HAL
open science

Gestion de métadonnées utilisant tissage et transformation de modèles

Marcos Didonet del Fabro

To cite this version:

Marcos Didonet del Fabro. Gestion de métadonnées utilisant tissage et transformation de modèles. Génie logiciel [cs.SE]. Université de Nantes, 2007. Français. NNT : . tel-00481520

HAL Id: tel-00481520

https://theses.hal.science/tel-00481520v1

Submitted on 6 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE STIM
« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATERIAUX »

Année 2007

No. attribué par la bibliothèque
|___|___|___|___|___|___|___|___|___|___|

Gestion de métadonnées utilisant tissage et transformation de modèles

THÈSE DE DOCTORAT

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Marcos DIDONET DEL FABRO

Le 11 septembre 2007 à l'UFR Sciences & Techniques, Université de Nantes,
devant le jury ci-dessous

Président : Michel Riveill		
Rapporteurs:	Michel Riveill, Professeur	Université de Nice-Sophia Antipolis
	Mokrane Bouzeghoub, Professeur	Université de Versailles
Examinateurs:	Jean Bézivin, Professeur	Université de Nantes
	Patrick Valduriez, Directeur de Recherche	INRIA
	Jeff Gray, Maître de Conférences	Université de l'Alabama à Birmingham
	Pierre-Louis Xech, Resp. Relations Recherche	Microsoft France

Directeurs de thèse: Jean Bézivin
Patrick Valduriez

Laboratoire: Laboratoire d'Informatique de Nantes Atlantique.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 - 44 322 Nantes, CEDEX 3. No. ED 366-314

Année 2007

Gestion de métadonnées utilisant tissage et transformation de modèles

Metadata management using model weaving and model transformation

THÈSE DE DOCTORAT

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Marcos DIDONET DEL FABRO

Université de Nantes

Gestion de métadonnées utilisant tissage et transformation de modèles

Résumé

L'interaction et l'interopérabilité entre différentes sources de données sont une préoccupation majeure dans plusieurs organisations. Ce problème devient plus important encore avec la multitude de formats de données, APIs et architectures existants. L'ingénierie dirigée par modèles (IDM) est un paradigme relativement nouveau qui permet de diminuer ces problèmes d'interopérabilité. L'IDM considère toutes les entités d'un système comme un modèle. Les plateformes IDM sont composées par des types de modèles différents. Les modèles de transformation sont des acteurs majeurs de cette approche. Ils sont utilisés pour définir des opérations entre modèles. Par contre, il y existe d'autres types d'interactions qui sont définies sur la base des liens. Une solution d'IDM complète doit supporter des différents types de liens. Les recherches en IDM se sont centrées dans l'étude des transformations de modèles. Par conséquence, il y a beaucoup de travail concernant différents types des liens, ainsi que leurs implications dans une plateforme IDM.

Cette thèse étudie des formes différentes de liens entre les éléments de modèles différents. Je montre, à partir d'une étude des nombreux travaux existants, que le point le plus critique de ces solutions est le manque de généricité, extensibilité et adaptabilité. Ensuite, je présente une solution d'IDM générique pour la gestion des liens entre les éléments de modèles. La solution s'appelle le tissage de modèles. Le tissage de modèles propose l'utilisation de modèles de tissage pour capturer des types différents de liens. Un modèle de tissage est conforme à un métamodèle noyau de tissage. J'introduis un ensemble des définitions pour les modèles de tissage et concepts liés. Ensuite, je montre comment les modèles de tissage et modèles de transformations sont une solution générique pour différents problèmes d'interopérabilité des données. Les modèles de tissage sont utilisés pour générer des modèles de transformations. Ensuite, je présente un outil adaptive et générique pour la création de modèles de tissage. L'approche sera validée en implémentant un outil de tissage appelé AMW (ATLAS Model Weaver). Cet outil sera utilisé comme solution de base pour différents cas d'applications.

Mots-clés: tissage de modèles, transformation de modèles, interopérabilité des données, ingénierie des modèles

Metadata management using model weaving and model transformation

Abstract

The interaction and interoperability between different data sources is a major concern in many organizations. The different formats of data, APIs, and architectures increases the incompatibilities, in a way that interoperability and interaction between components becomes a very difficult task. Model driven engineering (MDE) is a paradigm that enables diminishing interoperability problems by considering every entity as a model. MDE platforms are composed of different kinds of models. Some of the most important kinds of models are transformation models, which are used to define fixed operations between different models. In addition to fixed transformation operations, there are other kinds of interactions and relationships between models. A complete MDE solution must be capable of handling different kinds of relationships. Until now, most research has concentrated on studying transformation languages. This means additional efforts must be undertaken to study these relationships and their implications on a MDE platform.

This thesis studies different forms of relationships between models elements. We show through extensive related work that the major limitation of current solutions is the lack of genericity, extensibility and adaptability. We present a generic MDE solution for relationship management called model weaving. Model weaving proposes to capture different kinds of relationships between model elements in a weaving model. A weaving model conforms to extensions of a core weaving metamodel that supports basic relationship management. After proposing the unification of the conceptual foundations related to model weaving, we show how weaving models and transformation models are used as a generic approach for data interoperability. The weaving models are used to produce model transformations. Moreover, we present an adaptive framework for creating weaving models in a semi-automatic way. We validate our approach by developing a generic and adaptive tool called ATLAS Model Weaver (AMW), and by implementing several use cases from different application scenarios.

Keywords: model weaving, model transformations, data interoperability, model driven engineering
Discipline: Informatique

Acknowledgements

I would like to thank many people. First of all, I thank Jean Bézivin and Patrick Valduriez, who supervised me by during this thesis. They encouraged me through the good and not so good moments, helping me to arrive at this moment.

I thank the jury members, Mokrane Bouzegouhb, Michel Riveill, Jeff Gray and Pierre-Louis Xech. I thank specially Michel Riveill, who accompanied me since my first days in France. I thank INRIA and Microsoft Research for the financial support.

I thank my office colleagues. Frédéric, and our discussions about model driven engineering and other random things. Mikaël, when we discussed about everything and when he encouraged me from time to time. I thank Freddy, the only “non permanent” that was there during all these three years. I thank Hugo as well. I thank Ivan, for giving advice about writing, and for being a very helpful voisin, especially when going to the pubs to drink just one beer, of course. I thank Eduardo and Siloé for their friendship, beers and good meals! Thanks to Elodie, for all the administrative help!

I thank my friends from CTT for all the support, and specially the European ones, and the cool meetings through Europe and sometimes Brazil.

I thank my father, Dirceu, and my mother, Terezinha. They always encouraged me during this thesis and they spent a lot on phone bills! I thank my sister, Luciana, for all the support and for visiting me here as well.

Agora, a versão resumida em português.

Agradeço aos meus orientadores, Jean e Patrick. Agradeço os membros do juri: Mokrane Bouzegouhb, Michel Riveill, Jeff Gray and Pierre-Louis Xech. Agradeço o apoio financeiro do INRIA e da Microsoft. Agradeço o pessoal do laboratorio: o Fred, Freddy, Mikaël, Hugo, Elodie, Hannah, Ivan. Agradeço o Ivan, o Eduardo e a Siloé, e principalmente os panças do CTT pela parceria.

Agradeço em especial meus pais: Dirceu e Terezinha, e minha irmã, Luciana. Sem o apoio deles e os muitos telefonemas, não teria chegado onde estou.

Table of contents

1 Résumé étendu ... 1
2 Introduction ... 33
2.1 Context ... 33
2.2 Model Driven Engineering ... 34
2.3 Feature-based domain analysis ... 37
2.3.1 Representation ... 37
2.3.2 Computation ... 39
2.3.3 Utilization ... 41
2.3.4 Major issues identified ... 42
2.4 Presented approach ... 42
2.5 Thesis outline ... 43
3 State of the art ... 45
3.1 Introduction ... 45
3.2 Models ... 45
3.3 Mappings ... 47
3.4 Data interoperability ... 48
3.4.1 Centralized data interoperability ... 49
3.4.2 Distributed data interoperability ... 50
3.4.3 Mapping-based data interoperability ... 50
3.4.3.1 Matching ... 51
3.4.3.2 Comparison of adaptive matching solutions ... 55
3.4.3.3 Query discovery ... 58
3.4.3.4 Comparison of query discovery prototypes ... 59
3.4.4 Model management ... 60
3.4.4.1 Comparison of model management prototypes ... 63
3.4.5 MDE tools for interoperability ... 63
3.4.5.1 Analysis ... 65
3.5 Other application scenarios of mappings ... 65
3.5.1 Merge ... 65
3.5.2 Traceability ... 67
3.5.3 Mapping composition ... 67
3.6 Conclusions ... 68
4 Model weaving ... 71
4.1 Introduction ... 71
4.2 Models ... 71
4.2.1 Kernel MetaMetaModel (KM3) ... 73
4.3 Model weaving ... 74
4.3.1 Definitions ... 74
4.3.1.1 Core weaving metamodel ... 75
4.3.2 Extension operation ... 76
4.4 ATLAS Model Weaver tool ... 78
4.4.1 General description ... 78

4.4.1.1 MDE extensions ... 80
4.4.1.2 GUI extensions ... 81
4.5 Conclusions ... 84
5 Model-driven data interoperability ... 85
5.1 Introduction ... 85
5.2 Motivating example ... 86
5.3 Data interoperability metamodel extensions ... 88
5.3.1 Similarity expressions ... 89
5.3.2 Mapping expressions ... 90
5.3.3 Data value expressions ... 92
5.4 Interpreting data heterogeneity ... 93
5.4.1 Model transformations ... 94
5.4.2 Generic pattern of transformation ... 96
5.5 Conclusions ... 98
6 Matching transformations ... 101
6.1 Introduction ... 101
6.2 Motivating example ... 102
6.3 General overview ... 104
6.4 Matching transformations ... 105
6.4.1 Metamodel extensions ... 105
6.4.2 Creating weaving models ... 106
6.4.3 Calculating element similarity ... 108
6.4.3.1 Element-to-element similarities ... 108
6.4.3.2 Structural similarity ... 109
6.4.4 Selecting best links ... 114
6.4.4.1 Link filtering ... 114
6.4.4.2 Link rewriting ... 115
6.5 Extending the AMW tool to support matching transformations ... 117
6.6 General discussions ... 119
6.7 Conclusions ... 120
7 Case studies ... 121
7.1 Tool interoperability ... 121
7.1.1 Capturing the semantic heterogeneities ... 121
7.1.2 Interpreting the weaving model ... 122
7.1.3 Discussion ... 123
7.2 Bridge between SQL-DDL and KM3 ... 124
7.3 Data mapping between relational database and XML documents ... 127
7.4 Metamodel comparison and model migration ... 129
7.5 Traceability of model transformations ... 132
7.6 Merge in a Geographical Information System (GIS) ... 134
7.7 Model annotation ... 137
7.8 Calculating the difference between models ... 139
7.9 Conclusions ... 141
8 Conclusions ... 143
8.1 Issues on model weaving ... 143
8.2 Contributions of this thesis ... 143
8.3 Summary of contributions ... 145
8.4 Future work ... 145
8.4.1 Metamodel extensions ... 145
8.4.2 Production of transformations ... 146

8.4.3 Matching transformations 146

References 149

List of tables 157

List of figures 159

Appendix A 161

Glossary

AM3	ATLAS MegaModel Management
AMMA	ATLAS Model Management Architecture
AMW	ATLAS Model Weaver
API	Application Programming Interface
ATL	ATLAS Transformation Language
ATLAS	ATLantic dAta Systems
CVS	Concurrent Versions System
DSL	Domain-specific Language
DTD	Document Type Definition
DSWM	Domain-specific weaving metamodel
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
FAQ	Frequently Asked Questions
FDBMS	Federated Database Management System
GIS	Geographical Information Systems
GLAV	Global Local as View
GML	Graphical Markup Language
GMT	Generative Modeling Technologies
GUI	Graphical User Interface
HOT	Higher-order Transformation
HUTN	Human-Usable Textual Notation
INRIA	Institut National de Recherche en Informatique et en Automatique
JWNL	Java WordNet Library
LINA	Laboratoire d'Informatique de Nantes Atlantique
KM3	Kernel MetaMetaModel
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OWL	Web Ontology Language
QVT	Query / View / Transformation
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RDFS	RDF Schema
SBO	Semantic Bridge Ontology
SQL	Structured Query Language
SQL -DDL	SQL Data Definition Language
SVG	Scalable Vector Graphics

SWRL	Semantic Web Rule Language
TCS	Textual Concrete Language
UML	Unified Modeling Language
XMI	XML Model Interchange
XML	eXtensible Markup Language
XQuery	XML Query Language
XSD	XML Schema Definition
XSLT	eXtensible Stylesheet Language Transformation

1 Résumé étendu[1]

1.1 Introduction

L'idée de base dans l'Ingénierie Dirigée par Modèles (IDM) est de considérer les modèles comme entités de base. Un modèle est un artefact conforme à un métamodèle et qui représente un aspect donné d'un système. Les approches courantes de l'IDM ont trois niveaux de représentation pour des modèles : modèles terminaux, métamodèles et métamétamodèles [74]. Le métamodèle décrit les éléments d'un modèle terminal, la manière dont ils sont arrangés, liés et contraints. Le métamétamodèle est la représentation de base de tous les métamodèles et modèles terminaux d'un espace technique [86]. Les plateformes IDM sont composées de différents types de modèles. Un des types de modèles les plus important sont les modèles de transformation [75]. Les modèles de transformation sont utilisés pour définir des opérations entre modèles. Cependant, les transformations de modèles sont essentiellement conçues afin de définir des opérations fixes.

