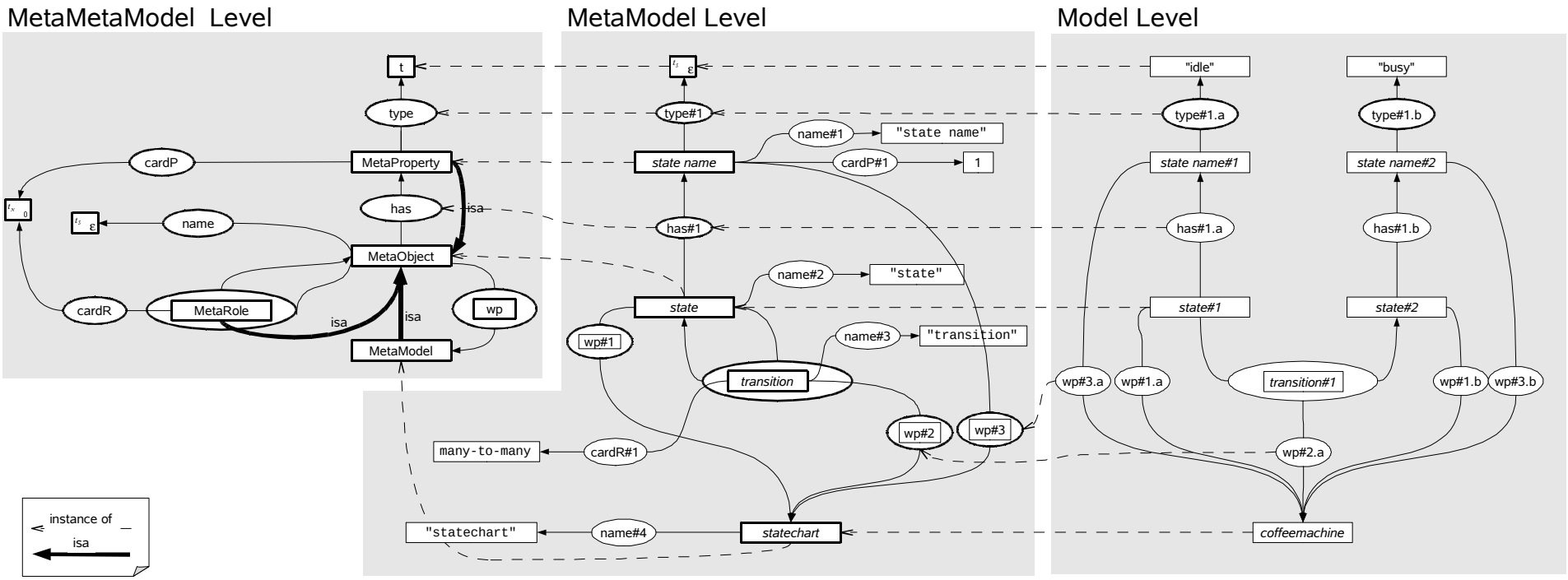

FIGURE 4.2 – MetaL appliqué au diagramme d'état, composé de trois niveaux (issu de [Englebert et Heymans, 2007]).

Chapitre 5

Exigences du problème

Ce chapitre expose l'importance de définir correctement les exigences du problème comme les caractéristiques requises pour les transformations de modèles et du langage destiné à les effectuer dans la section 5.2. Ensuite, la section 5.3 présente les caractéristiques recommandées pour les outils et langages de transformations, en passant par les exigences fonctionnelles et non fonctionnelles. Le cas d'étude servant à illustrer les transformations de modèles et de fil conducteur pour notre benchmark et choix d'outil est décrit à la section 5.4. Pour terminer, la section 5.5 expose les exigences d'un point de vue architectural, avec MetaDONE comme sujet principal.

5.1 Introduction

Il n'existe, à l'heure actuelle, il n'existe pas langages de transformation pouvant être considéré comme étant le meilleur. Afin de déterminer quel langage est le plus approprié pour un problème donné, il faut donc répondre à un ensemble de questions cruciales [Mens et Van Gorp, 2006] :

- Quel va être le type de transformations exécuté ?
- Quelles sont les caractéristiques importantes pour la transformation de modèles ?
- Quelles sont les caractéristiques importantes du langage ou outil devant mettre en place la solution ?
- Quels sont les mécanismes pouvant être utilisés pour la transformation de modèles ?

En effet, sans cette étape, les chances d'atteindre l'objectif sont fortement réduites, voire anéanties. Comme l'exprime Hubert : « les exigences devant être remplies par le langage de transformation doivent être claires, le cas échéant, le résultat se révélera probablement inefficace et rendra même impossible la résolution de certains problèmes de transformations » [Huber, 2008].

La définition des exigences est une étape cruciale. Les fonctionnalités requises doivent être clairement définies, tout en tenant compte des caractéristiques recommandées pour un outil aussi bien pour les exigences fonctionnelles que non fonctionnelles. Aussi, il est important de tenir compte des contraintes imposées dans un contexte particulier comme c'est ici le cas avec l'outil de méta-modélisation. Pour terminer, déterminer un cas d'étude peut aider à la définition et délimitation des exigences.

5.2 Fonctionnalités requises

Dans le cadre de ce travail, nous allons principalement nous intéresser aux outils permettant de mettre en œuvre des transformations de modèle vers modèle. Notre motivation principale est la résolution du cas d'étude présenté ultérieurement dans ce chapitre. Nous allons tout d'abord exposer dans la suite de ce chapitre les caractéristiques recommandées pour les outils et langages de transformations. Nous allons entre autre, jauger de l'étendue de leurs caractéristiques et veiller à la maturité du langage, sa maintenabilité, son support, sa concision, son interopérabilité, etc.

L'interopérabilité est ici un critère important, l'objectif est de trouver un outil adéquat pouvant être intégré à terme dans l'outil de méta-modélisation. L'échange d'informations dans un format compatible entre l'outil et le MetaCASE doit pouvoir être mis en place par divers moyens. L'outil doit supporter des technologies standards telles que MOF ou EMF et éventuellement pouvoir être étendu afin d'en supporter d'autres, comme celle de MetaDONE. Aussi, la communication doit être établie entre ces deux entités, l'outil doit donc offrir ce service tel que la possibilité d'être invoqué de manière externe par une API (*Application Programming Interface*) fournie, ou être incluse dans un autre outil, afin de pouvoir automatiser l'exécution des transformations, etc.

5.3 Caractéristiques recommandées pour les outils et langages de transformations

Il existe différentes caractéristiques jugées importantes pour les outils et langages de transformation [Tamura *et al.*, 2010] :

- automatisation ;
- mécanismes pour l'évolutivité et la réutilisation ;
- variabilité et testabilité ;
- formel, avec une fondation solide ;
- favorisation des approches déclaratives.

Aussi, plusieurs caractéristiques fonctionnelles et non-fonctionnelles ont été identifiées de manières plus précises et détaillées et sont recommandées pour un outil de transformation de modèles, [Tamura *et al.*, 2010] et [Mens et Van Gorp, 2006]. À noter qu'il existe une démarche similaire quant au choix d'un outils de transformations de graphes [Mens *et al.*, 2006].

5.3.1 Les exigences fonctionnelles

Les exigences fonctionnelles peuvent être classées en exigences fortes et faibles en fonction de l'importance des caractéristiques selon notre cas d'étude. Les premières sont essentielles pour réaliser des transformations d'une bonne qualité. Tandis que les secondes comme la bidirectionnalité ou la manipulation de transformation de modèles incomplets ne sont pas obligatoires pour la réalisation d'une transformation et pas nécessaires dans notre cas.

Exigences fortes

créer/lire/mettre à jour/supprimer les transformations (Create/Read/Update/Delete (CRUD)) : l'outil doit permettre d'effectuer toutes les opérations de base du cycle de vie d'une transformation ;

customisation ou réutilisation des transformations : cela peut être mis en place, entre autre par, l'utilisation de *templates*, la paramétrisation de fonctions ou l'héritage ;

grouper, composer et décomposer les transformations : permet d'augmenter la lecture, modularité, évolutivité et maintenabilité du langage de transformation. La subdivision d'une transformation complexe en plusieurs plus petites doit nécessiter un mécanisme de contrôle spécifiant la manière dont ces dernières doivent être combinées ;

capacité de tester, valider et vérifier les transformations : le but est de vérifier que le comportement des transformations est correct ;

spécifier des transformations génériques : évite la duplication de code ;

traçabilité et la propagation des changements : des mécanismes doivent être fournis par l'outil pour rendre cela possible.

Exigences faibles

spécifier des transformations bidirectionnelles : permet d'utiliser les règles de transformation pour la génération de la cible à partir de la source et également l'inverse, permettant ainsi la définition d'un nombre moins élevé de règles ;

suggestion d'application de transformations : « pour certains scénarii d'application, les outils dédiés devraient suggérer le type de transformation le plus approprié » [Mens et Van Gorp, 2006] ;

manipuler des modèles incomplets ou incorrects : comme expliqué par Mens, des modèles incomplets et incorrects doivent parfois être transformés. En effet, les modèles interviennent très tôt dans le cycle de développement, quand les besoins ne sont pas encore clairement et totalement définis ;

vérifier et garantir la correction des transformations : d'une part, il faut garantir la correction et complétion syntaxique. D'autre part, la correction sémantique ainsi que des propriétés telles que la terminaison (la transformation doit toujours mener à un résultat et par conséquent se terminer) et confluence (le résultat doit être unique) ;

5.3.2 Les exigences non-fonctionnelles

interopérabilité : l'outil doit pouvoir être facilement intégré avec d'autres outils. Cette exigence est ici importante car il sera amené à être intégré dans un MetaCASE ;

convivialité et utilité ;

verbosité *versus* concision : un compromis doit être trouvé pour fournir un équilibre. Si le langage est trop concis, n'offrant que trop peu de constructions syntaxiques, les transformations complexes seront difficiles à construire. De ce fait, le langage doit être assez verbeux que pour faciliter son utilisation ;

performance et évolutivité : l'outil doit être capable de manipuler de larges et complexes modèles/transformations ;

extensibilité : la facilité avec laquelle l'outil peut être étendu avec de nouvelles fonctionnalités ;

acceptation par la communauté d'utilisateurs : un outil apprécié et utilisé par les utilisateurs ne sera pas nécessairement celui se révélant le plus intéressant d'un point de vue théorique ;

utilisation de standards : l'outil doit permettre l'utilisation de standard tels que XML, UML, MOF et XMI. Cette exigence est ici moins importantes que l'interopérabilité.

5.4 Présentation théorique du cas d'étude

Le problème servant d'étude de cas pour illustrer les transformations de modèles dans une optique plus réaliste et qui aidera à la décision sur le choix d'un langage de transformation de modèles est le suivant :

Les transformations seront exprimées dans le domaine des DSL. L'objectif de la transformation est de tenter d'adapter une application conçue de manière abstraite (*AbstractProgram*) à la configuration d'un périphérique particulier (*CompliantProgram*), le méta-modèle est illustré à la figure 5.1. Il s'agirait d'une transformation endogène, qui copierait les artefacts où les modèles source et cible sont distincts.

Un programme peut donc être de type *AbstractProgram* ou *CompliantProgram*. Les dimensions et composantes du premier ne sont pas spécifiques à un périphérique (*Device*) donné, contrairement au second. Le programme est conçu comme une collection d'activités (*Activity*), à priori, indépendantes. Nous entendons par activité, une unité de présentation à l'écran pouvant nécessiter une technique de scrolling si celle-ci déborde de l'écran. Chacune de ces activités est régie par un ensemble de tâches (*Task*) qui peuvent ou non être dépendantes d'autres dans le temps. Par exemple : la tâche « taskDisplayMeeting » doit être précédée de « taskLogin ». Idéalement, les deux tâches sont regroupées dans la même activité. Une tâche est composée d'un ensemble de composants d'interface graphique (*Widget*), elle peut être définie comme « un objectif à atteindre par l'utilisateur à l'aide d'un système interactif » [Normand, 1992]. Un widget est un élément de base d'une interface graphique permettant d'afficher de l'information ou avec lequel l'utilisateur peut interagir avec le système [Wikipédia, 2015] comme une zone de texte (TextField) ou une image (Image). Un widget peut être composé d'autres widgets, par exemple un panel peut contenir des zones de texte. Ces composants possèdent un *Layout*, permettant de définir la manière dont les composants seront disposés sur l'interface graphique. Les activités sont indépendantes les unes des autres tandis que certaines peuvent se voir imposer une exécution en séquence.

La transformation a pour objet d'adapter le nombre de tâches par activités afin d'optimiser l'espace du périphérique et de diminuer le scrolling, afin de confondre l'activité avec l'écran physique (*CompliantProgram*). À l'inverse, une transformation duale qui agrégerait des activités entre elles afin de tirer le meilleur parti de la surface des tablettes pourrait également être imaginée. La première transformation peut bien entendu échouer s'il est impossible de réduire une activité. Si nous divisons des tâches reliées par une « précédence », cela doit entrainer une séquence pour autant que la séquence soit compatible.

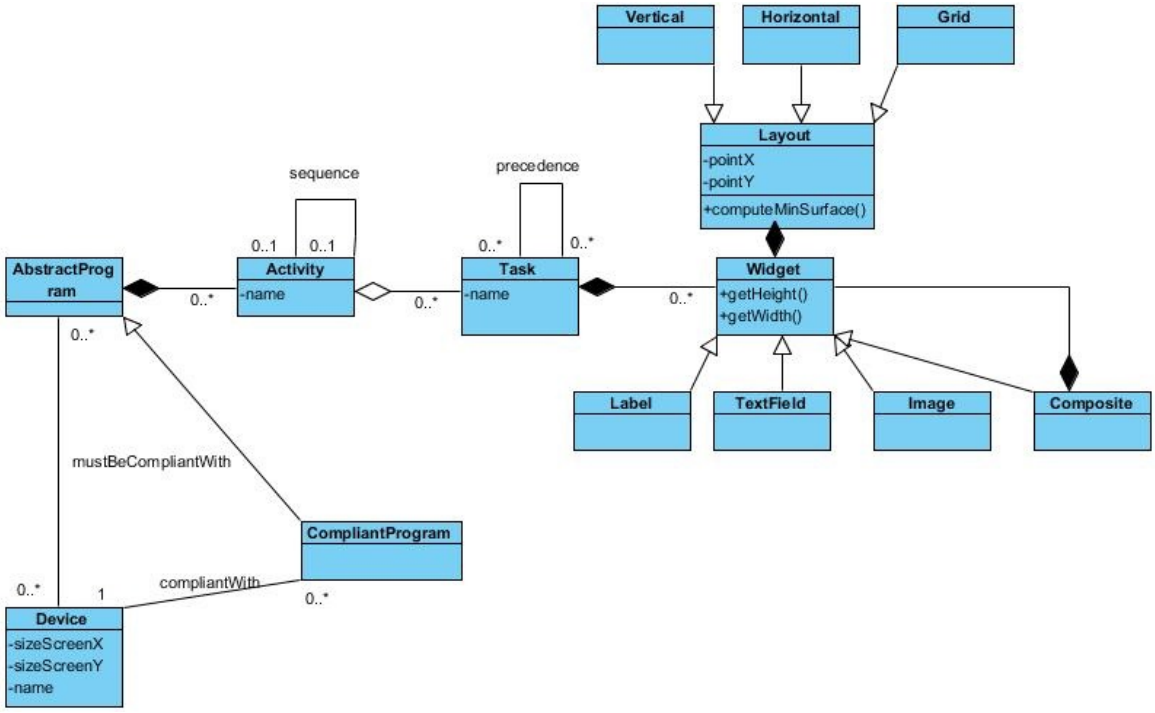

FIGURE 5.1 – Représentation UML du contexte du cas d'étude

5.5 Vision architecturale

MetaDONE est fondé sur le *framework* OSGi Equinox d'Eclipse. MetaDONE est voué à encore évoluer, le *framework* sur lequel il repose n'est donc pas une contrainte. Il n'est ainsi pas impossible qu'à l'avenir un autre *framework* d'OSGi soit employé.

D'un point de vue « interopérabilité », MetaDONE pourrait être adapté pour intégrer l'outil de transformation de modèles choisi. Par exemple, si l'outil permet l'importation et l'exportation de modèles au format XMI, il est tout à fait possible de développer un *plugin* pour importer et exporter des fichiers appartenant à ce même format dans MetaDONE. Par conséquent, les informations pourraient être exploitées par les deux outils. Actuellement, MetaDONE est déjà capable d'importer des méta-modèles de tel qu'EMF à l'aide d'un *loader*. Aussi, il pourrait être adapté afin de pouvoir utiliser une API fournie par un outil pour communiquer avec ce dernier.

En conclusion, il n'y aucune réelle contrainte d'un point de vue architectural identifiée.

5.6 Conclusion

L'importance de la définition des exigences du problème est ici mise en exergue. Nous pouvons constater qu'il existe une multitude de caractéristiques pouvant être mise en place par un outil de transformation de modèles. Cependant, la tâche de sélectionner les caractéristiques devant être offerte par un outil est simplifiée grâce aux travaux existants formulant des recommandations quant à ces celles-ci. À la lumière de ces faits, nous pouvons classer les outils en fonction de nos besoins et de ces recommandations afin de poursuivre notre travail de sélection, tout en tenant compte des contraintes imposées.

Chapitre 6

Pré-sélection d'outils actuels

Actuellement, nous n'avons pu écarter que certaines approches M2M. Pour départager les approches subsistantes, nous allons tout d'abord choisir un outil pour chacune d'elles afin de les évaluer et de les comparer plus en détails dans les chapitres ultérieurs à l'aide de notre cas d'étude.

La section 6.1 introduit le sujet à l'aide des pratiques courantes et les travaux existants en la matière de comparaisons d'outils à l'aide de benchmarks.

Ensuite, la section 6.2 explique la méthodologie appliquée pour pré-sélectionner des outils efficaces. Par après, la section 6.3 présente individuellement chaque candidat sélectionné, d'un point de vue de leur fonctionnement et caractéristiques générales. Aussi, des tableaux détaillés et comparatifs des outils y sont dressés.

Pour terminer, une conclusion vient clôturer ce chapitre 6.4.

6.1 Introduction

La pré-sélection d'outils n'est pas tâche aisée. Il faut préalablement avoir correctement défini les exigences et dimensions du problème, ce qui a été réalisé dans le chapitre précédent. Ainsi, nous avons déterminé les caractéristiques importantes qu'un outil de transformation doit posséder et nous pouvons pré-sélectionner des outils sur cette base. Pour ce faire, nous allons nous baser sur la documentation des outils et sur des travaux mettant en place des comparaisons d'outils de manière objective, avec des données représentative à l'aide de benchmark par exemple.

Nous pouvons citer quelques références en la matière comme celles traitant des compétitions d'outils de transformations de modèles (*Transformation Tool Contest* ou TTC). Celle de 2011 est décortiquée dans le compte rendu de Jakumeit [Jakumeit *et al.*, 2014]. Son but est de fournir une assistance dans le choix d'un outil répondant aux besoins d'ingénieurs logiciels. Cette compétition a lieu tous les ans depuis plusieurs années et mène à la publication de divers papiers sur les différents outils participants. Les langages évalués appartiennent aux approches basées sur les transformations de graphes et hybrides. Chaque outil est présenté et mis en place pour résoudre un benchmark de taille réduite de type « Hello World », consistant en un ensemble de tâches illustratives, mettant en œuvre les opérations basiques réalisables sur des modèles(CRUD). La simplicité de ces tâches permet de mettre en évidence les différences entre chaque outil mais cela n'offre pas la possibilité de tester leurs réelles forces et limites. [Mazanek, 2011] en effectue une présentation détaillée.

Les caractéristiques de chaque outil y sont exposées, tel que leur facilité d'apprentissage, les IDEs compatibles, l'organisation des transformations et leur réutilisation, leur domaine d'application, leur maturité, leur concision, etc. Cette synthèse nous permettra en partie de classifier certains outils y étant présentés.

Aussi, le travail de [Kolahdouz-Rahimi *et al.*, 2014] évalue à l'aide de la norme ISO 9126 (*International Organization for Standardization*) cinq outils avec leur mise en place dans différents scénarii afin de trouver le plus approprié dans le cas de refactoring de modèles.

Pour terminer, la thèse de Schubert [Schubert, 2010] compare différents outils et les évalue grâce à cette même norme ISO afin de trouver le plus adéquat dans le cadre de l'ingénierie qualité d'UML.

6.2 Méthodologie appliquée

Nous avons pré-sélectionné des outils en nous basant sur les travaux présentés ci-avant mais également sur la documentation spécifique de chaque outil. Tentons maintenant d'appliquer une méthodologie similaire pour tirer nos propres conclusions.

Dans un premier temps, nous avons écarté des outils en fonction de leur applicabilité et spécialisation car nous tentons d'en sélectionner les plus généraux et les moins restrictifs. Prenons par exemple le langage Edapt, il peut être écarté car il est principalement conçu pour la migration de modèles et l'évolution incrémentale de méta-modèles. Aussi, il n'est pas adapté pour les transformations où les méta-modèles source et cible sont totalement différents ou identiques [Jakumeit *et al.*, 2014], ce qui rend déjà ce langage très restrictif.

Ensuite, nous avons été attentifs aux caractéristiques possédées par les outils. Les candidats que nous retenons sont ceux possédant le plus de caractéristiques recommandées (exigences fonctionnelles et non fonctionnelles) présentées dans le chapitre précédent.

En outre, nous avons également évalué les outils d'un point de vue de leur maturité, leur facilité d'apprentissage et leur concision.

Nous avons constaté dans [Jakumeit *et al.*, 2014] que beaucoup de langages n'avaient pas de réel équilibre entre ces différentes caractéristiques. Par exemple VIATRA2 et Henshin obtiennent de mauvais résultats quant à leur concision sur des tâches basiques et leur facilité d'apprentissage est moyenne. Le constat contraire est effectué pour GReTL et UML-RSDS. Ceux possédant le meilleur équilibre et exposant des caractéristiques intéressantes sont ici Epsilon et GrGen.net.

Pour conclure, étant donné qu'il existe un grand nombre d'outils existants, notre démarche a tout d'abord tenté d'éliminer des outils de façon itérative en fonction de critères généraux exposés précédemment comme un manque de maturité, une applicabilité restreinte, des fonctionnalités limitées, etc. Ensuite, d'autres outils ont été pré-sélectionnés en fonction de leurs caractéristiques exposées, explicitées ci-après.

6.3 Liste des candidats à évaluer

Nous allons présenter les candidats pré-sélectionnés, ATL, Epsilon, GrGen.NET et Prolog et justifier les raisons de leur choix. Ensuite, les caractéristiques des outils seront analysées.

6.3.1 ATL

ATL est un langage de type hybride, combinant l'approche de type déclarative et impérative, s'appuyant sur le formalisme d'OCL. Bien qu'il soit hybride, le style à préférer est déclaratif. L'impératif n'est recommandé que pour réaliser des opérations complexes. Il possède son propre compilateur et machine virtuelle. Il a été créé par l'Institut National de Recherche en Informatique et en Automatique (INRIA) en réponse au langage QVT proposé par l'OMG. Ce langage est spécifié pour les transformations de modèles en général. ATL a été adopté comme langage et outil de support pour les transformations de modèles.

Principales composantes du langage

Nous allons présenter brièvement les principales composantes d'ATL, en nous basant sur le travail de Jouault présentant le langage et datant de 2008 [Jouault *et al.*, 2008]. Les transformations sont définies par des modules ATL. Un module est composé d'une entête, d'une section de gestion des importations, d'« helper » et de règles de transformation. L'entête fournit le nom de la transformation et déclare les modèles cibles et sources.

Il existe différents types de règles déclaratives (« matched rules ») :

standard rules : elles sont appliquées chaque fois qu'une correspondance est trouvée dans le modèle source ;

lazy rules : elles sont invoquées par une autre règle et appliquées à une correspondance trouvée, à chaque invocation de nouveaux éléments sont crées pour le modèle cible ;

unique lazy rules : elles possèdent le même fonctionnement général que les « lazy rules », excepté qu'elles ne sont appliquées qu'une seule fois à une correspondance donnée, aucun nouvel élément ne sera crée pour le modèle cible.

De plus, il y a deux types de construction impératives :

called rules : règles pouvant être invoquées par d'autres dans un style procédural ;

les blocs d'actions : séquences de déclarations impératives pouvant être employées dans des « called rules » ou « matched rules ».

Il existe également des règles abstraites permettant de mettre en place un héritage entre les règles :

abstract rules : règles pouvant étendre une autre règle à l'aide du mot clé « extends ». Cette dernière héritera du comportement de la règle abstraite.

Quant aux *helpers*, ils peuvent être vus comme des méthodes, ils rendent possible la factorisation de code ATL et peuvent être utilisés à divers endroits dans la transformation. Ils peuvent être de type attribut ou opération et être définis dans le contexte d'un élément du modèle ou du module. Ce sont des constructions impératives, tout comme les blocs *do* pouvant être définis au sein d'une règle.

Les règles ATL peuvent être invoquées de manière implicite, explicite ou par héritage. L'ordre d'exécution des règles est gérée automatiquement. Les conditions d'application de règles sont supportées à l'aide de prédicats dans la clause *from* [Huber, 2008]. Les transformations s'effectuent sur des modèles sources en lecture seule et produisent des modèles cibles

en écriture seule. Aucun changement dans les modèles n'est possible au cours de l'exécution. Aussi, le modèle cible ne peut être parcouru, contrairement au modèle source.

Caractéristiques du langage

D'un point de vue caractéristiques, la dernière version disponible est la 3.6.0, datant de janvier 2015. Cette version est compatible avec les dernières versions d'Eclipse. En moyenne, une version stable par an est publiée. Aussi, des releases contenant des corrections de bugs ainsi que des nouveautés paraissent plus fréquemment, c'est ici un bon indicateur de la maintenabilité et continuité de l'outil. ATL expose plusieurs caractéristiques importantes explicitées dans le chapitre 5 pour les transformations de modèles, reprises dans des tableaux *infra*.

De plus, ATL suscite un grand engouement de la part de la communauté. Cela peut se vérifier par le nombre de signalements de bugs et activités sur le forum du projet. Il est aussi utilisé dans le monde industriel par plusieurs partenaires comme cela peut être constaté sur son site internet [1]. Ce dernier rend également disponibles publiquement plus d'une centaines de projets couvrant de nombreux domaines. Quelques *screencasts* et tutoriels y sont également disponibles Une quantité considérable d'articles le présentent ou l'évaluent.

