Motifs de transformation de métamodèles

Ines Ragoubi

Rapport de stage

Master recherche en informatique

Motifs de transformation de métamodèles

Ines Ragoubi

Encadrants :
M. Mickaël Kerbœuf kerboeuf@univ-brest.fr
M. Jean-Philippe Babau Jean-Philippe.Babau@univ-brest.fr

Équipe :
LISyC-IDM

2 juin 2011

Résumé

L'IDM est un paradigme de l'ingénierie de logiciels où les modèles sont considérés comme une entité principale du processus de développement. Ils sont utilisés pour spécifier, concevoir, tester, documenter, maintenir et générer le code pour les applications à développer. Souvent, on utilise plusieurs langages de modélisation et de modèles au cours du processus de développement, qui ne peuvent rester insensible les uns des autres et doivent inter opérer. Ainsi, de nombreuses activités communes dans IDM impliquent la manipulation de plusieurs modèles, comme la transformation de modèle à modèle.
L'objectif de ce travail est de proposer une collection de transformations de métamodèles que nous qualifierons de *motifs de transformation de métamodèles*. Ce sont des transformations définies entre un métamodèle source et un métamodèle cible et qui se manifestent constamment. Ces motifs de transformation doivent être développés indépendamment des modèles sur lesquels ils seront appliqués afin de permettre leur réutilisation. Par conséquent, les transformations seront générées automatiquement. Identifiés dans un métamodèle plus large, ces motifs seront appliqués afin de pouvoir réutiliser des outils, définis pour ce métamodèle, sur des variantes structurellement différentes de ce dernier.

Table des matières

Introduction **3**

1 Contexte de l'étude et problématique **4**
1 Introduction à la modélisation . . . 4
2 L'ingénierie dirigée par les modèles . . . 5
2.1 Principales initiatives de l'IDM . . . 6
2.2 Les concepts de l'IDM . . . 7
2.3 Les langages dédiés de modélisation . . . 9
2.4 Transformation de modèles . . . 9
3 Approches de réutilisation de transformations . . . 12
3.1 L'approche de typage de modèle . . . 12
3.2 L'approche refactoring . . . 13
3.3 Approche proposée . . . 13

2 Identification des motifs de transformation de métamodèles **15**
1 Les patrons de conception . . . 15
1.1 Origine du concept . . . 15
1.2 Définition . . . 15
2 Patrons de transformation de métamodèles . . . 16
2.1 Modèles de transformations usuels . . . 16
2.2 Modèles de transformation structuraux . . . 19
2.3 Modèles de transformation à contraintes ModelType . . . 21

3 Mise en œuvre et outillage des patrons **23**
1 Description de l'environnement de travail . . . 23
1.1 Choix du langage d'implémentation . . . 23
1.2 L'environnement Kermeta . . . 23
1.3 Transformation de modèles avec Kermeta . . . 24
2 Vérification de la relation de matching entre modeltypes . . . 24
3 Implémentation des modèles de transformations . . . 25
4 Présentation du cas d'étude . . . 27
4.1 Description des métamodèles . . . 27
4.2 Problématique . . . 28
4.3 Processus de réutilisation . . . 28
4.4 Application des patrons de transformation . . . 29
4.5 résultats et commentaires . . . 30
Conclusion . . . 31
Bibliographie . . . 33

Introduction

Durant les dernières décennies, les techniques, les outils et les méthodes de génie logiciel évoluent de plus en plus rapidement pour fournir des solutions permettant non seulement de surmonter la complexité du développement des logiciels mais aussi de réutiliser des outils dans différents contextes [16].

Cette évolution a été marquée par le concept de l'ingénierie dirigée par les modèles (IDM), qui est une approche en plein essor où les modèles,en tant qu'abstraction du monde réel, deviennent des outils dans le processus de développement des logiciels.
En effet, la production, et en particulier l'édition, d'un modèle est rendue possible par l'usage des langages de modélisation, soit généralistes comme UML, soit spécifiques au domaine de modélisation (DSML pour *Domain Specific Modeling Language*). Les technologies basées sur les DSML offrent aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise. Ces langages dédiés sont généralement de petite taille et facile à utiliser.

La transformation de modèles joue un rôle primordial dans l'ingénierie dirigée par les modèles parce qu'elle permet de générer un modèle à partir d'un autre. L'écriture des transformations de modèles est une tâche complexe dans la mesure où elle doit répondre a des exigences diverses et hétérogènes et s'exprime au travers de langages dédiés comme ATL [17], SmartQVT [17] ou Kermeta [18] . Comme dans le développement logiciel classique, il apparait nécessaire de trouver des mécanismes appropriés pour la réutilisation de transformations.

L'objet de ce travail consiste à proposer des transformations de métamodèles que nous qualifierons de *motifs de transformation de métamodèles*. Ce sont des transformations définis entre un métamodèle source et un métamodèle cible et qui se manifestent constamment. Ensuite nous allons définir un processus pour générer automatiquement ces transformations afin de réutiliser des outils définis pour des métamodèles qui présentent le motif identifié.

La suite de ce document est organisée en trois chapitres. Le premier chapitre, définit le contexte et la problématique du sujet de stage. Il présente d'une part le principe et les concepts fondamentaux de l'IDM, définir les langages spécifiques au domaine de modélisation ainsi que les concepts de transformation de modèle ; d'autre part ce chapitre présente des approches existantes de réutilisation.
Le deuxième chapitre commence par introduire la notion des patrons de conception. Puis il présente la liste des motifs de transformation que nous avons définis. Dans le troisième chapitre, nous proposons une méthode pour mettre en œuvre les transformations définis dans le chapitre précédent afin de pouvoir les réutiliser. Ensuite nous allons implanter une étude de cas pour expérimenter les différents concepts présentés précédemment.

Chapitre 1

Contexte de l'étude et problématique

1 Introduction à la modélisation

La modélisation, au sens le plus large, est une construction abstraite pour mieux décrire la réalité. Dans [20], Robin Milner propose d'introduire la modélisation et les modèles au cœur du développement logiciel en évoquant le principe de ce qu'il appelle " *Tower of informatic models*". Il définit ce concept en tant que collection de différents modèles associés et reliés les uns aux autres. Selon lui, un modèle est un ensemble d'entités qui décrit et explique un système du monde réel. Ces entités ne sont pas forcément de même nature et n'ont pas les mêmes caractéristiques. Il faut donc les combiner afin de pouvoir modéliser le système en entier. L'auteur illustre ces propos par *une tour de modèle* avionique décrit dans la figure 1.1.

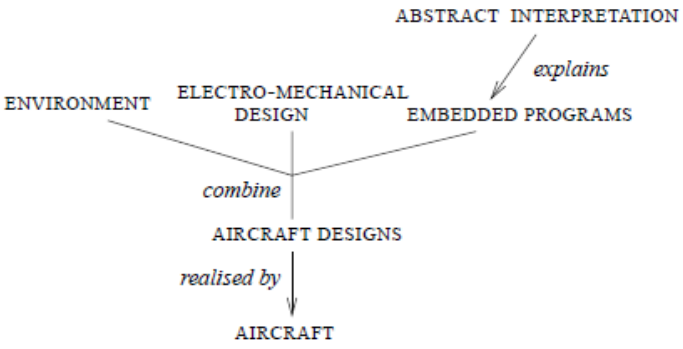

FIGURE 1.1 – Modèle simplifié de construction avionique [20]

Cette tour de modèle avionique est composée de trois niveaux d'abstraction et de leurs relations. Le premier niveau est le système à modéliser (avion), **représenté par** un modèle du système au deuxième niveau lui même, **combiné par** trois modèles hétérogènes, *un modèle de l'environnement* qui décrit l'environnement dans lequel le système opère (localité,température,vitesse du vent...) indépendamment des détails de traitements, *un modèle électromécanique* qui s'intéresse à

la conception électromécanique du système et *un modèle de programmes embarqués* qui décrit le code embarqué du système qui enfin lui même **expliqué par** par une interprétation abstraite qui est une simplification du programme, omettant certains détails et assurant des approximations, dans le but d'une analyse détaillée. Cet exemple montre que la modélisation des systèmes consiste non seulement à la conception des modèles mais aussi aux relations qui les relient.

2 L'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles (MDE [1]) est une approche prometteuse pour contourner la complexité de développement des logiciels ainsi que la difficulté des langages de troisièmes génération à réduire cette complexité [25]. L'intérêt pour cette approche a été fortement amplifié, en novembre 2000, lorsque l'OMG [2] a rendu publique son initiative MDA [3] qui vise à la définition d'un cadre normatif pour l'IDM [11, 22]. L'idée de cette initiative est de rendre les modèles *productifs* plutôt que *contemplatifs*. Je décris cette approche à travers la figure 1.2. La partie droite de la figure exprime le processus de développement en utilisant l'approche IDM, le principe est que le développeur commence à partir des exigences par construire un modèle du système à développer en se basant sur la notion des modèles productifs capables de générer du code contrairement à la partie gauche qui se contente des modèles contemplatifs, ce qui distingue l'IDM du développement basé sur la modélisation traditionnelle.

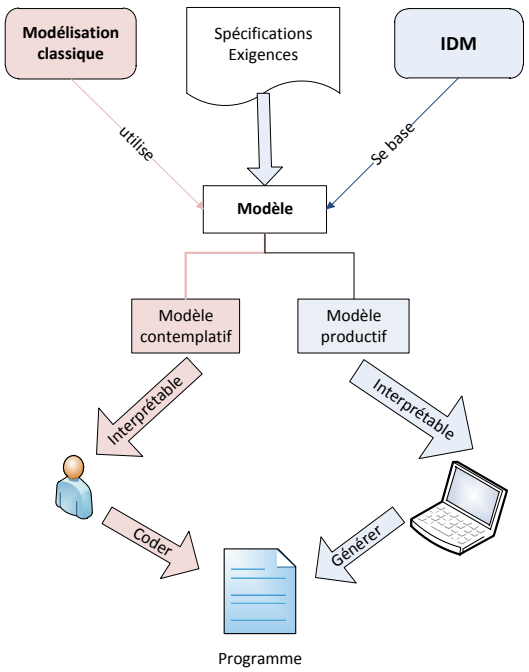

FIGURE 1.2 – Utilisation des modèles en génie logiciel

1. MDE : Model-Driven Engineering
2. OMG : Object Management Group
3. MDA : Model Driven Architecture

2.1 Principales initiatives de l'IDM

Cette section présente un aperçu des principales mises en œuvre de l'ingénierie dirigée par les modèles qui partagent un principe fondateur qui est l'utilisation des modèles, telles que l'Architecture dirigée par les modèles (MDA) [22], le MIC [4] [30], ou encore les usines à logiciel (*Software Factories*) [14].