En plus des opérations de transformation, il existe d'autres types d'interactions possibles entre modèles. L'établissement de liens entre éléments appartenant à différents modèles est un problème central dans de nombreuses applications, concernant différents domaines tels que l'intégration de données et schémas [15] [89] [103] [102], l'interopérabilité d'outils [41], la composition des interfaces utilisateur [129], la traçabilité entre modèles [72], et d'autres.

[1] Le chapitre suivant est un résumé étendu de cette thèse. Les idées introduites dans ce chapitre seront détaillés dans la suite de ce document.

La diversité des cas d'application motive la création d'une plateforme générique pour la gestion de liens. Dans ce but, trois aspects principaux doivent être considérés. D'abord, il est nécessaire de choisir un format de représentation approprié et de définir la sémantique des liens. Le format de représentation est un compromis entre simplicité et expressivité. Aujourd'hui, nous trouvons différents formats, allant des correspondances simples [103] aux graphes [120] [40]. L'éventail des solutions est habituellement ad hoc, c.-à-d., les liens entre les modèles sont définis avec l'objectif de fournir une solution simple et rapide. Ils ne peuvent pas être réutilisés ou étendues.

Le deuxième aspect concerne la création (calcul) de ces liens. Cela est étroitement lié aux approches de matching de schéma et d'ontologies. Il est important de trouver des mécanismes pour aider la création des liens entre les éléments de modèles. Cependant, il ne s'agit pas seulement de créer des techniques de matching de schéma ou d'ontologies. Plusieurs techniques existantes donnent déjà de bons résultats. Par contre, ils ne peuvent pas être réutilisés ou modifiés de manière facile. Ainsi, une solution générique doit fournir une plateforme facile à utiliser et extensible où de nouvelles méthodes peuvent être facilement intégrées.

Finalement, le troisième aspect est l'utilisation des liens. C'est un domaine très étendu, en raison du grand nombre des cas d'application. La représentation et la création de liens doivent être adaptées aux différentes utilisations. Nous ne connaissons aucune approche IDM qui puisse être utilisée dans différents cas d'application. Ainsi, l'objectif de cette thèse est le suivant:

Définition d'une solution générique pour la gestion de liens. Une solution générique doit supporter les aspects majeurs de la gestion de liens, i.e., la représentation, le calcul et l'utilisation de liens. Afin d'être applicable à différents cas d'application (e.g., traçabilité, interopérabilité, fusion), la solution proposée sera extensible pour supporter plusieurs types de liens.

Nous proposons le *tissage de modèles* comme une solution générique pour la gestion des liens entre éléments appartenant à des modèles différents. Le tissage de modèles propose l'utilisation des modèles de tissage, qui sont un type particulier de modèles capturant différents types de liens entre des éléments de modèles. Un modèle de tissage est conforme à un métamodèle de tissage. Nous définissons un métamodèle de tissage basé sur un ensemble de besoins génériques de la gestion des liens. Le métamodèle de tissage est extensible. Il est aussi possible de créer des métamodèles de tissage spécifiques à différents domaines d'application. Ceci a une importance significative, parce que la définition d'un métamodèle complet qui pourrait être utilisé dans tous les scénarios d'application n'est pas une solution pratique.

Les modèles de tissage sont créés en utilisant des méthodes diverses. Nous utilisons une approche semi-automatique. D'abord nous exécutons des *transformations de matching*. Les transformations de matching sont une approche pratique pour développer des techniques permettant la création des modèles de tissage. Les transformations de matching peuvent être adaptées pour prendre en compte différentes extensions de métamodèles. Une fois que les modèles de tissage sont créés, nous utilisons une interface graphique pour les raffiner manuellement.

Nous proposons une méthode générique pour produire des transformations de modèles à partir des modèles de tissage. Nous prenons en compte un ensemble d'observations sur la structure des modèles de transformations et de modèles de tissages pour définir une opération de gestion de modèles pour la génération de transformations.

Pour résumer, les contributions majeures de cette thèse sont les suivantes. Nous proposons une solution générique pour la gestion des liens entre éléments de modèles. L'adaptabilité et l'extensibilité sont les avantages principaux de notre approche. Nous expliquons les avantages d'une approche adaptative et extensible en ce qui concerne les trois aspects présentés. D'abord, nous présentons un métamodèle de tissage noyau qui est extensible. Ensuite, nous présentons une nouvelle opération de gestion de modèles utilisée pour produire des modèles de transformations basées sur les modèles de

tissage. Finalement, nous proposons un framework adaptatif pour créer et exécuter des transformations de matching. Nous validons notre approche en développant un outil appelé AMW (ATLAS Model Weaver) [39]. L'outil sera utilisé dans plusieurs scénarios d'applications pour valider notre approche.

Ce chapitre est organisé comme suit. La section 1.2 présente le tissage de modèles. La section 1.3 explique comment les modèles de tissage sont utilisés pour produire des transformations de modèles exécutables. La section 1.4 décrit différentes méthodes pour créer les modèles de tissage. La section 1.5 conclue.

1.2 Tissage de modèles

Le tissage de modèles est une approche générique couvrant tous les aspects de la gestion de liens: la représentation, le calcul et l'utilisation de liens. Les liens entre éléments de modèles sont enregistrés dans un modèle de tissage. D'abord, nous définissons les modèles, métamodèle de tissage et modèle de tissage. Ensuite, nous présentons un métamodèle de tissage noyau. En conclusion, nous présentons notre outil de tissage de modèle générique.

1.2.1 Définitions

Nous présentons les définitions de graphes, de modèle, et du métamodèle de tissage (suivant [41]).

Definition 1.1 (Multi-graphe orienté). Un multi-graphe orienté $G = (N_G, E_G, \Gamma_G)$ est composé d'un ensemble fini de nœuds N_G et d'un ensemble fini d'arrêtes E_G, une fonction $\Gamma_G : E_G \rightarrow N_G \times N_G$ reliant les arcs à leurs source et cible.

Definition 1.2 (Modèle). Un modèle $M = (G, \omega, \mu)$ est un triplet dont:

- $G = (N_G, E_G, \Gamma_G)$ est un multi-graphe orienté,
- ω est un modèle associé à un multi-graphe $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$,

- $\mu : N_G \cup E_G \rightarrow N_\omega$ est une fonction associant les éléments (nœuds et arrêtes) de G aux nœuds de G_ω. La fonction μ associe tous les nœuds et arrêts de G ($N_G \cup E_G$) avec un élément de ω (N_ω).

Definition 1.3 (Modèle de référence). Étant donnée un modèle $M_1 = (G_1, \omega_1, \mu_1)$, et un modèle $M_2 = (G_2, \omega_2, \mu_2)$, si $\omega_1 = M_2$, M_2 est appelé le modèle de référence de M_1.

Quelques modèles sont leur propre modèle de référence ($\omega = M$). Ceci permet d'arrêter la récursivité introduite dans cette définition. La relation entre un modèle et son modèle de référence est appelée *conformance*. Elle se note par *conformsTo* (or *c2*). Cette définition permet un nombre indéfini de niveaux. Cependant, nous avons observé dans différents domaines (XML, RDBMS, ontologies) que seulement trois niveaux sont nécessaires (cf. chapitre 2). Nous appelons ces niveaux métamétamodèle (M3), métamodèle (M2) et modèle terminal (M1).

Definition 1.4 (Métamétamodèle). Un métamétamodèle est un modèle qui est son propre modèle de référence.

Definition 1.5 (Métamodèle). Un métamodèle est un modèle tel que son modèle de référence est un métamétamodèle.

Definition 1.6 (Modèle terminal). Un modèle terminal est un modèle tel que son modèle de référence est un métamodèle.

Un modèle peut être conforme à un seul modèle de référence. Un modèle de référence peut avoir plusieurs modèles qui sont conforme à lui. Le métamétamodèle est la représentation de base de tous les métamodèles et modèles terminaux d'un domaine donné. En conséquence, le choix du métamétamodèle est déterminant pour développer une solution générique d'IDM.

Nous capturons les liens entre les éléments de modèles dans un modèle de tissage. Un modèle de tissage est conforme à un métamodèle de tissage. Le métamodèle de tissage définit les types de liens qui peuvent être créés. Nous commençons par définir les concepts de métamodèle et de modèle de tissage. Ensuite nous présentons un métamodèle de tissage noyau.

Definition 1.7 (Métamodèle de tissage). Un métamodèle de tissage est un modèle $MM_W = (G_M, \omega_M, \mu_M)$, qui définit des types de lien, tel que :

- $G_M = (N_M, E_M, \Gamma_M)$,
- $N_M = N_L \cup N_{LE} \cup N_O$, N_L indique les *types de liens*; N_{LE} indique les *types d'extrémités de liens* et N_O indique des nœuds auxiliaires,
- $\Gamma_M : E_M \rightarrow (N_L \times N_{LE}) \cup (N_O \times N_M)$, i.e., un *type de lien* fait référence à une ou plusieurs *extrémités de liens* et les nœuds auxiliaires font référence à n'importe quel type de nœud.

Definition 1.8 (Modèle de tissage). Un modèle de tissage est un modèle $M_W = (G_W, \omega_W, \mu_W)$, le graphe $G_W = (N_W, E_W, \Gamma_W)$, tel que son modèle de référence est un métamodèle de tissage ($\omega_W = MM_W$).

Cela signifie qu'un modèle de tissage contient des liens qui permettent de lier des éléments de différents modèles. Les éléments du modèle de tissage s'appellent les *éléments de tissage*. Un modèle de tissage est un modèle terminal. Les éléments de tissage qui sont conformes aux extrémités de liens ($\mu_W (N_W) = N_{LE}$) sont des références aux éléments des modèles liés. Pour obtenir la valeur réelle des éléments liés, les extrémités de lien sont associées à une fonction de déréférencement.

Nous illustrons le métamodèle et le modèle de tissage en utilisant l'expression $t = s_1 + s_2 + s_3 + s_4 / 4$. Le langage permettant de créer cette expression contient les opérateurs d'addition et de division, plus les tokens (les éléments du modèle). Le langage n'indique pas explicitement qu'il est possible de créer des expressions complexes. La sémantique est seulement connue si nous analysons l'expression elle-même. Dans notre solution, nous créons un type de lien qui capture la sémantique de la combinaison des opérations "+" et "/". Ce processus est la *promotion* de la sémantique dans le métamodèle de tissage. Le type de lien réfère à une extrémité de lien avec la cardinalité N (les éléments source), et à une extrémité de lien avec la cardinalité 1 (l'élément cible). L'expression (le lien entre les éléments) est créée dans un modèle de tissage conforme au métamodèle de tissage.

Nous appelons un ensemble de modèles liés, plus le modèle de tissage entre eux un *tissage de modèles*.

Definition 1.9 (Tissage de modèles). Un tissage de modèles est une tuple $<M_W, S_{WM}>$, ou:

- $M_W = (G_W, \omega_W, \mu_W)$ est un modèle de tissage,
- $S_{WM} = \{M_i = (G_i, \omega_i, \mu_i), i = [1..n]\}$ est un ensemble de modèles liées par M_W.