Tout cela représente une source d'informations intéressantes pour un développeur ou pour aider à choisir un outil de transformation de modèles. Aussi des exemples de transformations existantes donnent une impression positive sur les capacités de l'outil comme le projet « CPL2SPL », transformant un langage DSL vers un autre, avec des modèles ayant un méta-modèle important en termes de taille. Ou encore, nous pouvons citer le projet « Disaggregation », permettant de désagréger une classe UML en plusieurs autres sous certaines conditions, ce qui est similaire à notre cas d'étude. En effet, sous certaines conditions, nous allons devoir diviser des *Activity* en plusieurs comme expliqué en détails dans le chapitre 8. Le problème sur la désagrégation de classes UML est extrait du catalogue de K. Lano [Lano, 2000], centré sur les transformations sur des classes UML et diagrammes d'états, plusieurs autres projets développés avec ATL en sont tirés.

ATL est interopérable avec Eclipse, un compilateur et débugger y sont intégrés et l'éditeur propose une auto-complétion. Il peut être intégré de façon transparente à une application avec l'utilisation de code Java à l'aide de l'initiative d'« ATL launcher » [Cabot, 2015], contenant toutes les dépendances requisent pour l'employer de manière autonome. De surcroit, il utilise des standards comme la technologie EMF.

En outre, il a été évalué lors de compétition d'outils de langages de transformations et dans des travaux tels que [Bosems, 2011], où il se révèle ici plus performant que QVT pour le cas d'étude donné.

De plus, il est évalué dans [Kolahdouz-Rahimi *et al.*, 2014] où il n'est pas très bien classé contrairement à GrGen.NET. Ceci est dû au fait que le cas d'étude de ce travail est centré sur la migration, entrainant ainsi de mauvaises évaluations pour les critères spécifiques au problème. Toutefois, ce travail reste une référence pour juger de ses caractéristiques plus générales.

Pour terminer, il a aussi été évalué dans [Schubert, 2010], permettant d'éclaircir les dimensions de l'outil d'un point de vue de son utilité, sa portabilité, son efficacité, la continuité

1. http ://www.eclipse.org/atl. Date : 02/04/2015

du projet, sa maintenabilité et ses fonctionnalités. Toutes ces évaluations et faits permettent de conclure qu'ATL est un bon candidat.

6.3.2 ETL

Principales composantes du langage

Epsilon est un composant du Eclipse Modeling Project. ETL fait partie de la famille de langages proposée par Epsilon, il met en place une approche hybride en proposant des règles déclaratives et des blocs de type impératif. Il est spécifié pour effectuer de la gestion de modèles et des transformations en général.

Un de ses principaux avantages est qu'il est interopérable avec plusieurs langages additionnels spécialisés pour des tâches de transformations de modèles spécifiques. La motivation principale ayant conduit à l'élaboration de ce langage est l'existence de nombreux langages pouvant être qualifiés d'« isolés », spécifiques à certaines tâches, amenant un utilisateur à devoir apprendre différents langages et les combiner pour accomplir différentes tâches spécifiques sur des modèles [Kolovos *et al.*, 2008]. En effet, dans l'IDM il n'est pas rare dans le milieu de l'IDM de vouloir combiner différentes manipulations sur les modèles telles qu'une validation, une transformation pour terminer par une génération de code en langage de programmation voire une migration des modèles par la suite en cas d'évolution du business.

Epsilon est composé des différents langages suivants ([Kolovos *et al.*, 2006a]) :

Epsilon Object Langage (EOL) : destiné à la gestion de modèles et des transformations en général, il peut aussi être vu comme un template pour les autres langages car ceux-ci sont construit à partir d'EOL. Il peut être utilisé pour les problèmes non-spécifiques aux autres langages. Il est basé sur le langage OCL [Kolovos *et al.*, 2006b] ;

Epsilon Comparison Language (ECL) : destiné à la comparaison de modèles, homogènes ou hétérogènes ;

Epsilon Merging Language (EML) : destiné à la fusion de modèles, homogènes ou hétérogènes ;

Epsilon Validation Language (EVL) : destiné à la validation de modèle, il existe des similarités entre ce langage et OCL ;

Epsilon Wizard Language (EWL) : destiné au *refactoring* de modèles. Il permet aussi des transformations *in-place* interactives ;

Epsilon Generation Language (EGL) : destiné à la génération de texte à partir de modèle (M2T) ;

Epsilon Flock : destiné à la migration de modèles, il met en place une approche hybride et peut également migrer des modèles spécifiés dans des technologies de modélisation différentes, le tout de manière plus concise que la plupart des outils existants pour cette tâche [Rose *et al.*, 2010].

Nous allons à présent détailler les composantes d'ETL, ses transformations sont organisées sous forme de modules pouvant contenir des règles de transformation ainsi que des opérations écrites à l'aide du langage EOL. Chaque règle spécifie en paramètre un élément source et un ou plusieur(s) élément(s) cible(s). Les règles peuvent être déclarées de façon *abstract*, *lazy* et ou *primary*. Leur signification est identique à celle d'ATL.

Une règle peut étendre un nombre illimité d'autres règles. Elle peut aussi définir des *guards* pour limiter son applicabilité ainsi que des pre et post-conditions. Chaque règle contient un bloc *body* contenant du code EOL définissant la façon dont les éléments sources vont être transformés en éléments cibles.

La figure 6.1 illustre la syntaxe abstraite d'ETL.

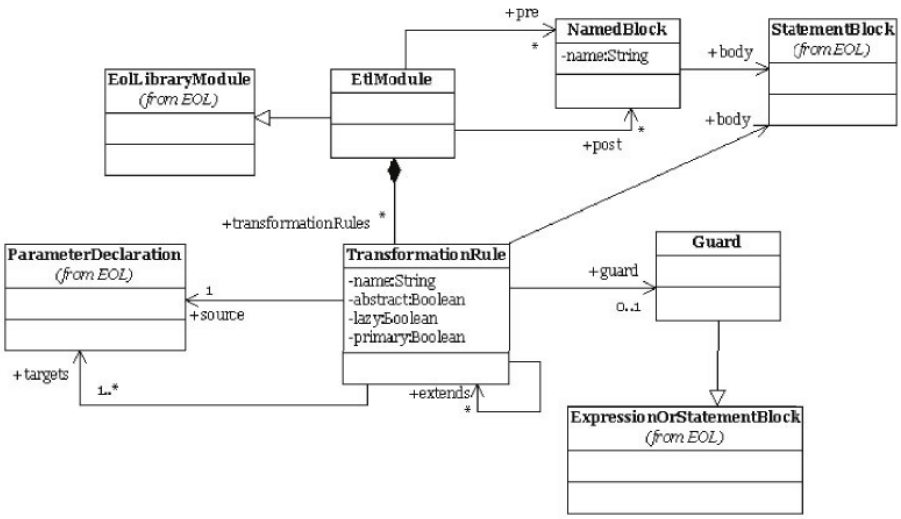

FIGURE 6.1 – Syntaxe abstraite d'ETL, tiré de [Kolovos *et al.*, 2008]

Enfin, un nombre arbitraire de modèles sources peut être transformé vers un nombre arbitraire de modèles cibles dans ETL. L'utilisateur peut indiquer pour chaque modèle impliqué dans la transformation s'il désire préserver son contenu ou non.

Caractéristiques du langage

Le panel de langages composant Epsilon pourrait prêter à penser que son apprentissage et utilisation sont laborieux mais chacun d'eux est basé sur le même langage de base, EOL. De ce fait, les langages comportent beaucoup de similarités simplifiant ainsi son apprentissage. Ce dernier est également facilité par des *screencasts* et une vingtaines de projets en exemple. Aussi, plusieurs articles traitant d'Epsilon et un livre en ligne sont également disponibles pour apprendre à manipuler les différents langages.

En outre, selon la communauté Epsilon, 35 projets *open-source* et 16 sociétés industrielles l'utilisent ainsi que plusieurs universités à des fins pédagogiques. La plupart de ces informations sont disponibles de manière plus détaillées sur le site de du projet Epsilon [2]. Un forum très actif s'y trouve également.

D'un point de vue interopérabilité, tous les langages sont supportés par Eclipse où un debbuger et compilateur sont disponibles. Un jar autonome est également mis à disposition

2. https ://eclipse.org/epsilon/. Date : 02/11/2014

afin qu'Epsilon ne soit pas dépendant d'un IDE, impliquant le fait qu'il peut être intégré à un outil de cette façon.

De plus, un avantage est que les transformations effectuées avec Epsilon sont indépendantes de la technologie utilisée pour représenter ou stocker les modèles exploités. Divers types de formats sont supportés tels qu'Ecore, XMI, XML, CVS, etc. Aussi, Epsilon peut être étendu afin de supporter de nouveaux types supplémentaires *via* le développement d'un driver spécifique par l'utilisateur. Par conséquent, il a le potentiel d'offrir une adéquation sémantique pour les méta-modèles de MetaDONE.

De surcroît, il est possible d'appeler du code Java depuis un programme Epsilon, permettant ainsi de faire face à des tâches non supportées par défaut par les langages ou ré-utiliser du code existant.

Pour terminer, Epsilon arrive en tête des votes des utilisateurs pour la résolution du cas d'étude dans la TTC 2011, les critères d'évaluation sont la compréhension du langage, sa concision et sa complétude. Il a également été évalué dans [Schubert, 2010], permettant ainsi d'éclaircir divers points sur les critères de l'outil.

6.3.3 GrGen.NET

GrGen.NET (*Graph Rewrite Generator*) est un système de réécriture de graphes (*graph rewrite system*) à usage général. Il combine des règles déclaratives basées sur les transformations de graphes avec des caractéristiques impératives afin de faire face aux transformations complexes. Il a été développé à l'université allemande de Karlsruhe et rendu public en 2003. Il est spécifié pour effectuer de la réécriture de graphes en général et est de type Turing-complet. C'est l'un des outils les plus rapides de catégorie et possédant une expressivité compétitive. Tous ces aspects rendent GrGen.NET adéquat à la transformation de modèles [Gelhausen *et al.*, 2008].

Principales composantes du langage

Tout d'abord, le langage dédié GRS (*Graph rewrite sequences*) est mis à disposition pour contrôler l'application des règles. Dans GrGen.NET, le mécanisme de contrôle des règles doit être géré de manière explicite. Il peut être qualifié de système de réécriture de graphes contrôlable, qui « est la clé pour rendre les outils de transformation de graphes utilisables à l'échelle industrielle », [Syriani et Vangheluwe, 2009].

GrGen.NET est un système de réécriture de graphes et non un outil de transformation de modèles. Ainsi, il supporte les fichiers GXL et GRS mais ne permet par l'importation de méta-modèles Ecore directement. Cependant, il offre la possibilité *via* un filtre de générer un modèle de graphe dans un format spécifique à GrGen.NET (.gm) à partir d'un fichier Ecore, en faisant correspondre ses éléments dans les types correspondants de GrGen.NET. Ensuite, le fichier XMI conforme à ce fichier Ecore et du fichier *.gm* généré peut être importé pour servir de modèle source et ainsi permettre la transformation, [Mazanek *et al.*, 2010].

La figure 6.2 donne une vue globale des composants et artifacts du système ainsi que les différents fichiers impliqués.

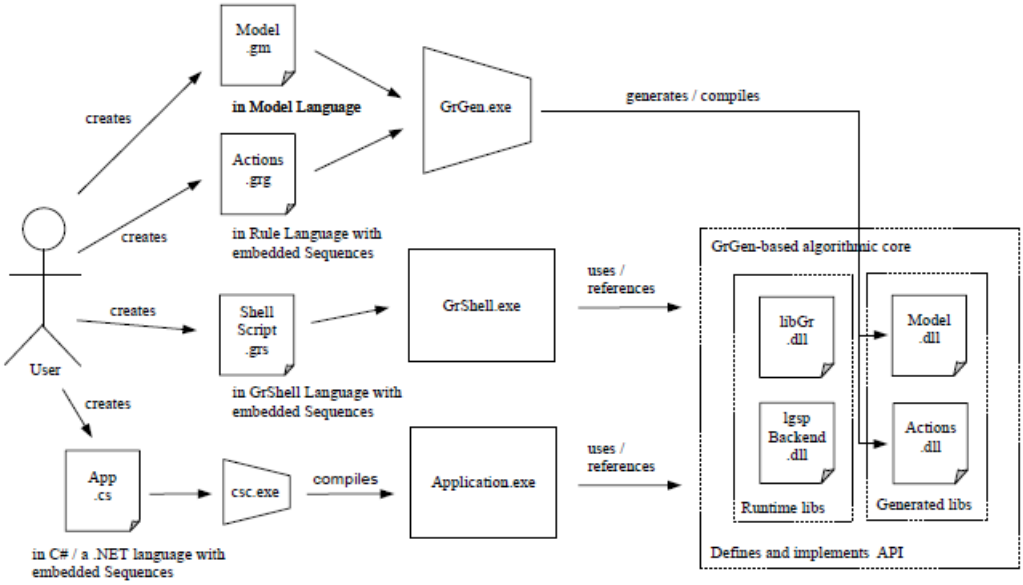

FIGURE 6.2 – Composants et *artifacts* du système de GrGen.NET

Ensuite, GrGen.NET supporte des règles fondées sur des *patterns* (*pattern-based rules*), constituées de *patterns left-hand-side* (LHS) et *right-hand-side*. « Un *pattern* est décrit par un graphlet, une spécification de nœuds et d'arcs connectés » [Jakumeit Edgar, 2014]. La partie LHS représente un *pattern* de pré-condition devant être vérifié dans le graphe hôte. La partie représente la post-condition après que la règle ait été appliquée sur le sous-graphe ciblé par la partie LHS [Syriani et Vangheluwe, 2013]. La partie de réécriture (*rewrite part*) permet d'effectuer des opérations CRUD sur les éléments du graphe. Elle peut être *modify-mode*, modifiant le sous-graphe ciblé ou *replace-mode*, spécifiant un nouveau sous-graphe.

GrGen.net met en place la possibilité d'utiliser le mécanisme de conditions d'applications négatives (NAC) afin d'exprimer des contraintes sur le graphe devant être transformés, [Blomer *et al.*, 2011].

La figure 6.3 illustre le fonctionnement des LHS, RHS et NAC. Cet exemple est tiré du travail de [Syriani et Vangheluwe, 2013], plus de détails peuvent y être trouvés ainsi que la définition du méta-modèle « PacMan ». Ici, la règle illustrée exprime le mouvement de l'objet *PacMan* vers la droite. Sa position initiale est en LHS et la finale est celle en RHS. La règle NAC empêche l'application de cette règle si un objet *Ghost* est déjà présent dans cet espace.

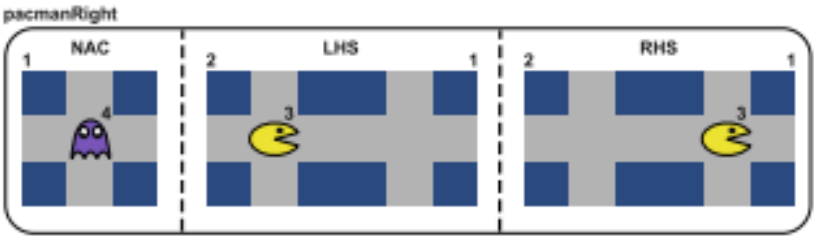

FIGURE 6.3 – Illustration du fonctionnement d'une règle [Syriani et Vangheluwe, 2013]

Les *patterns* complexes sont composés de *patterns* imbriqués (*nested pattern*) pouvant définir une structure *optionnel*, *multiple*, *iterated* ou des conditions d'application *negative* ou *independent* et de définitions et usages de *subpattern*. Les *subpatterns* peuvent être mis en place pour factoriser un *pattern* commun récurrent, améliorant ainsi la réutilisation [Jakumeit Edgar, 2014].

De plus, GrGen.NET offre des opérations et syntaxes permettant d'améliorer sa concision. Par exemple, il est possible de modifier le type d'un modèle vers un autre *via* un système de retypage, permettant ainsi de changer les types des éléments du graphe tout en gardant ses nœuds et arcs incidents et ce, en une seule ligne.

Pour terminer, le langage permet également la définition de fonctions et procédures.

Caractéristiques du langage

Plusieurs benchmarks soulignent que c'est l'un des outils les plus rapides et possédant un nombre supérieur de caractéristiques comparé aux autres de sa catégorie. Par exemple, la comparaison basée sur le benchmark de Varró [Geiß et Kroll, 2007] illustre que les performances de l'outil sont supérieures à celles d'AGG, VIATRA2, PROGRES ou encore Fujaba et ce, sur des modèles de large taille.

De surcroit, GrGen.net a reçu plusieurs *awards* aux GraBaTs (*International Workshop on Graph Based Tools*) tel que le *Best Live Contest Solution* concernant la flexibilité de la solution pour le cas d'étude en 2008 et le *Best Live Contest Solution*, mesurant les capacités de prototypage rapide des outils basé sur l'approche par transformation de graphes participants. Ainsi que le *Best Submitted Modeling Solution for Reverse Engineering* en 2009, évaluant les performances, concision, compréhension du langage entre autre. Il a déjà été mis en œuvre dans une cinquantaine de cas d'études et est utilisé dans différents projets comme indiqué sur le site de l'outil [3] et dans divers milieux tel que l'architecture, l'ingénierie mécanique ou encore la bio-chimie [Jakumeit *et al.*, 2014]. De plus, son *pattern matching* est d'une grande expressivité et performant.

Sa documentation tout comme son debugger et son outil de visualisation graphique (Ycomp) sont qualifiés d'excellents [Jakumeit *et al.*, 2014]. Un mode debug interactif et graphique est offert. De plus, son intégration peut être réalisée aisément à l'aide de l'API fournie (réalisée en langage .NET), [Jakumeit *et al.*, 2010]. Elle peut être également atteinte *via* le support du standard EMF et l'importation et exportation de fichiers Ecore et XMI, [Geiß et Kroll, 2008].

Ensuite, dans [Kolahdouz-Rahimi *et al.*, 2014], l'évaluation de ses caractéristiques à l'aide la norme *ISO 9126-1* est bonne. Bien que le cas d'étude du travail en question porte sur un problème de *refactoring*, c'est un bon indicateur pour jauger de la qualité de cet outil. En effet, certaines des caractéristiques évaluées sont assez générales pour l'outil et non spécifiques au cas d'étude tels que l'effort pour son apprentissage, son interopérabilité, sa maturité, etc.

Aussi, il est à noter qu'il arrive troisième dans le classement des utilisateurs dans la TTC2011. Sa spécification est considérée comme clairement structurée et un effort limité est à fournir pour sa compréhension. Kolahdouz tire la même conclusion quant à ces critères importants, ce qui motive notre choix. Pour terminer, il a été élu comme étant le meilleur outil pour la résolution du cas d'un TTC sur l'optimisation d'un compilateur [Buchwald

3. http://www.info.uni-karlsruhe.de/software/grgen/. Date : 20/12/2014

et Jakumeit, 2011], prouvant encore une fois sa maturité et sa capacité à gérer une grande quantité de données d'une manière performante.

6.3.4 Prolog

Prolog (acronyme de PROgrammation en LOGique) est un langage de programmation logique. Actuellement, il est peu utilisé pour concevoir des transformations de modèles.

Principales composantes du langage

En bref, Prolog est composé de faits et de règles, alimentant une base de connaissances. Cette base de connaissance peut être questionnée à l'aide de requêtes, nommées ici prédicats, unités de base du langage. Un prédicat est défini comme étant vrai et consiste en une tête et un nombre d'arguments.

Quant aux textes constants, ce sont les atomes. Prolog représente des données complexes par des termes composés. Un terme composé consiste en un foncteur, devant être un atome et des paramètres sans restriction de type. Un terme composé est identifié par sa tête et son arité. L'arité est le nombre de paramètres.

Caractéristiques du langage

Il existe plusieurs motivations nous poussant à adopter cet outil. D'une part, il est très répandu et connu par un vaste public. D'autre part, ce type de langage possède de multiples avantages pour la mise en œuvre de transformations de modèles, comme repris dans [Almendros-Jiménez et Iribarne, 2008]. Nous pouvons citer son interopérabilité avec d'autres outils par importation/exportation de fichiers au format XMI/XML car la plupart des outils Prolog possèdent des librairies XML. À partir de l'importation d'un fichier au format XMI, Prolog peut générer une représentation du modèle UML devant être transformé. Le modèle cible sera généré à l'aide de règles Prolog, représentant les règles de transformation. Le modèle cible sera à son tour exporté en fichier au format XMI pour être ensuite exploité par un autre outil. Ou encore, il peut être rendu interopérable avec l'utilisation de CORBA.

De plus, Almendros-Jiménez explique par quelle façon les représentations de méta-modèles UML peuvent être effectuées assez facilement en « faits » Prolog et comment les transformations ont la possibilité d'être mises en place *via* des règles Prolog. Cela a déjà été concrétisé dans l'outil MoMaT, il représente les modèles et les exploite à l'aide de Prolog[Störrle, 2007].

Mais surtout, c'est l'outil offrant la meilleure adéquation sémantique des méta-modèles (avec Epsilon) vis-à-vis de MetaDONE. En effet, il n'est lié à aucun standard particulier tel qu'EMF ou MOF et offre par conséquent la possibilité de représenter les méta-modèles avec respect par rapport à MetaDONE.

Pour terminer, un autre avantage est que Prolog est supporté par une multitude d'outils comme des IDEs, des compilateurs relativement efficaces, des debuggers, outils de visualisation, etc.

6.3.5 Évaluation partielle des candidats

Les tableaux suivants présentent et offrent un comparatif de la plupart des caractéristiques générales (tableau 6.1) fonctionnelles (tableau 6.2) et non-fonctionnelles (tableau 6.3)

importantes des langages retenus. Ces caractéristiques sont décrites à la section 5.3 du chapitre 5.

Caractéristiques générales

Le tableau 6.1 offre un récapitulatif des caractéristiques générales des langages pré-sélectionnés. Tout d'abord, l'**approche** mise en place par chaque outil est reprise : hybride (ATL et ETL) ou déclarative. Cette dernière étant subdivisée par l'approche basée sur les transformations de graphes (GrGen.net) et programmation logique (Prolog).

Ensuite, une distinction peut être faite entre le **domaine** d'application des graphes et celui des modèles. La principale différence est leur domaine technique, par exemple MOF ou Ecore pour les modèles et GXL pour les graphes. Néanmoins, Prolog peut être qualifié de neutre car il n'est pas lié à une technologie particulière. À noter qu'Epsilon peut être étendu par l'utilisateur à l'aide d'un driver pour supporter de nouvelles technologies.

En outre, les outils peuvent être catégorisés de **type** « rewriting », opérant en général sur un seul modèle. Tandis que d'autres peuvent être de type « mapping » et opérer sur plusieurs modèles. Un outil peut également combiner ces deux types .

Enfin, tous les outils sélectionnés bénéficient d'une bonne **maturité** dans leur domaine. En effet, ATL, Epsilon et GrGen.net existent depuis plus de 10 ans et sont mis en œuvre dans moult cas d'études et utilisés par plusieurs industries. Cependant, Prolog peut être vu comme moins mature compte tenu de la nécessité de développer une couche applicative spécifique pour la technologie visée.

Pour terminer, les **outils** mis à disposition de l'utilisateur sont recensés. D'une part, certains bénéficient d'une intégration dans l'IDE Eclipse, permettant ainsi d'avoir un environnement centralisé avec un compilateur, éditeur et debugger. D'autre part, certains comme GrGen.net ne bénéficient pas de cette intégration mais offrent des outils très puissants tel qu'un debugger interactif, permettant une représentation graphique (*ycomp*).

Langages / *Caractéristiques*	**ATL**	**Epsilon**	**GRGen.NET**	**Prolog**
Approche	hybride	hybride	basée sur transfo. de graphes	prog. logique
Domaine	modèles(ecore)	graphes/ modèles(tous+driver)	graphes/ modèles(tous)	neutre
Type	mapping/ rewriting	mapping/ rewriting	rewriting	mapping/ rewriting
Maturité	élevée	élevée	élevée	faible
Outils	intégr. Eclipse	intégr. Eclipse /jar autonome	débug. graph /compilateur	intégr. Eclipse /edit/compil./debug.

TABLE 6.1 – Exposition de caractéristiques générales des outils pré-sélectionnés

Caractéristiques fonctionnelles

D'un point de vue des caractéristiques fonctionnelles, tous les langages pré-sélectionnés permettent la réalisation des opérations **CRUD** sur les modèles. La **validation syntaxique** du modèle cible généré par ATL vis-à-vis de son méta-modèle n'est pas fournis [Huber, 2008], il n'y a pas d'outil formel disponible pour la vérification [Kolahdouz-Rahimi *et al.*, 2014]. Tout comme dans GrGen, la validation syntaxique n'est pas assurée mais « il est simple d'écrire des assertions afin de vérifier les contraintes du problème et de les évaluer ensuite dans le modèle cible » [Kolahdouz-Rahimi *et al.*, 2014]. Aussi, ils mettent en place une **traçabilité** implicite. Toutefois, Epsilon et GrGen.net offrent la possibilité de réaliser une traçabilité explicite. [Jakumeit *et al.*, 2014]

À propos de la **validation sémantique**, elle peut être effectuée à l'aide d'OCL dans Epsilon (avec EOL) et ATL. De plus, Epsilon met aussi à disposition le langage EVL, dédié à la spécification et vérification de contraintes sur les modèles [Kolovos *et al.*, 2010], il offre même un support semi-automatique quant à la correction de cas triviaux. Il peut être facilement combiné à l'utilisation d'ETL car tous les langages de la famille Epsilon sont interopérables. ATL effectue un contrôle sur l'applicabilité des règles, une erreur est levée si plusieurs règles peuvent être appliquées simultanément. Toutefois, cela ne permet pas d'assurer la confluence dans tous les cas, [Lano *et al.*, 2012]. Tant que les *called rules* et *lazy rules* ne sont pas employées, la terminaison et déterminisme sont assurés [Syriani et Vangheluwe, 2009]. La validation sémantique des modèles est réalisable dans Prolog à l'aide de règles comme testé de manière probante dans [Almendros-Jiménez et Iribarne, 2012].

Quant à GrGen, il est de type *Turing* complet, héritant ainsi des caractéristiques d'une machine de *Turing*. Il bénéficie d'une grande expressivité mais ne vérifie la couverture complète du modèle, la terminaison ou la confluence d'un problème, c'est par conséquent à l'utilisateur de prouver ces propriétés [Jakumeit Edgar, 2014].