Architecture dirigée par les modèles

L'Architecture Dirigée par les Modèles est une démarche proposée par l'OMG en 2001 [22]. Elle permet de séparer les spécifications fonctionnelles d'un système des spécifications d'implémentation sur une plate-forme donnée.
Le MDA [12] définit une architecture de spécifications à plusieurs niveaux :

- Computation Independent Model (CIM) dans lequel on s'intéresse aux caractéristiques requises du système et à l'environnement dans lequel il opère.
- Platform Independent Model(PIM) dans lequel les caractéristiques du système demeurent indépendants des plateformes.
- Platform Specific Model (PSM) dans lequel on intègre des éléments du PIM à des détails des PSM.
- Platform Description Model (PDM) modélise la plate-forme sur laquelle le système va être exécuté.

Dans le MDA, les modèles sont les artefacts de base, ils sont intégrés dans le processus du développement à travers une chaine de transformation du PIM en PSM et du PSM au code [22]. La figure 1.3 donne une vue générale d'un processus MDA, appelé aussi cycle de développement en Y en faisant apparaître les différents niveaux d'abstraction associés aux modèles.

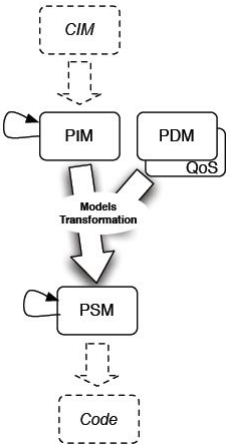

FIGURE 1.3 – Processus MDA [6]

Model-Integrated Computing

Le Model-Integrated Computing est une méthodologie dans laquelle les artefacts de base sont des modèles spécifiques au domaine d'application considéré. Le MIC repose sur une architecture à trois niveaux [30] :

- Le niveau *Meta* qui fournit des langages de méta-modélisation, des méta-modèles,des environnements de méta-modélisation et des méta-générateurs pour créer des outils spécifiques à un domaine qui seront utilisés dans le niveau MIPS (Model-Integrated Program Synthesis).

4. MIC : Model Integrated Computing

– Le niveau *MIPS* est constitué de langages de modélisation spécifiques à un domaine, et des outils pour la construction, l'analyse de modèles et la synthèse d'applications.
– Le niveau *application* fait référence à des applications synthétisé, à savoir le suivi, le contrôle, le diagnostic, la simulation et d'autres systèmes.

Software Factories

Les usines logicielles est une méthodologie développée par Microsoft. Cette approche intègre des innovations pour favoriser la transition d'une chaine de montage classique à une industrialisation du développement logiciel [14]. Elle s'appuie sur l'utilisation de *Software factory schema* qui est un document qui catégorise et résume les artefacts utilisés pour construire et maintenir un système (documents XML, les modèles, les fichiers de configuration...) et de *Software factory template* qui comporte des codes et des méta-données qui peuvent être chargés dans des outils extensibles comme les documents modèles qu'on peut charger dans Microsoft word à fin de réaliser un document spécifique.

2.2 Les concepts de l'IDM

L'IDM s'articule autour de trois concepts clés : les modèles, les métamodèles et les transformations.

Modèles et systèmes Un modèle peut être définit comme une abstraction suffisante pour décrire et expliquer une entité du monde réel. C'est un moyen de réduire la complexité d'un système afin de mieux comprendre son fonctionnement. Seidewitz [26] définit un modèle comme étant un ensemble d'entités décrivant le système à étudier il suppose qu'un modèle de la physique de Newton peut être décrit par un ensemble d'objets physiques et l'illustre par l'exemple du système solaire, un tel modèle est constitué par des entités décrivant les positions, les vitesses et les masses des planètes au moment où ils orbitent autour du soleil. Ces modèles peuvent être exprimés dans un langage formel pour pouvoir être formellement analysés.

Méta-modèle : langage de modélisation Traditionnellement dans l'IDM, les langages de modélisation sont eux-mêmes modélisés par ce qu'on appelle un *méta-modèle*. Ce méta-modèle représente les concepts du langage de modélisation utilisé et la sémantique qui leur est associée. Dans l'exemple du système solaire [26], le calcul vectoriel est le langage de modélisation primaire. On dit qu'un modèle est conforme à un méta-modèle si celui-ci respecte la définition et les contraintes du méta-modèle.

Selon [5], un langage de modélisation est définit généralement par une syntaxe abstraite,une sémantique et une syntaxe concrète :
– Syntaxe abstraite : elle décrit le vocabulaire et la grammaire des concepts fournis par le langage et leurs relations. Elle définit aussi les règles qui déterminent si un modèle écrit dans ce langage est valide ou non.
– La sémantique : définit la signification des concepts dans un langage et comment ils doivent être interprétés.
– Syntaxe concrète : Explique comment les syntaxes du langage peuvent être représentées. Cette syntaxe concrète peut être soit textuelle, soit graphique.
– Relations : Ces relations définissent des correspondances entre la syntaxe concrète et la syntaxe abstraite d'un même langage afin que celui-ci soit fonctionnel.

Méta-méta-modèle : langage de méta-modélisation Les langages utilisés pour exprimer les méta-modèles ont eux-même un modèle qu'on appelle méta-méta-modèle et qui se trouve être traditionnellement auto-descriptif. C'est en particulier le cas pour MDA. Un méta-modèle réflexif est exprimé dans le même langage de modélisation qu'il décrit. Les principaux langages de méta-modélisation existants sont MOF (pour *Meta-Objet Facilities*) [23], EMOF (pour *Essential MOF*), CMOF (pour *Complete MOF*) et ECore [10] défini par IBM et utilisé dans le framework

de modélisation d'Eclipse EMF (Eclipse Modeling Framework) [9].
La spécification de MOF adopte une architecture de méta-modélisation à quatre couches [23, 26].

M0 : Ce niveau représente le système réel à modéliser.
Exemple : Jeu d'échec dans la figure 1.4.

M1 : Ce niveau est composé du modèle représentant le système réel à modéliser.
Exemple : modèle de jeu d'échec via un DSL.

M2 : Dans ce niveau on trouve le méta-modèle décrivant le langage de modélisation.
Exemple : méta-modèle du DSL de jeu d'echec.

M3 : Un niveau qui comporte le méta-méta-modèle décrivant le langage de méta-modélisation.
Exemple : Ecore [10], MOF [23].

On retrouve ces couches dans la figure 1.4, qui illustre un système de jeu d'échec. L'échiquier est situé au niveau M0, chaque élément de ce système est modélisé au niveau M1. Dans ce niveau on retrouve le modèle qui **représente** l'échiquier, les pièces sont modélisés par une classe nommée pièce et qui a certains attributs décrivant la pièce, même chose pour le plateau et les cases. Ensuite on retrouve le méta-modèle au niveau M2 ce dernier **représente** le langage de modélisation et il est **conforme** à un méta-méta-modèle. Ce méta-méta-modèle est situé dans le niveau M3 il est **conforme** à lui même et représente le langage de méta-modélisation.

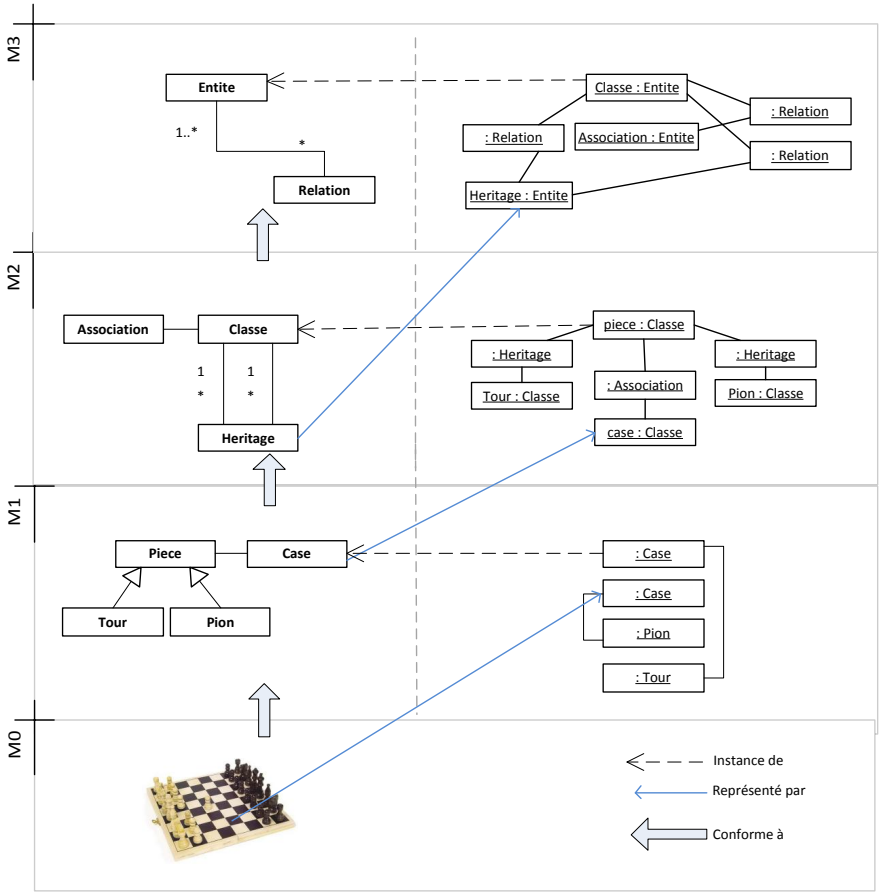

FIGURE 1.4 – Niveaux de modélisation : exemple du jeu d'échecs

L'IDM préconise l'utilisation des langages de modélisation dédiés à un domaine particulier qui offrent aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise. Le paragraphe suivant décrit la notion des DSL [5].

2.3 Les langages dédiés de modélisation

Un langage de modélisation spécifique au domaine (DSML) [31] est un langage restreint à un domaine particulier. Ces langages dédiés visent à identifier les concepts d'un domaine métier. Contrairement aux GPML (GPML pour *General Purpose Modeling Languages*) (exemlpe UML), les DSML définissent des langages plus restreints. Ces langages dédiés sont généralement de petite taille et facile à utiliser, pour des développeurs qui maîtrisent le domaine. Ils se conforment à des métamodèles tels que le MOF ou ECORE.

Parmi les DSML répandus, nous pouvons citer le langage de description d'architecture AADL (Architecture Analysis and Design Language), est un exemple de DSML du domaine de l'embarqué et du temps réel.

Adopter une approche DSL en génie logicielle implique des avantages et des inconvénients [31]. Un bon DSL est celui qui peut trouver un équilibre entre ces deux. D'une part les DSL permettent à des solutions d'être exprimées à un niveau d'abstraction spécifique au domaine du problème. En conséquence, les experts en matière du domaine eux-mêmes peuvent comprendre, modifier, et développer souvent des programmes de DSL. De plus puisque les programmes de DSL sont concis, auto-documentés en grande partie, ils peuvent donc être réutilisés à des fins différentes.