1.2.2 Métamodèle de tissage noyau

Nous définissons un métamodèle de tissage noyau pour supporter les aspects de base de la représentation de liens. Ce métamodèle est présenté dans [39]. Le métamodèle de tissage a un ensemble d'éléments, qui sont décrits ci-dessous:

- *WElement* est un élément abstrait dont tous les autres éléments héritent. Il a un nom et une description.
- *WModel* représente l'élément racine qui contient tous les éléments d'un modèle.
- *WLink* exprime un lien entre les éléments de modèles (sémantique de lien simple). Pour pouvoir exprimer des types et des sémantiques de liens différents, cet élément peut être étendu par différents métamodèles (j'expliquerai comment ajouter des liens différents dans la section suivante).
- *WLinkEnd* définit les types d'extrémité de liens. Chaque extrémité de lien représente un élément de modèle lié. Cela permet de créer des liens de cardinalité multiple.
- *WElementRef*'s sont associés à une fonction déréférencement qui prend comme paramètre la valeur de l'attribut *ref* et renvoie l'élément lié. Il y a également la fonction inverse qui prend l'élément lié comme paramètre et qui renvoie un identifiant.

Il est possible d'associer les fonctions directement aux extrémités de lien. Cependant, nous créons *WElementRef*s séparés pour qu'un même élément puisse être référencé par plusieurs extrémités de liens.

1.2.3 Opération d'extension

Le métamodèle de tissage n'est pas un métamodèle fixe. Il peut supporter différentes types de lien. Nous obtenons des types de liens différents en étendant le métamodèle de tissage en ajoutant des liens spécifiques à un domaine donné. Cela est réalisé grâce à l'opération d'extension de métamodèle.

Definition 1.10 (Opération d'extension de métamodèle). L'extension de métamodèle est une opération MM_R = Extend (MM_W, MM_E, M_{WD}), qui prend les métamodèles MM_W, MM_E et le modèle de tissage M_{WD} en entrée, et qui produit un nouveau métamodèle MM_R. Le métamodèle MM_W est étendu par MM_E, suivant les spécifications du modèle de tissage M_{WD}.

Comme expliqué précédemment, le métamodèle de tissage noyau n'est pas conçu pour supporter tous les types de lien existants. Pour supporter différents types de liens, et donc être applicables à des scénarios variés, nous introduisons différents sous-ensembles de métamodèles de tissage qui sont spécifiques à certains domaines, qui sont des extensions au métamodèle de tissage noyau. La définition de types de lien différents n'est pas une tâche aisée et exige souvent une connaissance détaillée du domaine d'application. Nous envisageons des différents types de liens:

- Composition: liens comme *Override*, *Merge*, *Delete*.
- Interoperabilité: liens comme *Equality*, *SourceToTarget*.
- Intégration de données: *Concatenation*, *Equality*, *IntToStr*.
- Traçabilité: *Origin*, *Source*, *Evolution*, *Modified*, *Added*.
- Alignement d'ontologies: *Equivalent*, *Equality*, *Resemblance*, *Proximity*.

A partir de cette liste (qui n'est pas exhaustive) nous pouvons voir que certains types de liens sont présents dans plusieurs domaines différents, par exemple les liens d'égalité sont disponibles dans

presque tous les scénarios, ce qui motive la création de différents ensembles d'extensions au métamodèle de tissage noyau. Les extensions sont réutilisées dans des applications différentes.

1.2.4 Outil ATLAS Model Weaver

Dans cette section nous présentons l'outil ATLAS Model Weaver (AMW). AMW est un outil générique et adaptatif pour manipuler les modèles de tissage conforme à des extensions de métamodèles différentes. L'extensibilité du métamodèle de tissage noyau a plusieurs implications sur la conception d'AMW. Le défi majeur est de développer un outil qui peut être facilement adapté et étendu. De cette façon, l'outil peut supporter les différents aspects de la gestion de liens.

1.2.4.1 Description générale

Les trois notions sur lesquelles nous avons basés la conception d'AMW sont: extension de métamodèle, extension d'outil et manipulation de modèle générique. L'outil emprunte les principes de la plateforme Eclipse [51]: construire un framework de base qui est extensible et utilisable en plusieurs domaines. L'architecture d'Eclipse est basée sur des contributions: nous contribuons à la plateforme avec un nouveau plugin (composant) et nous définissons également des points d'extension (un point d'entrée pour brancher des nouvelles contributions). Ce type d'architecture s'est avéré efficace et a été largement approuvé par la communauté de développement de logiciel. Nous appliquons les mêmes principes pour créer un framework extensible pour AMW.

L'idée principale de l'implémentation est d'avoir une interface utilisateur simple de l'outil de tissage et qui pourrait être partiellement re-générée sans devoir construire un outil spécifique pour chaque application de tissage. L'outil fournit un ensemble de fonctionnalités standard pour la gestion des modèles et des métamodèles de tissage. Il est construit comme une contribution à Eclipse EMF (Eclipse Modeling Framework) [55]. EMF fournit une API pour la manipulation de modèles. L'API accède aux modèles qui sont conformes au métamétamodèle Ecore [55].

Le framework est implémenté en étant basé sur le métamodèle de tissage noyau. Puisque les extensions de métamodèles de tissage sont des extensions des éléments tels que *WLinkEnd*, *WLink* ou *WElementRef*, le framework fournit une interface standard qui manipule ces éléments et ces extensions. Le framework définit différents points d'extension, dont différents composants sont branchés, ce qui permet d'enrichir l'int. Il y a deux catégories principales d'extensions: extensions IDM et extensions GUI (Graphical User Interface). Les extensions IDM permettent d'exécuter des tâches de gestion de modèles différentes, par exemple, traduire un modèle source en un modèle cible. Les extensions GUI fournissent les facilités graphiques pour appeler les extensions IDM. Le framework commande les interactions entre ces deux catégories d'extensions.

L'outil est disponible comme un composant de GMT (Generative Modeling Technologies) sur le site officiel d'Eclipse. L'outil a plus de 15.000 lignes de code. Le site fourni une documentation extensive, avec un Wiki, FAQ, le code source, un ensemble de cas d'étude, etc.

1.3 Interopérabilité de données dirigée par des modèles

Aujourd'hui, il existe différentes sources de données disponibles, avec des formats et sémantiques différents. En raison de la collaboration accrue entre les organisations et les environnements évoluant rapidement, il est souvent nécessaire d'utiliser des données venant de différentes sources dans une même entreprise. Cependant, les données produites par des organisations différentes sont souvent hétérogènes, avec des formats de données très différents, et rendent de ce fait l'interopérabilité de données difficile.

Dans cette section, nous présentons l'utilisation des modèles de tissage et de transformation comme une solution pratique pour réaliser l'interopérabilité de données. Notre solution est utilisée pour capturer les différents types d'hétérogénéités entre modèles d'une façon abstraite et déclarative.

Nous définissons différentes extensions de métamodèles qui sont capturent des expressions courantes dans des scénarios d'interopérabilité de données. Les modèles de tissage qui sont conformes

à ces extensions sont utilisés pour produire des transformations de modèles. Nous généralisons la production de transformations dans un pattern d'opération de gestion de modèles, ce qui permettra d'appliquer notre approche dans des scénarios similaires. Ceci est une opération fréquemment exécutée dans des plateformes de modélisation. Nous encapsulons ce pattern dans une opération appelée *TransfGen*.

Dans cette section, nous présentons d'abord un ensemble d'extensions de métamodèle pour l'interopérabilité de données. Ensuite, nous décrivons comment la production de transformations est encapsulée dans un pattern de modèle de transformation.

1.3.1 Extensions de métamodèle pour l'interopérabilité de données

Dans cette section nous présentons une vue d'ensemble des extensions de métamodèle qui capturent des expressions courantes d'interopérabilité de données. Nous considérons la nécessité de lier un métamodèle source avec un métamodèle cible. Les hétérogénéités sont capturées par un métamodèle de tissage. Nous présentons ces extensions en détail dans le chapitre 5.

1.3.1.1 Expressions de similarité

Les expressions de similarité représentent des liens de ressemblance entre les éléments de métamodèles. Ces expressions sont très courantes lors du développement des transformations. Il existe différents types d'expressions de similarité.

Égalité: les éléments de modèles qui représentent exactement la même information sont reliés par des liens d'égalité.

Équivalence: les éléments liés représentent de l'information similaire, mais pas exactement la même. Cependant, la sémantique de traduction peut être identique à celle des liens d'égalité, c.-à-d., un élément cible reçoit la valeur d'un élément source.

Equivalences typées: les définitions d'égalité et d'équivalence ne font pas de distinction entre les types d'éléments. L'addition de contraintes de type évite, par exemple, créer un lien entre une classe et un attribut (types différents), ou quand il n'est pas possible de faire de conversions de types.

Disjonction: deux éléments représentent de l'information incompatible.

Généralité: les éléments liés ont une relation d'héritage.

Non équivalence: il n'est pas toujours possible de trouver des équivalences entre tous les éléments de deux métamodèles qui nous voulons lier. Un élément sans équivalents peut être simplement ignoré. Cependant, il est important que le développeur d'application se rende compte de ce qui n'est pas liée.

1.3.1.2 Expressions complexes

Les expressions complexes lient un ensemble d'éléments source et un ensemble d'éléments cible. Le métamodèle de tissage encapsule ces expressions dans ses éléments. Les extensions de métamodèles sont créées séparément et rendues disponibles dans un dépôt partagé. Le formalisme de base qui définit ces expressions est caché dans le métamodèle de tissage. Les expressions de navigation et les calculs d'expressions sont définis dans une étape suivante.

Cependant, il n'est pas possible de définir des extensions de métamodèle pour chaque type d'expression existante, puisque ces expressions varient selon le domaine d'application. En outre, ces expressions sont souvent créées manuellement parce que les liens entre les éléments de modèles sont en général compliqués. La plus part du temps elles ne sont pas créées par des techniques automatiques, car cela impliquerait un certain raisonnement sémantique.

Plusieurs-à-un: ces expressions lient un ensemble d'éléments du métamodèle source avec un élément du métamodèle cible.

Un-à-plusieurs: ces expressions sont l'opposé des expressions *plusieurs-à-un*, c.-à-d., elles lient plusieurs éléments cibles à un seul élément source.

Multiple: Les expressions multiples relient un ensemble d'éléments des métamodèles source avec un ensemble d'éléments des métamodèles cibles. Ces expressions peuvent être créées en utilisant une combinaison des expressions précédentes. Cependant, cela réduit l'expressivité du lien.

Nouvelles valeurs sur la cible: ces expressions spécifient la nécessité de générer des valeurs dans le modèle cible qui n'ont pas une correspondance dans le modèle source. Ces valeurs peuvent être automatiquement produites ou peuvent prendre une valeur prédéfinie d'entrée par un utilisateur.

1.3.1.3 Expressions de valeurs de données

Les expressions de valeur de données diffèrent des expressions complexes parce qu'elles sont utilisés pour lier également les éléments des modèles terminaux et pas seulement les éléments des métamodèles. Les expressions de valeur de données spécifient une comparaison entre les éléments du modèle source et cible, pour les rendre compatibles.

1.3.2 Production de transformations

L’étape suivante après la définition des extensions de métamodèle est la création d’un modèle de tissage. Ensuite, les modèles de tissage sont utilisés pour produire des transformations de modèles qui peuvent être exécutées dans un moteur de transformation. Les transformations produites sont utilisées, par exemple, pour traduire un ensemble de modèles terminaux d'entrée en un ensemble de modèles terminaux de sortie.

Les modèles de tissage sont créés en utilisant l'interface graphique et adaptative d'AMW. L'interface interprète les extensions de métamodèle et propose un ensemble de menus pour créer les liens de tissage. Les modèles de tissage peuvent également être créés en utilisant des méthodes semi-automatiques. Nous proposons un pattern générique basé sur les extensions de tissage de métamodèles et utilisé pour produire des transformations de modèles. Ce pattern est utilisé pour implémenter une opération générique qui produit des modèles de transformations à partir des modèles de tissage. Les

transformations de modèles permettent d'exécuter des opérations de gestion de modèles. Nous définissons ci-dessous la transformation de modèles.

Definition 1.11 (Transformation de modèle). Une transformation de modèle est une opération qui prend un ensemble de modèles en entrée, visite les éléments de ces modèles et produit un ensemble de modèles en sortie.

Une transformation de modèle a la signature suivante:

$< OUT_1 : MM_{OUT1}, \dots , OUT_m : MM_{OUTm} > = T (< IN_1 : MM_{IN1}, \dots , IN_n : MM_{INn} >)$

T est le nom de l'opération; $<IN_1 - IN_n>$ est l'ensemble des modèles d'entrée ($n \geq 1$); les modèles d'entrée sont conformes aux métamodèles d'entrée; les métamodèles d'entrée peuvent être égaux ; $OUT_1 - OUT_m$ est l'ensemble des modèles de sortie ($m \geq 1$); les modèles de sortie sont conformes aux métamodèles de sortie; les métamodèles de sortie peuvent être égaux.