Cependant, le domaine de la vérification et validation de modèles étant très vaste et n'étant pas l'objet de ce mémoire, nous nous intéressons uniquement aux propriétés générales offertes par les langages. Pour plus d'informations, il existe des états de l'art tel que [Calegari et Szasz, 2013] vers lesquels nous renvoyons le lecteur.

Ensuite, la **réutilisation des transformations** est réalisable dans ATL et Epsilon *via* la notion d'héritage, GrGen.net et Prolog offrent un mécanisme de paramétrisation. Prolog peut également utiliser des *higher order predicates*. « Les prédicats d'un niveau supérieur d'abstraction peuvent être mis en place dans Prolog à l'aide de clause de Horn. Ce style encourage une abstraction plus élevée, plus de réutilisation et conduit à des programmes plus concis ».[Naish, 1996]

Quant à la **modularité** des transformations, ATL permet à ses helpers d'être inclus dans des librairies, pouvant elles-mêmes être importées dans des modules ATL. ETL permet l'import de module EOL. Dans GrGen, la modularité est possible par l'utilisation de *subpattern*, contenant des patterns communs.

La **bidirectionnalité** est uniquement possible avec GrGen.net [Blomer *et al.*, 2011].

Caractéristiques \ *Langages*	ATL	Epsilon	GRGen.NET	Prolog
CRUD	✓	✓	✓	✓
Réutilisation	héritage règles	héritage règles helpers	paramétrisation règles *higher order predicates*	paramétrisation
Validation syntaxique	X	X	X	X
Validation sémantique	OCL /conf. partielle	EOL/EVL	X	règles Prolog
Modularité transfo.	modules /librairies	modules	subpatterns	modules librairies
Traçabilité	implicite	implicite /explicite	implicite /explicite	implicite
Bidirectionnalité	X	X	✓	X

TABLE 6.2 – Exposition des caractéristiques fonctionnelles des outils pré-sélectionnés

Caractéristiques non fonctionnelles

Les caractéristiques non-fonctionnelles importantes sont explicitées pour chacun des outils.

Ces langages sont **conviviaux**, ils peuvent être qualifiés d'intuitifs et efficaces. Cependant, GrGen.NET n'étant pas intégré à un IDE est moins intuitif pour l'utilisateur. De plus, l'ordonnancement des règles doit être géré par ce dernier également (à l'aide du fichier .grs), demandant ainsi une certaine expérience. Toutefois, le langage est très efficace. L'utilisation d'ATL est jugée moyenne dû à son manque d'efficacité potentiel, comme l'obligation d'expliciter la recopie des éléments du modèle source vers le modèle cible (ce point sera détaillé par la suite). Quant à Prolog, il demande une grande maitrise du langage pour concevoir des transformations.

Concernant leur **concision**, elle est jugée élevée pour Espilon et GrGen.net [Jakumeit *et al.*, 2014], tandis que celle d'ATL et Prolog peut être qualifiée de moyenne. Par exemple, Epsilon et GrGen.net offrent des opérations concises pour le retypage de modèle, améliorant encore cette caractéristique. GrGen permet l'application de conditions négatives, rendant le processus de transformation plus compréhensible et diminuant le nombre de règles [Syriani et Vangheluwe, 2009].

D'un point de vue **extensibilité**, l'outil offrant le plus de mécanismes est Epsilon. Selon le site d'Epsilon : « Presque tous les aspects d'Epsilon sont extensibles. Un support pour un nouveau type de modèle peut être ajouté, l'outil développement basé sur Eclipse peut être étendu, des méthodes peuvent être ajoutées, ou même un nouveau langage se basant sur EOL peut même être implémenté ». De surcroit, il peut appeler des méthodes du langage Java afin d'effectuer des tâches non supportées nativement.

Ensuite, tous les langages excepté GrGen.NET sont intégrés dans l'IDE Eclipse. L'**interopérabilité** de GrGen.NET est possible à l'aide d'une API .net fournie, celle d'ATL *via* l'*ATL launcher* et Prolog peut aisément être utilisé avec CORBA. Quant à ETL, il met à disposition un jar autonome, pouvant être appelé par du code JAVA externe. Aussi, l'interopérabilité

peut être mise en place par *via* le partage de fichier XMI. Nous reviendrons plus en détails sur le sujet de l'interopérabilité dans un chapitre ultérieur.

De plus, ils **utilisent des standards** pour la technologie de représentation des modèles tel qu'EMF (dont XMI). De plus, ATL et Epsilon utilisent le standard OCL. Quant à GrGen, il supporte aussi le standard GXL.

Enfin, les outils basés sur un compilateur peuvent voir leurs performances améliorées contrairement aux outils basés sur un interpréteur. Ici, uniquement Epsilon se base sur un interpréteur. Selon la littérature, GrGen possède des **performances** élevées et peut concevoir des transformations complexes et sur des larges modèles [Jakumeit Edgar, 2014]. Les performances les moins probantes sont celles d'ATL.

Pour terminer, Prolog et Epsilon sont de bons candidats pour l'**adéquation sémantique** du méta-modèle de MetaDONE. Epsilon est déjà compatible avec plusieurs types de technologies (EMF, XML, CSV, Bibtex) et un driver peut être développé afin de le rendre compatible avec un nouveau type. Quant à Prolog, il n'est pas lié à une représentation spécifique.

Caractéristiques \ *Langages*	ATL	Epsilon	GRGen.NET	Prolog
Convivialité	moyenne	élevée	moyenne	moyenne
Concision	moyenne	élevée	élevée	moyenne
Extensibilité	moyenne	élevée	moyenne	moyenne
Interopérabilité	Eclipse /XMI/API	Eclipse/XMI/Jar	Eclipse /XMI/API	Eclipse /XMI/CORBA
Acceptation par la communauté	✓	✓	✓	Partielle
Utilisation de standards	OCL/XMI	OCL/XMI	XMI/GXL	XMI
Évolutivité/Performance	faible	élevée	élevée	moyenne
Adéquation sémantique méta-modèle	X	élevée	X	élevée

TABLE 6.3 – Exposition des caractéristiques non fonctionnelles des outils pré-sélectionnés

6.4 Conclusion

La littérature existante nous a permis de pré-sélectionner des outils mettant en place les diverses approches retenues : hybride et déclarative (basées sur les transformations de graphes et programmation logique). Ce choix a été guidé par les caractéristiques possédées par les outils tels que leur maturité, la possibilité de mise en place d'une interopérabilité, une utilisation de standards, une acceptation par la communauté, etc. Ces outils sont ATL, ETL, GrGen.net et Prolog, présentés de manière générale dans ce chapitre.

De plus, leurs caractéristiques fonctionnelles et non fonctionnelles ont été détaillées en tenant compte des recommandations formulées dans le chapitre des exigences, permettant ainsi à d'évaluer ces candidats partiellement.

Finalement, leur évaluation sera poursuivie dans un chapitre ultérieur suite à la réalisation du benchmark basé sur le cas d'étude.

Chapitre 7

Perspectives d'intégration d'un moteur de transformation dans un MetaCASE

Dans ce chapitre, nous tentons d'expliquer les concepts de bases liés à l'interopérabilité et aux environnements intégrés dans la section 7.2. La section 7.3 présente les différentes architectures d'intégrations existantes et les évaluent afin d'en élire une qui permettrait d'intégrer l'outil de transformations de modèles dans l'outil de méta-modélisation, MetaDONE. Ensuite, la section 7.4 présente des exemples de réalisations existantes d'intégration dans des outils de méta-modélisations faisant leurs preuves. Pour terminer, nous analysons la littérature à ce sujet afin de trouver des pistes pour nous permettre d'évaluer la façon la plus adéquate pour intégrer un outil (section 7.5).

7.1 Introduction

Dans le domaine de l'IDM, un grand nombre d'opérations sont nécessaires à l'utilisateur afin qu'il puisse effectuer différentes tâches sur les modèles. Toutefois, toutes ces opérations ne peuvent être fournies par un seul outil.

L'interopérabilité des outils est un enjeu important dans le domaine de l'IDM. Un environnement intégré doit être mis à disposition pour permettre cette dernière, de faire le lien entre ces diverses opérations mises à disposition par différents outils. Les outils de méta-modélisation ont la nécessité de permettre à différents outils hétérogènes d'être intégrés dans le même environnement centralisé.

Néanmoins, rendre des outils interopérables n'est pas aisé dû à leur hétérogènité et il y a de nombreuses dimensions à tenir en compte pour l'atteindre. En effet, il faut faire interagir différents outils qui devront partager des données, communiquer, etc. L'architecture d'intégration doit être bien définies pour atteindre un résultat probant.

7.2 Concepts et principes de bases

7.2.1 Interopérabilité

L'on entend par interopérabilité, « l'habilité pour un ensemble d'entités communicantes de partager des informations et d'exploiter ces informations *via* une sémantique opération-

nelle préalablement définie ». [Brownsword *et al.*, 2004]
Il existe différentes dimensions permettant de résoudre le problème de l'interopérabilité [Lewis et Wrage, 2004] :

l'interopérabilité syntaxique, rendant possible l'échange d'informations ;

l'interopérabilité sémantique, permettant l'interprétation commune du sens de l'information et de la manière dont l'exploitation de ces données doit être effectuée ;

l'interopérabilité technologique, permettant la coopération entre plusieurs entités logicielles dont les implémentations sont issues de technologies différentes. [Bondé, 2006]

7.2.2 Environnement intégré

L'intégration des outils dans un environnement intégré est un enjeu pour permettre l'interopérabilité. Pour rendre possible la création d'un environnement dans le domaine de l'IDM permettant à ses utilisateurs d'utiliser plusieurs outils différents d'une manière coordonnée, celui-ci doit fournir certaines fonctionnalités. Wasserman [Wasserman, 1990], l'un des travaux les plus cité du domaine et référencé par de nombreux autres auteurs tel que [Wicks, 2004], met en évidence cinq dimensions d'intégrations d'outils :

l'intégration des données : l'environnement permet à ses différents outils d'exploiter des données partagées (*via* un *repository* central ou une base de données par exemple). Elle possède cinq propriétés : l'interopérabilité des données, leur degré de redondance, leur consistance, leur synchronisation entre les différents outils les exploitant et la compréhension des données entre ces outils ;

l'intégration du contrôle : l'environnement permet aux outils de s'échanger mutuellement messages (à l'aide de CORBA ou RPC par exemple). Ces messages servent à diriger un outil afin qu'il effectue une opération correspondante aux messages reçus. Cette dimension permet de réduire les interactions entre le développeur et les outils et d'éviter toute manipulation manuelle complexe [Sriplakich, 2007] ;

l'intégration de la présentation : l'environnement met à disposition une interface utilisateur commune (pouvant être appelée *workbench*) afin de permettre à l'utilisateur d'interagir avec les outils de manière centralisée ;

l'intégration processus : l'environnement doit être capable de gérer l'exécution d'un processus qui requiert l'implication de plusieurs outils ;

l'intégration par plateforme : rend possible le fonctionnement de plusieurs outils sur la même plateforme afin qu'ils puissent utiliser des services communs fournit.

7.3 Évaluation d'architectures d'intégration

7.3.1 Extension d'un outil existant ou intégration d'un outil sous forme de librairie

Les caractéristiques souhaitées d'un outil peuvent être intégrées à un autre outil comme une extension (*plugin*), ou encore sous forme d'une librairie. Avec cette approche, un moyen doit être mis en place pour que l'outil général puisse utiliser l'autre par le biais d'une interface. Ainsi, les fonctionnalités et l'implémentation de l'outil composite seront encapsulées dans des interfaces permettant à l'outil dans lequel cette librairie sera importée de la manipuler. Ce type d'intégration offre un couplage fort. Cependant, si l'outil à intégrer possède

déjà une API, l'effort à fournir sera moindre.

Par exemple, l'intégration par plugin est employée par l'IDE Eclipse, des outils peuvent être mis en place en tant qu'extensions et utilisés dans l'IDE comme décrit dans [Wirpi, 2012]. Quelques réalisations ont déjà été mise en œuvre dont l'intégration de MetaEdit+ dans Eclipse, permettant aux développeurs de disposer d'un environnement intégré où ils peuvent à la fois effectuer leur développement et manipuler des modèles. Cette approche procure une meilleure intégration processus, les utilisateurs n'auront pas à fournir d'efforts quant à la gestion des outils et ne devront pas passer continuellement de l'un à l'autre.

7.3.2 Web service

Un service web est un mécanisme de communication entre applications distantes à travers un réseau. Il rend possible l'interopérabilité entre des logiciels hétérogènes. Il fonctionne à l'aide de protocoles standardisés et permet un couplage faible.

Cette approche est mise en place par ModelBus [Sriplakich, 2007], en utilisant des composants business ou mécanismes d'appels à des procédures distantes (*remote procedure call*), pouvant être effectué à l'aide de CORBA par exemple afin d'y intégrer des outils de *software engineering*. La mise en place d'une architecture semblable est utilisée par MetaEdit+.

7.3.3 Bus service

Un bus est un système de communication partagé entre plusieurs composants, il fournit un support fonctionnel pour l'intégration et la communication entre les outils. Les applications sont ainsi reliées par le bus où elles ont la possibilité de déposer des messages à destination d'autres applications. Les bus peuvent effectuer des transformations sur ces messages. Les bus sont vu comme des médiateurs, leur avantage est de fournir un couplage faible voir transparent.

7.3.4 Dans le cloud

Le *cloud* désigne un ensemble de processus qui consistent à utiliser la puissance de calcul et/ou de stockage de serveurs informatiques distants à travers un réseau, généralement Internet. Celui-ci peut apporter des bénéfices à l'IDM tel que l'exploitation de modèles très larges, le travail collaboratif, la résolution des problèmes d'interopérabilité, etc. Ce domaine émergent, combinant l'IDM et le cloud, se nomme le "*Modeling as a Service* (MaaS)" ou encore "*Modelling in the cloud*", [Bruneliere *et al.*, 2010].

Des recherches se tournent vers ce sujet, de manière générale comme [Bruneliere *et al.*, 2010] ou d'un point de vue stockage et transformations de modèles tel que [Clasen *et al.*, 2012]. L'ouvrage [Juracz *et al.*, 2013] traite également d'un outil ayant mis en place avec succès un architecture utilisant le *cloud* pour permettre du travail collaboratif.

Un avantage non négligeable concernant la transformation de modèle dans le *cloud* est l'abondance de ressources disponibles, le travail des transformations peut être optimisé et se voir réparti sur différentes unités de traitement. De plus, la disponibilité d'*engine* de transformations dans le *cloud* fournirait des services de gestion de modèles indépendant de toute plateforme.[Bruneliere *et al.*, 2010]

AToMPM est un autre exemple de réalisation fonctionnant complètement *online*. Il permet la manipulation de modèles dans le *cloud* [Syriani *et al.*, 2013b].

7.3.5 Architecture basée sur les composants

Cette architecture est composée d'une ensemble de composants intégrés, où chacun fournit un outil à l'environnement. Un composant peut être un *web service*, un progiciel, une ressource web ou encore un module encapsulant un ensemble de données ou fonctions [Wikipedia, 2015].

Pour mettre en œuvre des composants, un modèle de composants est nécessaire, il définit les entités et leur mode d'interaction et de composition. Ensuite, il faut une « infrastructure à composants, qui met en œuvre le modèle et permet de construire, déployer, administrer et exécuter des applications conformes au modèle. » [1]. Nous pouvons citer en exemple CORBA, OSGi ou Enterprise Java Beans (EJB).

En général, les composants ont une dépendance minimale entre eux afin de faciliter la réutilisation et un déploiement de type « plug-and-play » [Grundy *et al.*, 2000].

Selon [Grundy *et al.*, 2000], les outils basés sur une architecture de type composants peuvent être facilement intégrés ou étendus à d'autres outils. Ils sont aisément réutilisables, sont d'une grande interopérabilité et peuvent être contrôlés de manière externe par d'autres composants. Cependant afin d'obtenir une architecture efficace, les choix de conception doivent être bien pensés, par exemple le nombre de composants la constituant doit être correct, les composants communs doivent avoir un bon *design*, etc. L'intégration au niveau contrôle et données peut se révéler être plus efficace. L'auteur, conclu que cette approche offre de réels avantages. Elle est modulaire, facile à mettre en place et à intégrer, aussi bien pour les développeurs que pour les utilisateurs qui peuvent aisément les étendre.

7.4 Exemples de réalisations

7.4.1 Model Bus

ModelBus est un environnement d'ingénierie de modèles permettant un travail collaboratif en utilisant des outils hétérogènes dans un contexte de système distribué. C'est un exemple de réalisation ayant réussi à résoudre le problème d'interopérabilité avec succès.

L'interopérabilité des outils est rendue possible par un composant « adaptateur », liant un outil hétérogène à l'environnement. L'intégration des données est possible grâce à l'adaptateur, permettant à l'outil de charger les modèles d'un *workspace* afin de les manipuler et les sauvegarder. Concernant l'intégration du contrôle, elle est réalisée par une adaptation de l'approche *web services*, les outils peuvent ainsi mutuellement invoquer des opérations. [Sriplakich, 2007]

7.4.2 MetaEdit

MetaEdit est un outil de méta-modélisation, offrant un support pour le DSM (*Domain-specific modeling*), allant de la méta-modélisation à la modélisation, en passant par la génération de code. Son architecture repose sur son *MetaEngine*, les différents outils intégrés dans le MetaCase communiquent uniquement avec ce composant comme l'illustre la figure 7.1 ([Kelly *et al.*, 1996] et [MetaCase, 2014a]). Le *MetaEngine* gère les échanges entre le *repository* (ou base de données) contenant les données et les différents outils. L'intégration des

1. http://proton.inrialpes.fr/~krakowia/Enseignement/M2R-SL/CR/Flips/CR3-Composants-1.pdf. Date : 24/05/2015

données repose sur ce *repository* et *via* une API SOAP, évitant leur duplication et permettant une maintenabilité aisée de l'outil.

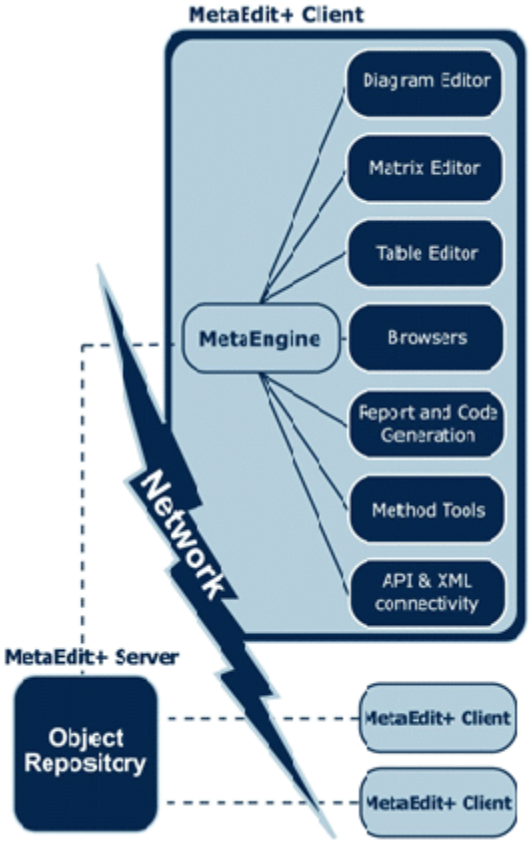

FIGURE 7.1 – Architecture de MetaEdit+, extrait de [MetaCase, 2014b]

7.5 Discussion sur le sujet de l'intégration d'outil dans le domaine du software engineering

Brown [Brown, 1993] suggère que l'intégration doit être simple, flexible et efficace. Il se focalise sur l'intégration d'un point de vue contrôle, par exemple à l'aide d'un serveur de message *broadcast multicast* ou *via* une communication directe point à point, elle permettrait une gestion plus facile. Il met également en exergue le fait que les outils doivent être extensibles et pouvoir être encapsulés dans d'autres outils et ce, d'une manière utilisant des standards tel que CORBA.

De son côté, Yang [Yang, 1994] préconise un couplage faible entre les différents outils, meilleure façon selon lui d'intégrer des outils dans un environnement d'ingénierie logiciel. Pour ce faire, il suggère une combinaison d'un système d'échange de messages et d'un *repository* central. Par conséquent, l'intégration est réalisée au niveau données et contrôle. Ses expériences ont démontré que le couplage faible est le plus efficace pour intégrer un outil dans un environnement de *software engineering*. Elles prenaient en compte les critères de

comportement, temps de réponse, de coût d'intégration et de l'indépendance des composants.

Nous avons extrait ci-dessus deux avis intéressants pour notre cas sur le sujet de l'intégration d'outil dans le domaine du software engineering. Toutefois, il peut être constaté dans l'état de l'art de Wicks [Wicks, 2004] que divers travaux dans le milieu tentent de classifier les différentes approches existantes ou de recommander une certaine approche mais beaucoup d'avis divergent sur le sujet.

7.6 Conclusion

Bien que les sujets de l'interopérabilité et des architectures d'intégration soient importants et complexes, il existe peu de littérature à ce sujet, particulièrement récentes. Tout comme leur mise en place pour un outil de méta-modélisation. Toutefois, des réalisations probantes existent telles que MetaEdit+ ou ModelBus.

Nous avons pu constater que d'une part, il existe différents niveaux sur lesquels l'intégration peut être accomplie, au niveau des données, du contrôle, de la présentation, du processus et/ou encore de la plateforme. D'autre part, il existe une multitude d'architectures d'intégration pouvant être élaborées, possédant chacune des forces et des faiblesses et un couplage pouvant varier de faible à élevé en fonction des choix effectués.

Des solutions semblent concluantes et peuvent être mise en place de manière réaliste telles que l'intégration des données *via* un *repository* central et l'intégration du contrôle par des échanges de messages. De plus, certaines perspectives paraissent prometteuses pour l'avenir telle qu'une architecture dans le *cloud*, offrant divers avantages tel qu'un travail collaboratif sur les modèles et offrant une puissance non négligeable, elle doit néanmoins encore faire ses preuves.

Pour conclure, il n'existe pas de solution pouvant être élue comme la meilleure, il faut tenir compte des contraintes de chaque outil. Aussi, il faut veiller à réaliser une intégration simple, flexible, efficace. La mise en place de tel environnement défie la systématisation et requiert de la créativité.

Chapitre 8

Présentation détaillée du cas d'étude

Ce chapitre expose en détails le fonctionnement de la transformation du cas d'étude, présente les modèles impliqués dans cette dernière et illustre le tout à l'aide d'un exemple. Cette présentation fait suite à celle plus générale effectuée à la section 5.4 du chapitre 5.

Tout d'abord, la section 8.2 présente la logique générale avec laquelle le cas d'étude est implémenté ainsi que les principales simplifications effectuées. Ensuite, la section 8.3 offre des illustrations des modèles en entrée et en sortie de la transformation avec une notation *ad-hoc* pour faciliter la compréhension du problème, les éléments en entrée et sortie sont détaillés et la mise en évidence d'éléments tels que les séquences entre activités est effectuée. Des mockups y sont aussi présentés afin permettre une compréhension claire et générale du problème d'une manière plus visuelle et concrète.

8.1 Introduction

Actuellement, le panel de périphériques disponibles est très large. Le développement d'applications compatibles avec tous ces supports est un enjeu. En effet, tout propriétaire d'application veut maximiser la portée de cette dernière et les utilisateurs souhaiteraient également avoir la possibilité d'utiliser leurs applications préférées sur tous leurs périphériques. Développer une application spécifique à chaque support n'est pas une solution adéquate.

C'est ici que les transformations de modèles entrent en action. Il est facilement possible de créer un modèle représentant un logiciel abstrait qui, par le biais de ces transformations, serait transformé en un programme adéquat à chacun de ces périphériques en tenant compte de leur contraintes pour un résultat et affichage adéquat.

8.2 Principe général de la transformation

Un programme est conçu comme une collection d'activités, elles-mêmes composées d'un ensemble de tâches. Une activité est dite conforme si ses dimensions sont optimisées pour l'écran physique du périphérique associé. L'objectif de la transformation est d'adapter le nombre de tâches par activités afin de les rendre conforme. Pour vérifier la conformité d'une activité, les dimensions de chacune des tâches la composant sont calculées et additionnées aux autres. Si ce total devient supérieur à la taille du périphérique donné, la création d'une nouvelle activité est réalisée. La nouvelle activité se verra assignée les tâches qui rendaient l'activité originale non conforme, réduisant ainsi les dimensions de cette dernière. S'il existe

des tâches ayant une contrainte de précédence se voyant assignées à des activités distinctes, une contrainte de séquence est ajoutée entre ces activités. Toutefois, la transformation peut échouer si une activité ne peut être réduite car une ou plusieurs de ses tâches possède des dimensions trop grandes, inadaptées pour le périphérique.

8.2.1 Hypothèses simplificatrices et perspectives

Des hypothèses simplificatrices sont émises sur le modèle en entrée « *AbstractProgram* » afin de se focaliser lors de l'implémentation de la transformation sur les tâches centrales présentées précédemment, limitant ainsi sa taille et l'effort de développement à fournir car l'implémentation de cette transformation doit être réalisée dans plusieurs langages de transformation de modèles. L'objectif initial est ici de réaliser une transformation pour permettre la comparaison des différents langages.

- une tâche est composée d'un *Widget* composite principal de type *Panel*, contenant lui-même tous les autres widgets de la tâche tels que les boutons, labels, etc. Nous supposons que les éléments composant la tâche sont optimisés par défaut dans l'espace. De ce fait, pour calculer la dimension d'une tâche, la transformation récupère simplement les dimensions du *Panel*, et évite ainsi de parcourir et additionner la taille de chaque *Widget* composant ;

- une activité est dite non conforme si la somme des tailles de ses tâches est supérieure à la taille de l'écran du périphérique (*Device*) donné. Une ou plusieurs tâches peuvent donc être responsables de la non-conformité d'une activité. Cependant, dans le cas où les dimensions d'une tâche ne sont pas conformes à celle du périphérique donné, nous notifions l'utilisateur que la transformation ne peut être exécutée. Actuellement, nous ne réduisons pas la taille des tâches et de ses widgets afin d'optimiser l'espace.
  Or, la transformation pourrait être étendue afin de réduire au minimum la taille de chacun de ses widgets afin de rendre la tâche conforme. Cela pourrait rencontrer évidemment quelques limitations, comme le fait que chaque élément réduit doit toujours être lisible et accessible à l'utilisateur afin de permettre une interaction. Aussi dans le cas contraire, les widgets et tâches pourraient être agrandis afin de couvrir une surface optimale pour le périphérique donné ;

- dans notre cas, le modèle en entrée est relativement simple et amène à une division d'une activité en maximum deux activités. Actuellement, seule notre implémentation de la transformation avec le langage ETL permet la gestion de modèles plus complexes conduisant à une division d'une activité en n activités. Les autres transformations pourraient être étendues relativement aisément pour permettre cette gestion également.