En contre partie, les avantages liés à l'utilisation des DSL sont nuancés par le fait qu'un langage dédié doit disposer d'un ensemble d'outils spécifiques qui sont par nature difficiles à adopter à d'autres langages à cause de leur hétérogénéité, il est donc difficile d'assurer et de maintenir l'interopérabilité. De plus les coûts de conception, la mise en œuvre, le maintien d'un DSL et l'éducation pour les utilisateurs sont élevés, ainsi que leur disponibilité est limitée.
En effet, le choix d'une des approches DSML ou GPML, dépend du contexte de développement et du domaine abordé.

2.4 Transformation de modèles

Un concept clé de l'IDM, consiste à manipuler les modèles à travers des transformations, ces transformations assurent le passage d'un ou plusieurs modèles sources à un ou plusieurs modèles cibles [7].Dans la suite, différents types de transformation de modèles sont expliquées dans [17,19].

Transformation horizontale de modèle Une transformation de modèle horizontal est une transformation, où le modèle source et le modèle cible appartiennent au même niveau d'abstraction. Un exemple pour ce type particulier de transformation est le *refactoring*, où le modèle cible, par rapport au modèle source, change dans sa structure interne sans changer le comportement. Par exemple le renommage d'un élément dans le modèle ne change rien au comportement du modèle voir figure 1.5.

Transformation verticale de modèle Contrairement à la transformation de modèle horizontale, différents niveaux d'abstraction sont utilisés dans le modèle source et cible. Un exemple de cette transformation le *raffinement*.

Transformation endogène de modèle C'est la transformation qui affecte les modèles exprimés dans le même langage. Les deux modèles source et cible sont conformes au même métamodèle. Exemples de transformations endogènes,*l'optimisation* comme la fusion ou l'élimination de code mort ou encore *le refactoring*.

5. DSL pour Domain Specific Language

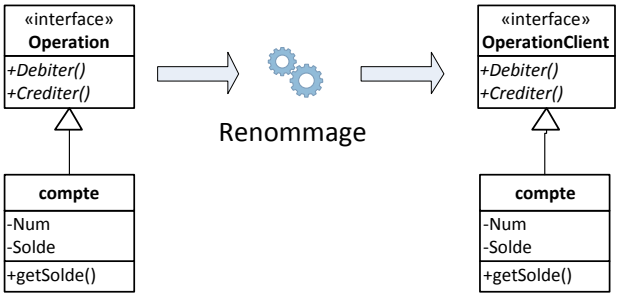

FIGURE 1.5 – Renommage de l'interface operation

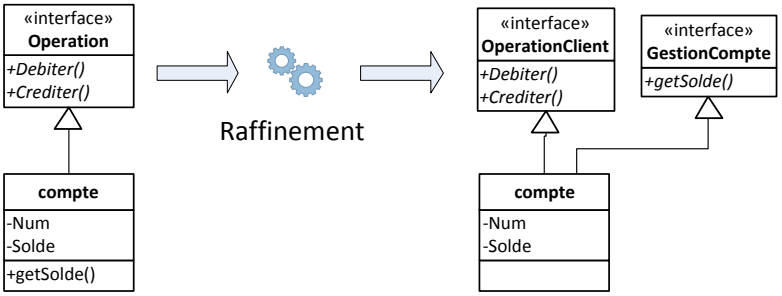

FIGURE 1.6 – Raffinement : Ajout d'interface

Transformation exogène de modèle Contrairement aux transformations endogènes, les transformations exogènes sont exprimées entre les modèles conformes à des différents méta-modèles. Les transformations exogènes peuvent également être appelé *translation*. Un exemple pour les transformations exogènes est la génération de code ou de documentation ou l'ingénierie inverse par exemple la décompilation. Par exemple, un diagramme de classes UML peut être traduit en code Java. La traduction d'un code Java en diagramme de classe UML est un exemple pour le reverse engineering. Ces familles de transformations peuvent être résumé par le tableau 2.4.

	Horizontale	Verticale
Endogène	Refactoring	Rafinement
Exogène	Migration de langage	Génération de code

TABLE 1.1 – Familles de transformations de modèles

Techniques de transformation de modèles

Il existe deux approches principales pour la transformation model-to-model : impérative et déclarative [1].
La première est fondée sur des règles ou des instructions qui indiquent explicitement quand et comment les éléments du modèle cible devrait être créés à partir des éléments de la source. En général, on parcourt la source dans un certain ordre et on génère la cible. Dans l'approche déclarative,

une description du mapping (relation de correspondance) entre la source et la cible est fournie. Cette description indique la relation entre les deux modèles plutôt que la façon de créer et de relier leurs éléments. Dans ce contexte, une approche déclarative de transformation de modèles est proposée dans [1, 15]. Dans cette approche la transformation est définie par la spécification de patrons déclaratives. Ce sont des contraintes sur des triples graphes composés par deux graphes et un troisième intermédiaire. Ces contraintes doivent être satisfaites par le résultat de la transformation.

Problématiques liées aux transformations de modèles

Une des problématiques concerne le langage utilisé pour décrire la transformation de modèles. Ceci est important, car les domaines d'application de la technologie de transformation de modèles peuvent être diverses. Des langages de transformation de modèles ont été classifié dans [7, 19], ainsi que des outils ont été évalué dans [17] mais ces études ont conclu qu'aucun outils n'est réellement meilleur sans l'absolu qu'un autre.
Néanmoins, un ensemble de critères peut aider à déterminer l'outil de transformation approprié pour un problème de transformation particulier [3] :

- **Niveau d'abstraction** : Les transformations de modèles peuvent soit introduire de nouveaux détails, réduire la quantité de détails ou laisser les modèles inchangés. Une transformation verticale peut changer le niveau d'abstraction, contrairement à une transformation horizontale qui change la représentation d'un modèle mais ne change pas le niveau d'abstraction.
- **Méta-modèle** : Différencier si les méta-modèles source et cible sont identiques ou différents.
- **Espace technique utilisé** : Les modèles sont représentés en utilisant différents espaces techniques (ensemble d'outils techniques)et langages (cf Ci-après).
- **Nombre de modèles utilisé** : Le nombre minimum est un, lorsque les modèles source et cible sont les mêmes, mais la plupart des transformations utilise deux modèles. Certaines transformations peuvent impliquer plusieurs modèles source et de combiner les informations qui s'y trouvent dans un modèle cible.
- **Type de la cible utilisé** : La cible peut être modèle si la transformation est *model-to-model* ou texte si la transformation est *model-to-text*.
- **Préservation des propriétés** : Les transformations peuvent être appliquées de telle façon que le modèle source et le modèle cible ont une propriété commune, qui n'est pas changé par la transformation (syntaxe, comportement, sémantique).

Langages pour la transformation des modèles

De nombreux langages sont à ce jour disponibles pour écrire des transformations de modèle. Parmi ces langages, on retrouve :

- **ATLAS Transformation Language** (ATL) est un langage hybride (déclaratif et impératif) qui permet de définir une transformation sous la forme d'un ensemble de règle. Il est défini par un modèle MOF pour sa syntaxe abstraite et possède une syntaxe concrète textuelle. Pour accéder aux éléments d'un modèle, ATL utilise des requêtes sous forme d'expressions OCL [3].
- **SmartQVT** C'est une implémentation du langage standardisé QVT-Operational qui permet la transformation de modèles à l'aide de requêtes. SmartQVT est un langage impératif de transformation. Il permet d'expliciter le détail des opérations de transformation. [3].
- **Kermeta** est un langage open-source pour modélisation et la méta-modélisation, développé par l'équipe Triskell à l'IRISA [18]. Il a été conçu comme extension du EMOF, en ajoutant notamment la possibilité d'exprimer la sémantique et le comportement des méta-modèles avec un langage d'action impératif et orienté-objet. Son approche impérative pour modéliser les résultats de transformation est différents par rapport à ATL et SmartQVT, qui suivent principalement le paradigme déclaratif. Au lieu de règles, Kermeta utilise des opérations,

qui sont fondamentalement très similaires à des opérations ou méthodes dans les langages de programmation orientés objet, tels que Java. Ce langage est largement décris dans la section 1.2,dans la mesure où il est notre langage d'implémentation.

3 Approches de réutilisation de transformations

Vu les différentes problématiques liées à l'écriture des transformations,ce processus apparait comme une tâche complexe. L'idée de la réutilisation des transformations est un moyen possible pour contourner cette complexité et de réduire le coût et le temps de développement de ces transformations [28]. En effet, une transformation peut souvent s'appliquer à un autre modèle, s'ils partagent des concepts semblables. Rendre une transformation de modèle réutilisable, est une problématique qui a été étudiée dans certains travaux qui concernent en particulier le typage des modèles [27] et la refactorisation de modèles [28].

3.1 L'approche de typage de modèle

La notion de typage de modèles a été implémentée dans le langage Kermeta [18] afin d'améliorer la réutilisation du code existant entre des variantes de métamodèles.
En effet,les modèles sont définis comme des graphes d'objets [29] pas forcement connexes. Ayant définit un modèle,Steel définit comme type d'un modèle, les types de tous les éléments du modèle c'est-à-dire, les nœuds et tous les arcs possibles. Les types des arcs sont modelisés en MOF 2.0 par des propriétés, qui sont déjà contenues par des classes, alors le type d'un modèle consiste tout simplement d'un ensemble de classes.

La notion de conformité(ou de substituabilité) des types de modèles *matching* [29] est employée pour décider si un type de modèles donné peut être utilisé quand un autre est attendu, elle a été adapté à une relation plus générale de matching que de Bruce utilise dans ses travaux [4]. Le matching en Kermeta ne s'appuie pas sur les noms des classes, mais prend en compte tous les détails de multiplicités et autres caractéristiques structurelles présentes en Kermeta. En effet,il consiste à assurer que pour chaque propriété ou opération sur un certain type, le type correspondant possède une propriété avec la même signature.

En effet, dans le langage kermeta, les collections sont traitées différemment des singletons. En conséquence, plusieurs valeurs des propriétés (ou opérations, ou les paramètres) ne peut se conformer à une seule valeur, et vice-versa [29].
Dans sa thèse Steel traite un exemple de transformation de modèle qui prend en entrée une machine d'état et produit une table de correspondance entre la situation actuelle, un événement qui arrivent, et l'état résultant.
Il considère certaines variantes de machines d'états en modifiant la multiplicités de la référence de l'état initial et teste si la transformation écrite pour un métamodèle déterminé pourra marcher ou non avec les variantes qui lui sont conformes. Il obtient donc des relations de non conformité entre certains modèles. Par exemple le modèle machine à état ayant l'état initial avec une référence 0..* n'est pas conforme au modèle ayant l'état initial avec une référence 0..1.

Un autre exemple, décrit dans [27], pour faire des transformations de modèles réutilisables à travers des méta-modèles structurellement différents qui repose aussi sur la notion de *type de modèle*. Il est alors possible de définir une transformation qui s'applique à un type de modèle qui sera *de facto* réutilisable dès lors qu'on l'appliquera à des modèles conformes au type attendu.

La figure 1.7 présente une transformation T écrite pour un modèle source A, peut être réutilisé pour un modèle B *sous-type* de A pour donner un autre modèle cible C. Un modèle A sous-type de B, est un modèle conforme à A.