Notre approche considère que les transformations sont des modèles. Ainsi, l'opération *T* est spécifiée dans un modèle de transformation $T = (G_T, MM_T, \mu_T)$. Un modèle de transformation est toujours un modèle terminal. *T* est conforme à un métamodèle de transformation MM_T. Cela signifie que toutes les opérations appliquées sur des modèles peuvent aussi être appliquées aux transformations, y compris des transformations de transformations (les avantages de considérer des transformations comme des modèles sont expliqués plus tard dans cette section).

1.3.2.1 Pattern générique de transformation

Nous encapsulons la tâche de production de transformations à partir d'un modèle de tissage dans un pattern générique de transformation. La définition de ce pattern générique de transformation est fondée sur trois faits. Premièrement, le métamodèle de tissage noyau définit le concept de liens, d'extrémités de liens et d'extensions de ces éléments. Deuxièmement, les langages courants de transformations ont des structures semblables. Troisièmement, nous utilisons des modèles déclaratifs

de transformation qui indiquent seulement quoi transformer, et pas comment transformer. Le pattern de transformation exprime la sémantique d'exécution du modèle de tissage: il transforme les différents types de liens en expressions dans un langage de transformation spécifique.

Nous utilisons des transformations d'ordre supérieur (HOT en anglais) pour définir le pattern générique. Les HOT's prennent en entrée un modèle de tissage conforme à une extension du métamodèle de tissage et le transforme en un modèle de transformation.

Definition 1.12 (Transformation d'ordre supérieur). Une transformation d'ordre supérieur est une transformation $T_{OUT} : MM_T = T_{HOT} (T_{IN} : MM_T)$, tel que les modèles d'entré et/ou de sortie sont des modèles de transformation. Les transformations d'ordre supérieur prennent un modèle de transformation en entrée et produisent un modèle de transformation en sortie.

Ce pattern défini avec des HOTs est la base pour créer une opération de gestion de modèles appelée *TransfGen*. Nous définissons cette opération ci-dessous.

Definition 1.13 (Opération *TransfGen*). *TransfGen* est une transformation d'ordre supérieur qui prend un modèle de tissage M_W en entrée et qui produit un modèle de transformation M_T en sortie. Le modèle de tissage est conforme à une extension de métamodèle MM_W.

L'opération *TransfGen* permet d'encapsuler la tâche de production de transformations. De cette façon, il est possible de bien séparer cette tâche et de proposer cette solution générique. Le pattern pour implémenter peut être utilisé comme base pour d'autres implémentations.

1.4 Transformations de matching

Nous avons vu dans les sections précédentes que les transformations de modèles sont utilisées pour exprimer différentes opérations entre modèles. Par conséquent, il y a un nombre croissant de transformations de modèles qui sont développées pour différents scénarios d'applications. Par exemple, il y a des transformations pour supporter l'interopérabilité de données, pour traduire des représentations textuelles en représentations graphiques, ou pour fusionner plusieurs modèles.

Nous avons vu dans la section 1.3 comment les modèles de tissage sont utilisés pour produire des modèles de transformations en capturant différents types de liens. Les liens sont utilisés comme spécification pour des patterns de transformation fréquemment utilisés. Cependant, le processus de création des modèles de tissage peut être partiellement automatisé. Une méthode semi-automatique basée sur des patterns bien définis apporte beaucoup d'avantages : elle diminue le temps de création des transformations; elle diminue les erreurs qui peuvent se produire avec un codage manuel; elle augmente la qualité du code produit. Le processus de création de liens entre éléments s'appelle *matching*. Il y a plusieurs solutions qui proposent de créer des liens entre différents modèles. Cependant, ces solutions ne peuvent pas être adaptées ou étendues facilement, pour supporter différentes extensions de métamodèles, ce qui rend plus difficile le développement de nouvelles techniques.

Dans cette section, nous présentons une solution adaptative pour développer différentes techniques de matching pour semi-automatiser le développement des modèles de tissage. Nous proposons l'exécution des transformations de matching. Les transformations de matching sont des transformations qui produisent des liens entre un ensemble d'éléments appartenant à des modèles différents. Ces liens sont capturés par un modèle de tissage. Le modèle de tissage est conforme aux extensions du métamodèle de tissage noyau.

1.4.1 Transformations de matching

Dans cette section nous allons présenter comment implémenter des techniques de matching. Nous définissons une opération pour chaque technique de matching. Le but est de créer les liens entre un ensemble de modèles d'entrée et de créer un modèle de tissage.

Definition 1.14 (Matching). Matching est le processus d'appariement d'éléments appartenant à des modèles différents.

Le processus de matching utilise différents techniques pour créer des liens entre éléments de modèles. Nous définissons une opération de gestion de modèle pour chaque technique différente. Le but est de trouver les liens entre les éléments d'un ensemble de modèles d'entrée et puis de créer un modèle de tissage. Le processus entier est encapsulé dans une opération appelée *Match*. L'opération prend deux modèles M_a et M_b en entrée et produit un modèle de tissage M_w en sortie. M_a et M_b sont conformes à respectivement MM_a et MM_b ; M_w est conforme à MM_w.

$M_w : MM_w = Match\ (M_a : MM_a, M_b : MM_b)$.

Nous implémentons ces opérations en utilisant des transformations de modèles. Ceci signifie que les opérations de matching sont implémentées avec des transformations de modèles spécifiques. Ces transformations s'appellent *transformations de matching*.

Definition 1.15 (Transformation de matching). Une transformation de matching est une transformation spécifique de domaine T qui prend deux ou plusieurs modèles en entrée, et qui les transforme en un modèle de tissage M_W.

$< OUT_1 : MM_{OUT1}, \ldots , OUT_n : MM_{OUTn} > = T\ (< IN_1 : MM_{IN1}, \ldots , IN_m : MM_{INm} >)$

Les transformations de matching implémentent différentes tecniques produisant les modèles de tissage. Nous pouvons donc considérer que les transformations de matching *transforment* un ensemble de modèles en un modèle de tissage. Ces transformations peuvent être adaptées pour supporter différents types de liens.

Le processus complet de création des modèles de tissage est semi-automatique, c.-à-d., c'est un processus interactif qui alterne entre l'exécution automatique des transformations de matching et le raffinement manuel des modèles de tissage dans l'outil AMW. Nous expliquons les différents types de transformations de matching implémentées dans les sections suivantes.

1.4.1.1 Création de modèles de tissage

Les transformations qui créent les modèles de tissage sont le premier type de transformations de matching exécutées. La transformation qui crée les modèles de tissage s'appelle *CreateWeaving*. La transformation prend deux modèles M_a et M_b en entrée et les transforme en un modèle de tissage M_w. M_a est conforme à MM_a, M_b est conforme à MM_b et M_w est conforme à MM_w.

$M_w : MM_w = CreateWeaving\ (M_a : MM_a, M_b : MM_b)$.

Cette transformation lie un ensemble d'éléments d'un type donné de M_a avec un ensemble d'éléments d'un type donné de M_b. Elle crée un produit cartésien restreint $M_a \times M_b$. L'opération crée un lien entre chaque paire d'éléments. Cependant, l'exécution d'un produit cartésien peut créer trop d'éléments si les modèles d'entrée sont importants. Considérez par exemple deux modèles avec 100 éléments chacun. Le produit cartésien crée un modèle de tissage avec au moins 100 x 100 = 10.000 éléments. Ces éléments sont capturés par les extensions de *WLinkEnd*. D'ailleurs, il y a un élément additionnel contenant la sémantique du lien (une extension de *WLink*). Pour cette raison, nous utilisons des versions restreintes du produit cartésien qui prennent en compte le type des éléments.

L'opération peut aussi être adaptée pour modifier des modèles de tissage (pour créer ou supprimer d'autres liens). Dans ce cas elle a un modèle de tissage comme paramètre d'entrée.

$M_w : MM_w = CreateWeaving\ (M_a : MM_a, M_b : MM_b, M_w' : MM_w)$.

1.4.1.2 Calcul de la similarité des éléments

Le deuxième type de transformation de matching calcule une valeur de similarité entre les paires d'éléments. Cette valeur de similarité est utilisée pour évaluer la proximité sémantique entre les éléments liés. Un lien avec une valeur élevée de similarité indique qu'il y a une bonne probabilité pour que l'élément source soit traduit en un élément cible.

Nous définissons une transformation de modèle appelée *AssignSimilarity*. La transformation prend un modèle de tissage M_w' et un poids (*weight*) en entrée, et elle produit un modèle de tissage M_w en

sortie. Les modèles d'entrée et de sortie sont conformes au même métamodèle de tissage MM_w. Le modèle de tissage de sortie a les nouvelles valeurs de similarité.

$M_w : MM_w = AssignSimilarity\ (M_w' : MM_w, weight: double)$.

Le paramètre *weight* est utilisé pour limiter les valeurs de similarité entre [0-weight]. Ce paramètre permet d'ajuster l'impact d'une méthode donnée de similarité. Par exemple, une méthode similarité qui compare les noms des éléments peut avoir le poids 0.8, et une méthode de similarité qui compare les types des éléments peut avoir le poids 0.2. Ceci signifie qu'un ensemble d'éléments est considéré plus semblable s'ils ont le même nom que le même type. Différentes transformations de matching peuvent être exécutées pour obtenir une valeur plus précise de similarité. Nous appliquons des méthodes d'élément-à-élément et les méthodes structurelles, qui sont présentées ci-dessous.

- **Similarités élément-à-élément**

Des similarités d'élément-à-élément sont calculées en prenant les paires d'éléments liées et en comparant les propriétés des éléments de différentes manières. Les transformations de matching d'élément-à-élément sont les techniques de matchings les plus utilisées. Nous développons différentes transformations des matchings, chacune appliquant une méthode différente.

- *Similarité de chaîne de caractères:* les noms des éléments de modèles sont considérés des chaînes de caractères (strings). Les noms sont comparés en utilisant des méthodes de comparaison de chaînes de caractères telles que la distance de Levenshtein, et n-grammes [33].
- *Dictionnaire des synonymes:* les noms sont comparés en utilisant un dictionnaire des synonymes (nous utilisons WordNet [57]). Ce dictionnaire fournit un arbre des synonymes. La similarité entre deux termes (noms d'élément) est calculée selon la distance entre ces termes dans l'arbre de synonyme. De cette façon il est possible, par exemple, d'augmenter la valeur de similarité entre les éléments qui ne donne pas de bons résultats en utilisant des méthodes de comparaison de chaîne de caractères.

Plusieurs techniques d'élément-à-élément sont déjà implémentées et sont disponibles dans des APIs publiques. Nous étendrons ainsi le moteur de transformation d'ATL pour pouvoir appeler des méthodes d'APIs externes. Le moteur de transformation fournit les méthodes d'emballage qui peuvent être appliquées à chaque élément d'un modèle. De cette façon nous sommes capables d'utiliser des APIs tel que SimMetrics [128], qui contient des méthodes de similarité des chaînes de caractères, et JWNL API [76], qui accèdent à la base de données de WordNet.

- **Similarité structurelle**

Les similarités structurelles sont calculées en utilisant les propriétés internes des éléments de modèles, par exemple, types, cardinalité, et les liens entre les éléments des modèles, par exemple, l'arbre de composition ou d'héritage. Ces données sont codées dans les métamodèles.

- *Propriétés internes*

Les éléments de modèles ont un ensemble de propriétés, telles que le type, la cardinalité, l'ordre, la longueur, etc. Considérez deux éléments de modèles $a \in M_a$ et $b \in M_b$; M_a et M_b sont des modèles terminaux différents, mais sont conforme au même métamodèle. Une transformation de matching compare les propriétés de a avec les propriétés de b. Si une propriété donnée a la même valeur, elle additionne 1(un) à une valeur provisoire de similarité. Cette valeur provisoire est multipliée par le paramètre de poids et ajoutée à la valeur initiale de similarité. Cependant, cette comparaison générique est valide seulement si M_a et M_b sont conforme au même métamodèle. Quand les métamodèles sont différents, l'opération doit être adaptée pour chaque propriété différente.

Considérez deux metamodels différents, KM3 et SQL-DDL (les métamodèles complets peuvent être trouvés dans le zoo AM3 [3]). Nous considérons deux éléments de ces métamodèles, *Attribute* de KM3 et *Column* de SQL-DDL. Un *Attribute* a des propriétés telles que le *type, lower, upper, isOrdered,* ou *isUnique*. Un *Column* a les propriétés suivantes : *default, type, keys*, ou *canBeNull*. Ces propriétés ne peuvent pas être directement comparées en utilisant une technique générique, parce que leurs valeurs ne sont pas compatibles et il n'y a aucune équivalence de noms. Par exemple, la

transformation doit tenir compte que le *canBeNull* est un booléen. La même information est capturée en analysant la valeur de la propriété *lower*.