8.3 Illustration ad-hoc des modèles du cas d'étude

Les figures de cette section illustrent à l'aide d'une notation *ad-hoc* les modèles utilisés en entrée et celui après transformation, afin de faciliter la compréhension du cas d'étude et juger la correction des transformations si besoin.

8.3.1 Modèles en entrée (AbstractProgram et Device)

Représentation sous forme ad-hoc

La figure 8.1 illustre les modèles *AbstractProgram* et *Device* utilisés en entrée de la transformation. Ceux-ci seront utilisés pour les différentes implémentations de la transformation. Un *AbstractProgram* doit être conforme à un périphérique donné (*Device*). Dans notre cas, nous fournissons un périphérique dont le nom est « tabletteNexus » et ayant une hauteur d'écran de 900 pixels et une largeur d'écran de 600 pixels.

L'objectif de cette application est de permettre à un utilisateur identifié de consulter et gérer ses différents rendez-vous.

Le programme abstrait, est composé de trois activités principales distinctes, composées de différentes tâches. Chacune de ces activités est ici indépendante, il n'y a pas de séquence entre-elles. Concernant les tâches, il existe une relation de précédence entre deux tâches dans la première activité « activityLogin » (entre « taskLogin » et « taskDiplayMeetingMustBeInNewActivity ») et il en existe une également dans la seconde activité se nommant « activityHandleMeetingCompliant ».
Chaque tâche est composée d'un *Widget* de type *Panel* contenant tous les autres widgets composant la tâche. Ce panel possède une hauteur et une largeur. Une tâche pourrait contenir plusieurs *Panel* mais nous avons fait le choix de simplifier le problème comme explicité à la section précédente. Ici, nous pouvons attirer l'attention sur « panelTaskTooBig » et « panelLogoutTooBig », qui forceront la création d'une nouvelle activité car elles ont une dimension identique au *Device* et elles ne peuvent donc pas cohabiter avec une autre tâche dans une même activité.

Pour terminer, nous n'avons pas explicité tous les sous widgets car ils auraient alourdi le schéma avec des informations de moindre importance.

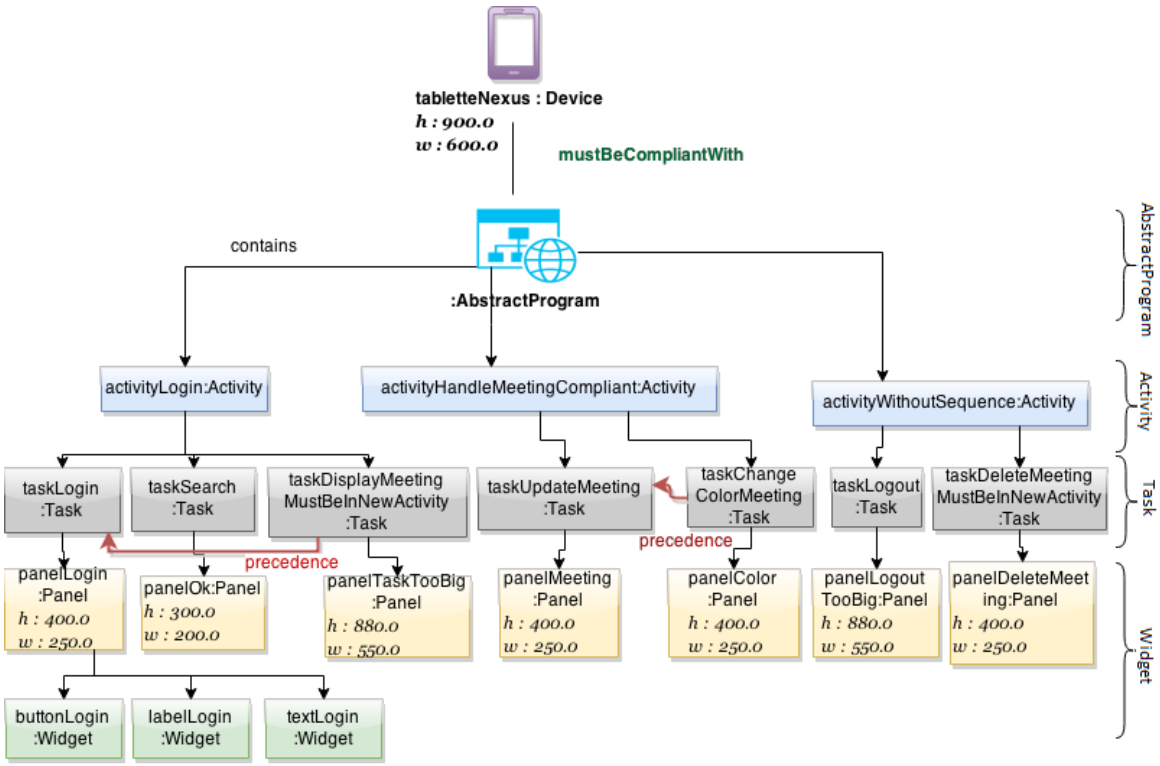

FIGURE 8.1 – Illustration du modèle en entrée de la transformation (*AbstractProgram* et *Device*)

Représentation sous forme de mockups

La figure 8.2 une des activités du modèle en entrée « ActivityLogin » mais cette fois sous forme d'un mockup.

Nous pouvons y retrouver les trois tâches de cette activité : « taskLogin » (en haut à gauche), « taskSearch » (en haut à droite) et « taskDisplayMeeting »(dans la partie du bas). Elles permettent respectivement à un utilisateur de s'identifier auprès de l'application, d'afficher ses différents rendez-vous et d'effectuer une recherche. C'est une activité très simple permettant d'illustrer la transformation clairement.

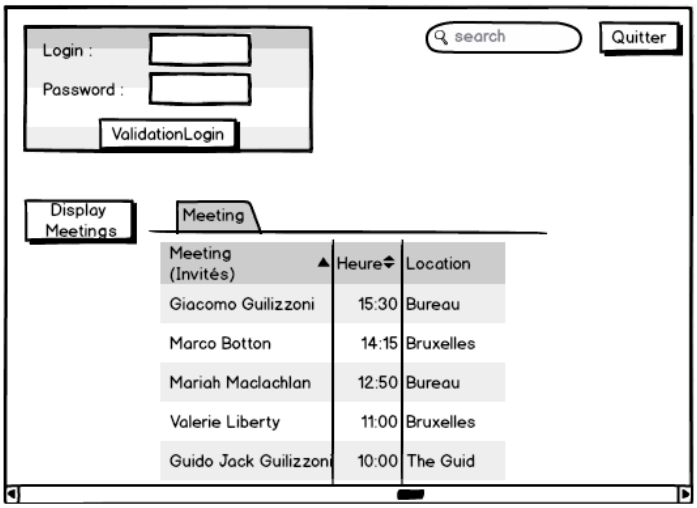

FIGURE 8.2 – Mockup représentant l'« activityLogin » de l'*AbstractProgram*

8.3.2 Modèle résultant de la transformation en sortie (CompliantProgram)

Représentation sous forme ad-hoc

La figure 8.3 illustre le modèle *CompliantProgram* devant résulter de la transformation. Les activités conformes y sont simplement copiées ainsi que l'ensemble de leurs composants. Tandis que les activités non-conformes sont subdivisées. Ainsi, deux nouvelles activités doivent être créées suite à la transformation : « activityLoginSplit1 » et « activityWithoutSequenceSplit2 », en jaune sur la figure. Une contrainte de séquence doit être ajoutée entre « activityLogin » et cette nouvelle activité dû à la séparation de deux tâches contenant une relation de précédence.

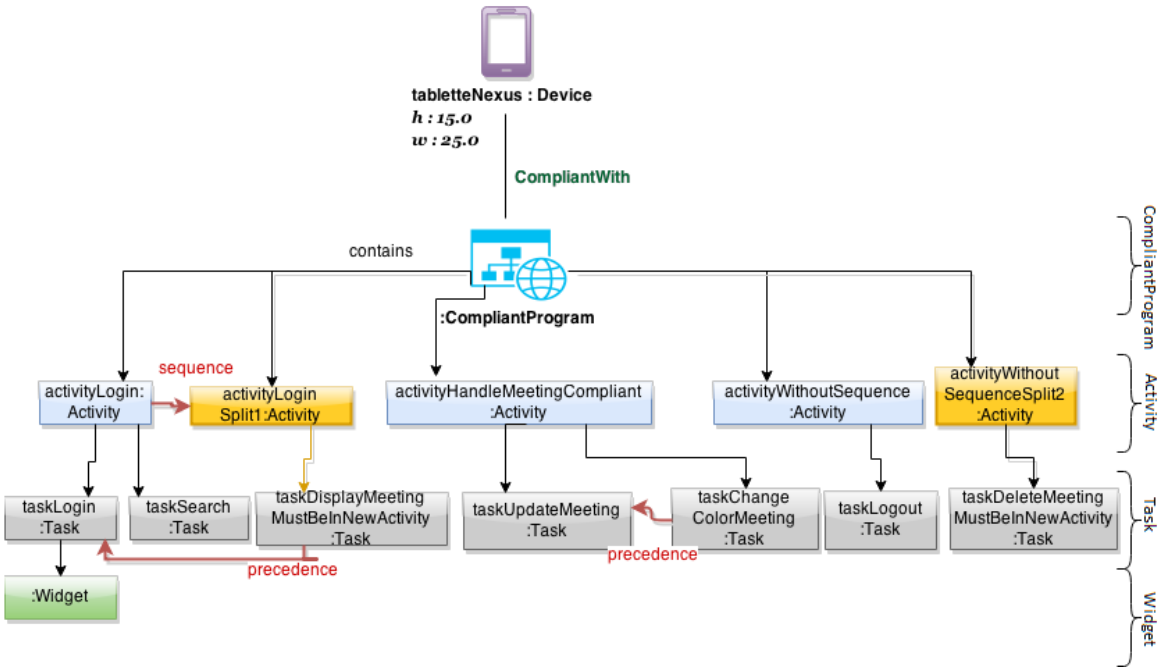

FIGURE 8.3 – Illustration du modèle cible résultant de la transformation (*CompliantProgram*)

Représentation sous forme de mockups

Les figures suivantes illustrent une des activités du modèle en sortie « ActivityLogin » mais cette fois sous forme d'un mockup. La figure 8.4 représente l'activité « ActivityLogin » ne contenant plus que les tâches « taskSearch » et « taskLogin ». En effet, les autres tâches ont été supprimées et copiées dans une nouvelle activité afin de permettre l'activité actuelle de devenir conforme au périphérique.

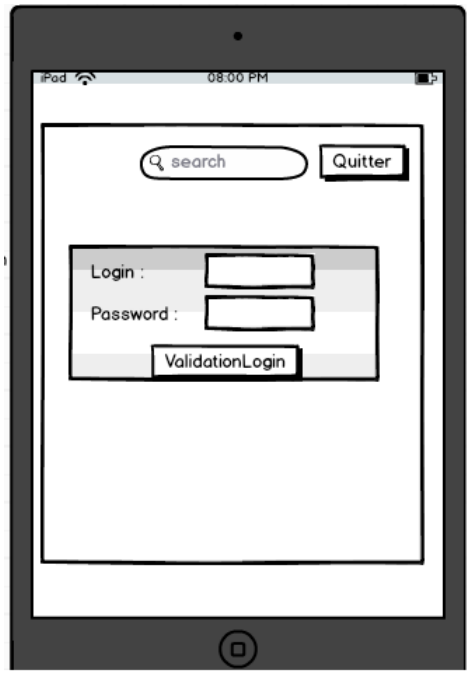

FIGURE 8.4 – Mockup d'un activité rendue conforme dans le modèle cible *CompliantProgram*

La figure 8.5 représente la nouvelle activité créée contenant la tâche « taskDisplayMeeting » :

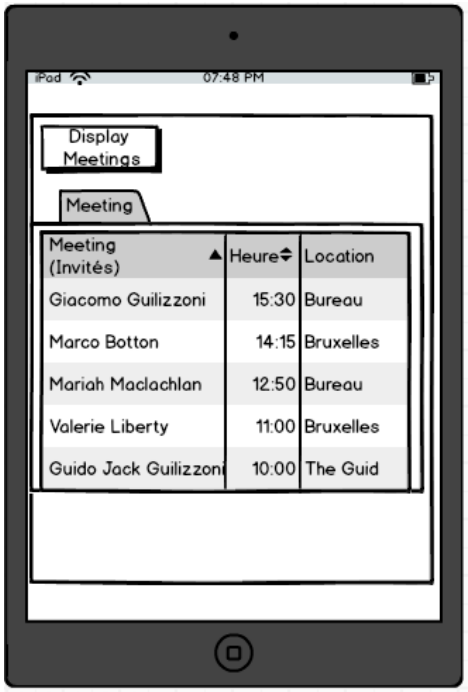

FIGURE 8.5 – Mockup d'un nouvelle activité crée dans le modèle cible *CompliantProgram*

8.4 Conclusion

L'IDM peut être employé dans plusieurs domaines et réduit considérablement la complexité et coût des développements *via* leur automatisation.

Le problème présenté offre un cas d'étude concret et visuel à l'étude des transformations de modèles. Elles sont utilisées pour automatiser l'adaptation d'une application abstraite à une application adéquate pour un périphérique donné en tenant compte de ses contraintes. Bien qu'il soit simplifié dans le cadre de ce travail, il reste d'une complexité intéressante et montre ainsi la force des transformations de modèles. Un cas d'étude réaliste permet de juger de manière objective les outils de transformations de modèles utilisés pour le mettre en place.

Chapitre 9

Benchmark et évaluation des outils sélectionnés

Dans ce chapitre, nous présentons les moyens mis en place afin de concevoir la transformation du cas d'étude dans les différents langages (section 9.2) . Ensuite, les différentes implémentations de la transformation sont évaluées, permettant ainsi de comparer les outils à la section 9.3. Les implémentations des transformations sont consultables en annexes A et le code y est expliqué à l'aide de commentaires. Pour terminer, une conclusion générale de ces évaluations est formulée (section 9.4).

9.1 Introduction

La comparaison d'outils à l'aide de benchmark est une pratique relativement répandue permettant de déterminer la solution la plus appropriée pour un scénario donné à l'aide de mesures représentatives. Pour réaliser une évaluation objective, des standards tels que la norme ISO-9126 [ISO/IEC, 2015] peuvent être utilisés. Cette norme définit un langage commun pour modéliser la qualité d'un logiciel. Elle propose de classer les outils selon un ensemble structuré de caractéristiques et sous-caractéristiques. Cette étape est indispensable quant à la sélection d'un outil adéquat.

9.2 Présentation des ressources

Afin de pouvoir procéder à la conception de la transformation et de la tester, nous devons disposer des ressources nécessaires tels le modèle source et son méta-modèle, ce dernier étant également celui du modèle cible. Les modèles de l'*AbstractProgram* et du *CompliantProgram* ont été réalisé sous un format XMI afin d'être exploités lors par les transformations réalisées avec les différents langages sélectionnés. Leur méta-modèle est représenté par un fichier Ecore, standard des modèles XMI.

9.2.1 Méta-modèle Ecore

Tout d'abord, un méta-modèle Ecore a été créée afin de permettre la mise en œuvre des transformations. La figure 9.1 illustre ce méta-modèle. Comme explicité précédemment, c'est le standard pour représenter les méta-modèles. Celui-ci sera utilisé dans le cadre de l'implémentation de la transformation avec ATL, Epsilon et GrGen.net et Prolog.

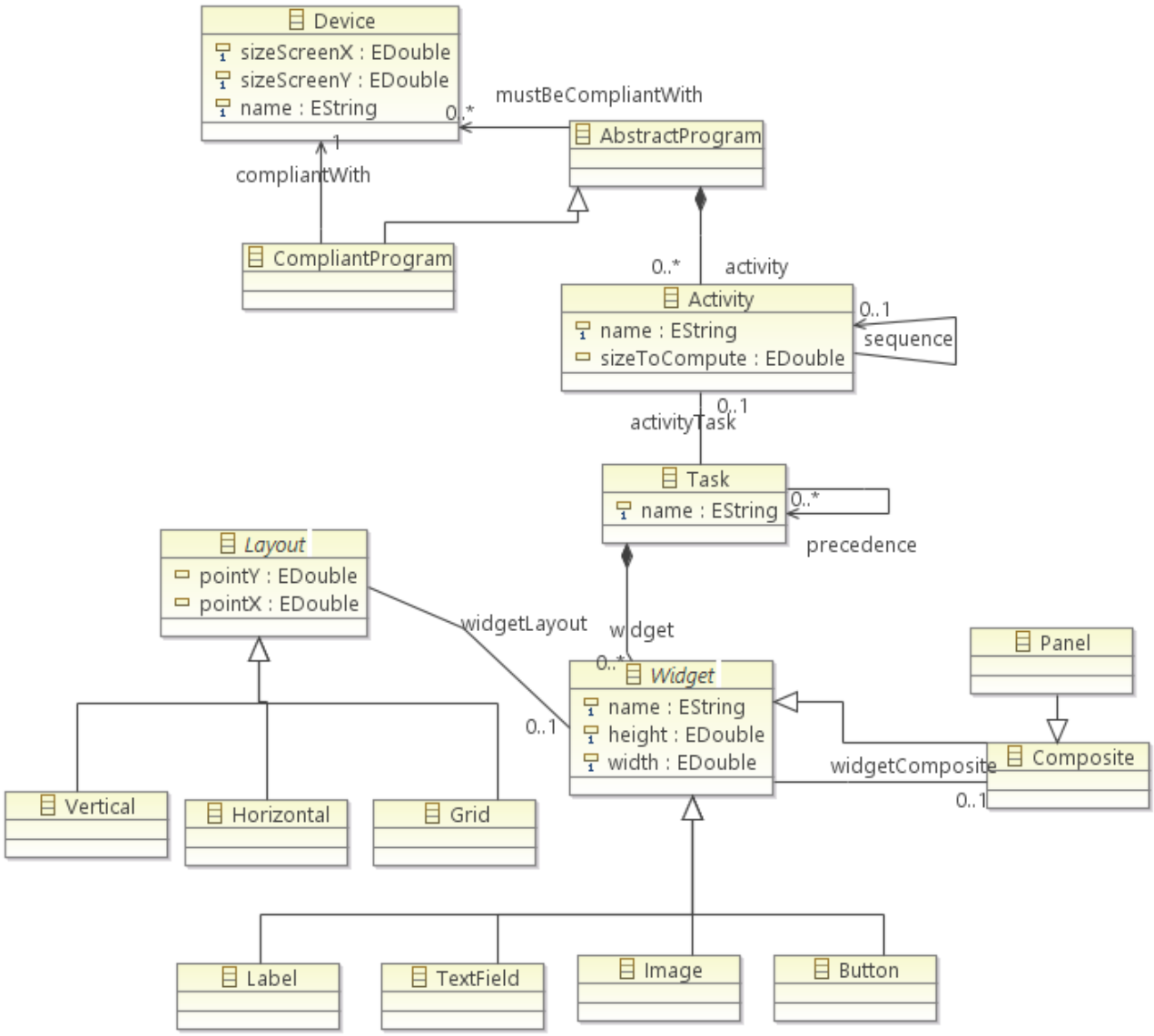

FIGURE 9.1 – Méta-modèle Ecore du cas d'étude

Modèles source en format XMI

La figure 9.2 illustre la représentation d'un fichier XMI dans Eclipse représentant l'instanciation du modèle en entrée, tous ses composants peuvent ici être visualisés.

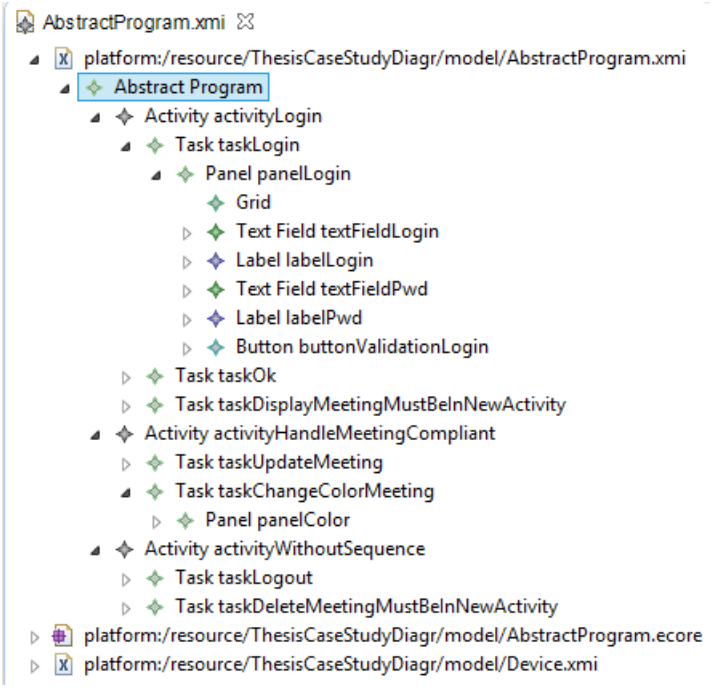

FIGURE 9.2 – Modèle source (*AbstractProgram*) en représentation XMI sous l'IDE Eclipse

9.2.2 Modèles cible résultant en format XMI

La figure 9.3 illustre également la représentation d'un fichier XMI dans Eclipse représentant l'instanciation du modèle en sortie généré.

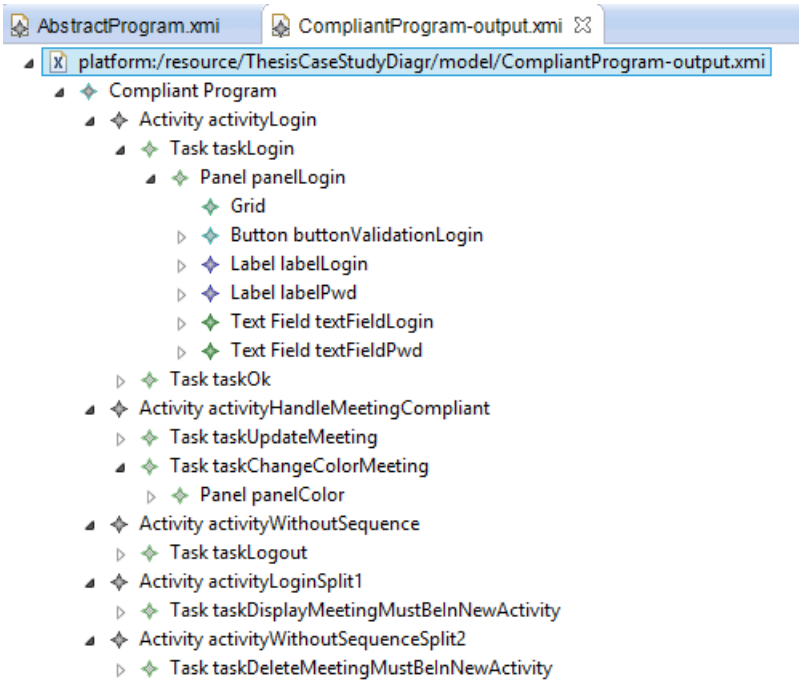

FIGURE 9.3 – Modèle cible résultant (*CompliantProgram*) en représentation XMI sous Eclipse

9.3 Évaluation des outils

Les langages sont évalués suite à l'implémentation de la transformation pour le cas d'étude. Ils sont principalement jugés sur les caractéristiques suivantes, tirées de la norme ISO 9126 :

utilisabilité : d'un point de vue de la difficulté d'apprentissage en termes d'effort et de temps, de la facilité de compréhension de la spécification du langage, les outils mis à disposition pour son utilisation ou encore son acceptation par la communauté ;

fonctionnalité : leur adéquation pour concevoir une transformation appropriée, d'un point de vue niveau d'abstraction du langage et utilisation des ressources aussi bien humaine comme l'effort de développement à fournir, que matériel tel que le temps d'exécution. Son interopérabilité et son utilisation de standards et la correction de la transformation seront également jugés ;

efficacité : la possibilité pour la transformation réalisées de fournir des performances raisonnables et ce avec des modèles de tailles diverses. Ou encore les outils mis à disposition pour améliorer l'efficacité de la transformation ;

maintenabilité : la taille de l'implémentation, la modularité et complexité du langage ainsi que la possibilité de factorisation de code ;

maturité : le nombre d'années que l'outil est disponible publiquement, le nombre de cas d'études existant, son utilisation par la communauté et industries.

Toutes les caractéristiques générales de la norme ISO ne sont pas prises en compte car elles ne sont pas toutes aussi pertinentes pour cette évaluation que celles sélectionnées, par exemple la portabilité n'est pas retenue. Nous n'avons également repris que les sous-caractéristiques les plus judicieuses. Les critères évalués ne sont pas pondérés bien que l'interopérabilité et la maturité soient des critères auxquels nous sommes particulièrement attentifs dans le cadre de ce travail. Les outils sont jugés de manière globale en tenant compte des exigences explicitées précédemment.

9.3.1 ATL

Cette implémentation a été réalisée avec la version *Luna* de l'IDE Eclipse et du plugin ATL 3.6.0 (ATL SDK - ATLAS Transformation Language SDK). La machine virtuelle d'ATL utilisée est « EMF-specific VM ».

Il existe également la machine virtuelle (VM) « EMF Transformation Virtual Machine » (EMFTVM), principalement une VM de recherche permettant des expérimentations et possédant des caractéristiques plus avancées du langage. La *release* stable 3.1.6 du plugin Eclipse d'ATL, datant du 15 janvier 2015, comprend par défaut EMFTVM. Cependant, la conception de la transformation avait déjà débuté avant que cette dernière ne soit rendue publique et une migration nécessitait quelques changements sur la transformation existante. Les principales différences sont qu'EMFTVM est plus puissante et avancée pour certaines caractéristiques d'ATL, telles que pour le mode « refining », expliqué plus loin dans la sous-section fonctionnalité. Toutefois, cette machine virtuelle n'existe que depuis 2011 et sa maturité est donc moins élevée.