Pour résumer la relation de *sous-typage* qui permet de déterminer qu'un modèle peut être concerné par une transformation réutilisable est cependant limitée par certaines contraintes struc-

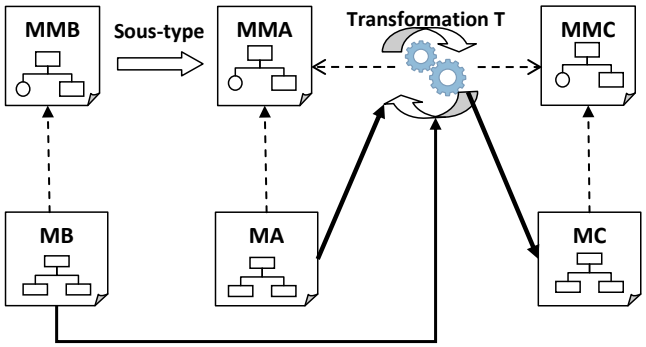

FIGURE 1.7 – Technique basée sur le typage

turelles. Par exemple, une variation dans le nom des attributs est un motif d'exclusion par la relation de sous-typage [27].

3.2 L'approche refactoring

Pour s'extraire des contraintes structurelles imposées par le typage des modèles [27], une autre approche consiste à réécrire en parties les modèles [28]. Cette approche décrite dans la figure 1.8 propose de transformer les instances conformes à un certain méta-modèle en instances conformes au méta-modèle du domaine de définition de la transformation que l'on souhaite réutiliser. Cette approche nécessite autant de refactoring que de méta-modèles sur lesquels on souhaite appliquer la transformation réutilisable.

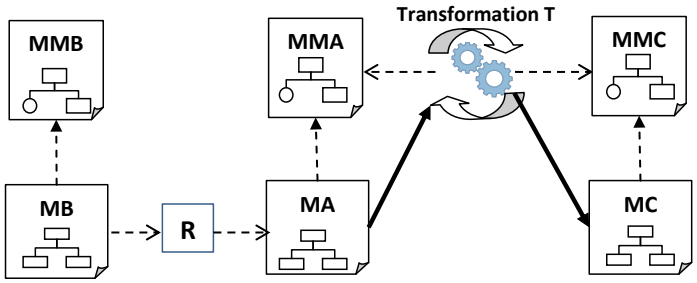

FIGURE 1.8 – Technique basée sur le refactoring

3.3 Approche proposée

Les langages de transformation de modèles proposent des *primitives* qui permettent de mettre en relation les éléments *source* et *cible* d'une transformation. La transformation elle-même reste à spécifier dans le cas d'un langage déclaratif ou à programmer dans le cas d'un langage impératif. Or, souvent on constate que des transformations de modèles écrite pour un modèle source peuvent s'appliquer à d'autres modèles qui partagent les mêmes concepts.

Afin d'éviter certaines contraintes structurelles qui limitent la réutilisation des modèles [27], on propose de définir des opérations de refactoring "type" basées sur des schémas-types, récurrents, de transformations de modèles 1.9,qu'on qualifie de *motifs* ou *pattern*.

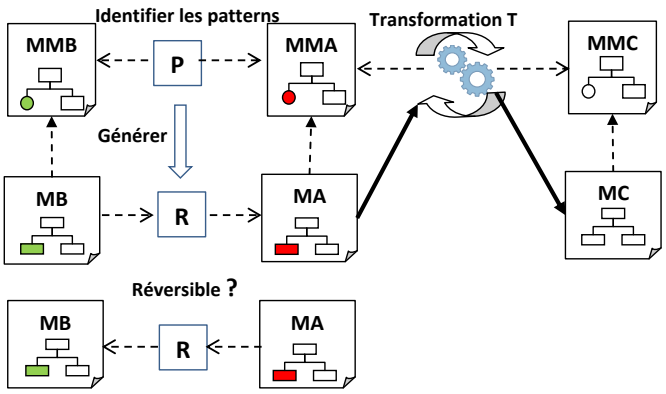

FIGURE 1.9 – Description de l'approche proposée

À titre d'illustration, la figure 1.10 représente la transformation d'une chaîne d'objets en collection ordonnée.

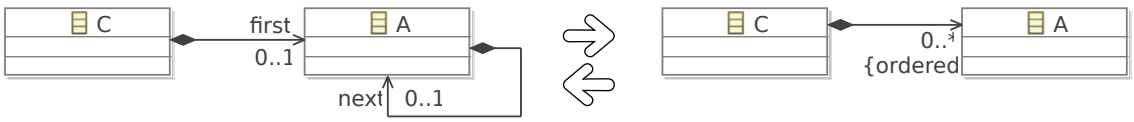

FIGURE 1.10 – chaîne / collection ordonnée

Ces transformations sont complexes dans la mesure où elle mettent en relation deux *sous-ensembles* des méta-modèles source et cible. Elles s'apparentent à de la refactorisation, mais au niveau méta-modèle. On souhaite ensuite pouvoir générer des transformations au niveau modèle à partir des motifs identifiés.

Chapitre 2

Identification des motifs de transformation de métamodèles

Ce chapitre présente l'origine et la définition des patrons de conception. Ce concept est en particulier celui qu'on a adopté pour proposer des transformations entre métamodèle source et cible.

1 Les patrons de conception

1.1 Origine du concept

L'origine des modèles de conception est dérivée des travaux de l'architecte Christopher Alexander [2] sur les modèles architecturaux. Selon Alexander, un modèle de conception possède 3 éléments essentiels : problème, solution et conséquence ; qui permettent à un architecte de rapidement adresser les problèmes d'une manière connue et acceptée. En 1995, ce concept s'est répandu largement dans l'industrie du logiciel depuis la publication de "Design Patterns" de Gamma et al [13]. Qui présente un catalogue de 23 modèles de conception qui s'adressent aux problèmes et solutions rencontrés lors de la modélisation d'applications.

1.2 Définition

Selon Christopher Alexander [2], chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, de telle façon que l'on puisse la réutiliser des millions de fois et ce jamais de la même manière. Bien que ceci a été dit en matière d'édifice et de villes, ce qu'il dit s'applique aussi aux modèles de conception orientée-objet. Cependant, un modèle de conception décrit une solution qui résout un problème récurrent de conception dans un contexte particulier.
L'utilisation des patrons de conception offre de nombreux avantages. Tout d'abord cela permet de bénéficier du savoir faire d'experts dans des contextes éprouvés pour résoudre les problèmes de conception sans être amené à réinventer des solutions. Ainsi on gagne en rapidité et facilité de conception. De plus, ils permettent d'utiliser un vocabulaire commun de conception pour décrire les modèles.
Dans les travaux de Gamma et al. [13], les patrons proposés ont été classés selon leur rôle et leur domaine d'application. On distingue entre 3 types, les modèles créateurs, structuraux et comportementaux. Les patrons créateurs permettent d'instancier et de configurer des classes et des objets. Les patrons structuraux permettent d'organiser les classes d'une application. Enfin, les patrons de comportement concernent les interactions entre classes et la collaboration entre eux. L'objectif de la section suivante est de présenter la liste de motifs de transformation de métamodèles identifiés.

2 Patrons de transformation de métamodèles

Nous proposons de définir une collection de modèles de conception pour la transformation de modèles. Un patron de transformation de métamodèle, est alors une solution réutilisable à un problème de transformation récurrent. Les patrons, que nous proposons, sont classés en trois types, des patrons de transformation usuels, structuraux et des patrons à contraintes structurelles de typage de modèles décrites dans la section 3.1. Ils seront décris dans la section suivante.

2.1 Modèles de transformations usuels

Cette catégorie décrit des patrons de transformation de métamodèles usuels, où la transformation est définie entre métamodèle source et métamodèle cible.

Nom du motif : ArrayToList

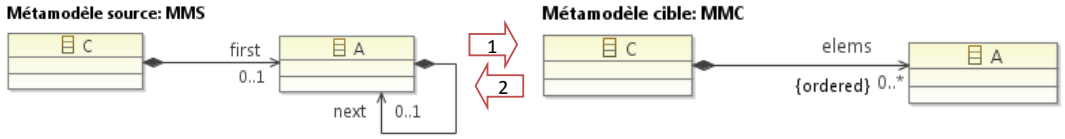

FIGURE 2.1 – Motif de transformation ArrayToList

Description : Le métamodèle source MMS représente un tableau d'objets tandis que le métamodèle cible MMC présente une liste d'objets ordonnés.

Principe : Les éléments C et A du métamodèle source doivent être transformés respectivement en C et A du métamodèle cible. L'ordre des éléments du tableau doit être conservé lors de la transformation vers liste. Le premier élément MMS.C.first doit être ajouté à la première position de la liste MMC.elems.

Algorithm 1 source2cible

```
MMC.C ← MMS.C
variable CurrentElem : MMS.A
CurrentElem ← MMS.first
repeat
  if CurrentElem ≠ vide then
    MMC.elems.add(CurrentElem)
    CurrentElem ← CurrentElem.next
  end if
until CurrentElem==vide
```

Algorithm 2 cible2source

```
MMS.C ← MMC.C
variable CurrentElem : MMS.A
CurrentElem ← MMC.elems.first
MMS.C.first ← CurrentElem
repeat
  if CurrentElem ≠ vide then
    MMS.CurrentElem.next.add(CurrentElem)
    CurrentElem ← CurrentElem.next
  end if
until CurrentElem==vide
```

Nom du motif : MutualExclusion and Polymorphism

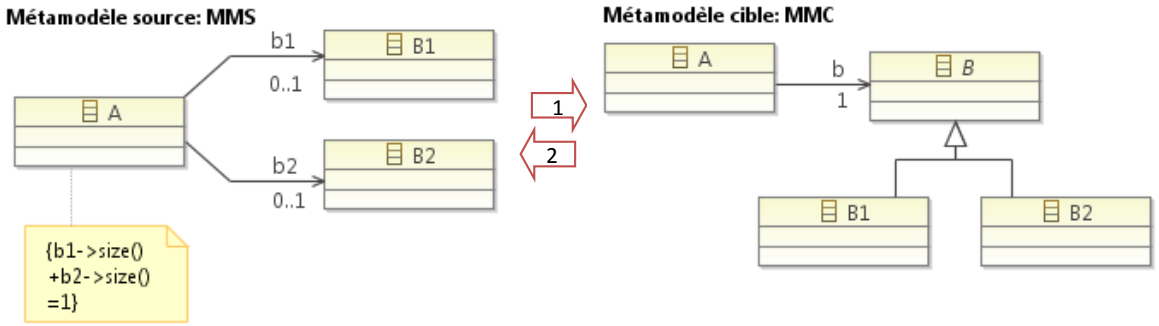

FIGURE 2.2 – Mutual Exclusion and Polymorphism

Description : Ce motif représente une transformation de deux références d'objets vers une référence d'objet polymorphe.