- *Relations entre les éléments*

Il y a différents types de relations entre les éléments d'un même métamodèle, par exemple, l'arbre de composition ou d'héritage. La plupart des méthodes structurelles existantes qui exploitent les relations entre les éléments se basent sur la supposition suivante: si deux éléments de modèles sont semblables, les voisins de ces éléments sont susceptibles d'être semblables. Par exemple, si un lien entre deux attributs de deux modèles différents a une valeur élevée de similarité, les classes contenants ces attributs ont une bonne probabilité d'être similaires.

Nous créons une transformation inspirés de l'algorithme Similarity Flooding (SF) [101]. L'idée principale de SF est de propager la valeur de similarité entre une paire d'éléments vers une paire d'éléments qui sont reliés par des arrêtes avec une même étiquette.

Nous proposons de créer de différents types de modèles de propagation basés sur différentes relations structurelles ou sémantiques entre les éléments des métamodèles. Ceci permet d'avoir différentes manières de propager la similarité entre les liens, non seulement basés sur la valeur de l'étiquette des arrêtes, parce que cette supposition est trop restrictive, elle ne peut pas capturer différentes relations sémantiques entre les modèles. En revanche, elle est également trop générique, parce que nous ne pouvons pas créer des modèles spécifiques de propagation.

Notre approche permet de construire différents modèles de propagation selon le scénario d'application. La question principale est la création des éléments et des valeurs appropriés de propagation entre un ensemble de liens. Nous développons trois types différents de propagation basés sur cette règle générique. Nous les présentons ci-dessous.

Arbre de composition: cette méthode de propagation d'arbre permet de propager la similarité entre les éléments qui ont les relations de compositions, par exemple les classes et ces attributs ou ces

références (il faut noter que ce n'est pas la composition entre les classes, mais entre les classes et ses membres).

Arbre de relations: cette méthode de propagation tient compte du type des références de deux classes liées. Par exemple, considérez les liens entre les classes *(a,b)* et *(c,d)*; *a* a une référence vers *c* et *b* a une référence vers *d*. Le modèle de propagation est utilisé pour propager la similarité entre ces deux liens.

Arbre d'héritage: cette méthode permet de propager la valeur de similarité du lien entre deux classes vers les liens entre les classes qui héritent de ces deux classes. Cette méthode peut être considérée comme une extension à la méthode de propagation d'arbre de relations. Cependant, elle tient compte seulement des références qui représentent des relations d'héritage.

1.4.1.3 Sélection des meilleurs liens

Le troisième type de transformations de matching choisit seulement les liens qui satisfont un ensemble de conditions. Les liens choisis sont inclus dans le modèle de tissage final ou ré-écrits en différents types de liens. Ces transformations de matching sont généralisées par l'opération *Select<method>*.

$M_w : MM_w = Select<condition> (M_w' : MM_w)$.

L'opération prend un modèle de tissage M_w' en entrée et produit un autre modèle de tissage M_w en sortie. Les deux modèles de tissage sont conformes au même métamodèle de tissage MM_w. L'étiquette *condition* dénote les critères de sélection. Des liens sont choisis en utilisant deux méthodes : filtrage de liens et réécriture de liens. Ces méthodes sont expliquées ci-dessous.

1.4.1.4 Filtrage de liens

Il y a différents types de méthodes de filtrage de lien. La méthode la plus simple (et également la plus utilisée) est de choisir un seuil minimum et de choisir seulement les liens qui ont une valeur de

similarité plus haute que ce seuil. Le plus grand inconvénient de cette méthode est le choix d'une valeur correcte. Créer un nouveau modèle de tissage basé sur de seuils trop petits peut rapporter trop de faux positifs, c.-à-d., qui ne devraient pas être créés. En revanche, les seuils trop élevés peuvent exclure des faux négatifs.

Dans des scénarios typiques d'interopérabilité de données, une méthode courante est le choix des liens avec les valeurs de similitude les plus élevées pour chaque élément source. Cette méthode rend normalement de bons résultats parce que les transformations d'interopérabilité de données doivent traduire tous les éléments du modèle source (ou la plupart des éléments) en un modèle cible. Ainsi, il est nécessaire d'obtenir un lien entre chaque élément d'un métamodèle source avec les éléments d'un métamodèle cible. Ces liens seront utilisés pour générer des transformations des modèles.

1.4.1.5 Réécriture de liens

Les méthodes de réécriture de liens analysent les relations entre les liens d'un modèle de tissage déjà filtré. Ces relations sont utilisées pour transformer des liens simples (par exemple, *Equivalent*, *Equality*) en types de liens qui capturent différents patterns de transformation. Les patterns communs sont, par exemple, des conversions de données, concaténation, etc. Par exemple, si plus d'un élément source est lié avec le même élément cible, ce lien peut être réécrit comme un lien de concaténation. La forme la plus commune de réécriture de lien est l’imbrication entre les éléments avec des relations de composition, par exemple classes et attributs, ou tables et colonnes.

Plus de la création des liens complexes, les transformations de réécriture de liens peuvent créer les liens qui enregistrent différents types d’informations sur le processus de matching global. Après l'exécution d'un ensemble de transformations de matching, il est normal que quelques éléments du métamodèle source ne soient liés avec aucun élément du métamodèle cible, et vice-versa. Nous créons une méthode de réécriture de liens qui permet d'enregistrer les éléments source et/ou cible qui ne sont références par aucun lien. Ce type de lien peut être utilisé à différents buts: pour vérifier si le modèle

de tissage résultant est correct, pour enregistrer quels éléments ne peuvent pas être traduits d'un modèle à l'autre, ou pour les utiliser comme entrée pour créer des algorithmes de différence.

En résumé, les transformations de matching permettent d'implémenter différentes techniques de matching pour créer de modèles de tissage. Les transformations de matching peuvent être facilement modifiées pour supporter différentes extensions au métamodèle noyau. L'intégration de ces transformations dans l'outil AMW permet de paramétrer l'exécution de ces transformations.

1.5 Conclusions

Dans ce chapitre, nous avons présenté une vue générale des solutions proposées dans cette thèse. Nous avons présenté notre solution générique d'IDM pour la gestion de liens, appelée *tissage de modèles*. Nous avons séparé le problème de gestion des liens entre les éléments de différents modèles en trois aspects majeurs: représentation, calcul et utilisation. La diversité des scénarios d'application a motivé le développement d'une solution générique et extensible, capable de capturer différents types de liens.

Nous avons proposé l'utilisation des modèles de tissage pour capturer les liens entre les éléments de modèles différents. Les modèles de tissage sont conformes aux extensions d'un métamodèle de tissage noyau. Le métamodèle de tissage décrit les types de liens qui peuvent être créés. Les extensions de métamodèles permettent la création des métamodèles de tissage avec un vocabulaire plus près des domaines d'application. Un métamodèle extensible a beaucoup d'implications dans le processus global, parce que toutes les implémentations devront prendre en compte l'extensibilité. Un tel métamodèle affecte comment les modèles de tissage sont créés et utilisés.

Nous avons fait un inventaire de l'utilisation des modèles de tissage dans plusieurs scénarios d'application, et en particulier dans l'interopérabilité de données (cf. l'état de l'art). Les modèles de tissage sont utilisés comme spécifications pour produire des modèles de transformations. Nous avons classifié différents types d'hétérogénéités, qui sont capturées par différentes extensions au métamodèle

de tissage noyau. Les métamodèles de tissage englobent plusieurs patterns de transformations utilisés couramment. Nous avons défini un pattern générique pour traduire les modèles de tissage en modèles de transformation. Ce pattern encapsule le processus de production de transformations en une opération de gestion de modèles. Cette opération s'appelle *TransfGen*. Cette opération peut servir de base pour d'autres implémentations.

Nous montre la possibilité d'utiliser des transformations de modèles pour implémenter des différentes techniques de matching. Ces transformations sont appelées transformations de matching. Ces transformations peuvent être adaptées pour supporter différentes extensions de métamodèles. Les modèles de tissage sont créés en utilisant une interface utilisateur adaptative, et en utilisant les transformations de matching. Le développement et l'intégration des transformations de matching dans un outil générique permettent de développer différentes techniques de matchings existantes, et d'une façon très rapide. Ceci est très important pour définir une solution générique.

La diversité des cas d'utilisations (divers cas d'utilisation sont présentés dans le chapitre 7) démontre qu'il n'est pas possible de manipuler efficacement chaque besoin des différents scénarios d'application en utilisant des mécanismes trop généralistes, tels que des langages de transformation, ou des modèles de correspondances fixes. Nous présentons différents cas d'utilisation dans le chapitre 7. La plupart de ces cas d'utilisation sont basées sur des scénarios de taille réelle, avec des modèles de niveau raisonnable en taille et en complexité. Ceci démontrera que notre solution a atteint un niveau raisonnable de maturité, permettant de l'utiliser dans des scénarios industriels.

Il y a quelques défis à résoudre, par exemple comment améliorer des transformations de matching existantes pour devenir de plus en plus performantes, de ce fait diminuant l'intervention humaine sur la création des modèles de tissage. Une autre question importante est la création de différents sous-ensembles d'extensions de métamodèles de tissage qui englobent les patterns les plus fréquemment utilisés pour différents scénarios d'application, menant à la standardisation des domaines.

1.6 Bibliographie

[1] Abiteboul, S, Cluet, S, Milo, T. Correspondence and Translation for Heterogeneous Data. In proc. of ICDT 1997, Dephi, Greece, pp 351-363

[2] AM3. ATLAS Megamodel Management. Ref. site: http://www.eclipse.org/gmt/am3, 03/2007

[3] AM3 Atlantic Zoo. Reference site: http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/, 10/2006

[4] AMW. The ATLAS Model Weaver. Reference site: http://www.eclipse.org/gmt/amw, 05/2007

[5] An, Y, Borgida, A, Miller, R, Mylopoulos, J. A Semantic Approach to Discovering Schema Mapping Expressions. In proc. of ICDE, Istanbul, Turkey, 2007 (to appear)

[6] An, Y, Borgida, A, Mylopoulos, J. Inferring Complex Semantic Mappings between Relational Tables and Ontologies from Simple Correspondences. In proc. of ODBASE'05, 2005, Agia Napa, Cyprus, pp 1152-1169

[7] An, Y, Borgida, A, Mylopoulos, J. Constructing Complex Semantic Mappings between XML Data and Ontologies. In proc. of ISWC'05, 2005, Galway, Ireland, pp 6-20

[8] Arenas, M, Libkin, L. XML data exchange: consistency and query answering. In proc. of the Symposium on Principles of Database Systems (PODS), 2005, Baltimore, USA, pp 13-24

[9] ATL: ATLAS Transformation Language. Reference site: http://www.eclipse.org/m2m/atl, 05/2007

[10] Atzeni, P, Cappellari, P, Bernstein, P A. A Multilevel Dictionary for Model Management. In proc. of ER 2005, Klagenfurt, Austria, pp 160-175

[11] Atzeni, P, Cappellari, P, Bernstein, P A. ModelGen: Model Independent Schema Translation. In proc. of ICDE 2005, Tokyo, Japan, pp 1111-1112

[12] Atzeni, P, Cappellari, P, Bernstein, P A. Model independent schema and data translation. In proc. of EDBT 2006, Munich, Germany, pp 368-385

[13] Barbero, M, Didonet Del Fabro, M, Bézivin, J. Traceability and Provenance Issues in Global Model Management. In: 3rd ECMDA-Traceability Workshop, Haifa, Israel 2007

[14] Barbero, M, Jouault, F, Gray, J, Bézivin, J. A Practical Approach to Model Extension. ECMDA-FA 2007, Haifa, Israel, pp 32-42

[15] Batini, C, Lenzerini, M, Navathe, S B. A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys, 18(4):323–364, 1986

[16] Bergamaschi, S, Castano, S, Vincini, M, Beneventano, D. Semantic Integration of Heterogeneous Information Sources. Data and Knowledge Engineering, 36(3), 215-249, 2001

[17] Berlin, J, Motro, A. Database Schema Matching Using Machine Learning with Feature Selection. In proc. of 14th Intl. Conf. Advanced Information Systems Engineering (CAiSE) 2002, Toronto, Canada, pp 452-466

[18] Bernstein, P A. Applying Model Management to Classical Meta Data Problems. In proc. of the 1st CIDR 2003, Asilomar, USA, pp 209-220