Remarque : Nous avons fourni deux méta-modèles distincts pour la transformation, un représentant la source et l'autre la cible mais possédant tous deux un contenu identique. L'objectif étant d'éviter toute confusion sur les règles OCL de type *MetaModel!<nameElement>->allInstances()*, permettant de retourner toutes les instances d'une classe appartenant au méta-modèle donné. De ce fait, nous pouvons uniquement renvoyer les éléments du méta-modèle source, comme nous le faisons dans le « helper » *isActivitiesCompliant()*.

Utilisabilité

L'installation, configuration et prise en main d'ATL ont pu être réalisées rapidement et simplement grâce à l'existence d'une bonne documentation et des tutoriels sur le site de l'outil. Ensuite, l'intégration dans Eclipse offre une auto-complétion du langage et l'utilisation du débugger améliore son utilisation et sa maintenabilité. De plus, les ressources disponibles pour la compréhension du langage et son apprentissage sont abondants. Le forum fournit également la réponse à de nombreuses questions rencontrées par un développeur.
Un autre avantage est la gestion intégrée de méta-modèles Ecore et de modèles XMI dans Eclipse, permettant ainsi d'avoir un environnement de travail unique. L'effort d'apprentissage est moyen, la documentation est claire tout comme la spécification du langage. Les nombreux cas d'études disponibles aident à simplifier l'apprentissage du langage. En effet, ils sont pertinents et de bonne qualité. Pour conclure, c'est un langage accepté par la communauté.

Fonctionnalité

L'effort de développement pour la réalisation de la transformation peut être qualifié de moyen. D'une part, certaines constructions du langage ne sont parfois pas intuitives. D'autre part, l'effort est minimisé et compensé par le grand nombre de projets existants disponibles. La qualité et quantité de ces derniers est élevée et ils sont assez diversifiés, par conséquent il est possible de trouver un exemple pour beaucoup de tâches. Le code source de la transformation peut être retrouvé en annexe A.1. La complexité syntaxique et structurelle du langage est modérée.

L'implémentation est composée de plusieurs règles dont la plupart n'effectuent qu'une simple copie d'éléments du modèle source vers le modèle cible mais augmentant la taille totale de l'implémentation. Toutefois, la notion d'héritage de règles permet de limiter le code redondant et sa taille totale comme c'est le cas pour les modèles de type « Widget ». Étant donné que nos méta-modèles cibles et sources sont similaires, nous aurions pu utiliser le « refining mode » d'ATL, permettant de copier implicitement les éléments du modèle source vers le modèle cible et ainsi éviter ce code superflu. Cependant, nous employons la machine virtuelle « EMF-specific VM » et selon la documentation sur le site d'ATL, le « refining mode » dans cette dernière entrainerait des limitations car des éléments standards d'ATL n'y sont pas supportés comme les blocs d'actions, la fourniture de plusieurs éléments en entrée, etc. Ceci est planifié pour les prochaines versions et est déjà supporté avec la machine virtuelle « EMFTVM ».

Ensuite, plusieurs « helpers » ont été mis en place pour factoriser du code afin d'éviter qu'il soit redondant comme `computeSize` ou `getSizeTask`, respectivement mis à disposition pour calculer la dimension d'un élément donné et de retourner celle d'une tâche. Ceux-ci sont ensuite simplement appelés par les règles, aidant ainsi à améliorer la modularité et maintenabilité du code.

Il existe principalement deux règles où la logique du problème se concentre, la « matched rule » se nommant `ReduceActivity` et la « called rule » se nommant `Composition` permettant respectivement de réduire une activité existante et d'en créer une nouvelle. Les constructions impératives comme les « called rules » et « actions blocks » ont été d'une grande utilité pour résoudre le problème aisément. Le fait qu'ATL mette en place une approche hybride permet de trouver une solution simple à des parties du problème plus complexes. La partie déclarative d'ATL possède un niveau élevé d'abstraction mais notre cas d'étude requiert une partie impérative pour les parties plus complexes. Le niveau d'abstraction est ici moyen.

Concernant les inconvénients, le langage ne fournit pas la possibilité de définir des variables globales, pouvant être une difficulté pour le développeur en fonction du problème donné. Des alternatives peuvent être cependant trouvées. Aussi, la possibilité de lever une exception n'a pas été trouvée afin de notifier l'utilisateur que la transformation rencontre un problème ou ne peut être effectuée.

Pour terminer, son interopérabilité est possible grâce à *ATL Launcher* et l'appel de code JAVA ou encore *via* l'importation et exportation de fichiers XMI, comme explicité précédemment. ATL utilise les standards MOF, Ecore et OCL. Pour terminer, le modèle obtenu par le biais de la transformation est correct et valide.

Maintenabilité

ATL possède le plus grand nombre de lignes de code, ce qui ne contribue pas à une maintenabilité aisée. Aussi, le fait le « *refining mode* » n'est pas utilisé rend le code moins maintenable. En effet, si les méta-modèles évoluent, la transformation devra probablement être adaptée afin de pouvoir gérer ces changements et plusieurs règles seront ainsi impactées ou devront être ajoutées en cas de l'ajout d'éléments. La taille de la transformation est élevée.

La possibilité d'utiliser l'héritage entre règles, l'importation de modules est bénéfique pour la maintenabilité.

Efficacité

Le temps d'exécution de cette transformation est en moyenne de 0.4 seconde en utilisant 500.000ko de mémoire. « ATL Profiler » est un outil disponible afin d'évaluer facilement l'efficacité d'une transformation ATL ou encore d'aider le développeur à déceler un problème comme une portion de code inefficace dans son code. Cet outil explicite le nombre d'appels pour chaque opération effectuée pendant l'exécution, le temps d'exécution total, la mémoire utilisée, etc. Cet outil augmente l'efficacité du langage ainsi que sa maintenabilité.
Un récapitulatif des temps d'exécution est présenté dans le tableau 9.2, nous pouvons y constater que la performance d'ATL pour notre cas d'étude est la plus faible.

Maturité

La maturité du langage est très avancée et bénéficie du suivi d'une communauté conséquente. L'outil existe depuis 2003 environ et est accepté par la communauté et adopté par plusieurs industries comme expliqué précédemment. Il est également mis en œuvre dans de nombreux cas d'études. Pour terminer, des releases sont fréquemment rendues publique.

9.3.2 Epsilon (ETL)

L'implémentation a été conçue à l'aide de l'IDE Eclipse (*Luna*) Epsilon v1.2, comprenant déjà tous les plugins nécessaires à l'utilisation des langages de la famille Epsilon. La fourniture d'une distribution complète d'Eclipse contenant Epsilon et toutes ses dépendances permet un gain de temps et évite des problèmes de configuration, particulièrement pour les utilisateurs débutants avec Eclipse.

Utilisabilité

La possibilité de l'implémentation sous Eclipse entraîne les même avantages que cités pour ATL, à une différence près que l'auto-complétion était moins performante et elle n'était pas disponible pour accéder aux éléments des modèles dans la version utilisée. Il est également possible de se passer d'un IDE et d'utiliser le jar autonome mis à disposition par Epsilon afin d'utiliser les langages directement au sein d'une application Java ou Android [1]. L'interopérabilité de l'outil se voit ainsi améliorée. La configuration et prise en main de l'outil a été aisée également grâce la documentation et tutoriels disponibles.

L'effort d'apprentissage est faible, grâce à la mise à disposition d'un *ebook* clair, bien structuré et concis passant en revue les différents langages de la famille Epsilon disponibles avec exemples à l'appui. Le nombre de cas d'étude existant améliore également ce dernier. En outre, un forum très actif existe, offrant une aide rapide à l'utilisateur. La complexité syntaxique et structurelle du langage est faible, il est facile à prendre en main et à utiliser. C'est un langage accepté par la communauté.

Fonctionnalité

L'effort de développement peut être qualifié de faible. L'implémentation de la transformation a été réalisée deux fois avec un langage spécifique de la famille Epsilon, une fois à l'aide ETL et l'autre avec Flock. Les deux transformations ont été conçues avec la même facilité. Nous avons voulu illustrer la force d'Epsilon en illustrant la facilité de passer d'un langage à un autre et la diversité des langages proposés.

Il est aisé de passer d'un langage à l'autre et EOL aide grandement à cette tâche. En effet, le module EOL peut simplement être importé et réutilisé dans ETL et Flock. Le module EOL contient les opérations communes et plus complexes telles que le calcul des dimensions des éléments, connaître si une activité est conforme ou non (`isActivityTooBig`) ou encore récupérer la liste des tâches conformes pour une activité non conforme (`getAllFittingTasks`).

Tout d'abord, l'avantage de Flock pour le cas d'étude actuel est la possibilité d'utiliser le retypage (*via* le mot clé `retype`) permettant de changer le type du modèle *AbstractProgram* en *CompliantProgram*. Il est ainsi possible d'éviter la copie des éléments conformes et de se concentrer sur le cœur de la transformation même, rendre les autres *Activity* conformes.

```
1                retype AbstractProgram to CompliantProgram;
```

La solution avec Flock contient deux règles. La première (`AbstractProgram`) permet de contrôler la faisabilité de la transformation et levant une exception si ce n'est pas le cas. La seconde (`Activity`) permet de créer une nouvelle activité et d'y ajouter les tâches rendant l'activité initiale non conforme. Cette implémentation possède peu de lignes de code.

1. https://www.android.com/. Date : 04/05/2015

Ensuite, la solution avec ETL contient une règle supplémentaire ayant comme responsabilité de copier les activités conformes vers le modèle cible. Contrairement à ATL, il n'est pas nécessaire de spécifier tous les éléments à recopier vers le modèle cible, les sous-éléments des *Activity* sont recopiés implicitement, ce qui rend la taille de la transformation peu élevée et facilement maintenable. Le niveau d'abstraction est élevé concernant sa partie déclarative mais le fait que les langages soient hybrides permet de résoudre le problème facilement. Par exemple, le boucle ci-dessous permet de généraliser la gestion de la création de n activités si l'activité originale contient plusieurs tâches non conformes, ne pouvant être assignée à une seule nouvelle activité. Tant qu'il reste des tâches non conformes dans une activité originale, une nouvelle activité est créée et se voit attribuer une partie de ces tâches.

```
2                    while(abstractActivity.getAllFittingTasks().size>0) {...}
```

Epsilon utilise les standards MOF, Ecore et OCL. Son interopérabilité est possible *via* l'importation et exportation de fichiers XMI, la possibilité d'utilisation du langage JAVA ou du jar autonome pour appeler les différents langages d'Epsilon.

Pour terminer, le modèle obtenu par le biais de ces deux transformations est correct et valide.

Maintenabilité

La maintenabilité est grande grâce à la possibilité de réutilisation de modules EOL dans les différentes implémentations de la transformation. Aussi, la copie implicite de sous-éléments dans ETL l'augmente également et rend la transformation résistante face à la possible évolution des méta-modèles. En effet, la transformation ne devra pas être modifiée si des sous-éléments de *Activity* se voient être ajoutés ou supprimés. Ce constat peut être également fait concernant Flock grâce à son système de retypage. Aussi, la possibilité d'héritage entre les règles augmente la maintenabilité.

Efficacité

Les transformations réalisées avec Epsilon peuvent être d'une grande performance car chaque langage est spécialisé dans une tâche spécifique de l'IDM, Epsilon Flock est reconnu comme étant très efficace pour la migration de modèles.

Tout comme dans ATL, Epsilon propose un outil de profilage afin d'évaluer facilement l'efficacité d'une transformation ou encore d'aider le développeur à déceler un problème comme une portion de code inefficace.

Un récapitulatif des temps d'exécution est présenté dans le tableau 9.2, nous pouvons y constater que la performance d'ETL pour notre cas d'étude est moyenne.

Pour terminer, il existe un projet sur le *repository*[2] du projet Epsilon, permettant de tester les performances d'ETL avec le cas d'étude tiré d'un TTC dont l'objet est du *reengineering* où un graphe syntaxique abstrait (*abstract syntax graph*) doit être transformé en une machine à états [Horn, 2011]. Un des challenge est la performance et l'évolutivité. La transformation a été testée sur des modèles larges également et il en ressort qu'elle peut être exécutée en 4 minutes pour un modèle de 100 Mo, ce qui est 10 fois moins rapide que GReTL dans ce cas (lui-même étant 10 fois moins rapide que GrGen.net). Ce qui confirme la supériorité de GrGen.net d'un point de vue de la performance. Cependant, des patchs semblent avoir été développé pour y remédier et améliorer ses performances.

2. https:\//git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/plain/performance/TransformLargeSourceModel/README.textile. Date : 22/04/2015

Maturité

L'outil existe depuis 2006 environ et bénéficie d'une grande maturité. Il est accepté par la communauté et adopté par plusieurs industries comme expliqué lors de sa présentation. Il est également mis en œuvre dans de nombreux cas d'études. Son forum est très actif et chaque question reçoit une réponse rapide. Pour terminer, des releases sont fréquemment rendues publique.

9.3.3 GrGen.NET

La transformation a été développée dans Notepad++ et la version v4.4 de GrGen.NET a été utilisée. GrGen.NET est un langage mettant en place une approche par transformation de graphes donc les modèles concrets tels que les *Activity* sont représentés sous-forme de nœuds et les relations entre chaque modèle par des arcs. La figure 9.4 illustre cette représentation pour la partie centrale du modèle cible.

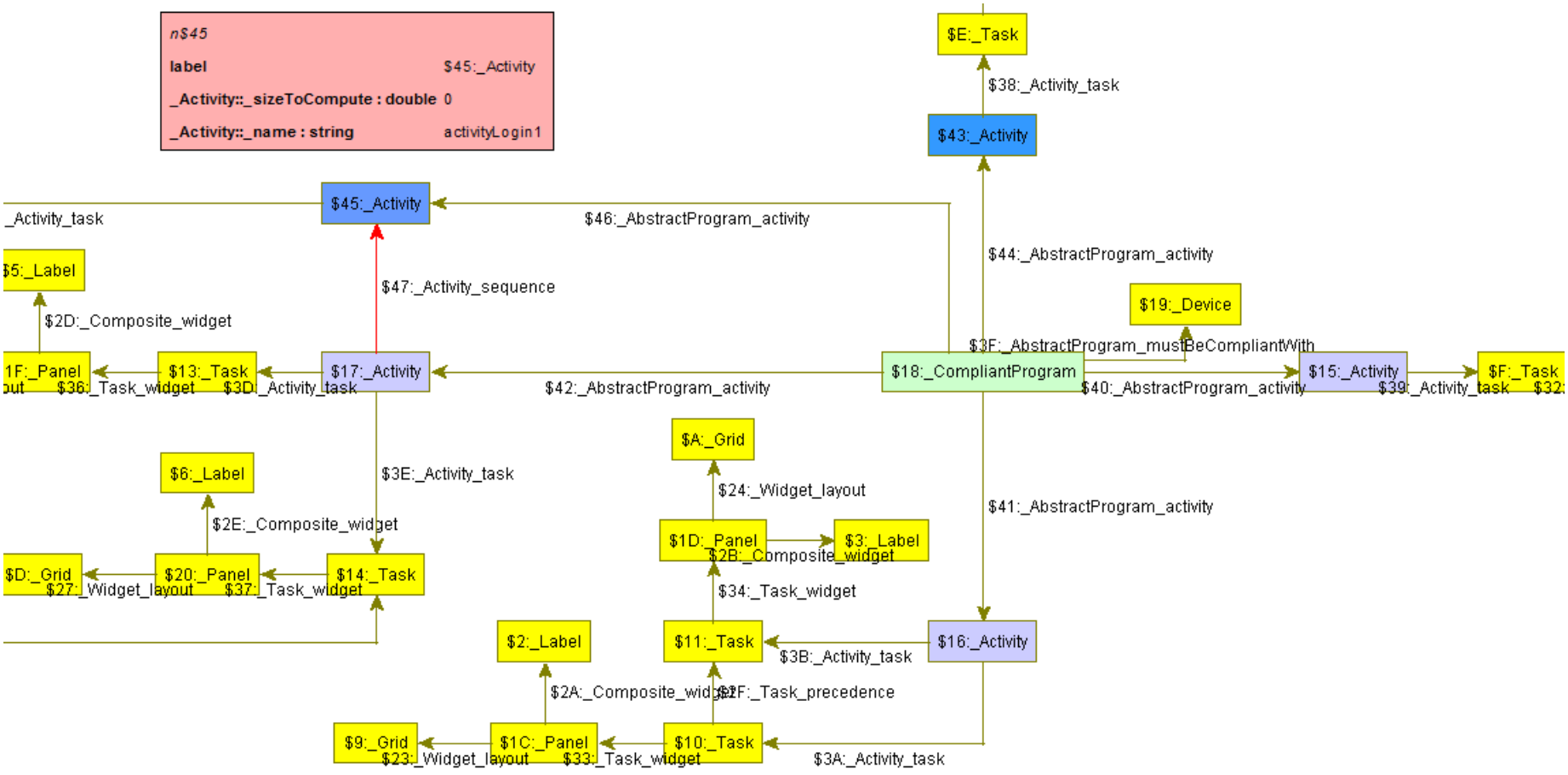

FIGURE 9.4 – Modèle cible résultant (CompliantProgram) sous forme de graphes (Ycomp-GrGen.net)

Utilisabilité

Le manque d'intégration de l'outil dans un IDE soulève quelques désavantages comme le manque d'auto-complétion et d'environnement intégré. Cependant, un fichier de configuration est fournie pour permettre une coloration syntaxique dans plusieurs éditeurs de texte comme Notepad++. Aussi, un fichier .grs doit être réalisé par l'utilisateur afin d'indiquer lui-même les différentes règles à exécuter ainsi que spécifier des conditions à ces exécutions. Il doit également spécifier les options désirées pour la transformation telles que l'utilisation du débugger simple et/ou le débugger graphique *Ycomp*. Toutefois, cela laisse un grand contrôle à l'utilisateur. *Ycomp* est très intéressant et puissant, c'est l'un des grands avantages de ce langage.

L'effort d'apprentissage peut être qualifié de moyen. Les cas d'études ne sont pas accompagnés de documents explicatifs, contrairement aux autres langages. Sa documentation est bien plus dense et il est moins simple pour un débutant de s'y repérer. Toutefois, l'avantage est que cette dernière est très complète et permet de refléter une certaine puissance de l'outil.

Plusieurs problèmes ont été rencontrés lors de son utilisation, comme explicité dans la sous-section *maturité* ci-dessous. Ces problèmes peuvent entrainer une perte de temps pour l'utilisateur et demande un effort de compréhension et d'analyse. De plus, aucun forum n'est disponible or c'est une source d'informations intéressante et facile d'accès à tous. Cependant, l'équipe en charge du projet est très réactive par contact mail. A la lumière de ces faits, l'effort de développement peut être catégorisé de moyen voir élevé dû aux problèmes rencontrés.

Pour terminer, bien qu'il mette en place une approche basée sur les transformations de graphes, sa gestion de méta-modèle Ecore et d'importation et d'exportation de fichiers XMI est très efficace et facile de prise en main.

Fonctionnalité

La complexité syntaxique et structurelle est modérée voir élevée pour un débutant. En effet, les structures sont assez complexes et de nombreux sous-blocs composent une seule règle avec chacun un rôle bien distinct. L'effort de développement peut être qualifié de moyen. Le niveau d'abstraction du langage est élevé.

Ici la règle `abstractToCompliant` permet de retyper un *AbstractProgram* vers un *CompliantProgram*, évitant ainsi la copie des différents éléments.

Les manipulations sur les modèles peuvent être effectuée facilement grâce à la puissance du langage et l'utilisation d'un représentation par graphes. Par exemple, la règle « addSequenceBtwActivities » invoquée dans la règle `createActivity` à l'aide du mot clé *exec* permettant d'ajouter facilement une séquence entre deux activités. Ou encore `redirectNotFittingTasks` permettant de rediriger les arcs des tâches rendant une activité non conforme vers une nouvelle activité. Cela évite des copies/créations et suppression de tâches.

Un pattern est mis en place afin de modulariser le code tel que `searchNotFittingTasks`, son fonctionnement est similaire au *helper* du même nom dans ATL, il permet de vérifier chaque activité et d'en créer de nouvelles pour les cas de non-conformité en appelant la règle `createActivity`. Le bloc *iterated* assure que tous les constructions y étant encapsulées sont appliquées tant que des éléments peuvent être matchés. Dans ce pattern, la gestion des précédences peut être effectuée facilement à l'aide de la méthode `reachableEdges`, permettant de récupérer les arcs entre deux nœuds donnés. Quant à son interopérabilité est possible *via* l'importation et exportation de fichiers XMI et l'utilisation de son API en .net fournie.

Aussi, l'outil permet l'utilisation de standards tels qu'Ecore, XMI et GXL.

De plus, son interopérabilité est facilitée par l'API (.NET) fournie ou par l'importation et exportation de fichier XMI par exemple.

Pour terminer, la transformation génère un modèle cible correct.

Maintenabilité

Chaque règle possède une responsabilité propre. De plus, la possibilité d'utiliser des *patterns*, procédures et fonctions augmentent la maintenabilité. La possibilité de retyper le *AbstractProgram* en un *ConcreteProgram* améliore également cette caractéristique, rendant la transformation résistante aux évolutions futures du méta-modèle. Aussi, le code gagne en concision.

```
1            compliantProgram:_abstractPrograms::_CompliantProgram<abstractProgram>;
```

De plus, « les *subpatterns* permettant la réutilisation de pattern commun et le matching de sous-structures *via* la récursion en leur sein » [Jakumeit Edgar, 2014].

Efficacité

Un récapitulatif des temps d'exécution est présenté dans le tableau 9.2, nous pouvons y constater que la performance de GrGen.net est la meilleure, comme le confirme la littérature en général.

Maturité

L'outil existe depuis 2003 et bénéficie d'une certaine maturité. Toutefois, bien que le langage soit relativement mature et très bien évalué dans la littérature en général, nous avons pu ressentir qu'il était moins maintenu ou avait un caractère moins industrialisé qu'ATL et Epsilon. En effet, lors de sa première utilisation, nous avons constaté que l'outil n'était pas compatible avec Windows 8. Or, ce système d'exploitation est disponible depuis 2012. Ensuite, deux autres bugs ont été rapportés et fixés rapidement par l'équipe de GrGen.NET, entrainant la sortie publique de la release v4.4.1 sur le site internet de l'outil contenant ces résolutions de bug. Ces derniers bugs étaient liés à la possibilité d'itérer dans un *subpattern* ou la gestion de plusieurs façons différentes le *casting* des éléments dans un objet de type *Set*.

9.3.4 Prolog

L'implémentation de la transformation à l'aide de Prolog nécessite un effort de développement non négligeable et n'a donc pas été réalisée dans le cadre de ce mémoire. En effet, la transformation en elle-même peut être réalisée dans ce langage avec une complexité limitée mais un approfondissement ce dernier dépassant le cadre du mémoire est tout de même demandé. Dès lors, nous allons nous baser sur les travaux existants dans le domaine tels que [Almendros-Jiménez et Iribarne, 2008], [Almendros-Jiménez et Iribarne, 2013] ou encore [Störrle, 2009] afin d'envisager les perspectives possibles et extrapoler nos conclusions.

Utilisabilité

L'avantage de Prolog est sa connaissance par un grand nombre de développeurs et l'abondance de documentation à son sujet. L'effort d'apprentissage à fournir pour parvenir à concevoir des transformations est élevé. En effet, il faut gérer une multitude de points tels que

l'importation des modèles sources et l'exportation des modèles cibles ainsi que leur représentation, leur validation, la traçabilité, etc. Ce qui demande une connaissance avancée du langage. De plus, il faut tout concevoir « from scratch » car il n'existe pas réellement d'exemples ou beaucoup d'autres outils de la sorte.

D'un point de vue des outils mis à disposition, il existe maintes implémentations de Prolog fournissant un éditeur de texte, debugger, parfois un plugin Eclipse, etc. Prolog est accepté et reconnu par la communauté des développeurs en général. Cependant, il n'est pas répandu dans le milieu de l'IDM en comparaison à d'autres outils tels qu'ATL ou ETL.

De plus, « ses expressions sont plus simples et concises que leur correspondance avec OCL »[Störrle, 2009]. L'utilisabilité d'OCL et ses performances sont assez médiocre et Prolog se révèle plus efficace [Störrle, 2011]. Par conséquent, il y a des initiatives telles que LQF (*Logical Query Facility*) [Störrle, 2009] tentant de réaliser une API de haut niveau à l'aide de Prolog pour effectuer des requêtes sur des modèles par l'utilisateur.

À la lumière de ces faits, nous pouvons conclure que les capacités de Prolog sont grandes et qu'il possède des avantages face à d'autres approches, utilisant OCL pour effectuer des requêtes sur les modèles par exemple.

Fonctionnalité

La figure 9.6 illustre une possibilité de fonctionnement général d'une transformation effectuée en programmation logique telle que nous aurions pu la réaliser. Comme expliqué dans [Almendros-Jiménez et Iribarne, 2008], chaque élément du méta-modèle est représenté sous forme de faits Prolog. Le méta-modèle pourra ensuite être instancié sous cette forme à son tour. Les règles de transformations seront définies à l'aide de règles logiques. Des *model query rules* définiront les nouveaux éléments du modèle cible. Tandis que des règles de transformation se chargeront de traduire les nouveaux éléments définis par les *model query rules* dans le même type de représentation que le modèle source. Un principe similaire est utilisé dans l'outil MoMaT [Störrle, 2007], les modèles y sont représentés sous forme de faits Prolog. La figure 9.5 illustre l'architecture de cet outil, nous permettant ainsi de visualiser ce qui existe et peut être mis en place.