Principe : L'élément MMS.A doit être transformé en MMC.A en tenant en considération que :

- b1.size() + b2.size() =1
- b=b1 ou b2

Nom du motif : Distribute and Factorize

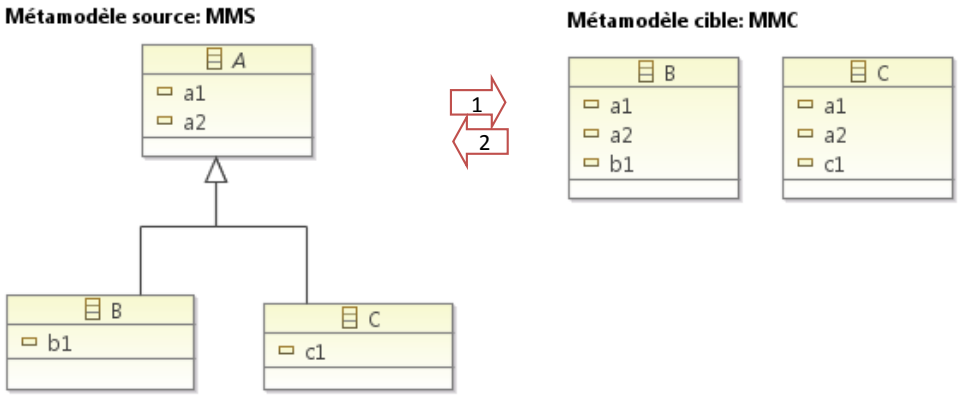

FIGURE 2.3 – Distribute and factorize

Description : La transformation de source vers cible représente la suppression d'une généralisation par distribution d'attributs communs.
La transformation de cible vers source consiste à créer une généralisation en factorisant les attributs communs.

Principe : L'élément A est abstract. Les éléments B et C de MMS doivent être transformés respectivement en B et C du dans MMC. Créer les attributs communs MMS.a1 et MMS.a2 et les ajouter comme attributs dans MMC.B et MMC.C

Nom du motif : Divide and Combine

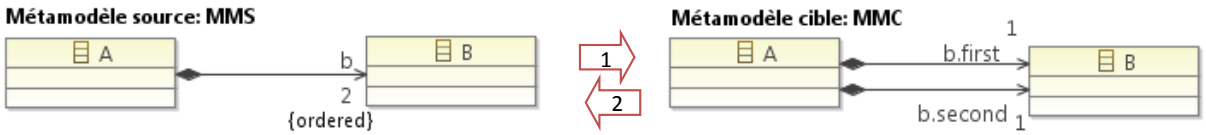

FIGURE 2.4 – Divide and Combine

Description : Le métamodèle source présente une liste de deux éléments. Le métamodèle cible présente deux listes qui contiennent chacune un seul élément.

Principe : Le principe de la transformation source à cible consiste à diviser la liste du métamodèle source en deux listes dans la cible. La liste dans MMS doit être ordonnée. La transformation inverse consiste à combiner les deux listes en une seule.

Algorithm 3 source2cible

MMC.A ← MMS.A
MMC.A.b.first ← MMS.A.b[0]
MMC.A.b.second ← MMS.A.b[1]

Algorithm 4 cible2source

MMS.A ← MMC.A
MMS.A.b[0] ← MMC.b.first
MMS.A.b[1] ← MMC.b.second

2.2 Modèles de transformation structuraux

Notre objectif étant l'identification de patrons de transformation de métamodèles, on s'est référé donc, dans notre étude, aux patrons de conception structuraux proposés par Gamma et al [13] ainsi que d'autres modèles proposés dans [8].

Nom du motif : Tree

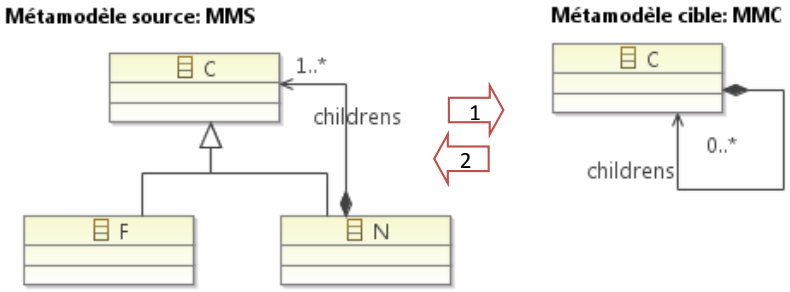

FIGURE 2.5 – Motif tree

Description : Métamodèle source : L'élément C représente un composant, c'est une classe abstraite. L'élément F représente les objets feuille. Une feuille est un composant qui n'a pas d'enfants. L'élément N définit un composant qui possède impérativement des enfants. Métamodèle cible : Le composant C peut être doté ou pas d'enfants. S'il a des enfants, il est alors considéré comme un nœud, sinon c'est une feuille.

Principe : Source vers cible : Chaque nœud N est transformé en un composant C qui possède des enfants. Une feuille F est transformée en un composant C n'ayant pas d'enfants. Cible vers source : Chaque composant C, s'il possède des enfants, il est transformé en un nœud N sinon il est transformé en une feuille F.

Algorithm 5 source2cible

```
var CurrentChild : MMS.C
CurrentChild ← MMS.N.childrens.first
repeat
  if MMS.N.childrens ≠ vide then
    if CurrentChild isInstanceOf N then
      MMC.childrens.add(MMC.C.new)
      recursif call(CurrentChild)
    else
      MMC.childrens.add(MMC.C.new)
    end if
    CurrentChild.next
  end if
until CurrentChild== vide
```

Nom du motif : Façade

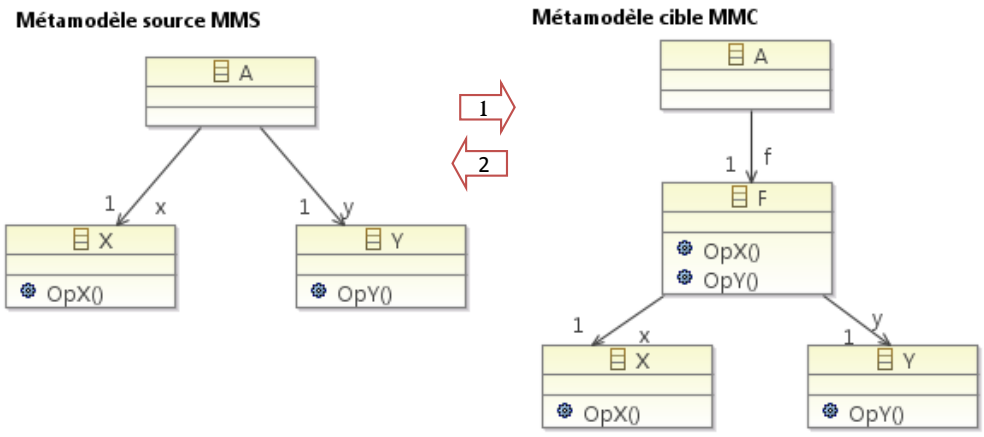

FIGURE 2.6 – Motif façade

Description : Le métamodèle source fournit un objet A ayant accès à deux autres objets X et Y. Dans le métamodèle cible, l'objet A a accès qu'à l'objet F qui lui permet d'accéder aux autres classes.

Principe : La transformation consiste à unifier l'accès en introduisant un nouvel objet F dans le métamodèle cible qui unifie l'accès à l'ensemble des classes.

Nom du motif : Espace de nommage

Description : Inspiré de [8] ce modèle présente une transformation d'un métamodèle source présentant le motif élément nommé vers un [24] métamodèle cible présentant une variante de ce motif.
L'élément N : Correspond à l'espace de nommage c'est à dire le contexte dans lequel est définit l'élément nommé.
L'élément E : représente un élément auquel on fait référence par son nom ou identifiant, l'élément nommé.
L'élément C : c'est l'objet qui manipule l'élément nommé à travers sa référence.

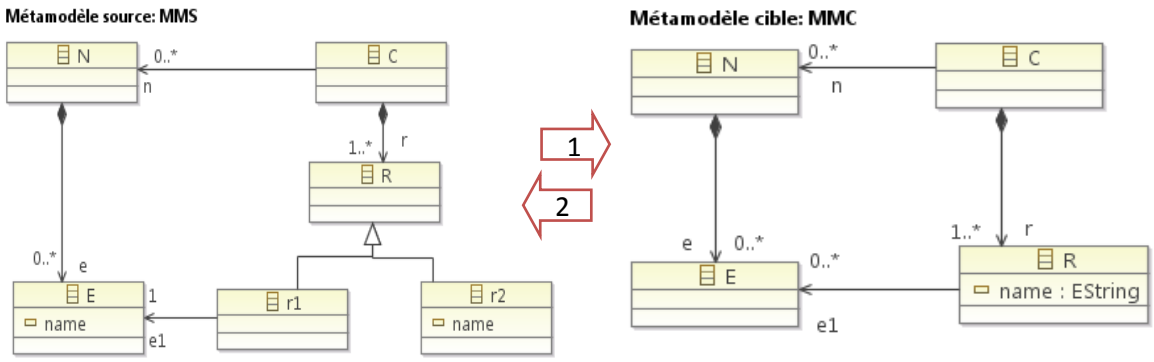

FIGURE 2.7 – Motif espace de nommage

L'élément R2 : Correspond à une référence non résolue, désigne l'élément nommé par l'intermédiaire de son nom.
L'élément R1 : Correspond à une référence résolue qui désigne l'élément nommé en pointant directement sur lui.

Principe : Cette transformation consiste à passer, d'une hiérarchie d'héritage qui consiste à étendre les propriétés de l'objet R en deux objets plus spécifiques, vers une hiérarchie de généralisation où l'objet R capture les propriétés de r1 et r2.

2.3 Modèles de transformation à contraintes ModelType

Dans cette section, nous allons présenter quatre motifs de transformation qui ont une relation avec les contraintes structurelles de modelType 3.1, dans les trois premiers motifs nous allons considérer des variantes de métamodèles où à chaque fois on modifie la multiplicité de la référence a.

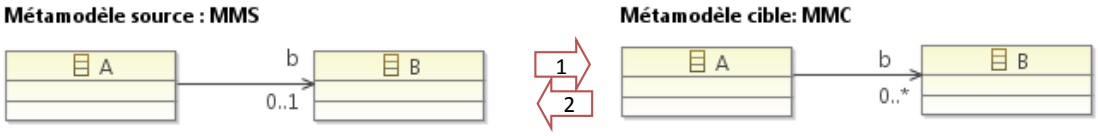

FIGURE 2.8 – Transformation d'une référence de 0..1 à 0..*

Description : Ce motif présente la transformation entre un métamodèle source dans lequel la multiplicité est 0..1 vers un métamodèle cible avec une multiplicité 0..*.

Principe : La transformation source vers cible consiste à créer L'élément A. Pour la transformation inverse, il existe la supposition où A est associée à plusieurs B. Dans ce cas, en partant de MMS A sera associée à un seul B.