[19] Bernstein, PA, Haas, L M, Jarke, M, Rahm, E, Wiederhold, G. Panel: Is Generic Metadata Management Feasible? In proc. of VLDB 2000, Cairo Egypt, pp 660-662

[20] Bernstein, P A, Melnik, S, Petropoulos, M, Quix, C. Industrial-strength Schema Matching. ACM SIGMOD Record 33(4), 2004

[21] Bézivin, J, Bouzitouna, S, Didonet Del Fabro, M, Gervais, M, Jouault, F, Kolovos, D, Kurtev, I, Paige, R. A Canonical Scheme for Model Composition. In proc. of Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, LNCS 4066, edited by Arend Rensink and Jos Warmer. Springer-Verlag, Bilbao, Spain, pp 346-360

[22] Bézivin, J, Brunelière, H, Allilaire, F, Barbero, M, Didonet Del Fabro, M, Jouault, F. OMG MDA Tool Capabilities RFI. Response of ATLAS Group – INRIA, Response date: 2007-03-05. Reference site: http://www.omg.org/docs/mda-user/07-03-03.pdf

[23] Bézivin, J, Didonet Del Fabro, M, Jouault, F, Kurtev, I, Valduriez, P. Model Engineering: From Principles to Applications, Springer - 2007 (to appear)

[24] Bilke, A, Naumann, F. Schema Matching using Duplicates. In proc. Intl. Conf. Data Engineering (ICDE), 2005, Tokyo, Japan, pp 69-80

[25] Boronat, A, Carsi, J, Ramos, I. Exogenous Model Merging by means of Model Management Operators. In proc. of the Third Workshop on Software Evolution through Transformations: Embracing the Change, Recife, Brazil, 2006

[26] Brockmans, S, Haase, P, Stuckenschmidt, H. Formalism-Independent Specification of Ontology Mappings - A Metamodeling Approach. In proc. of ODBASE 06, Montpellier, France, pp 901-908

[27] Brunet, G, Chechik, M, Easterbrook, S, Nejati, S, Niu, N, Sabetzadeh, M. A manifesto for model merging. In proc. of Workshop. on Global Integrated Model Management (in ICSE'06), Shangai, China, 2006

[28] Bugzilla Bug Tracking Tool. Reference site: http://www.bugzilla.org, 06/ 2006

[29] Buneman, P, Davidson, S B, Kosky, A. Theoretical Aspects of Schema Merging. In proc. of Intl. Conf. on Extending Database Technology (EDBT). LNCS; vol. 580. Springer, Vienna, Austria, pp 152-167, 1992

[30] CBOP, DSTC, IBM (2003) MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-03

[31] Cicchetti, A, Di Ruscio, D, Pierantonio, A. A Metamodel-independent approach to difference representation. Journal of Object Technology (Special Issue from *TOOLS Europe 2007*), June 2007, 20 pages

[32] Clifton, C, Housman, E, Rosenthal, A. Experience with a Combined Approach to Attribute-Matching Across Heterogeneous Databases. In proc. of IFIP 2.6 Working Conf. Database Semantics, 1996

[33] Cohen, W, Ravikumar, P, Fienberg, S E. A Comparison of String Distance Metrics for Name-Matching Tasks. In proc. of IIWeb 2003, Acapulco, Mexico, pp 73-78

[34] Czarnecki, K, Eisenecker, U. Generative Programming. Methods, Tools and Applications. Addison-Wesley, 2000, ISBN 0-201-30977-77

[35] Czarnecki, K, Helsen, S. Classification of Model Transformation Approaches. In proc. of OOPSLA'03 Workshop on the Generative Techniques in the Context of Model-Driven Architecture, Anaheim, California, USA, 2003

[36] Czarnecki, K, Helsen, S. Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, special issue on Model-Driven Software Development. 45(3), 2006, pp 621-645

[37] DAML+OIL: Reference Description (W3C) (12/18/2001). Reference site: http://www.w3.org/TR/daml+oil-reference

[38] Dhamanka, R, Lee, Y, Doan, A H, Halevy, A, Domingos P. iMAP: Discovering Complex Semantic Matches between Database Schemas. In proc. of SIGMOD 2004, Paris, France, pp 383-394

[39] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, Gueltas, G. AMW: a generic model weaver. In proc. of 1ères Journées sur l'Ingénierie Dirigée par les Modèles, 2005, Paris, France, pp 105-114

[40] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Valduriez, P. Applying Generic Model Management to Data Mapping. In proc. of BDA 2005, Saint-Malo, France, pp 343-355

[41] Didonet Del Fabro, M, Bézivin, J, Valduriez, P. Model-Driven Tool Interoperability: An Application in Bug Tracking. In proc. of The 5th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE'06), LNCS 4275, edited by R. Meersman and Z. Tari et al. Springer-Verlag Berlin Heidelberg 2006, Montpellier, France, pp 863-881

[42] Didonet Del Fabro, M, Bézivin, J, Valduriez, P. Weaving Models with the Eclipse AMW plugin. In proc. of Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany

[43] Didonet Del Fabro, M, Valduriez, P. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In proc. of The 22nd Annual ACM SAC, MT 2007 - Model Transformation Track, Seoul (Korea), pp 963-970

[44] Dion, B. Efficient Development of Safe Railway Applications Software with EN 50128 Objectives using SCADE Suite, Innotrans 2006

[45] Do, H H. Schema Matching and Mapping-based Data Integration. Verlag Dr. Müller (VDM), ISBN 3-86550-997-5, Phd thesis, Univesity of Leipzig, Germany, 2005

[46] Do, H H, Rahm, E. COMA - A System for Flexible Combination of Schema Matching Approaches. In proc. of VLDB 2002, Hong Kong, China, pp 610-621

[47] Doan, A H. Learning to Map between Structured Representations of Data. Phd Thesis. University of Washington. 2002

[48] Doan, A H, Domingos, P, Levy, A. Learning source descriptions for data integration. In proc. of WebDB Workshop 2000, Dallas, USA, pp 81-92

[49] Doan, A, Madhavan, J, Domingos, P, Halevy, A Y. Learning to map between ontologies on the semantic web. In proc. of WWW 2002, Honolulu, Hawai, USA, pp 662-673

[50] Doan A, Halevy A. Semantic Integration Research in the Database Community: A Brief Survey. AI Magazine, Special Issue on Semantic Integration, Spring 2005, pp 83-94

[51] Eclipse Foundation. Reference site: http://www.eclipse.org.

[52] Ehrig, M, Haase, P, Hefke, M, Stojanovic, N. Similarity for Ontologies - A Comprehensive Framework. In proc. of ECIS 2005, Regensburg, Germany

[53] Ehrig, M, Staab, S. QOM - Quick Ontology Mapping. In proc. of Intl. Semantic Web Conf. (ISWC), 2004, Hiroshima, Japan, pp 683-697

[54] Embley, DW, Jackmann, D, Xu, L: Multifaceted Exploitation of Metadata for Attribute Match Discovery in Information Integration. In proc. of Intl. Workshop Information Integration on the Web (WIIW), 2001, Rio de Janeiro, Brazil

[55] EMF. Eclipse Modeling Framework. Reference site: http://www.eclipse.org/emf, 04/2007

[56] Euzenat, J. An API for Ontology Alignment. In proc. of ISWC 2004, Hiroshima, Japan, pp 698-712

[57] Fellbaum, C. WordNet, an Electronic Lexical Database. MIT Press, 1998. Reference site: http://wordnet.princeton.edu/

[58] Flanakin, M. Web Log. Comments and complaints on software and technology in general. Comparison: Web-based Tracker. 08/08/2005. http://geekswithblogs.net/flanakin/articles/CompareWebTrackers.aspx

[59] Fletcher, G, Wyss, C M. Relational data mapping in MIQIS. SIGMOD (Demo), 2005, Baltimore, Maryland, USA

[60] Fuxman, A, Hernandez, M A, Ho, H, Miller, R J, Papotti, P, Popa, L. Nested Mappings: Schema Mapping Reloaded. In proc. of VLDB 2006, Chicago, Illinois, USA, pp 67-78

[61] Garcia-Molina, H, Hammer, J, Ireland, K, Papakonstantinou, Y, Ullman, J, Widom, J. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In proc. of the AAAI Symposium on Information Gathering, Stanford, California, USA, March 1995

[62] Geographic matters. An ESRI White Paper - September 2002

[63] Georg, G, France, R, Ray, I. Composing Aspect Models. In proc. of The 4th AOSD Modeling With UML Workshop UML 2003. San Francisco, CA, USA

[64] Giunchiglia, F, Shvaiko, P, Yatskevich, M. S-Match: an Algorithm and an Implementation of Semantic Matching. In proc. of 1st European Semantic Web Symposium (ESWS), 2004, Crete, Greece, pp 61-75

[65] Giunchiglia, F, Yatskevich, M. Efficient Semantic Matching. In proc. of 2nd European Semantic Web Conf. (ESWC), 2005, Crete, Greece, pp 272-289

[66] GMT Project. Generative Modeling Technologies. Reference site: http://www.eclipse.org/gmt/, 05/2007

[67] Gray, J, Lin, Y, Zhang, J. Automating Change Evolution in Model-Driven Engineering, IEEE Computer (Special Issue on Model-Driven Engineering - Doug Schmidt, ed.), vol. 39, no. 2, February 2006, pp 51-58

[68] Harel, D, Politi, M. Modeling Reactive Systems With Statecharts. The Statemate Approach. McGraw Hill, 1998

[69] Hoshiai, T, Yamane, Y, Nakamura, D, Tsuda, H. A Semantic Category Matching Approach to Ontology Alignment. In proc. of 3rd Intl. Workshop Evaluation of Ontology based Tools, 2004

[70] Hull, R, King, R. Semantic Database Modeling: Survey, Applications, and Research Issues. ACM Comput. Surv. 19(3): 201-260 (1987)

[71] Jossic, A, Didonet Del Fabro, M, Lerat, J, Bézivin, J, Jouault, F. Model Integration with Model Weaving: a Case Study in System Architecture. In proc. of International Conference on Systems Engineering and Modeling – ICSEM 2007, Haifa, Israel

[72] Jouault, F. Loosely Coupled Traceability for ATL. In proc. of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany. 2005

[73] Jouault, F. Contribution à l'étude des langages de transformation de modèles. Ph.D. thesis, Université de Nantes. 2006

[74] Jouault, F, Bézivin, J. KM3: a DSL for Metamodel Specification. In proc. of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp 171-185

[75] Jouault, F, Kurtev, I. Transforming Models with ATL. In proc. of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, pp 128-138

[76] JWNL (Java WordNet Library). Ref. site: http://sourceforge.net/projects/jwordnet. August 2006

[77] Kalfoglou, Y, Schorlemmer, W M. Ontology Mapping: The State of the Art. Semantic Interoperability and Integration, 2005

[78] Kalnins, A, Barzdins, J, Celms, E. Model Transformation Language MOLA. Edited by Uwe ASSMANN, Mehmet AKSIT et Arend RENSINK, réds., Model Driven Architecture, European MDAWorkshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers, LNCS 3599, pp 62-76

[79] Karsai, G, Agrawal, A, Shi, F, Sprinkle, J. On the Use of Graph Transformation in the Formal Specification of Model Interpreters.J. UCS, 9(11):1296–1321, 2003

[80] Kedad, Z, Bouzeghoub, M. Discovering View Expressions from a Multi-Source Information System. In proc. of CoopIS 1999. Edinburgh, Scotland, pp 57-68

[81] Kedad, Z, Xue, X. Mapping discovery for XML data integration. In proc. of CoopIS 2005, Agia Napa, Cyprus, November 2005, pp 166-182

[82] Kementsietsidis, A, Arenas, M, Miller, R J. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In proc. of ACM SIGMOD Intl. Conf. Management of Data, 2003, San Diego, USA, pp 325-336

[83] Kensche, D, Quix, C, Chatti, M A, Jarke, M. GeRoMe: A Generic Role Based Metamodel for Model Management. OTM Conferences (2) 2005, Agia Napa, Cyprus, pp 1206-1224

[84] Kolovos, D S. Extensible Platform for Specification of Integrated. Languages for mOdel maNagement (Epsilon), Official Web-Site. http://www.cs.york.ac.uk/~dkolovos/epsilon. 04/2007

[85] Kolovos, D, Paige, R F, Polack, F A C. Merging Models with the Epsilon Merging Language (EML), In proc. of ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006), Genova, Italy, pp 215-229

[86] Kurtev, I, Bézivin, J, Aksit, M. Technological Spaces: An Initial Appraisal. In proc. of CoopIS, DOA'2002 Federated Conferences, Industrial track, 2002, Irvine, California, USA