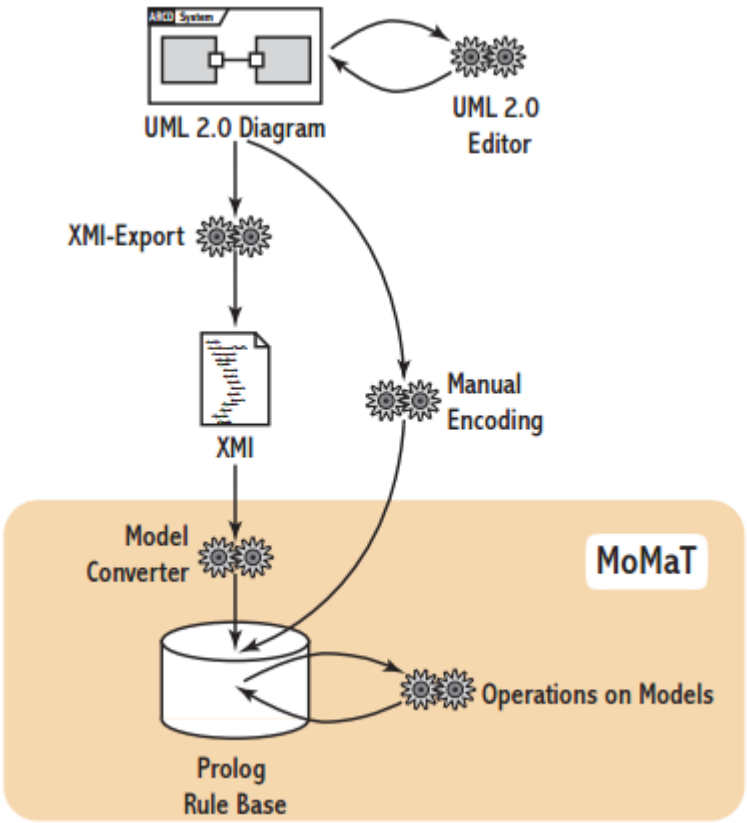

FIGURE 9.5 – Architecture de MoMaT, issue de [Störrle, 2005]

Ensuite un interpréteur Prolog pourra être utilisé afin d'obtenir les nouveaux faits Prolog représentant le modèle cible *via* les transformations précédemment définies.

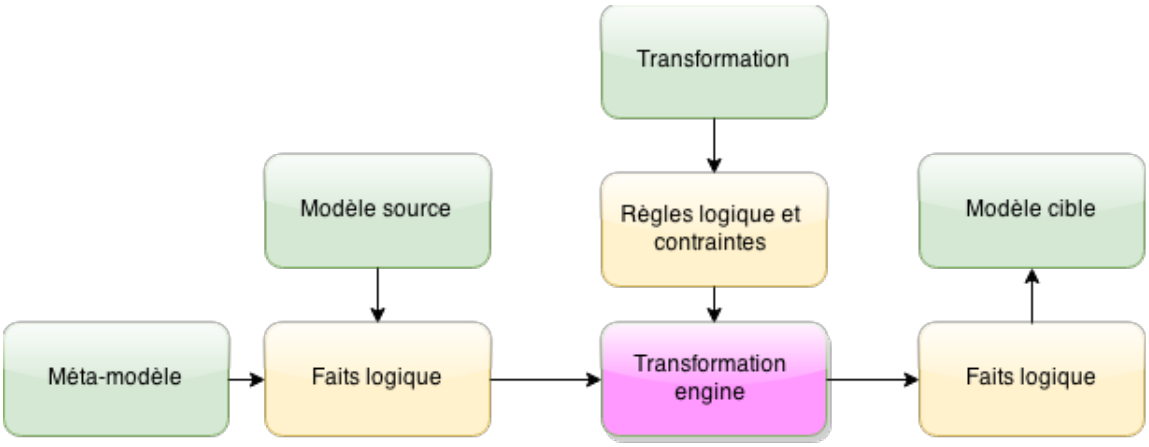

FIGURE 9.6 – Programmation logique : représentation et fonctionnement

Le langage PTL utilise Prolog comme moteur de transformation. Il emploie des règles de style ATL pour effectuer le mapping entre le modèle source et cible tandis que des règles logiques sont utilisées comme « helpers ». Ces règles de style ATL sont encodée par des règles

Prolog. Le site[3] de PTL met à disposition du code pouvant être utilisé pour importer et exporter des fichiers au format XMI que nous pourrions ré-utiliser. Aussi, ce framework met en œuvre plusieurs cas d'études provenant de l'ATL ZOO[4].

Il existe plusieurs implémentations de Prolog compatibles avec l'IDE Eclipse et le langage Java tels que SWI-Prolog[5] *via* l'interface JPL[6] ou B-Prolog[7].

Prolog est compatible avec le standard XMI. En effet, la plupart des implémentations de Prolog disposent d'une librairie permettant la gestion des XML, facilitant ainsi l'importation et exportation des modèles en format XMI et augmente aussi son interopérabilité. Il peut également s'interfacer avec CORBA comme décrit dans [Steiner, 2001]. Aussi, un compilateur[8] CORBA IDL existe pour l'outil (SWI-)Prolog.

Le niveau d'abstraction du langage est élevé mais l'effort de développement total pourrait s'avérer élevé dû à la multitude de dimensions à gérer en plus de conception de la transformation.

Efficacité

Prolog offre un haut niveau d'abstraction et une grande puissance de calcul. Il est plus efficace qu'OCL pour effectuer des requêtes sur les modèles par exemple [Störrle, 2011].

Quant aux réalisations existantes, selon [Störrle, 2007] MoMaT est capable de gérer des modèles très larges et offre des opérations avancées sur les modèles.

Selon le site de PTL, ses performances ont été testées dans le cadre de la validation de contraintes à l'aide d'un benchmark[9] incluant la manipulation de modèles relativement larges avec succès. Cependant, aucune information supplémentaire ou métriques n'est disponible à ce sujet. Il n'est donc pas aisé de quantifier et évaluer son efficacité.

Maintenabilité

La taille du code de la transformation en Prolog serait faible voire moyenne car il possède un niveau d'abstraction élevé. Aussi, le code peut être réutilisable, d'un niveau d'abstraction élevé et concis à l'aide des *higher order predicates* comme expliqué dans [Sadilek et Wachsmuth, 2008]. Des faits et contraintes peuvent être facilement supprimés ou ajoutés sans impacter le reste du code, rendant le code résistant aux changements.

Maturité

La maturité du langage Prolog n'est plus à démontrer. Toutefois, sa mise en place pour la conception de transformation de modèles manque encore de maturité. En effet, la littérature à ce sujet est encore limitée et peu de réalisations réellement abouties existent sur ce sujet ou restent très académiques.

Par exemple, PTL met à disposition l'utilisation d'un debbuger et d'un système de traçabilité afin d'aider les développeurs lors de leur implémentation de transformations. Une

3. http://indalog.ual.es/mdd/ptl2. Date : 4/03/2015
4. http://www.eclipse.org/atl/atlTransformations/. Date : 19/05/2015
5. http://www.swi-prolog.org/. Date : 16/04/2015
6. http://www.swi-prolog.org/packages/jpl/java_api/. Date : 16/04/2015
7. http://www.picat-lang.org/bprolog/. Date : 16/04/2015
8. http://www.cs.vu.nl/~eliens/documents/corba/pl/plcorba.html. Date : 08/04/2015
9. http://incquery.net/publications/trainbenchmark-scp. Date : 28/04/2015

validation pour vérifier les contraintes des modèles et des transformations est également disponible. Aussi, un plugin Eclipse a été développé afin de permettre l'intégration de PTL à cet IDE. Cependant, PTL n'est pas encore très mature. Moins d'une dizaine d'exemples sont disponibles (repris de l'ATL Zoo) et ceux-ci ont une difficulté faible voir moyenne. Quant à MoMaT, il est surtout développé dans le milieu académique mais aussi dans deux projets industriels importants (à la date où l'article a été écrit) [Störrle, 2007]. La maturité de Prolog pour la réalisation de transformations peut être qualifiée de moyenne.

9.3.5 Comparaison des outils

Le tableau 9.1 offre un récapitulatif des constatations précédentes. Les valeurs +1, +2 et +3 correspondent respectivement à une appréciation faible, moyenne et élevée.

	Langages				
Caractéristiques	***Caractéristique***	***Attributs***	ATL	Epsilon	GRGen.NET	Prolog
	Utilisabilité	Facilité apprentissage	+2	+3	+2	+1
		Effort de dév.	+1	+3	+3	+2
	Fonctionnalité	Complexité	+2	+3	+2	+2
		Taille du code de la transformation	+1	+3	+3	+2
		Nombre de lignes de code	203	99	95	/
		Niveau d'abstraction	+2	+2	+3	+3
	Maturité		+3	+3	+3	+2
	Efficacité	Performance	+1	+2	+3	+3
	Interopérabilité		+2	+3	+2	+3

Table 9.1 – Exposition des caractéristiques fonctionnelles des outils

Afin de comparer aisément les performances de chaque langage, le tableau 9.2 reprend les temps d'exécution de chacune des transformations pour différents modèles sources. Ces derniers sont avec chargement des modèles inclus.

Les temps d'exécutions ne sont pas très représentatifs des capacités réelles des outils mais permet tout de même de les comparer, les transformations ne sont pas vraiment optimisées car elles ont été conçues par un débutant en la matière. Les meilleures performances sont de loin celles de GrGen.net. Celles d'ETL sont moyennes. Quant à ATL, ce sont les plus mauvaises.

	Langages		
Modèle source	**ATL**	**ETL**	**GrGen.Net**
Modèle présenté (5ko)	15 ms	27 ms	40 ms
Modèle 5Mo	134.756 ms	69.908ms	453 ms
Modèle 10Mo	517.776 ms	283.170 ms	938 ms

Table 9.2 – Temps d'exécution selon des modèles de différentes tailles

9.4 Conclusion

Une évaluation des outils a pu être menée de façon objective grâce à la norme ISO-9126. Chaque outil est jugé sur ses caractéristiques d'un point de vue utilisabilité, fonctionnalité, efficacité, maintenabilité et maturité. Cependant, certains de ces critères peuvent se révéler être un peu subjectif tels que la facilité d'apprentissage. Ces critères pourraient être réévalués en fonction du contexte donné, d'un cas d'étude différent, etc. Ils aident tout de même de mettre en évidence les avantages et inconvénients principaux de chaque outils.

9.4.1 Discussion

Un choix multicritère a été effectué et le résultat tend vers Epsilon. Il est l'outil le plus interopérable et la possibilité de développer un driver pour étendre les technologies supportées par l'outil est un avantage. Aussi, il est très concis, le plus facile d'apprentissage, mature, extensible, etc. Quant à GrGen.net, il est très intéressant et pourrait être classé « second ». C'est l'outil le plus performant mais il semble manquer de maturité.

ATL ne se démarque pas et possède quelques faiblesses telles que sa performance ou son manque de concision. De plus, la maturité quant à sa nouvelle machine virtuelle peut être qualifiée de faible. Concernant Prolog, il n'existe pas de couche applicative disponible aussi mature que pour les autres outils dans le domaine des transformations de modèles. Aussi, concevoir des transformations « from scratch » pourrait nécessiter une charge non négligeable de travail. Par conséquent, il pourrait être plus avantageux de s'intéresser et suivre l'évolution des solutions existantes ayant été citées afin d'en tirer profit et d'éventuellement les utiliser.

Chapitre 10

Architecture d'intégration dans MetaDONE

Dans ce chapitre, les possibilités d'intégration d'un outil de transformation de modèles sont exposées. La première section expose les possibilités pour rendre compatibles les différents de (méta-)modèles utilisés dans MetaDONE et l'outil de transformation de modèles. La seconde section propose des pistes d'intégration possibles entre MetaDONE et les outils de transformations évalués en tenant compte des dimensions vues précédemment. Bien qu'une préférence ait été formulée pour ETL, nous discutons tout de même brièvement de l'intégration des autres outils.

10.1 Gestion de la technologie de représentation des (méta-) modèles

L'intégration des outils peut s'effectuer au niveau de la technologie de représentation des (méta-) modèles afin de permettre un partage de données compatibles entre les outils. Les pistes envisagées sont la réalisation d'un *loader/exporter* dans MetaDONE et la conception d'un driver pour Epsilon afin qu'il supporte le méta-méta-modèle de MetaDONE.

10.1.1 Importation/exportation de (méta-) modèles dans MetaDONE

Actuellement, MetaDONE est capable d'importer des méta-modèles de type KM3, OWL [McGuinness *et al.*, 2004] et EMF à l'aide d'un *loader*. Ces méta-modèles permettent ensuite d'instancier un modèle.

Il est envisageable d'implémenter un *loader* et *exporter* de (méta-) modèles sur deux niveaux. Deux possibilités pour cette implémentation sont possibles, décrites ci-après.

Dans le premier cas, un *class diagram* représentant un méta-modèle EMF (*Place*) est importé dans MetaDONE. Ensuite, un mapping implicite est effectué entre ce *class diagram* et un méta-modèle dans MetaDONE (*Place :MO*[1]). Le méta-modèle EMF devient donc un méta-modèle MetaDONE, rendant ainsi possible l'échange de leurs instances respectives (*P1*). Il est ainsi permis d'importer et d'exporter des (méta-)modèles EMF dans MetaDONE. La figure 10.1 illustre les différents niveaux d'abstraction et les mappings possibles. Les concepts utilisés sont issus d'un méta-modèle de Petri Net (figure 10.2).

1. Meta-Object

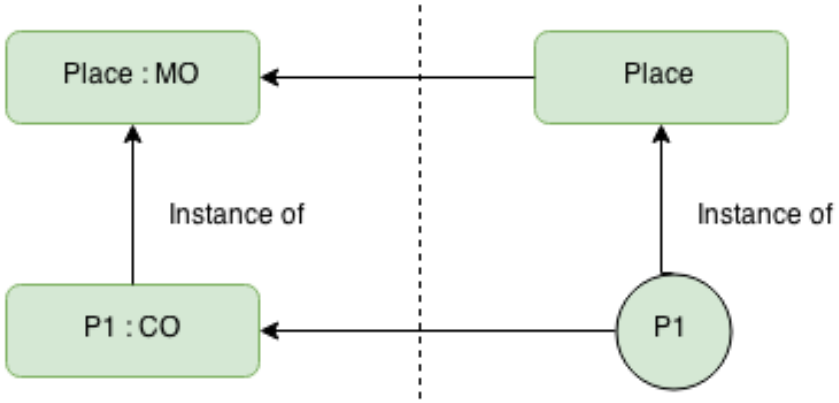

FIGURE 10.1 – Possibilités d'intégrer un *Loader* et *exporter* sur deux niveaux dans MetaDONE (option 1)

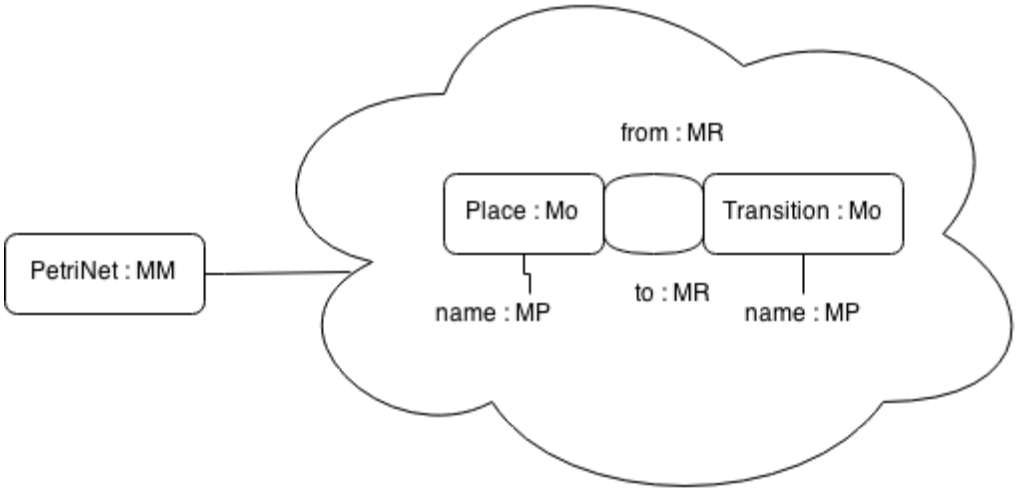

FIGURE 10.2 – Méta-modèle de Petri Net

Dans le second cas (figure 10.3), la mise en œuvre est divisée en deux phases. Tout d'abord, il doit exister préalablement un méta-modèle du langage EMF dans MetaDONE (*EMFClass :MO*). Lors de la phase 1, un méta-modèle EMF sera importé en tant que modèle MetaDONE (*Place :CO*), instance du méta-modèle mentionné précédemment et non plus comme méta-modèle. Ensuite, la phase 2 fait intervenir une étape de traduction qui transforme les objets (*Place :CO*[2]) représentant le méta-modèle EMF dans MetaDONE en méta-objet dans ce dernier (en mauve). Il est alors permis d'importer les objets EMF (*P1*) dans MetaDONE (*P1 :CO*).

2. Concrete-Object

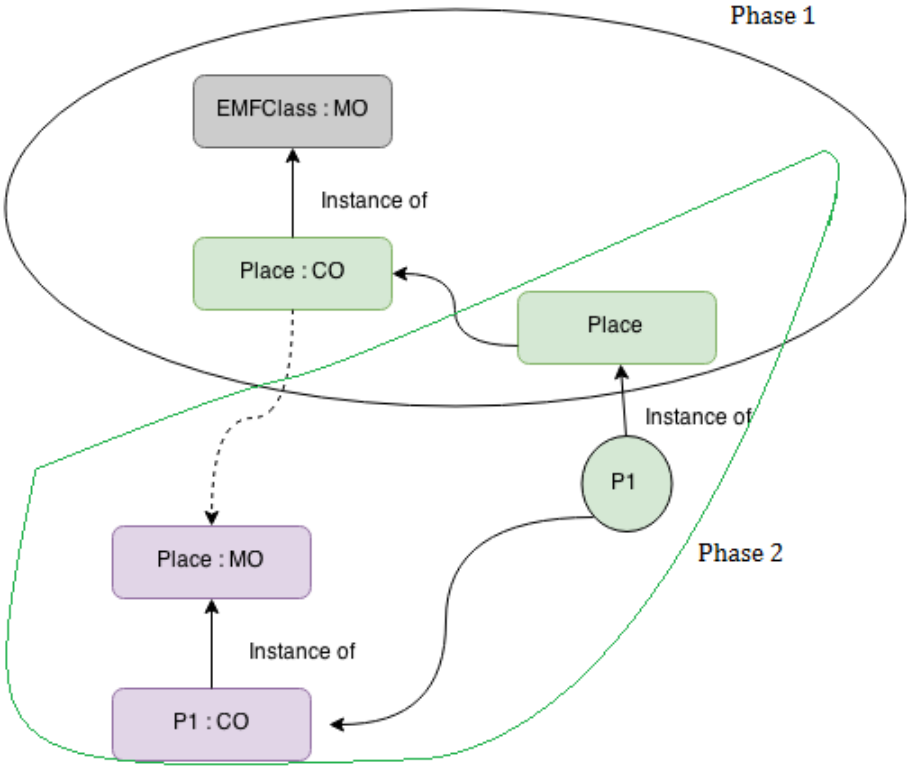

FIGURE 10.3 – Possibilités d'intégrer un *Loader* et *exporter* sur deux niveaux dans MetaDONE (option 2)

Pour terminer, la première option est sans doute plus simple à mettre en œuvre. Cependant, elle a le désavantage d'éventuellement perdre des informations lors des conversions. Toutefois, ce désavantage existe également dans la deuxième option, mais le modèle d'origine est stocké dans le *repository*, rendant possible la mise en place d'une traçabilité éventuelle.

10.1.2 Conception d'un driver Epsilon pour supporter le méta-méta-modèle de MetaDONE

Epsilon fournit un support robuste pour EMF mais n'est lié à aucune technologie particulière concernant sa gestion de modèles. En effet, Epsilon repose sur un *open model connectivity framework* pouvant être étendu par un développeur en lui fournissant un nouveau *driver* afin que les langages puisse supporter des technologies supplémentaires.

La couche *Epsilon Model Connectivity* (EMC) fournit une interface uniforme permettant à un programme de type Epsilon d'interagir avec un modèle conforme aux technologies supportées. La figure 10.4 illustre l'architecture d'Epsilon où EMC occupe une place centrale. Différents *drivers* spécifiques à une technologie ont déjà été définis tels qu'EMF, MetaEdit+ ou encore MDR (*Meta Data Repository*) [Kolovos *et al.*, 2010]. MDR est une base de données crée pour stocker des informations sur les structures contenant les données.

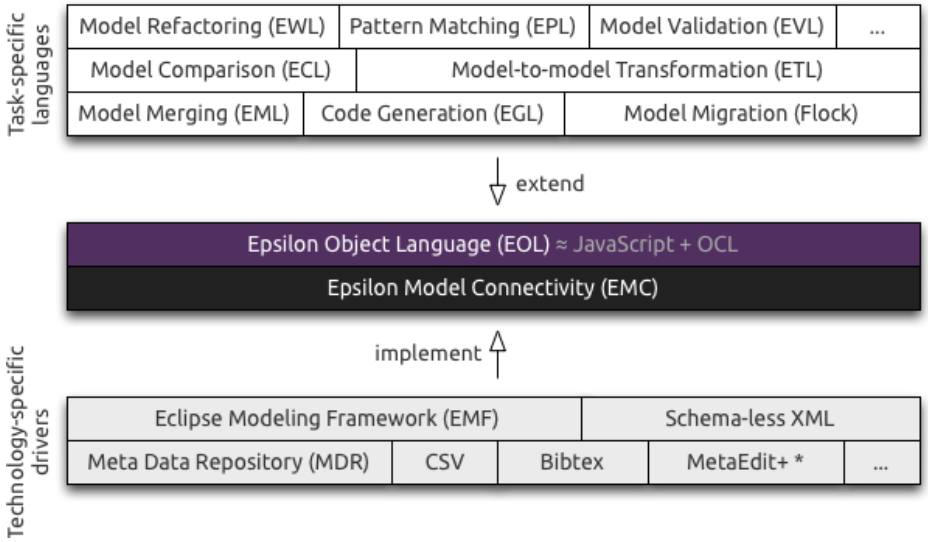

FIGURE 10.4 – Représentation de l'architecture d'Epsilon Model Connectivity (EMC), tiré du site de l'outil

EMC met à disposition les classes et interfaces nécessaires à la conception d'un nouveau *driver*. Par exemple, les éléments d'un modèle peuvent être gérés (*query* et modifications) avec les méthodes de l'interface *IModel*. La classe *ModelRepository* est un conteneur offrant des services *via* le *design pattern* Façade[3] afin de permettre aux outils reposant sur EMC d'accéder aux modèles, elle permet également de gérer les transactions. Des informations plus détaillées et complètes à ce sujet sont disponibles à ce sujet dans l'e-book d'Epsilon.

EMC pose des hypothèses minimales quant à la structure et l'organisation de la technologie de représentation du modèle. Il s'abstient de redéfinir les concepts tel que les méta-modèles, types, etc. Une approche légère est employée, utilisant des chaînes de caractères (*string*) pour définir les noms et objets des plateformes de l'implémentation cible comme les éléments du modèles. L'objectif principal de la mise en place de cette approche étant de rendre EMC résistants aux éventuels changements des implémentations des technologies actuelles mais aussi d'adopter de nouvelles technologies sans changements. De plus, cela permet de préserver les performances contrairement à une approche plus lourde consistant à fournir des *wrapping objects* pour les objets natifs représentant des éléments et type de la nouvelle technologie à supporter. [Kolovos *et al.*, 2010]

3. http://fr.wikipedia.org/wiki/Fa%C3%A7ade_%28patron_de_conception%29. Date : 20/05/2015

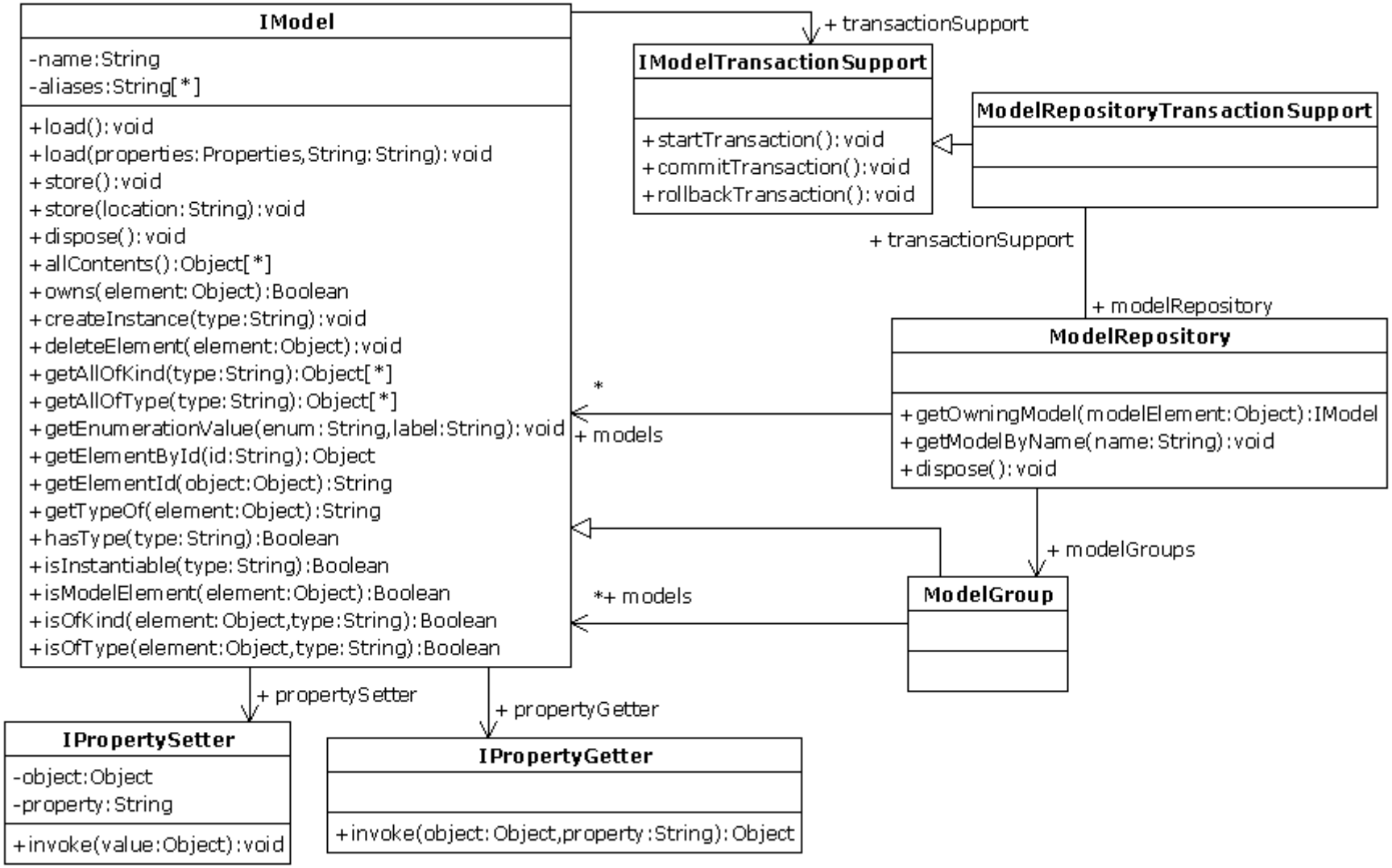

FIGURE 10.5 – Vue d'ensemble du *design* d'EMC ([Kolovos *et al.*, 2010])

L'avantage de cette option est la préservation de la technologie de représentation de (méta-)modèles dans MetaDONE. Ainsi, les modèles gardent toute leur expressivité et l'outil ne doit pas subir de grandes modifications.