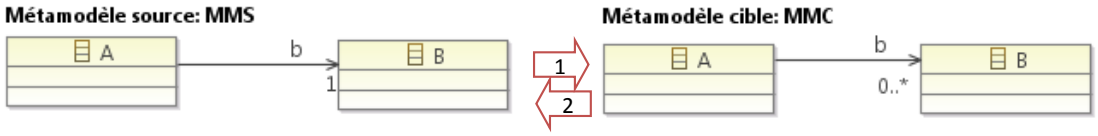

FIGURE 2.9 – Transformation d'une référence de 1 à 0..*

Description : Dans ce motif la transformation se fait entre un métamodèle source dans lequel la multiplicité est 1 vers un métamodèle cible avec une multiplicité 0..*. Dans ce cas aussi, l'hypothèse d'avoir dans MMC, l'élément A associé à plusieurs

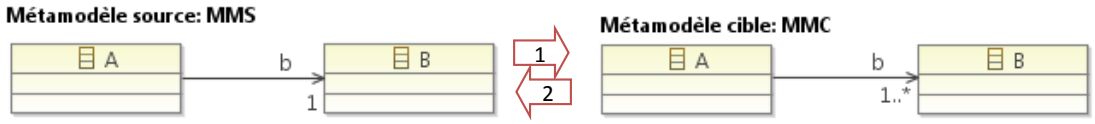

FIGURE 2.10 – Transformation d'une référence de 1 à 1..*

Description : Dans ce motif la transformation se fait entre un métamodèle source dans lequel la multiplicité est 1 vers un métamodèle cible avec une multiplicité 1..*.

Nom du motif : renommage

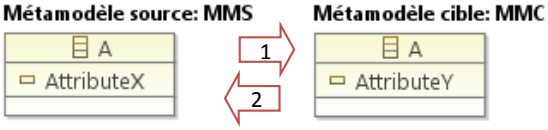

FIGURE 2.11 – Modèle de transformation renommage d'attribut

Description : L'élément A représente des objets ayant un attribut nommé X dans le métamodèle source et un attribut nommé Y dans le métamodèle cible.

Principe : Renommer l'attribut X en Y.

Chapitre 3

Mise en œuvre et outillage des patrons

Dans ce chapitre, nous allons mettre en œuvre les transformations proposées au cours du chapitre précédent. On vise à proposer des outils capables d'appliquer ces motifs de transformations. Une fois identifiés, ces derniers seront réutilisés sur des outils définis pour des métamodèles source et cible plus larges. La section suivante présente l'environnement d'implémentation des transformations.

1 Description de l'environnement de travail

1.1 Choix du langage d'implémentation

Comme il a été décrit dans la section 2.4 , il existe de nombreux langages pour écrire des transformations de modèle. Notre choix s'est posé sur kermeta, qui est une approche impérative permettant d'écrire des transformations d'une manière différente par rapport à ATL et SmartQVT. Qui suivent principalement le paradigme déclaratif. Au lieu de règles, Kermeta permet de définir des comportements, fondamentalement très similaires à des opérations ou méthodes dans les langages de programmation orientés objet, tels que Java. En plus kermeta permet de définir des aspects qui permettent de rajouter de nouveaux éléments aux métamodèles.
D'autre part, puisque kermeta supporte EMF, le chargement et l'enregistrement des données des métamodèles se fait au sein de la transformation elle même. Il s'effectue à travers un code simple, par contre dans ATL et SmartQVT, ceci se fait en dehors du code. Pour implémenter les transformations définies précédemment et pouvoir les générer automatiquement, kermeta s'avère comme le langage adaptée à nos besoins.

1.2 L'environnement Kermeta

Dans le domaine de l'IDM, il existe de nombreux langages disponibles pour l'écriture de métamodèles, tel que MOF et ECORE. Ces langages, permettent de définir des structures sans fournir de support pour la définition de sémantiques, de comportements ou d'algorithmes. D'où est née Kermeta qui propose un méta-langage plus expressif. Il est définit comme un cœur (figure 3.1) contenant les concepts communs aux différents langages du domaine de l'IDM. Ce langage permet de spécifier des métamodèles, leur sémantique opérationnelle mais aussi, il permet de décrire la structure et le comportement des modèles [11]. Il a été intégré à l'IDE Eclipse sous la forme d'un plugin.

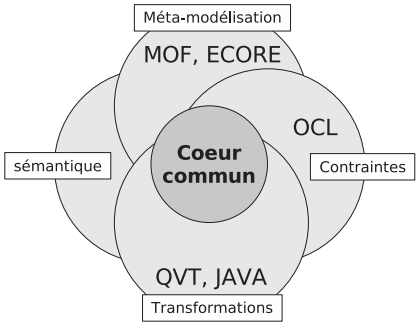

FIGURE 3.1 – Le cœur de Kermeta [11]

1.3 Transformation de modèles avec Kermeta

Le processus de transformation se base sur deux grandes étapes :

Définition des métamodèles : Une transformation est définie sur la base des métamodèles de sa source et de sa cible.Par conséquent, pour faire la transformation nous devons commencer par définir les métamodèles d'entrée et de sortie. La définition d'un métamodèle en Kermeta peut se faire de 3 manières :

- A travers un éditeur graphique de diagramme Ecore ;
- a travers l'éditeur EMF
- en écrivant un programme Kermeta qui correspond à la définition du métamodèle, ou en le générant à partir de son métamodèle ecore. Il est possible aussi d'en rajouter des comportements.

Définition de la transformation : Une fois, définis ces métamodèles peuvent être utilisés directement dans le programme Kermeta. La première phase consiste à créer pour chaque élément du modèle source, l'élément correspondant dans le modèle cible. Lors de la seconde phase, il est nécessaire de conserver un lien entre ces éléments.
Ceci est utile et permet de faciliter la tâche de la transformation inverse. Pour garder ces informations de traçabilité, Kermeta offre la possibilité d'utiliser la table de hachage. Le processus de trace utilisé est définis sous forme d'une classe Trace simple [21]. Cette classe permet de stocker les éléments source et cible entre deux ensembles d'objets. Elle a été améliorée par la suite par l'équipe Triskell en lui rajoutant la possibilité de trace bidirectionnel.

2 Vérification de la relation de matching entre modeltypes

Dans cette section, nous allons vérifier la relation de substituabilité décrite dans la section3.1 pour les modèles de transformation identifiés. Si cette relation est vérifiée on pourra réutiliser des outils définis pour un métamodèle sur des variantes de ce dernier. En effet, le principe de typage des modèles définit dans la section 3.1, permet de faire des transformations génériques sur des instances de différents métamodèles à condition que la relation de matching ou la substituabilité est assurée entre les modeltypes correspondants. Le processus de vérification de la relation de conformité entre les deux variantes est le suivant :

La première étape consiste à définir deux métamodèles, MMDomaine et MMLarge illustré dans la figure 3.2 , respectivement un modeltype domainMT et largeMT (figure 3.3). Les deux métamodèles ont des éléments différents, MMLarge a plus d'éléments comparé à MMDomaine.
Définir un modeltype, pour un métamodèle bien déterminé, consiste à lister ses classes sous cette

forme : modeltype nom_modeltype {*nom_package* :: *nom_classe*}.

Une fois les modeltypes définis, on écrit du code propre a MMDomaine. Pour cela, on définit une classe générique typé par le modeltype correspondant, c'est à dire domainMT. Le code est présenté dans la figure 3.3. On veut réutiliser ce code en l'appliquant à largeMT. Le model checker de kermeta indiquera si les modeltypes ne sont pas conformes, si c'est le cas, on ne pourra pas réutiliser le code pour largeMT.

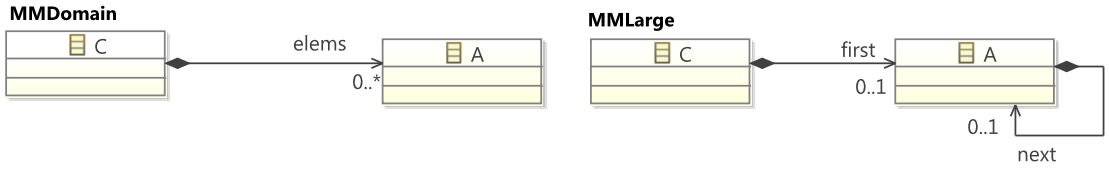

FIGURE 3.2 – MMDomaine et MMLarge

En effet, la relation de matching n'a pas été vérifiée pour la plupart des motifs qu'on a identifiés, il est donc pas possible d'appliquer l'approche modeltype pour ces modèles.

```
package PatternArrayToList::Transformation;

require "../MetamodelTrgt/MMDomain.ecore"
require kermeta

// on définit le modeltype correspondant au MMDomaine
modeltype domainMT{mmdomain::C,mmdomain::A}

// On définit une classe générique typée avec domainMT
class Code<MT:domainMT>
{
    operation test(trgMT:MT) : Void is do

    end
}
```

```
package PatternArrayToList::Transformation;
require kermeta
require "domainCode.kmt"
require "../MetamodelSrc/MMLarge.ecore"

// on définit le modeltype correspondant à largeMT
modeltype largeMT {mmlarge::C,mmlarge::A}

class Test
{
    operation test() : Void is do
        // on utilise le code pour le modeltype largeMT
        var fonctiontgt:Code<largeMT> init Code<largeMT>.new
    end
}
```

Type PatternTabToList::Transformation::largeMT is not a conformant type binding for the variable MT : PatternTa
Model type 'largeMT' does not match model type 'domainMT': no match for required class 'C'.

FIGURE 3.3 – Vérification de conformité

3 Implémentation des modèles de transformations

La définition des transformations que nous avons définis dans le chapitre précédent doit être indépendante des modèles sur lesquels ces transformations seront appliquées afin de permettre leur réutilisation. Par conséquence, nous allons générer ces transformations.
L'implémentation des modèles de transformations définis dans le chapitre précédent,a été faite pour certains patrons, dont on peut citer, le patron ArrayToList et Divide and Combine de la première catégorie, le patron Tree de la deuxième catégorie et les patrons à contraintes modelType.

Le processus est illustrée dans la figure 3.4.
Ce processus présente l'implantation d'une fonction **Generate** permettant de générer la transformation décrite dans le pattern, le principe est le suivant :

- Charger les ressources des fichier source.ecore et cible.ecore
- Récupérer pour chacun son package et les classes définies dedans.