[87] Kurtev, I, Didonet Del Fabro, M. A DSL for Definition of Model Composition Operators. In proc. of 2nd Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, ECOOP 2006, Nantes, France. July 2006

[88] Lawley, M, Steel, J. Practical Declarative Model Transformation with Tefkat. In proc. of Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844, pp 139-150. Springer Berlin / Heidelberg, January 2006

[89] Lenzerini, M. Data Integration: A Theoretical Perspective. In proc. of PODS 2002, Madison, Wiscosin, USA, pp 233-246

[90] Li, W S, Clifton, C. SemInt - A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Network. Data and Knowledge Engineering 33(1), 49-84, 2000, 85

[91] Lin, Y, Gray, J, Jouault, F. DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems (Special Issue on Model-Driven Systems Development), Fall 2007

[92] M2M. Model-to-Model Transformation project. Ref. site: http://www.eclipse.org/m2m/, 05/2007

[93] Maedche, A, Motik, B, Silva, N, Volz, R. MAFRA - a mapping framework for distributed ontologies. In proc. of EKAW 2002, Siguenza, Spain, pp 235-250

[94] Madhavan, J, Bernstein, P A, Chen, K, Halevy, A, Shenoy, P. Corpus-based Schema Matching. In proc. of Workshop Information Integration on the Web at IJCAI, 2003, Acapulco, Mexico, pp 1567-1572

[95] Madhavan, J, Bernstein, P A, Domingos, P, Halevy, A. Representing and Reasoning about Mappings between Domain Models. In proc. of AAAI/IAAI 2002, Edmonton, Canada, pp 80-86

[96] Madhavan, J, Bernstein, P A, Rahm, E. Generic Schema Matching Using Cupid, In proc. of VLDB 2001, Roma, Italy, pp 49-58

[97] Madhavan, J, Halevy, A Y. Composing Mappings Among Data Sources. In proc. of VLDB 2003, Berlin, Germany, pp 572-583

[98] Mantis Bug Tracking System. Reference site: http://www.mantisbt.org/, 06/2006

[99] Melnik, S. Generic Model Management: Concepts and Algorithms, Ph.D. Dissertation, University of Leipzig, Springer LNCS 2967, 2004

[100] Melnik, S, Bernstein, P A, Halevy, A, Rahm, E. Supporting Executable Mappings in Model Management. In proc. of SIGMOD 2005, Maryland, USA, pp 167-178

[101] Melnik, S, Molina, H G, Rahm, E. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In proc. of ICDE 2002, San Jose, California, USA, pp 117-128

[102] Miller, R, Haas, L, Hernández, M. Schema Mapping as Query Discovery. In proc. of VLDB 2000, Cairo, Egypt, pp 77-88

[103] Miller, R J, Hernandez, M A, Haas, L M, Yan, L-L, Ho, C T H, Fagin, R, Popa, L. The Clio Project: Managing Heterogeneity. In proc. of SIGMOD Record 30, 1, 2001, pp 78–83

[104] Milo, T, Zohar, S. Using Schema Matching to Simplify Heterogeneous Data Translation. In proc. of VLDB 1998, New York City, USA, pp 122-133

[105] Mitra, P, Wiederhold, G. Resolving Terminological Heterogeneity in Ontologies. In proc. of Workshop on Ontologies and Semantic Interoperability at the 15th European Conference on Artificial Intelligence (ECAI), 2002, Lyon, France

[106] Mitra, P, Wiederhold, G, Kersten, M L. A Graph-Oriented Model for Articulation of Ontology Interdependencies. In proc. of EDBT 2000. Konstanz, Germany, pp 86-100

[107] Morishima, A, Okawara, T, Tanaka, J, Ishikawa, K. SMART: a tool for semantic-driven creation of complex XML mappings. In proc. of SIGMOD 2005 (demo), Maryland, USA, pp 909-911

[108] Nash, A, Bernstein, P A, Melnik, S. Composition of mappings given by embedded dependencies. In proc. of PODS 2005, Maryland, USA, pp 172-183

[109] Nejati, S, Sabetzadeh, M, Chechik, M, Easterbrook, S, Zave, P. Matching and Merging of Statecharts Specifications, In proc. of 29th ICSE, Minneapolis, USA, May 2007

[110] Noy, N, Musen, M. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In proc. of AAAI/IAAI 2000, Austin, Texas, USA, pp 450-455

[111] Omelayenko, B. RDFT: A Mapping Meta-Ontology for Web Service Integration. Knowledge Transformation for the Semantic Web 2003. pp 137-153

[112] OMG. Human Usable Textual Notation (HUTN) Specification, Final Adopted Specification. (ptc-02-12-01)

[113] OMG. Meta Object Facility (MOF) Specification, version 1.4, OMG Document formal/2002-04-03, 2002. Reference site : http://www.omg.org/technology/documents/formal/mof.htm

[114] OMG. MOF QVT Final Adopted Specification, OMG Document ptc/2005-11-01, 2005. Reference site http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[115] OMG. Object Management Group. Reference site: http://www.omg.org. 04/2007

[116] OpenGIS. Geographic Markup Language (GML) Implementation Specification, January, 29th, 2003. https://portal.opengeospatial.org/files/?artifact_id=7174

[117] OWL. Web Ontology Language. 10/02/2004. Reference site: http://www.w3.org/2004/OWL/

[118] Palopoli, L, Terracina, G, Ursino, D. The System DIKE - Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses. In proc. of ADBIS-DASFAA 2000, Prague, Czech Republic, pp 108-117

[119] Popa, L, Velegrakis, Y, Hernandez, M, Miller, R J, Fagin, R. Translating Web Data. In proc. of 28th International Conference for Very Large Databases (VLDB 2002), August 2002, Hong Kong, China, pp 598-609

[120] Pottinger, R A, Bernstein, P A. Merging Models Based on Given Correspondences. In proc. of VLDB 2003. Berlin, Germany, pp 862-873

[121] Rahm, E, Bernstein, P A. A Survey of Approaches to Automatic Schema Matching, VLDB Journal 10, 4 (Dec. 2001), pp 334-350

[122] Rahm, E, Bernstein, P A. An Online Bibliography on Schema Evolution. SIGMOD Record 35(4): 30-31 (2006)

[123] Ramesh, B, Jarke, M. Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering. V27 , Issue 1 (January 2001), pp 58-93

[124] RDF – Resource Description Framework. W3C Specification. Reference site: http://www.w3.org/RDF/

[125] Scalable Vector Graphics Specification (1.1) W3C Recommendation. 14 January 2003. http://www.w3.org/TR/SVG/

[126] Sheth, A P, Larson, J A. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Comput. Surv. 22(3): 183-236 (1990)

[127] Shvaiko, P, Euzenat, J. A Survey of Schema-Based Matching Approaches. Journal of Data Semantics IV: 146-171 (2005)

[128] SimMetrics. Developed by Sam Chapman. Reference site: http://sourceforge.net/ projects/simmetrics/, 08/2006

[129] Stirewalt, R E K, Rugaber, S. Automating user interface generation by model composition. In proc. of The IEEE International Conference on Automated Software Engineering, 1999

[130] Taentzer, G, Ehrig, K, Guerra, E, Lara, J, Lengyel, L, Levendovszky, T, Prange, U, Varro, D, Varro-Gyapay, S. Model Transformation by Graph Transformation: A Comparative Study. In proc. of the Model Transformations in Practice (MTiP) Workshop at MoDELS 2005, 2005

[131] UML. Unified Modeling Language. OMG Specification. Version 2.1.1. 01/04/2005. Reference site: http://www.omg.org/technology/documents/formal/uml.htm

[132] Vangheluwe, H, de Lara, J. Domain-Specific Modelling with AToM3. In proc. of 4th OOPSLA Workshop on Domain-Specific Modeling, Vancouver, Canada, October 2004

[133] Varro, D, Varro, G, Pataricza, A. Designing the Automatic Transformation of Visual Languages. Science of Computer Programming, 44(2):205–227, August 2002

[134] Velegrakis, Y, Miller, R J, Mylopoulos, J. Representing and Querying Data Transformations. In International Conference on Data Engineering (ICDE) 2005, Tokyo, Japan, pp 81-92

[135] Velegrakis, Y, Miller, R J, Popa, L. Mapping Adaptation under Evolving Schemas. In proc. of VLDB 2003, Berlin, Germany, pp 584-595

[136] Wang, J, Wen, J, Lochovsky, F, Ma, W. Instance-based Schema Matching for Web Databases by Domain-specific Query Probing. In proc. of VLDB, 2004, Toronto, Canada, pp 408-419

[137] Wiederhold, G. Database Design (2nd edition). New York: McGraw-Hill, 1982

[138] Willink, E. UMLX: A graphical transformation language for MDA. In proc. of Arend RENSINK, ed., CTIT Technical Report TR-CTIT-03-27, pp 13–24, Enschede, The Netherlands, june 2003. University of Twente

[139] XSD: XML Schema. W3C Specification. 28/10/2004. Reference site: http://www.w3.org/XML/Schema

[140] Xu, L, Embley, D. Discovering Direct and Indirect Matches for Schema Elements. In proc. of Intl. Conf. Database Systems for Advanced Applications DASFAA 2003, Kyoto, Japan, pp 39-46

2 Introduction

2.1 Context

Complex information systems manipulate large amounts of data. The large number of such systems leads to a significant number of data sources with different formats and semantics. These systems are often composed of a set of smaller components that interoperate, which, in turn, manipulate specific data. The way these components interact and exchange data form the system as a whole. The interaction and interoperability between different data sources is a major concern in many organizations. The different data formats, APIs, and architectures increases the incompatibilities, in a way that interaction between heterogeneous components becomes a very difficult task.

In order to cope with interoperability issues, model driven engineering (MDE) has emerged. MDE's basic assumption is to consider models as first-class entities. A model represents a given aspect of a system, which can be the data sources, the relationships between them, or even the platform code. Current MDE approaches usually have three representation levels for models: metametamodel, metamodel and terminal models [74]. The terminal model represents a given aspect of a system. The metamodel describes the various kinds of the elements of a terminal model and the way they are arranged, related and constrained. The metametamodel is the base representation format of all metamodels and models of one technical space [86].

MDE platforms are composed of different kinds of models. One of the most important kinds of models are transformation models [73]. Transformation models are used to define operations between model elements. A transformation model defines how a set of input models is transformed into a set of output models. Transformation models are usually general-purpose models based on a fixed language.

In addition to fixed transformation operations, there are other kinds of interactions and relationships between models. Once a set of models is created separately, they must be composed or put in relation to be able to interact and form the system as a whole. The most common scenario is to obtain a new model from an existing one. This situation is common in data translation scenarios [102] [40]. The relationships are used as specifications to produce data transformations.

Another scenario is the creation of a new model from two or more models, e.g., composition of models. In schema integration [89], it is often called *merging*. A new schema is created to provide a unified vision of different data sources. The merge definition from [120] and [29] requires transferring all the data from the original schemas into the merged schema (information preservation constraint). However, there are situations where only parts of the model are composed. For instance, in data warehouse systems only a subset of data may be necessary.

In other contexts, a transformed model may keep track of the model from which it has been generated. This is called traceability in recent model transformation solutions [30]. If traceability data is saved, it is possible to restore the original models. Models may also be put in relation using a secondary model, which must be maintained during the entire development process.

Until now, most research efforts in MDE concentrated on studying transformation languages. We are not aware of a MDE solution that has studied the establishment of relationships in detail. This means additional efforts must be undertaken to study these relationships and their implications on MDE platforms. Consequently, the objective of this thesis is the following:

Define a generic relationship management solution. It must provide generic mechanisms that support the main issues in relationship management, i.e., the representation, the computation and the utilization of relationships. The conceptual foundations of different approaches must be unified on a set of common definitions. The solution must provide easy adaptation mechanisms to be used in different application scenarios (e.g., traceability, interoperability, annotation, merge).

We propose a generic MDE solution for relationship management called model weaving. The purpose of model weaving covers the *representation*, *computation* and *utilization* of various relationships between elements pertaining to different models. The scope of this thesis is the study of the *conceptual*, *practical* and *applicative* aspects related to model weaving. This includes the inventory of actual and potential applications of the approach.

The conceptual investigation on model weaving encompasses the central work on unification. This implies the inventory of various solutions to different related problems and demonstrations that may be considered as variants of model weaving techniques. The conceptual investigation on model weaving has the ambition to propose a graph-based foundation covering the different aspects of the approach.