10.2 Intégration des outils

L'intégration des données pourrait être atteinte *via* un *rep*ository central où les modèles seraient stockés et partagés entre MetaDONE et l'outil de transformation de modèles. Toutefois, un *repository* est déjà employé au sein de MetaDONE et des problèmes de performance lié à ce dernier sont rencontrés en mode écriture, cette possibilité n'est donc pas envisageable actuellement. D'autres alternatives pourraient être l'utilisation de fichiers intermédiaires ou l'échange de messages.

De plus, l'intégration du contrôle, permettant aux outils de s'échanger des messages afin de déclencher des opérations, pourrait être réalisée aisément à l'aide de l'API fournie par un outil. Cette possibilité combinée à celle proposée pour l'intégration des données permettrait une intégration simple, flexible et efficace. Les outils concernés par cette option sont GrGen.net et ATL. Autrement, le standard CORBA pourrait être mis en place avec un outil tel que Prolog par exemple.

Aussi, comme le suggère [Brownsword *et al.*, 2004], l'outil pourrait être encapsulé dans MetaDONE. Cette mise en œuvre est tout à fait possible pour ETL grâce au jar autonome mis à disposition par l'outil. Le jar d'Epsilon peut être inclus au sein de MetaDONE et par conséquent, ETL (et les autres langages de la famille Epsilon) pourrait être disponible à l'aide de code JAVA [4] afin de mettre en place des transformations ou en exécuter.

10.3 Discussion

Une solution combinant l'option de la conception d'un *driver* avec Epsilon et l'implémentation d'un *loader* et *exporter* peut être envisagée. Elle permettrait de s'affranchir de la dépendance aux outils Epsilon liée au *driver* développé. Elle rendrait également possible l'utilisation de standards par MetaDONE tel qu'EMF tout en continuant à laisser la possibilité à l'utilisateur de travailler avec des modèles de types MetaDONE, Ecore ou autre. En outre, les (méta-)modèles existants de type MetaDONE seraient toujours supportés et compatibles grâce au développement d'un driver Epsilon.

Quant à l'intégration du contrôle, le jar d'Epsilon peut être inclus au sein de MetaDONE lui permettant d'utiliser ETL et les autres langages de la famille Epsilon. Enfin, l'intégration des données pourraient être mise en place par l'une des pistes citées mais cela nécessite un approfondissement.

4. https://eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.standalone. Date : 11/05/2015

Chapitre 11

Conclusion

Dans ce mémoire, nous proposons des pistes pour sélectionner un outil de transformation de modèles et l'intégrer dans un outil de méta-modélisation (MetaDONE). Pour ce faire, les exigences précises du problème sont établies et un cas d'étude concret est défini afin d'évaluer les différents outils. Nous répertorions ensuite différentes architectures d'intégration réalisables pour intégrer l'outil dans MetaDONE en tenant compte des contraintes posées par ces derniers.

Dans les premiers chapitres, nous situons la question des transformations de modèles dans son contexte par la définition des concepts liés à l'ingénierie dirigée par les modèles.

Nous dressons par la suite un état de l'art sur les transformations de modèles. En effet, nous y détaillons les principales composantes et caractéristiques pouvant être possédées par les transformations de modèles et exposons les avantages et inconvénients des principales approches modèles vers modèles existantes. Ainsi, nous pouvons déterminer que les approches les plus adéquates sont celles de type hybride et relationnelle.

Par après, l'importance de définir correctement les exigences d'un problème est mise en exergue. Nous les établissons ensuite dans le cadre de ce travail. Ainsi, les caractéristiques recommandées pour un outil de transformation de modèles sont présentées. Un cas d'étude devant servir de benchmark pour évaluer des outils est défini également, tout comme les contraintes liées à l'architecture de MetaDONE. Ce qui nous permet de pré-sélectionner des outils de transformations de modèles mettant en place les approches choisies précédemment.

Ensuite, nous définissons l'importance de l'interopérabilité dans un environnement intégré. Les différents concepts utiles sont détaillés et plusieurs architectures d'intégration existantes sont exposées.

Les chapitres suivants dressent une méthode pour évaluer les outils pré-sélectionnés, ces derniers étant ATL, Epsilon (plus précisément ETL), GrGen.net et Prolog. Dans un premier temps, nous présentons ces outils et la justification de leur choix. Leurs caractéristiques fonctionnelles et non fonctionnelles sont ensuite détaillées et comparées. Dans un deuxième temps, nous les utilisons pour concevoir la transformation de modèles du cas d'étude servant de benchmark. Dès lors, une évaluation objective peut être réalisée à l'aide de mesures représentatives. Pour ce faire, nous nous basons sur un standard qui est la norme ISO-9126.

Enfin, à la lumière des faits précédents, nous dressons les possibilités d'intégrer un outil dans MetaDONE.

Nous constatons suite à cette évaluation que l'outil le plus intéressant se révèle être ETL. En effet, il expose de nombreuses caractéristiques importantes, c'est un outil mature, interopérable, concis, facile d'apprentissage, utilisant des standards, etc. Aussi, c'est un des candidats offrant la meilleure adéquation sémantique avec les méta-modèles de MetaDONE grâce à son extensibilité.

Divers pistes sont envisageables quant à l'intégration de cet outil de transformation de modèles dans MetaDONE.

D'une part, la technologie de représentation des (méta-)modèles peut être rendue compatible *via* le développement d'un driver spécifique pour le méta-modèle MetaDONE, de manière à étendre les technologies supportées par Epsilon avec ce dernier. Ou encore, un plugin peut être implémenté dans MetaDONE afin qu'il puisse exploiter des modèles standards employés par l'outil de transformation tel qu'Ecore.

D'autre part, une intégration au niveau contrôle pourrait être effectuée *via* l'appel des transformations d'ETL en Java au sein de MetaDONE, rendu possible par le jar autonome mis à disposition par Epsilon.

De plus, MetaDONE pourrait être étendu en utilisant les autres langages de transformations de modèle de la famille Epsilon car ils sont interopérables. Chaque tâche spécifique pouvant être réalisée sur un modèle pourrait ainsi être résolue à l'aide d'un langage adéquat, telle que la validation, transformations, migration, etc. Cependant, ce jugement pourrait être ré-évalué en fonction d'un contexte donné.

Cependant, GrGen.NET reste un outil à prendre en considération si la performance et la gestion de larges modèles est une nécessité car il expose les meilleures performances. C'est un bon outil possédant également des caractéristiques importantes mais il se révèle moins mature, un peu plus complexe qu'ETL et ne permet pas d'adéquation sémantique avec les (méta-)modèles de MetaDONE. Quant à son intégration, elle peut être réalisée à l'aide de son API fournie.

Pour terminer, l'IDM est un domaine en constante évolution et un large panel d'outils de transformation de modèles ne cessant de se développer existe, ce qui permet d'envisager des perspectives supplémentaires tel que la prise en considération d'autres outils. De même que pour les possibilités d'intégration d'outils, la mise en place d'architecture est une activité nécessitant de la créativité et dépendant d'un contexte, par conséquent d'autres pistes peuvent être proposées. De surcroît, une approche différente peut également être abordée pour évaluer différents outils que celle mise en œuvre dans ce travail.

Liste des tableaux

Table des figures

Annexe A

Implémentation de la transformation du cas d'étude

A.1 ATL

```
-- @path AbstractPrograms=/AbstractToCompliantProgram/AbstractProgram.ecore
-- @path AbstractPrograms=/AbstractToCompliantProgram/AbstractProgramTarget.ecore

--uses strings;
module AbstractToCompliant;
create OUT: AbstractProgramTarget from IN: AbstractPrograms;

--******************HELPERS*****************
--******************************************

--==================================================================
--==Helper to get a counter to generate unique IDs for new activities==
--==================================================================
helper def : idCounter : Integer = 1;

--==================================================
--==Helper to calculate a size from given parameters=
--==================================================
helper def : computeSize (height: Real, width : Real): Real = height*width;

--==============================================================================
--==Make the abstractProgram OclUndefined if the transfo cannot be processed (if one activity cannot be reduced)=
--==============================================================================
--alternative to 'error' print in console
helper context AbstractPrograms!AbstractProgram def: abstractProgramCompliant(): OclAny =
    if (self.activity->select(a | not(thisModule.isActivitiesCompliant()))->notEmpty())
        then
            OclUndefined
        else
            self
        endif;
--==============================================================================
--==Control if all tasks of the activity are compliant, not too big (activity can be reduced)=
--==============================================================================
--use to print 'error' in consol if the transformation cannot be process
helper def: isActivitiesCompliant(): Boolean =
--Parcours toutes les instances tasks de toutes les Activity de AbstractPrograms
AbstractPrograms!Task->allInstances()->select(t|(not(t.isTaskSizeCompliant())))->isEmpty();
```

```
--=========================================================
--==Control if the activity is compliant or must be reduced=
--=========================================================
helper context AbstractPrograms!Activity def: isActivityTooBig(): Boolean =
    if self.task->collect(t|((t.getSizeTask())))->sum() > self.getSizeDevice()
    then true
    else false
    endif;

--==============================================================
--==Control if a task is compliant (not too big) with the device=
--==============================================================
    --used in isActivitiesCompliant helper
helper context AbstractPrograms!Task def: isTaskSizeCompliant(): Boolean =
self.widget->select(w|w.oclIsTypeOf(AbstractPrograms!Panel))->exists(var |
    thisModule.computeSize(var.height, var.width) <= thisModule.computeSize(self.
        activityTask.abstractProgActivity.mustBeCompliantWith.first().sizeScreenX,
        self.activityTask.abstractProgActivity.mustBeCompliantWith.first().
        sizeScreenY));

--===================================
--==Return the size of a given task=
--===================================
helper context AbstractPrograms!Task def: getSizeTask(): Real =
thisModule.computeSize(self.widget->select(w|w.oclIsTypeOf(AbstractPrograms!Panel))->
    first().height, self.widget->select(w|w.oclIsTypeOf(AbstractPrograms!Panel))->
    first().width);

--=================================
--==Return the size of the device=
--=================================
helper context AbstractPrograms!Activity def: getSizeDevice(): Real =
thisModule.computeSize(self.abstractProgActivity.mustBeCompliantWith.first().
    sizeScreenX,self.abstractProgActivity.mustBeCompliantWith.first().sizeScreenY);

--=================================================================================
--==Return a list containing the tasks too big for the activity and for which a new
    activity must be created=
--=================================================================================
helper context AbstractPrograms!Activity def: getAllFittingTasks : Sequence(
    AbstractPrograms!Task) =
    self.task->iterate(t; resTask: Sequence(AbstractPrograms!Task) = Sequence{}|
    --init resTask
    if resTask.isEmpty()
    then resTask.including(t)
    else
        if (resTask->collect(res|(res.getSizeTask()))->sum() + t.getSizeTask() < self
            .getSizeDevice())
        then
            resTask.including(t)
        else
            resTask
        endif
    endif
);

--******************RULES*****************
--****************************************

-----------------------------------------------------------
--Root matched rule (abstractProgram to CompliantProgram)-
-----------------------------------------------------------
rule AbstractToCompliant {
    from
        abstractProgram : AbstractPrograms!AbstractProgram
        -- can be used to control if the transformation can be processed
--  using {
```

```
--        abstractProgramTestIfPossible : AbstractPrograms!AbstractProgram=abstractProgram.abstractProgramCompliant();
--  }
    to
        compliantProgram : AbstractProgramTarget!CompliantProgram (
            compliantWith <- device,
            --THROW VMException: Unable to access activity on OclUndefined in transfo is not possible
--          activity <- abstractProgramTestIfPossible.activity
            activity <- abstractProgram.activity

        ),
        device: AbstractProgramTarget!Device (
            name <- abstractProgram.mustBeCompliantWith.first().name,
            sizeScreenX <- abstractProgram.mustBeCompliantWith.first().sizeScreenX,
            sizeScreenY <- abstractProgram.mustBeCompliantWith.first().sizeScreenY
        )
        do{
            --verifie si une des activites ne possede pas une tache non reductible, si c'est le cas : print en console
            if (abstractProgram.activity->select(a | not(thisModule.isActivitiesCompliant()))->notEmpty()) {
                self.debug('**********ERROR************* One task is not compliant and cannot be reduce !');
            }
            else {
                self.debug('OK, all tasks are compliants');
            }
        }
}

-------------------------------------------------------------------------------
-- Match and copy activity if the size of its tasks is compliant with the device-
-------------------------------------------------------------------------------
rule Activity {
    from
        abstractActivity : AbstractPrograms!Activity
        -- verifie que la liste des taches non compliant est vide
        (not(abstractActivity.isActivityTooBig()))

    to
        compliantActivity : AbstractProgramTarget!Activity (
            name <- abstractActivity.name,
            task <- abstractActivity.task,
            abstractProgActivity<-abstractActivity.abstractProgActivity
        )
}

-------------------------------------------------------------------------------
-- Split an Activity in several activities if its tasks are too big for the device-
-------------------------------------------------------------------------------
--**Precondition : the activity must not be compliant/too big with the device
-- The boolean return by the isActivityTooBig must not be true

--**Post : return compliants activities : split the current activity in 2 and the "sequence" attribute of the current activity is filled
            --if a task contains a "precedence" attribute with an other task
rule ReduceActivity {
    from
        abstractActivity : AbstractPrograms!Activity
        (abstractActivity.isActivityTooBig())
    using {
        tasksCompliant : Sequence(AbstractPrograms!Task)=abstractActivity.getAllFittingTasks;
        --iterate because excluding(sequence.object) and not excluding(sequence)
        newTasksActivity : Sequence(AbstractProgramTarget!Task) = abstractActivity.task->
                iterate(t; res: Sequence(AbstractProgramTarget!Task) = Sequence{} |
```

```
158                         if tasksCompliant.includes(t)
159                         then res
160                         else res.including(t)
161                         endif
162                 );
163     }
164     to
165         outputActivitySplit : AbstractProgramTarget!Activity  (
166             name<-abstractActivity.name,
167             --copy compliant tasks
168             task <- tasksCompliant
169         )
170     do {
171
172         self.debug('Create␣Sequence␣between␣current␣activity␣␣' +
                outputActivitySplit.name +'␣and␣the␣new␣one␣?␣:␣' + newTasksActivity->
                select(t | not(t.precedence.isEmpty()))->notEmpty());
173         --create new activity with others taks and control if "precedence" in task
                exists
174         if (newTasksActivity->select(t | not(t.precedence.isEmpty()))->notEmpty()) {
175             -- fill the sequence attribute of the current activity with the ref of
                    the new activity created
176             outputActivitySplit.sequence <- newTasksActivity->iterate(a; acc :
                    Sequence(AbstractPrograms!Activity)=Sequence{}|
177                                                     acc->append(thisModule.
                                                            Composition(
                                                            abstractActivity,
                                                            newTasksActivity)));
178         } else {
179             -- create a new activity
180             -- can be extends to create several activities ...
181             newTasksActivity->iterate(a; acc : Sequence(AbstractPrograms!Activity)=
                    Sequence{}|
182                                                     acc->append(thisModule.
                                                            Composition(
                                                            abstractActivity,
                                                            newTasksActivity)));
183         }
184     }
185 }
186
187 ---------------------------------------
188 --create a new activity (called rule)-
189 ---------------------------------------
190 rule Composition (abstractActivity :AbstractProgram!Activity, activityToCreate :
        AbstractProgramsTarget!Activity ){
191     to
192         newActivity : AbstractProgramTarget!Activity (
193             --generate unique name
194             name <- abstractActivity.name+'Split'+thisModule.idCounter,
195             task <- activityToCreate->select(t | t.oclIsTypeOf(AbstractPrograms!Task)
                    ),
196             abstractProgActivity <-abstractActivity.abstractProgActivity
197         )
198
199     do {
200         --increment id counter
201         thisModule.idCounter <- thisModule.idCounter + 1;
202         self.debug('Create␣New␣Activity␣!␣-Split-␣' + '␣named␣:␣--' +newActivity.name
                +'--');
203         --return the new activity created
204         newActivity;
205     }
206 }
207
208 --copy task
209 rule Task {
210     from
```

```
211         abstractTask : AbstractPrograms!Task
212     to
213         compliantTask : AbstractProgramTarget!Task (
214             name <- abstractTask.name,
215             widget <- abstractTask.widget,
216             precedence <-abstractTask.precedence
217         )
218 }
219
220 -----------------------
221 --Sub elements mapping-
222 -----------------------
223
224 abstract rule Widget {
225     from
226         abstractWidget : AbstractPrograms!Widget (abstractWidget.oclIsTypeOf(
            AbstractPrograms!Composite)<>true)
227     to
228         compliantWidget : AbstractProgramsTarget!Widget (
229             name <- abstractWidget.name,
230             width <- abstractWidget.width,
231             height <- abstractWidget.height,
232             layout <- abstractWidget.layout,
233             widgetComposite <- abstractWidget.widgetComposite
234         )
235 }
236
237 rule Panel extends Widget{
238     from abstractWidget : AbstractPrograms!Panel
239     to compliantWidget : AbstractProgramTarget!Panel
240 }
241
242 rule Button extends Widget{
243     from abstractWidget : AbstractPrograms!Button
244     to compliantWidget : AbstractProgramTarget!Button
245 }
246
247 rule Label extends Widget{
248     from abstractWidget : AbstractPrograms!Label
249     to compliantWidget : AbstractProgramTarget!Label
250 }
251
252 rule TextField extends Widget{
253     from abstractWidget : AbstractPrograms!TextField
254     to compliantWidget : AbstractProgramTarget!TextField
255 }
256
257 rule Image extends Widget {
258     from abstractWidget : AbstractPrograms!Image
259     to compliantWidget : AbstractProgramTarget!Image
260 }
261
262 rule Composite{
263     from abstractComposite : AbstractPrograms!Composite (abstractComposite.
            oclIsTypeOf(AbstractPrograms!Composite))
264     to compliantComposite : AbstractProgramTarget!Composite
265 }
266
267 --Layout Part
268
269 rule Grid{
270     from abstractLayout : AbstractPrograms!Grid
271     to compliantLayout : AbstractProgramTarget!Grid (
272             widgetLayout <- abstractLayout.widgetLayout
273         )
274 }
275
276 rule Horizontal {
```

```
    from abstractLayout : AbstractPrograms!Horizontal
    to compliantLayout : AbstractProgramTarget!Horizontal (
            widgetLayout <- abstractLayout.widgetLayout
        )
}

rule Vertical {
    from abstractLayout : AbstractPrograms!Vertical
    to compliantLayout : AbstractProgramTarget!Vertical (
            widgetLayout <- abstractLayout.widgetLayout
        )
}
```

A.2 Epsilon

A.2.1 Code EOL

```
--=========================================================================
--==Control if all tasks of the activity are compliant, not too big (activity can be reduced)=
--=========================================================================
operation isActivitiesCompliant(): Boolean {
    return AbstractProgram!Task->allInstances()->select(t|t.isTaskSizeNotCompliant())->isEmpty();
}

--=========================================================
--==Control if the task can be compliant the device=
--=========================================================
operation AbstractProgram!Task isTaskSizeNotCompliant(): Boolean {
    return self.widget->select(w|w.isKindOf(AbstractProgram!Panel))->exists(p | computeSize(p.height, p.width) > computeSize(self.activityTask.abstractProgActivity.mustBeCompliantWith.first().sizeScreenX,self.activityTask.abstractProgActivity.mustBeCompliantWith.first().sizeScreenY));
}

--==========================================================
--==Control if the activity is compliant or must be reduced=
--==========================================================
operation AbstractProgram!Activity isActivityTooBig(): Boolean {
    return self.task->collect(t|((t.getSizeTask())))->sum() > self.getSizeDevice();
}

--=========================================================
--==operation to calculate a size from given parameters=
--=========================================================
operation computeSize (height: Real, width : Real) : Real {
    return height*width;
}

--=========================================================================
--==operation to calculate the get the size of the device from an activity=
--=========================================================================
operation AbstractProgram!Activity getSizeDevice() : Real {
    return computeSize(self.abstractProgActivity.mustBeCompliantWith.first().sizeScreenX,self.abstractProgActivity.mustBeCompliantWith.first().sizeScreenY);
}

--=====================================
--==Return the size of a given task=
--=====================================
operation AbstractProgram!Task getSizeTask() : Real {
    return computeSize(self.widget->select(w|w.isKindOf(AbstractProgram!Panel))->first().height, self.widget->select(w|w.isKindOf(AbstractProgram!Panel))->first().width);
}

--=============================================================================
--==Return a list containing the tasks too big for the activity and for which a new activity must be created=
--=============================================================================
operation AbstractProgram!Activity getAllNotFittingTasks() : Sequence(AbstractProgram!Task) {
    var resTask : Sequence=Sequence{};

    for (t : AbstractProgram!Task in self.task) {
            sumSize=sumSize+t.getSizeTask();
           if (sumSize > self.getSizeDevice()){
               resTask.add(t);
```

```
            }
    }
    sumSize=0.0;
    return resTask;
}
--==========================================================================
---==Return a list containing the tasks compliant for the activity and for which a new activity must be created=
--==========================================================================
operation AbstractProgram!Activity getAllFittingTasks() : Sequence(AbstractProgram!Task) {
    var resTask : Sequence=Sequence{};

    for (t : AbstractProgram!Task in self.task) {
        if (resTask.isEmpty()) {
            resTask.add(t);
            sumSize=t.getSizeTask();
        }
        else {
            if (resTask->collect(res|(res.getSizeTask()))->sum() + t.getSizeTask() < self.getSizeDevice()){
                resTask.add(t);
            }
        }
    }
    return resTask;
}
```

A.2.2 Flock

```
import "Abstract2CompliantUtil.eol";

--Initialize 'global' variables
pre{
    var sumSize: Real=0.0;
    var idCounter: Integer=1;
}

--==================================================================
---==Retype an AbstractProgram to a CompliantProgram (Copy All Elements)
--==================================================================
retype AbstractProgram to CompliantProgram

--==================================================================
---==Control if the transformation can be perform
---==And set the attribute of the compliantWith of the CompliantProgram =
--==================================================================
migrate AbstractProgram {
    if (not(isActivitiesCompliant())) {
        throw "Transformation cannot be perform. One or more activities are not reductible!";
    }
    migrated.compliantWith =original.mustBeCompliantWith.first;
}

--==========================================================================
---==Migrate the Activity Element
---==Delete the not fitting Task of an activity and migrate them in a new Activity
--==========================================================================
migrate Activity {
    --get the task too big for the activity
    var notFittingTask: Sequence(AbstractProgram!Task)=migrated.getAllNotFittingTasks();
    --Create a new activity (and set its attributes)for the not fitting task
    for (task in notFittingTask) {
        var newActivity : new CompliantProgram!Activity;
```

```
35          newActivity.name=original.name+'Split'+idCounter;
36          newActivity.abstractProgActivity=original.abstractProgActivity;
37          --Handle sequence attribute for new activity
38          if(task.precedence.size()>0){
39              migrated.sequence=newActivity;
40          }
41          --Add task to the new activity
42          newActivity.task.add(task);
43          migrated.abstractProgActivity.activity.add(newActivity);
44          idCounter=idCounter+1;
45      }
46      --delete the task too big for the activity
47      delete migrated.task.select(t|notFittingTask.includes(t));
48  }
```

A.2.3 ETL

```
1   import "Abstract2CompliantUtil.eol";
2
3   pre{
4       "Running␣ETL".println();
5       --Initialize 'global' variables
6       var sumSize: Real=0.0;
7       var idCounter: Integer=1;
8   }
9
10  ---------------------------------------------------------------------------
11  -- Match AbstractProgram to CompliantProgram and verify that the transformation can
        be perform
12  ---------------------------------------------------------------------------
13  rule AbstractProgram2Compliant
14      transform s : AbstractProgram!AbstractProgram
15      to t : CompliantProgram!CompliantProgram {
16
17      if (not(isActivitiesCompliant())) {
18          throw "Transformation␣cannot␣be␣perform.␣One␣or␣more␣activities␣are␣not␣
                reductible!";
19      }
20          t.compliantWith=s.mustBeCompliantWith.first;
21  }
22
23  ---------------------------------------------------------------------------
24  -- Split an Activity in several activities if its tasks are too big for the device-
25  ---------------------------------------------------------------------------
26  rule ReduceActivity
27      transform abstractActivity : AbstractProgram!Activity
28      to compliantProg:CompliantProgram!CompliantProgram, compliantActivity :
            CompliantProgram!Activity {
29
30          guard : abstractActivity.isActivityTooBig()
31              "ReduceActivity".println();
32              //copy fitting tasks part
33              //-----------------------------------
34              //retrieve the final CompliantProgram which must contains the activity
35              var compliantProgFinal = CompliantProgram!CompliantProgram->allInstances
                    ().first();
36              compliantActivity.name= abstractActivity.name;
37              //add the fitting tasks
38              compliantActivity.task.addAll(abstractActivity.getAllFittingTasks());
39
40              //handle not fitting tasks part
41              //-----------------------------------
42              //create a new activity while there are tasks not fitting
43              while(abstractActivity.getAllFittingTasks().size>0) {
44                  var newAct : new CompliantProgram!Activity;
45                  newAct.name=abstractActivity.name+'Split'+idCounter;
```

```
46                  idCounter=idCounter+1;
47                  //handle sequence attribute
48                  if(abstractActivity.task.select(t | not(t.precedence.isEmpty()))->
                        notEmpty()){
49                      compliantActivity.sequence.add(newAct);
50                  }
51                  //Compliant task has been removed with the getAllFittingTasks(), only
                         the others are still in the abstractActivity
52                  newAct.task.addAll(abstractActivity.getAllFittingTasks());
53                  compliantProgFinal.activity.add(compliantActivity);
54                  compliantProgFinal.activity.add(newAct);
55              }
56                  delete(compliantProg);
57 }
58
59 ------------------------------------------------------------------------------
60 -- Match and copy activity if the size of its tasks is compliant with the device-
61 ------------------------------------------------------------------------------
62 rule AbstractActivityToCompliant
63     transform abstractActivity : AbstractProgram!Activity
64     to compliantProg:CompliantProgram!CompliantProgram, compliantActivity :
           CompliantProgram!Activity {
65
66         guard : abstractActivity.isActivityTooBig()==false
67             var compliantProgFinal = CompliantProgram!CompliantProgram->allInstances
                   ().first();
68             "AbstractActivityToCompliant".println();
69             compliantActivity.name= abstractActivity.name;
70             compliantActivity.task.addAll(abstractActivity.task);
71             compliantProgFinal.activity.add(compliantActivity);
72             delete(compliantProg);
73
74 }
```