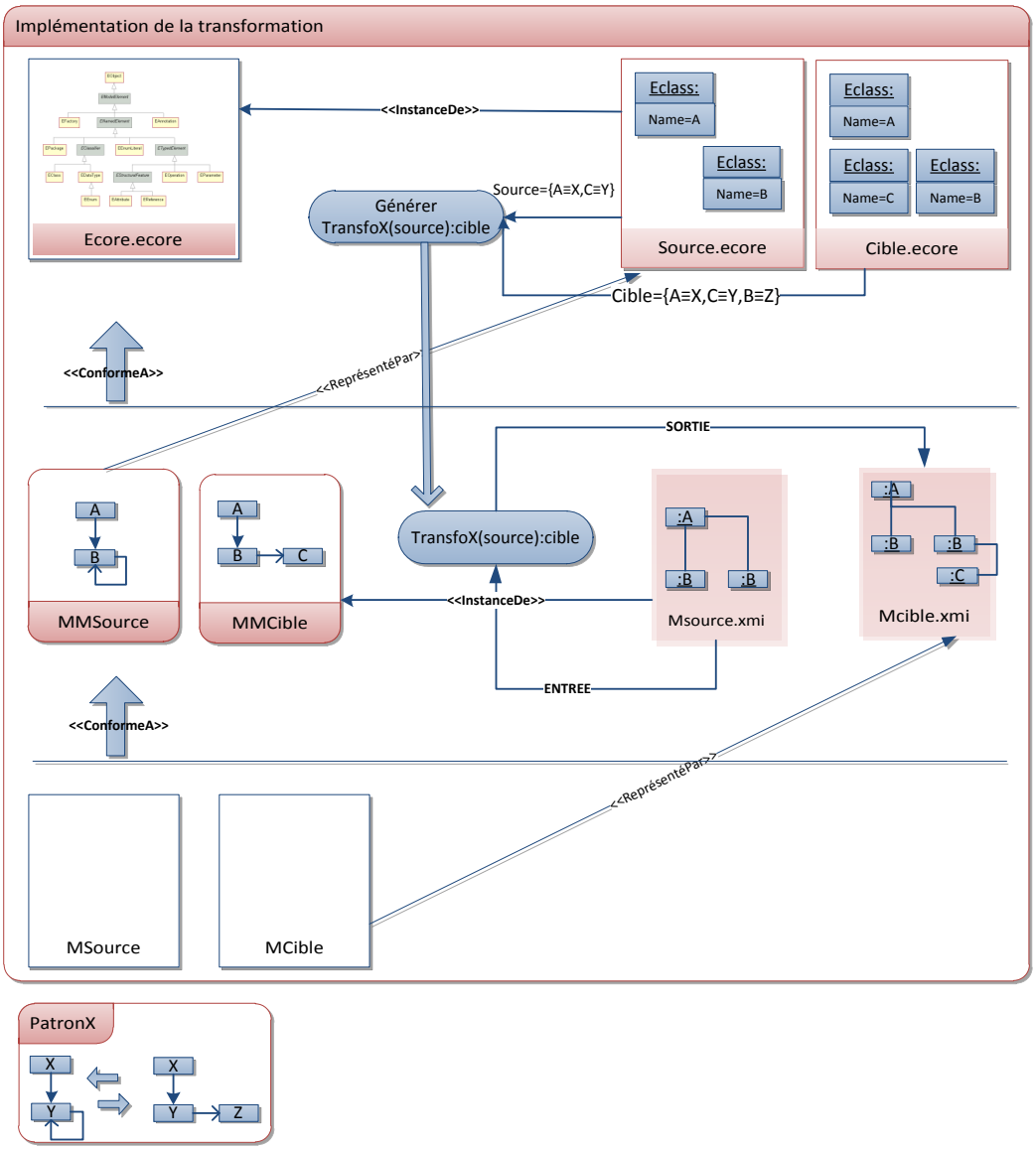

FIGURE 3.4 – Implémentation des modèles de transformations

- Faire correspondre chaque classe à celle qui lui est équivalente dans le pattern d'origine.
- Ecrire la transformation définie pour ce pattern ensuite l'ajouter à un fichier que nous appelons transfo.kmt
- Enregistrer le fichier et l'exécuter pour transformer les instances du modèle source.xmi en cible.xmi

Le principe est le même pour générer la transformation inverse.

4 Présentation du cas d'étude

Pour expérimenter les différents concepts présentés précédemment, nous allons considérer l'étude de cas suivante :

Nous disposons de deux outils, **expression2machine** et **Run**. Expression2machine est une transformation qui prend en paramètre une expression arithmétique et qui renvoie en résultat une machine à pile dont l'exécution produit le résultat de l'évaluation de l'expression. L'outil Run est une fonction qui exécute la machine à pile. L'outil **expression2machine** est défini pour accepter des modèles conformes à un métamodèle bien défini **MMArithm** et renvoyer des modèles conformes à un certain métamodèle **MMmAPile** où on pourra exécuter la fonction Run. Ceci est illustré dans la figure 3.5.

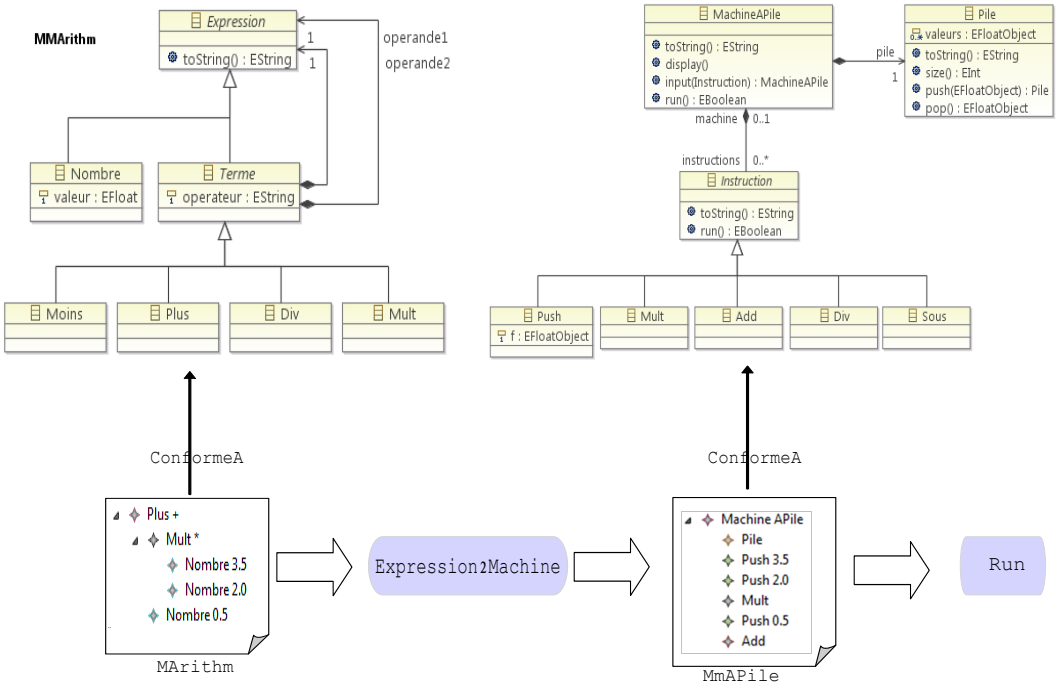

FIGURE 3.5 – Expression2Machine

4.1 Description des métamodèles

Expression arithmétique

Une expression arithmétique est soit un nombre, soit un terme qui représente un opérateur appliqué à deux autres expressions arithmétiques appelées opérandes. Un terme, peut être une addition, une soustraction, une multiplication ou une division.

Machine à pile

Une machine à pile est une machine à calculer munie d'une pile et d'une liste d'instructions. La pile peut contenir des valeurs et on peut afficher son contenu. Les instructions peuvent être :

- Push n : insère une valeur n au sommet de la pile.
- Add : dépile les deux valeurs situés au sommet de la pile, les additionne, et place le résultat en sommet de la pile.

- Sous : dépile les deux valeurs situés au sommet de la pile, les soustrait, et place le résultat en sommet de la pile.
- Mult : dépile les deux valeurs situés au sommet de la pile, les multiplie, et place le résultat en sommet de la pile.
- Div : dépile les deux valeurs situés au sommet de la pile, les divise, et place le résultat en sommet de la pile.

Une machine à pile peut être exécutée. Elle exécute (à travers la fonction Run) dans l'ordre chacune des instructions. Chaque instruction exécutée est supprimée de la liste. La machine s'arrête quand elle n'a plus d'instructions à exécuter.

4.2 Problématique

Nous disposons d'une variante (MMVariante) du métamodèle MMArithm dont les instances sont considérés comme des entrées potentielles pour la transformation expression2machine.
La différence se situe au niveau de la structure des opérandes, dans le métamodèle initial, un terme est composé d'une liste contenant deux opérandes de type expression, par contre dans la variante, un terme est composé de deux listes contenants chacune une opérande de type expression.
On souhaite exécuter la fonction run pour les instances de cette variante du métamodèle afin produire le résultat de l'expression arithmétique.

Nous disposons aussi, d'une variante du métamodèle MMmAPile, pour laquelle on souhaite réutiliser la fonction run. La question qui se pose est, comment réutiliser cet outil pour les variantes dont on dispose ?

4.3 Processus de réutilisation

Afin de pouvoir réutiliser les outils expression2machine et Run pour évaluer la valeur d'une expression arithmétique, nous proposons d'appliquer des patrons de transformation aux variantes de métamodèles dont nous disposons.
La première étape consiste à adapter MMVariante à MMArithm, c'est à dire appliquer un pattern X qui nous permet de retrouver la même structure que celle de MMArithm, de tel sorte qu'on peut réutiliser l'outil **expression2machine**.
La deuxième étape consiste à appliquer à la variante de MMmAPile un pattern Y.
Ce processus est illustré dans la figure 3.6

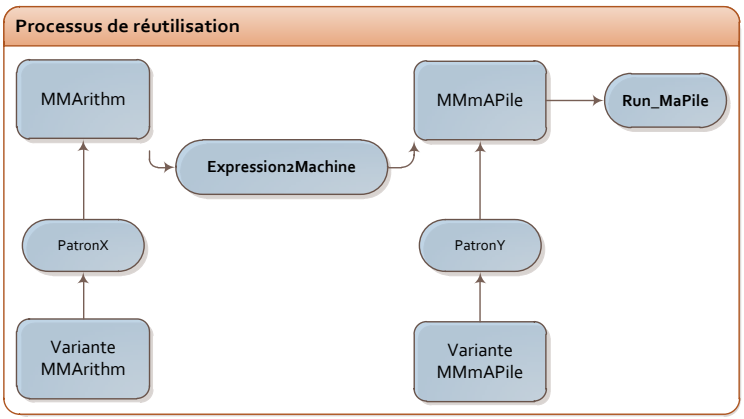

FIGURE 3.6 – Réutilisation des outils

4.4 Application des patrons de transformation

Il reste à identifier les patterns de transformation correspondants à X et Y.
Il s'agit du pattern "Divide and Combine" pour X et le pattern "ArrayToList" pour Y définis dans la section 2.1.
En effet, le patron "Divide and combine" apparait dans le métamodèle de la variante de MMArithm, où l'élément A correspond à l'élément Terme, et l'élément B correspond à l'élément expression.
Pour ce patron il suffit d'appliquer la transformation de source à cible, nous n'avons pas besoin d'appliquer la transformation inverse, puisque le résultat de la fonction run n'effectuera pas de modification au métamodèle en entrée mais il s'agit simplement de calculer la valeur de l'expression.