A.3 GrGen.net

```
// GrGen.NET solution to transform an abstractProgram to a CompliantProgram for a given Device
#using "abstract_program__ecore.gm"

//-------------------------------------------------------
//- Main/first rule
//-------------------------------------------------------
rule abstractToCompliant {
    abstractProgram:_abstractPrograms::_AbstractProgram;
    modify {
        //Retyping _AbstractProgram to CompliantProgram
        compliantProgram:_abstractPrograms::_CompliantProgram<abstractProgram>;
    }
}

//-------------------------------------------------------------------------
//- Iterate over each activity and control if the activity is compliant or must be split-
//-------------------------------------------------------------------------------
rule activityToCompliant {
    device:_abstractPrograms::_Device;
    iterated{
            activity:_abstractPrograms::_Activity;
            ts:searchNotFittingTasks(activity,device);

            //activity -:_abstractPrograms::_Activity_task -> task1:_abstractPrograms::_Task ;

        //if { activity._name == "activityWithoutSequence"; } // OK process
        modify{
            emit("␣\n=====Current␣activity␣name=====␣",activity._name);
            ts(); //call modify body of 'searchNotFittingTasks' pattern
            eval {}
        }
    }
}

//---------------------------------------
//-Add a sequence between two activities-
//---------------------------------------
rule addSequenceBtwActivities(activityToSplit:_abstractPrograms::_Activity, newActivity:_abstractPrograms::_Activity)
{
    modify{
        eval{
            emit("Create␣a␣sequence␣between␣:␣"+ newActivity._name+ "␣and␣"+ activityToSplit._name);//+newActivity._name
            add(_abstractPrograms::_Activity_sequence,activityToSplit,newActivity);
        }
    }
}

//-----------------------------------------------------------------
//- rule used to redirect the tasks not compliant to a new activity
//-----------------------------------------------------------------
rule redirectNotFittingTasks(activityToSplit:_abstractPrograms::_Activity, newActivity:_abstractPrograms::_Activity, ref tasksToAdd:set<Node>)
{
      activityToSplit -edgeActTask:_abstractPrograms::_Activity_task -> task:_abstractPrograms::_Task;
      if{task in tasksToAdd;}
      modify{
        //redirect the edge of the not fitting task to the new activity
        newActivity !-edgeActTask->! task;
      }
}
```

```
60
61 //-----------------------------------------------------------------
62 //- Create a new activity and control if a sequence must be created)-
63 //-----------------------------------------------------------------
64 rule createActivity(activToSplit:_abstractPrograms::_Activity, ref tasksToAdd:set<
       Node>,var createSequence:boolean) {
65     abstractProgram:_abstractPrograms::_AbstractProgram;
66     compliantProgram:_abstractPrograms::_CompliantProgram<abstractProgram>;
67         modify {
68             //create a new activity in the compliantProgram
69             compliantProgram -e:_abstractPrograms::_AbstractProgram_activity->
                   newActivity:_abstractPrograms::_Activity;
70             eval{
71                 //call a rule to put these tasks in the new activity
72                 exec ([redirectNotFittingTasks(activToSplit, newActivity, tasksToAdd)
                       ]);
73                 newActivity._name =activToSplit._name+1;
74                 emit("␣\n␣-␣**Create␣new␣activity␣!␣with␣name␣:␣"+ newActivity._name
                       +"␣␣-␣**␣createSequence␣??␣:␣"+ createSequence );
75                 //call the rule to create a new activity
76                 if(createSequence){
77                     exec(addSequenceBtwActivities(activToSplit,newActivity));
78                 }
79             }
80         }
81 }
82
83 //-----------------------------------------------------------------
84 //- Search the tasks which made the activity not compliant-
85 //- (calculate the size of each task)
86 //-----------------------------------------------------------------
87 pattern searchNotFittingTasks(activity:_abstractPrograms::_Activity, device:
       _abstractPrograms::_Device)
88 {
89     //iterated over each task of an activity to calculate its size
90     iterated{
91         activity -:_abstractPrograms::_Activity_task -> task:_abstractPrograms::_Task
               ;
92         task-->panel:_abstractPrograms::_Panel;
93             modify {
94                 //contains the precedences found between tasks
95                 def ref precedences:set<Edge> = reachableEdges(task,_abstractPrograms
                       ::_Task_precedence);
96                 eval{
97                     //Control if the task is compliant
98                     if(panel._width>device._sizeScreenX || panel._height>device.
                           _sizeScreenY){
99                         emit("␣\n\n␣**********ERROR*************␣One␣task␣is␣not␣
                               compliant␣and␣cannot␣be␣reduce␣!␣\n␣\n" );
100                    }
101                    //size of the current task
102                    yield sum =sum + computeSize(panel._height,panel._width);
103
104                    //if the size of the activity is not compliant, the notFitting
                           are put in a set
105                    if(sum>computeSize(device._sizeScreenX,device._sizeScreenY)){
106                        emit("␣\n␣-␣**␣Current␣Activity␣not␣compliant,␣too␣big␣**␣␣"
                               + activity._name );
107                        tasksTooBig.add(task);
108                        yield isActivityTooBig =true;
109                        //check if a sequence between the current task and the new
                               task must be created
110                        if(precedences.size()>0){
111                            yield isPrecedenceExist =true;
112                            emit("␣\n␣-␣**␣A␣precedence␣exist,␣a␣new␣sequence␣must␣be
                                   ␣create␣!:␣" + isPrecedenceExist);
113                        }
114                    }
```

```
                }
            }
    }

    modify{
        //Initialized/reset variable
        def var isActivityTooBig:boolean = false;
        def var sum:double = 0.0;
        //contain the not fitting tasks
        def ref tasksTooBig:set<Node> = set<Node>{};
        def var isPrecedenceExist:boolean = false;
        def ref precedences:set<Edge> = set<Edge>{};

        eval{
            //handle the activity not compliant, create a new activity to put the notFitting tasks in it
            if(isActivityTooBig){
                exec (createActivity(activity,tasksTooBig,isPrecedenceExist));
            }
        }
    }
}
//==================================================
//==Function used to calculate a size from given parameters=
//==================================================
function computeSize (var height:double, var width :double) : double
{
    return (height*width);
}
```

A.3.1 Fichier GRS

```
#//import the metamodel, input model and the transformation rules
import abstract_program.ecore abstractProgram.xmi Abstract_to_compliant.grg
#//execute the given rules
debug exec abstractToCompliant && activityToCompliant
#//show the graph, the transformation
show graph ycomp
#//generate the output xmi
export compliantProgramResult.xmi
quit
```

Bibliographie

Alliance, O. (2003). *Osgi service platform, release 3*. IOS Press, Inc.

Almendros-Jiménez, J. M. et Iribarne, L. (2008). A framework for model transformation in logic programming.

Almendros-Jiménez, J. M. et Iribarne, L. (2012). Model validation in ontology based transformations. *arXiv preprint arXiv :1210.6111*.

Almendros-Jiménez, J. M. et Iribarne, L. (2013). A model transformation language based on logic programming. *In SOFSEM 2013 : Theory and Practice of Computer Science*, pages 382–394. Springer.

AndroMDA (2014). Andromda 3.5-snapshot. http://www.andromda.org/. Date of access : 2015-01-30.

Anjorin, A., Lauder, M., Patzina, S. et Schürr, A. (2011). emoflon : Leveraging emf and professional case tools. *Informatik*, page 281.

Arendt, T., Biermann, E., Jurack, S., Krause, C. et Taentzer, G. (2010). Henshin : advanced concepts and tools for in-place emf model transformations. *In Model Driven Engineering Languages and Systems*, pages 121–135. Springer.

Atkinson, C. et Kühne, T. (2002). Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1):5–22.

Balasubramanian, D., Narayanan, A., van Buskirk, C. et Karsai, G. (2007). The graph rewriting and transformation language : Great. *Electronic Communications of the EASST*, 1.

Barroca, B., Lúcio, L., Amaral, V., Félix, R. et Sousa, V. (2011). Dsltrans : A turing incomplete transformation language. *In Software Language Engineering*, pages 296–305. Springer.

Bézivin, J. (2004). Sur les principes de base de l'ingénierie des modèles. *L'Objet*, 10(4):147–157.

Biehl, M. (2010). Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*.

Blomer, J., Geiss, R. et Jakumeit, E. (2011). The grgen .net user manual.

Bondé, L. (2006). *Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP*. Thèse de doctorat, Thèse de doctorat, Université des Sciences et Technologies de Lille.

Boocock, P. (2003). Jamda model compiler framework. http://jamda.sourceforge.net/docs/index.html. Date of access : 2015-01-30.

Bordbar, B., Howells, G., Evans, M. et Staikopoulos, A. (2007). Model transformation from owl-s to bpel via sitra. *In Model Driven Architecture-Foundations and Applications*, pages 43–58. Springer.

Bosems, S. (2011). A performance analysis of model transformations and tools.

Braun, P. et Marschall, F. (2003). Botl–the bidirectional object oriented transformation language.

Brown, A. W. (1993). Control integration through message-passing in a software development environment. *Software Engineering Journal*, 8(3):121–131.

Brownsword, L. L., Carney, D. J., Fisher, D., Lewis, G. et Meyers, C. (2004). Current perspectives on interoperability. Rapport technique, DTIC Document.

Bruneliere, H., Cabot, J., Jouault, F. *et al.* (2010). Combining model-driven engineering and cloud computing. *In Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10 : Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*.

Buchwald, S. et Jakumeit, E. (2011). Compiler optimization : A case for the transformation tool contest. *arXiv preprint arXiv :1111.4737*.

Cabot, J. (2015). Executing ATL transformations from java.

Calegari, D. et Szasz, N. (2013). Verification of model transformations : A survey of the state-of-the-art. *Electronic Notes in Theoretical Computer Science*, 292:5–25.

Clasen, C., Del Fabro, M. D., Tisi, M. *et al.* (2012). Transforming very large models in the cloud : a research roadmap. *In First International Workshop on Model-Driven Engineering on and for the Cloud*.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J. et Quesada, J. F. (2002). Maude : Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243.

Codagen (2002). Codagen technologies and model-driven architecture (mda). http://www.omg.org/mda/mda_files/CodagenMDA-Final.pdf. Date of access : 2015-01-30.

Combemale, B. (2008). Ingénierie dirigée par les modèles (idm)–état de l'art.

Consortium, W. W. W. (2007). XSL transformations XSLT version 2.0. http://www.w3.org/TR/xslt20/. Date of access : 2015-01-30.

Consortium, W. W. W. (2010). Xquery 1.0 : An xml query language (second edition). http://www.w3.org/TR/xquery/. Date of access : 2015-01-30.

Czarnecki, K. et Helsen, S. (2003). Classification of model transformation approaches. *In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. Citeseer.

Czarnecki, K. et Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

De Lara, J. et Vangheluwe, H. (2002). Atom3 : A tool for multi-formalism and meta-modelling. *In Fundamental approaches to software engineering*, pages 174–188. Springer.

Diaw, S., Lbath, R. et Coulette, B. (2010). État de l'art sur le développement logiciel basé sur les transformations de modèles. *Technique et Science Informatiques*, 29(4-5):505–536.

Ebert, J., Süttenbach, R. et Uhe, I. (1997). Meta-case in practice : a case for kogge. *In Advanced Information Systems Engineering*, pages 203–216. Springer.

Englebert, V. (2007). Metal2 : a formal specification. Rapport technique, University of Namur.

Englebert, V. (2015). The metadone research project. https://sites.google.com/site/precisemetadoneresearchproject/. Date of access : 2015-01-30.

Englebert, V. et Heymans, P. (2007). Towards more extensible metacase tools. *In Advanced Information Systems Engineering*, pages 454–468. Springer.

Englebert, V. et Magusiak, K. (2013). The grasyla 2.0 language edition 1.2 (draft). Rapport technique, University of Namur.

Favre, J.-M. (2004). Towards a basic theory to model model driven engineering. *In 3rd Workshop in Software Model Engineering, WiSME*, pages 262–271. Citeseer.

Friedman-Hill, E. (2003). *JESS in Action*. Manning Greenwich, CT.

Geiss, R. et Kroll, M. (2007). On improvements of the varro benchmark for graph transformation tools. *Universität Karlsruhe, IPD Goos, Tech. Rep*, 7(12):2007.

Geiss, R. et Kroll, M. (2008). Grgen. net : A fast, expressive, and general purpose graph rewrite tool. *In Applications of Graph Transformations with Industrial Relevance*, pages 568–569. Springer.

Gelhausen, T., Derre, B. et Geiss, R. (2008). Customizing grgen. net for model transformation. *In Proceedings of the third international workshop on Graph and model transformations*, pages 17–24. ACM.

Gerber, A., Lawley, M., Raymond, K., Steel, J. et Wood, A. (2002). Transformation : The missing link of MDA. *In Graph Transformation*, pages 90–105. Springer.

Ghamarian, A. H., de Mol, M., Rensink, A., Zambon, E. et Zimakova, M. (2012). Modelling and analysis using groove. *International journal on software tools for technology transfer*, 14(1):15–40.

Group, O. M. (2012). Common object request broker architecture (CORBA). http://www.omg.org/spec/CORBA/3.3/. Date of access : 2015-01-30.

Grundy, J., Mugridge, W. et Hosking, J. (2000). Constructing component-based software engineering environments : issues and experiences. *Information and Software Technology*, 42(2):103–114.

Hasso Plattner Institute (HPI), University of Potsdam, G. (2015). Mdelab projects. https://www.hpi.uni-potsdam.de/giese/public/mdelab/. Date of access : 2015-04-03.

Horn, D.-I. T. (2011). Model transformations for program understanding : A reengineering challenge.

Huber, P. (2008). *The model transformation language jungle : an evaluation and extension of existing approaches*. na.

IBM (2015). XDE—xde model importer in ibm rational software architect. http://www.ibm.com/developerworks/rational/library/08/0226_gangulli-ali/. Date of access : 2015-01-30.

ISO/IEC (2015). International standard iso/iec fdis 9126-1. http://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf. Date of access : 2015-04-08.

Jakumeit, E., Buchwald, S. et Kroll, M. (2010). Grgen. net. *International Journal on Software Tools for Technology Transfer*, 12(3-4):263–271.

Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Hegedüs, Á., Herrmannsdörfer, M., Horn, T., Kalnina, E., Krause, C., Lano, K. *et al.* (2014). A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, 85:41–99.

Jakumeit Edgar, Blomer Jakob, G. R. (2014). *The GrGen.NET User Manual*.

Jézéquel, J.-M., Barais, O. et Fleurey, F. (2011). Model driven language engineering with kermeta. *In Generative and Transformational Techniques in Software Engineering III*, pages 201–221. Springer.

Jouault, F., Allilaire, F., Bézivin, J. et Kurtev, I. (2008). Atl : A model transformation tool. *Science of computer programming*, 72(1):31–39.

Jouault, F. et Bézivin, J. (2006). Km3 : a dsl for metamodel specification. *In Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer.

Juracz, L., Lattmann, Z., Levendovszky, T., Hemingway, G., Gaggioli, W., Netterville, T., Pap, G., Smyth, K. et Howard, L. (2013). Vehicleforge : A cloud-based infrastructure for collaborative model-based design. *In MDHPCL@ MoDELS*, pages 25–36.

Kalnins, A., Barzdins, J. et Celms, E. (2005). Model transformation language mola. *In Model Driven Architecture*, pages 62–76. Springer.

Kelly, S., Lyytinen, K. et Rossi, M. (1996). Metaedit+ a fully configurable multi-user and multi-tool case and came environment. *In Advanced Information Systems Engineering*, pages 1–21. Springer.

Kifer, M. et Lausen, G. (1989). F-logic : a higher-order language for reasoning about objects, inheritance, and scheme. *In ACM SIGMOD Record*, volume 18, pages 134–146. ACM.

Klassen, L. et Wagner, R. (2012). Emorf-a tool for model transformations. *Electronic Communications of the EASST*, 54.

Kleppe, A. G., Warmer, J. B. et Bast, W. (2003). *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional.

Koenig, D., Glover, A., King, P., Laforge, G. et Skeet, J. (2007). *Groovy in action*, volume 91. Manning.

Kolahdouz-Rahimi, S., Lano, K., Pillay, S., Troya, J. et Van Gorp, P. (2014). Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, 85:5–40.

Kolovos, D., Rose, L., Paige, R. et Garcia-Dominguez, A. (2010). The epsilon book. *Structure*, 178:1–10.

Kolovos, D. S., Paige, R. F. et Polack, F. A. (2006a). Eclipse development tools for epsilon. *In Eclipse Summit Europe, Eclipse Modeling Symposium*, volume 20062, page 200.

Kolovos, D. S., Paige, R. F. et Polack, F. A. (2006b). The epsilon object language (eol). *In Model Driven Architecture–Foundations and Applications*, pages 128–142. Springer Berlin Heidelberg.

Kolovos, D. S., Paige, R. F. et Polack, F. A. (2008). The epsilon transformation language. *In Theory and practice of model transformations*, pages 46–60. Springer.

Königs, A. (2005). Model transformation with triple graph grammars. *In Model Transformations in Practice Satellite Workshop of MODELS*, page 166.

Kurtev, I. (2008). State of the art of qvt : A model transformation language standard. *In Applications of graph transformations with industrial relevance*, pages 377–393. Springer.

Lano, K. (2000). Catalogue of model transformations.

Lano, K. et Kolahdouz-Rahimi, S. (2010). Specification and verification of model transformations using uml-rsds. *In Integrated Formal Methods*, pages 199–214. Springer.

Lano, K., Kolahdouz-Rahimi, S. et Poernomo, I. (2012). Comparative evaluation of model transformation specification approaches. *International Journal of Software and Informatics*, 6(2):233–269.

Lawley, M. et Steel, J. (2006). Practical declarative model transformation with tefkat. *In Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer.

Lepper, M. *et al.* (2011). Solving the ttc 2011 compiler optimization task with metatools. *arXiv preprint arXiv :1111.4744*.

Levendovszky, T., Lengyel, L., Mezei, G. et Charaf, H. (2005). A systematic approach to metamodeling environments and model transformation systems in vmts. *Electronic Notes in Theoretical Computer Science*, 127(1):65–75.

Lewis, G. A. et Wrage, L. (2004). Approaches to constructive interoperability.

Mazanek, S. (2011). Helloworld ! an instructive case for the transformation tool contest. *arXiv preprint arXiv :1111.4739*.

Mazanek, S., Rensink, A. et Van Gorp, P. (2010). Transformation tool contest 2010. *Malaga, Spain*, 41.

McGuinness, D. L., Van Harmelen, F. *et al.* (2004). Owl web ontology language overview. *W3C recommendation*, 10.

Mens, T. (2010). Model transformation : A survey of the state-of-the-art. *In* Gerard, S., Babau, J.-P. et Champeau, J., éditeurs : *Model Driven Engineering for Distributed Real-Time Embedded Systems*. Wiley - ISTE.

Mens, T. et Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142.

Mens, T., Van Gorp, P., Varró, D. et Karsai, G. (2006). Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, 152:143–159.

Mernik, M., Heering, J. et Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344.

MetaCase (2014a). Metaedit+, system administrator's guide. Date of access : 2014-11-04.

MetaCase (2014b). Metaedit+, system administrator's guide. Date of access : 2014-11-04.

Naish, L. (1996). Higher-order logic programming in prolog. Rapport technique, University of Melbourne.

Normand, V. (1992). *Le modèle SIROCO : de la spécification conceptuelle des interfaces utilisateur à leur réalisation*. Thèse de doctorat, Université Joseph-Fourier-Grenoble I.

OMG (2011). Meta object facility (mof) 2.0 query/view/ transformation specification. OMG Document : formal/2011-01-01.

OMG (2014a). *Meta Object Facility (MOF) 2.4.2 Core Specification*.

OMG (2014b). Object constraint language. OMG Document : formal/2014-02-03.

OptimalJ (2002). Optimalj tutorials 3.1. http://www.uio.no/studier/emner/matnat/ifi/INF5120/v04/verktoy/OptimalJTutorials.pdf. Date of access : 2015-01-30.

Patrascoiu, O. (2004). Yatl : Yet another transformation language-reference manual version 1.0.

Pieter Van Gorp, H. S. et Muliawan, O. (2015). Motmot : Model driven, template based, model transformer. http://www.fots.ua.ac.be/motmot/index.php. Date of access : 2015-04-03.

Pivl, T. (2005). Java metadata interface (jmi).

Rose, L. M., Kolovos, D. S., Paige, R. F. et Polack, F. A. (2010). Model migration with epsilon flock. *In Theory and Practice of Model Transformations*, pages 184–198. Springer.

Saada, H., Dolques, X., Huchard, M., Nebut, C. et Sahraoui, H. (2012). *Generation of operational transformation rules from examples of model transformations*. Springer.

Sadilek, D. A. et Wachsmuth, G. (2008). Prological model processing.

Schätz, B. (2010). Uml model migration with pete. *Transformation Tool Contest 2010 1-2 July 2010, Malaga, Spain*, page 85.

Schubert, L. (2010). *An Evaluation of Model Transformation Languages for UML Quality Engineering*. Thèse de doctorat, Master's thesis, Georg-August-Universität Göttingen, 2010.[cited at p. 101].

Schürr, A., Winter, A. et Zündorf, A. (1999). The progres approach : language and environment, handbook of graph grammars and computing by graph transformation : vol. 2 : applications, languages, and tools.

Sendall, S. et Kozaczynski, W. (2003). Model transformation the heart and soul of model-driven software development. Rapport technique.

Somogyi, Z., Henderson, F. J. et Conway, T. C. (1995). Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512.

Sriplakich, P. (2007). *ModelBus : un environnement réparti et ouvert pour l'ingénierie de modèles*. Thèse de doctorat, Paris 6.

Steiner, T. (2001). Prolog goes middleware - java-based embedding of logic servers. *PC Artificial Intelligence*, pages 25 – 27.

Störrle, H. (2005). A lightweight platform for experimenting with model driven development. Rapport technique, Technical Report TR0503, University of Munich.

Störrle, H. (2007). A prolog-based approach to representing and querying software engineering models. *VLL*, 274:71–83.

Störrle, H. (2009). A logical model query interface.

Störrle, H. (2011). Vmql : A visual language for ad-hoc model querying. *Journal of Visual Languages & Computing*, 22(1):3–29.

Syriani, E., Gray, J. et Vangheluwe, H. (2013a). Modeling a model transformation language. *In Domain Engineering*, pages 211–237. Springer.

Syriani, E. et Vangheluwe, H. (2009). Matters of model transformation. *School of Computer Science, McGill University*.

Syriani, E. et Vangheluwe, H. (2013). A modular timed graph transformation language for simulation-based design. *Software & Systems Modeling*, 12(2):387–414.

Syriani, E., Vangheluwe, H. et LaShomb, B. (2015). T-core : a framework for custom-built model transformation engines. *Software & Systems Modeling*, pages 1–29.

Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S. et Ergin, H. (2013b). Atompm : A web-based modeling environment. *In Demos/Posters/StudentResearch@ MoDELS*, pages 21–25.

Taentzer, G. (2004). Agg : A graph transformation environment for modeling and validation of software. *In Applications of Graph Transformations with Industrial Relevance*, pages 446–453. Springer.

Tamura, G., Cleve, A. *et al.* (2010). A comparison of taxonomies for model transformation languages. *Paradigma*, 4(1):1–14.

Technology, D. S. (2010). Angie an introduction. https://delta-software.com/fr/component/jdownloads/finish/17/189.html?Itemid=1680. Date of access : 2015-01-30.

Tisi, M., Martínez, S., Jouault, F., Cabot, J. *et al.* (2011). Refining models with rule-based model transformations.

Varró, D. et Balogh, A. (2007). The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234.

Voudouris, C., Dorne, R., Lesaint, D. et Liret, A. (2001). iopt : A software toolkit for heuristic search methods.

Wagner, R. (2006). Developing model transformations with fujaba. *In Proc. of the 4th International Fujaba Days*, pages 06–275.

Walls, C., Richards, N. et Oberg, R. (2004). *XDoclet in action*. Manning.

Wasserman, A. I. (1990). Tool integration in software engineering environments. *In Software Engineering Environments*, pages 137–149. Springer.

Wicks, M. (2004). Tool integration in software engineering : The state of the art in 2004. *Integration The VLSI Journal*, pages 1–26.

Wikipédia (2014). Ingénierie dirigée par les modèles — wikipédia, l'encyclopédie libre. [En ligne ; Page disponible le 22-mai-2015].

Wikipédia (2015). Composant d'interface graphique — wikipédia, l'encyclopédie libre. [En ligne ; Page disponible le 23-mai-2015].

Wikipedia (2015). Component-based software engineering — wikipedia, the free encyclopedia. [Online ; accessed 24-May-2015].

Willink, E. D. (2003). Umlx : A graphical transformation language for mda. *In A. Rensink (Ed.) Proceedings of the Workshop on Model Driven Architecture : Foundations and Applications*, pages 13–24.

Winter, A., Kullbach, B. et Riediger, V. (2002). An overview of the gxl graph exchange language. *In Software Visualization*, pages 324–336. Springer.

Wirpi, O. (2012). Seamless integration of metaedit+ and eclipse to combine modeling and coding.

Yang, Y. (1994). Tool interfaces for software development : models, experiments and evaluation. *In TENCON'94. IEEE Region 10's Ninth Annual International Conference. Theme : Frontiers of Computer Technology. Proceedings of 1994*, pages 776–780. IEEE.