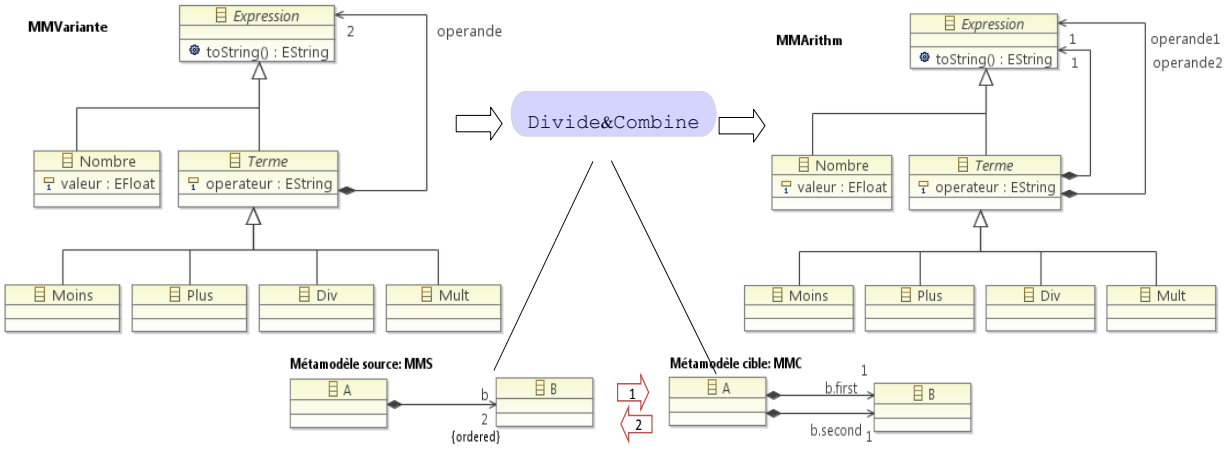

FIGURE 3.7 – Appliquer le motif Divide and combine

Pour le patron "ArrayToList", il s'agit de faire correspondre l'élément MMS.C à MMSource.machineApile, la référence MMS.C.first à la référence MMSource.machineApile.instruction, et l'élément MMS.A à l'élément MMSource.Instruction.
Pour la cible, MMC.C correspond à MMCible.MachineAPile, MMCible.MachineAPile.instructions à MMC.C.elems et enfin MMC.A correspond à MMCible.Instruction.
Pour la variante du MMmAPile, la transformation appliquée est bidirectionnelle, puisque le résultat d'exécution de la fonction run sera stocké dans la pile et la liste d'instructions sera vidée. Par conséquent, le principe est le même pour la transformation inverse, on fait correspondre chaque élément avec celui qui lui convient dans le métamodèle cible.

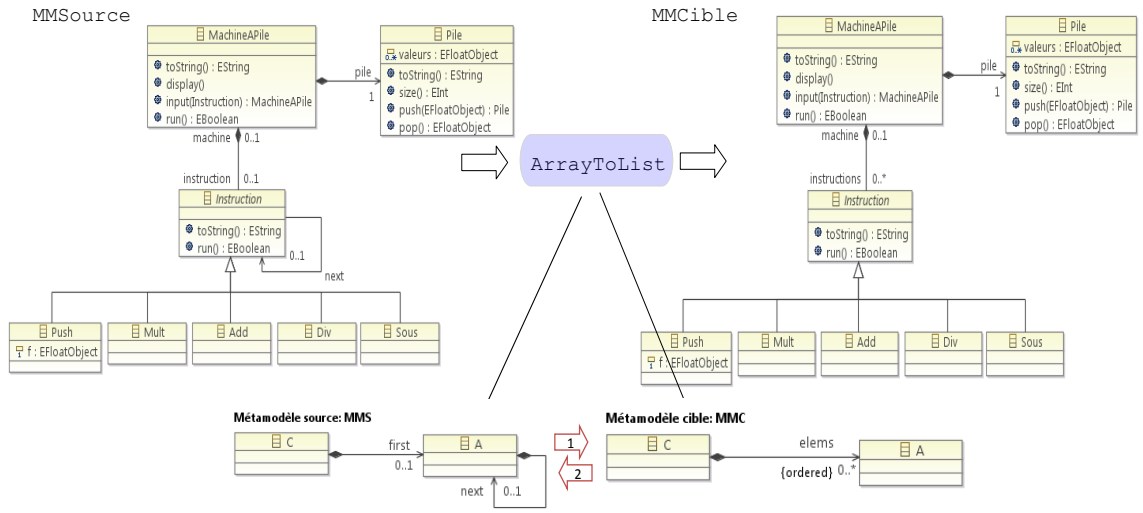

FIGURE 3.8 – Appliquer le motif ArrayToList

4.5 résultats et commentaires

Dans l'étude de cas décrite précédemment, nous disposons d'outils conçus à l'origine pour accepter des instances conformes à un métamodèle d'entrée bien déterminé et renvoyer en sortie des instances conformes à un métamodèle précis. Notre but est de pouvoir réutiliser ces outils pour des variantes structurellement différentes de celles d'origine. Nous avons expérimenté ces variantes, et nous avons réussi à appliquer notre approche sur ces métamodèles bien qu'ils étaient structurellement différents.

Conclusion

Une première phase du stage, a été consacrée à une étude bibliographique. Dans une première partie, nous avons introduitz la notion de modélisation et en particulier les artefacts de l'IDM, les langages spécifiques de modélisation ainsi que les transformations de modèles.
Dans la deuxième partie, nous avons passé en revue certains des travaux qui ont essayé de spécifier des solutions pour la réutilisation d'outils dans IDM comme la notion de substituabilité des modèles types dans Kermeta.

Dans la deuxième phase, l'objectif était de se familiariser à l'environnement de travail kermeta en développant des transformations avec ou sans la notion de modèle type, ainsi que la métamodélisation en se basant sur Ecore.

L'objectif principal de mon travail consiste à proposer des solutions qui nous permettent de réutiliser des outils avec des modèles dont le méta-modèle est une variante de celui avec lequel l'outil a été conçu à l'origine. Ces solutions, consistent en une collection de transformation récurrentes et usuelles qu'on appelle motifs de transformation de métamodèle.

Dans la plupart des transformations de modèles identifiées, il existe certains principes que nous avons pris en considération pour faciliter et valider la production de transformations de modèles. Après l'identification et la classification de ces patterns, nous avons choisi de les mettre en œuvre dans le langage kermeta et générer les transformations correspondantes. Malgré que ces transformations semblent simples à écrire, leur développement d'une façon indépendante du métamodèle sur lequel elles seront appliquées est complexe.

Dans la fin de ce rapport, nous avons introduit une étude de cas pour expérimenter les différents concepts présentés précédemment. Cet exemple a montré que les outils identifiés ont permis d'expérimenter et d'appliquer notre approche.
Une dernière phase, qui peut être considérée comme perspective, est la formalisation de la notion d'expressivité des sources et cibles de transformations.
Plus précisément, il s'agira de définir une sémantique dénotationnelle ensembliste des métamodèles mis en relation par une transformation afin de déterminer l'équivalence ou la diminution de l'expressivité du métamodèle source une fois transformé dans le métamodèle cible.

Bibliographie

[1]

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A pattern language. Towns, buildings, construction.* Oxford University Press, 1977.

[3] M. Biehl. Literature study on model transformations. Technical report, Royal Institute of Technology Stockholm, Sweden, 2010.

[4] K. B. Bruce and J. C. Vanderwaart. Semantics-driven language design : Statically type-safe virtual types in object-oriented languages. In *IN ELECTRONIC NOTES IN THEORETICAL COMPUTER SCIENCE*. Elsevier Science Publishers, 1999.

[5] T. Clark, A. Evans, P. Sammut, and J. Willans. Applied Metamodelling - A Foundation for Language Driven Development. 2004.

[6] B. Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle.* Phd thesis, Institut National Polytechnique de Toulouse, Toulouse, France, juillet 2008.

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

[8] L. Duchien and C. Dumoulin, editors. *Actes des 2ème journées sur l'Ingénierie Dirigée par les Modèles (IDM'06)*, Lille, France, jun 2006. ISBN 10 : 2–7261–1290–8.

[9] Eclipse. The eclipse modeling framework (emf) overview. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html. Consulte le 25 novembre 2010.

[10] Ecore. The eclipse modeling framework project home page. http://www.eclipse.org/modeling/emf/.

[11] F. Fleurey. *Langage et méthode pour une ingénierie des modèles fiable.* PhD thesis, Université de Rennes 1, 2006.

[12] R. France and B. Rumpe. Model-driven development of complex software : A research roadmap. *Future of Software Engineering*, pages 37–54, 2007.

[13] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[14] J. Greenfield. Software factories : Assembling applications with patterns, models, frameworks, and tools. *Microsoft Corporation*, 2004.

[15] E. Guerra, J. de Lara, D. Kolovos, and R. Paige. Inter-modelling : From theory to practice. In D. Petriu, N. Rouquette, and y. Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 376–391. Springer Berlin / Heidelberg, 2010.

[16] B. Hailpern and P.Tarr. Model-driven development : The good, the bad, and the ugly. *IBM Systems*, 45 :451 – 461, 2006.

[17] P. Huber. The model transformation language jungle - an evaluation and extension of existing approaches. Master's thesis, 2008.

[18] Kermeta. Kermeta - breathe life into your metamodels. http://www.kermeta.org/.

[19] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152 :125–142, March 2006.

[20] R. Milner. The tower of informatic models. In *From semantics to Computer Science*, 2009.

[21] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. On Executable Meta-Languages applied to Model Transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaïque, Oct. 2005.

[22] OMG. Mda specifications : The architecture of choice for a changing world. http://www.omg.org/mda/specs.htm. Consulte le 23 novembre 2010.

[23] OMG. Meta object facility (mof) core specification. http://www.omg.org/spec/MOF/2.0/. Consulte le 26 novembre 2010.

[24] G. Savaton and J. Delatour. Motif pour la métamodélisation : Élément nommé. In *2ème Journées sur l'Ingénierie Dirigée par les Modèles (IDM'06), Juin 2006, Lille.*

[25] D. C. Schmidt. Guest editors introduction : Model-driven engineering. *Computer*, 39 :25–31, 2006.

[26] E. Seidewitz. What models mean. *IEEE SOFTWARE*, 20(5) :26–32, 2003.

[27] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *Journal of Software and Systems Modeling (SoSyM)*, tba :346–379, 2010.

[28] J. Sánchez Cuadrado and J. García Molina. Approaches for model transformation reuse : Factorization and composition. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin / Heidelberg, 2008.

[29] J. Steel. *Typage de modeles.* PhD thesis, Universite de Rennes 1, 2007.

[30] J. Sztipanovits, G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, and A. Misra. Multigraph : an architecture for model-integrated computing. In *Proc. Conf. First IEEE Int Engineering of Complex Computer Systems Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP*, pages 361–368, 1995.

[31] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35 :26–36, June 2000.