The practical validation of model weaving has been conducted in relation to the iterative definition of the conceptual foundations. The application in different scenarios has enabled a constant evolution on the conceptual definitions. The experiments conducted in this thesis intend to demonstrate that different domains can take profit of a unified solution by using a set of common techniques. Model weaving is a new research field, which is motivated by the lack of foundations in existing MDE platforms with respect to the establishment of relationships between elements of different models.

In this chapter, we present a domain analysis that encompasses all aspects of model weaving. This analysis has the purpose of investigating each one of these aspects, i.e., representation, computation and utilization, to be able to delimitate the issues studied during this thesis. First, we introduce a set of MDE principles and concepts. These concepts are not meant to be formal. They are introduced here to present the domain analysis. Then, we present the domain analysis. Finally, we present an overview of the proposed approach.

2.2 Model Driven Engineering

The basic assumption in MDE is to consider models as first-class entities. The main implication of this assumption is that models are software artifacts that can be modified, updated, or processed for different purposes. Different operations can be applied on models. This differs from the traditional view of software development where models are used essentially for general documentation.

A model represents a system. The relation between a model and a system is of major importance in model driven engineering. The system belongs to the real world. It is formed by several entities, properties and constraints that interact together. A model is typically an entity that represents a given aspect of a system, focusing on a precise goal. We define systems and models below:

Definition 2.1 (System). A system is a group of interacting, interrelated, or interdependent elements forming a complex whole (from Wikipedia.org).

Definition 2.2 (Model). A model is an artifact that represents a system. A model is formed by a set of model elements.

We illustrate in Figure 2.1 the relation between a model and the system it represents. Consider a complex library system, where students and professors can lend books, reserve, print hard copy, and other request. A model represents this library system.

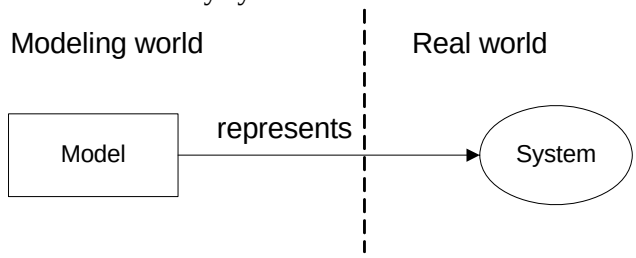


Figure 2.1 A model representing a system

The different entities of a system are captured by the model elements. The model elements have a set of properties, and may have relations between themselves. The nature of the model elements, i.e., their type, set of properties, and possible relations, are defined in a metamodel.

Definition 2.3 (Metamodel). A metamodel is a model that defines the type of the elements and relationships of a model.

A model always conforms to a metamodel. This relation is called *conformance* (often abbreviated as *c2*, for conforms to). The conformance relation has a different nature than the representation relation between a model and a system. A metamodel model may be considered as the type of a given model, because it defines a set of constraints for creating the model. A metamodel conforms to a metametamodel.

Definition 2.4 (Metametamodel). A metametamodel is a model that specifies the base representation for all models and metamodels for a given context. A metametamodel conforms to itself.

Figure 2.2 shows this three-level architecture. M3 is the metametamodel. M2 is the metamodel. M1 is the model, which represents the system. The system corresponds to the M0 level. The M0 is not part of the modeling world.

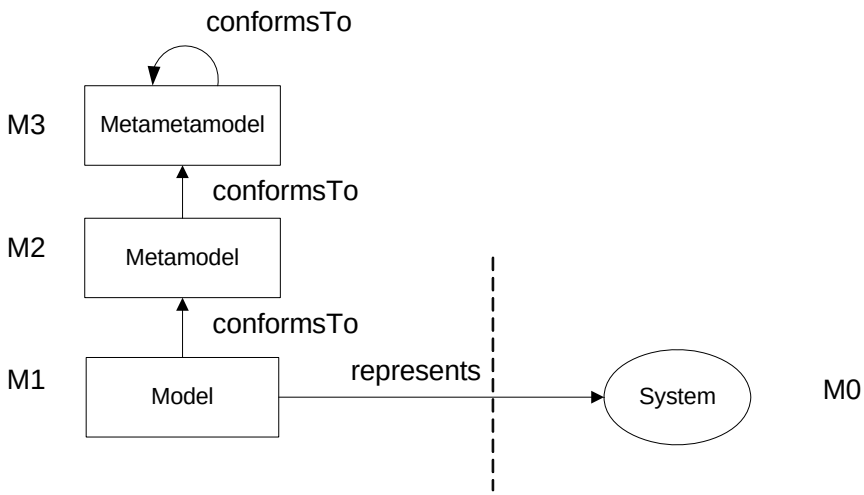


Figure 2.2 Three-level modeling architecture

This three-level architecture is general enough to be applied in different contexts. We illustrate these levels in different domains as follows[2]:

- Relational databases: the relational data corresponds to the M1 level, the schemas to the M2 level and the definition of schemas (a schema as a set of tables; tables have columns; columns have types, etc.) corresponds to the M3 level.
- XML: the XML documents correspond to the M1 level, the XML schema definitions (XSD) correspond to the M2 level, and the definition of XML as a nested structure (composed by elements; elements have sub elements and attributes) is equivalent to the M3 level.
- Structured files: the file corresponds to the M1 level; the grammar definition corresponds to the M2 level; the EBNF definition is the M3 level.

All the models and metamodels of a MDE platform are designed following this three level architecture. These models are isolated entities. The operations between these different models are defined using model transformations.

Definition 2.5 (Model transformation). Model transformation is an operation that takes a set of models as input, visits the elements of these models and produces a set of models as output.

Model transformations are defined using transformation models. Transformation models are used to define general-purpose and fixed operations between different models. There are several research efforts that study model transformations; for instance, ATL [73], GReAT [79], C-SAW [67] or VIATRA [133].

We illustrate in Figure 2.3 the base schema of a model transformation operation. Consider to transforming the input model M_A into the output model M_B. In this illustration, we do not consider multiple input or output models, however, this schema can be extended to support multiple input and/or output models. M_A conforms to metamodel MM_A (as indicated by the *c2* arrows). M_B conforms to metamodel MM_B. The transformation model M_T conforms to the transformation metamodel MM_T. MM_T defines the set of possible operations that may be defined. The transformation model contains the operations that are executed to transform M_A into M_B. All the metamodels conform to the same metametamodel.

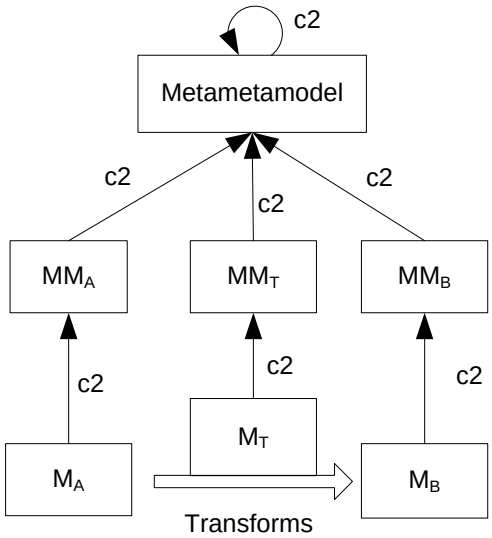


Figure 2.3 Model transformation base schema

[2] The MDE terminology used here is not always equivalent with different domains. However, the three-level architecture can be frequently identified.

There are several approaches that follow this schema (or a similar one), for instance ATL [73] and QVT [114]. Other transformation platforms do not have the notion of model, such as XSLT transformations. However, it is possible to establish equivalences between these platforms and the MDE concepts introduced here.

With this reduced set of concepts it is possible to define different kinds of model and to execute transformations between them. However, these concepts do not address the issue of establishing different kinds of relationships between these model elements (other than directed transformation models[3]).

2.3 Feature-based domain analysis

A domain analysis has the purpose of collecting relevant information about a domain and to integrate it in some kind of model. The domain analysis of model weaving is conducted using feature models [34]. We follow the approach of [36], which uses feature models to classify different model transformation approaches. Feature models define a set of requirements and concepts of a domain. The feature models are organized into connected hierarchies of common and variable features characterizing a given concept. Each different hierarchy is represented by a feature diagram.

Consider two models *Ma* and *Mb*, and two model elements, $a \in Ma$ and $b \in Mb$. A relationship *R (a, b)* indicates that *a* is somehow related to *b*. There are three aspects that must be considered when creating *R*: the representation, computation and utilization, as illustrated in the feature diagram in Figure 2.4. Each aspect is depicted by a different feature. The cardinality means that the features are mandatory [1..1], optional [0..1] or repetitive [1..N] (this notation follows the new notation for feature diagrams presented at [34]).We describe these features below.

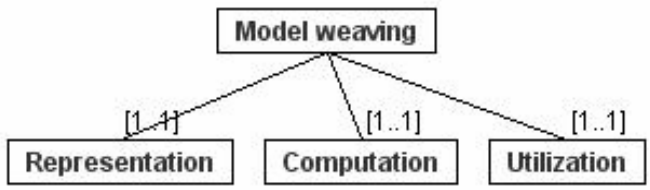


Figure 2.4 Model weaving feature diagram

Representation. The representation format the kinds of relationships that can be created, their syntax and semantics, and if (and how) they are stored.

Computation. The computation of relationships is typically a complex task that involves human intervention. Finding relationships between different model elements usually requires knowledge about the application domain that cannot be automatically interpreted by computers. The process of computing relationships is called *matching.*

Utilization. Relationships are used in different application scenarios, such as schema and data integration [15] [89] [102], aspects composition [63], tool interoperability [41], composition of user interfaces [129], traceability of transformations [72], model annotation, or model difference [31] [91].

2.3.1 Representation

We create a new feature diagram for each one of these features. The feature diagram in Figure 2.5 details the different issues covering the representation of relationships. A generic model weaving approach should support these issues. The features connected by an angle are said to be grouped

[3] Model transformations may be considered as a directed relationship between a source and a target model.

features, because they share the cardinality (e.g., <1-2>, which means a concept may have one feature present, or both at the same time).

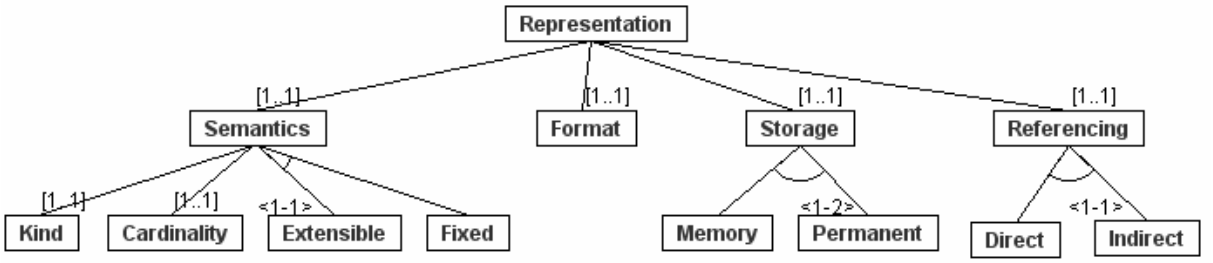


Figure 2.5 Representation feature diagram

Semantics. The semantics of the relationship define how the relationships should be interpreted. The semantics has four base features.

The *kind* of the relationships depicts its meaning. There are several different kinds of relationships, for instance *equality*, *equivalence*, *generalization*, *extension*, or *union*.

The *cardinality* indicates how many elements are connected by one relationship, i.e. 1:1, 1:N, N:N or N:1.

Most part of existing solutions supports relationships with cardinality 1:1 and representing an equivalence relation between two elements. However, complex relationships may have multiple cardinalities and semantics; for instance, to concatenate a set of alphanumeric elements, or to calculate the average between a set of elements.

The *extensible* and *fixed* features are grouped with cardinality <1-1>. This means that relationships are created using extensible or fixed semantics, but not both. Relationships with extensible semantics can be used in different application scenarios.

Format. The format depicts the standard that is used to represent the relationships; for instance, ontologies, database schemas, XML documents, or text files. The format of the relationships is closely related with the semantics. This means some formats are more adapted to represent specific kinds of relationships. For example, XML documents are more adapted to represent nested relationships than text files.

Storage. After the semantics are defined in a precise way, it may be necessary to store the relationships. The relationships can be saved in permanent storage, or can be kept in memory uniquely at the moment they are processed by some application.

Referencing. This feature specifies how the model elements are referenced by the relationships. The *direct* approach adds additional information about the relationships directly in the linked elements (e.g., the relationship kind). The *indirect* approach saves the relationships in separated entities (this is closely related with the storage format), which prevents modifying the model elements with additional information.

The extra information about links is not relevant to the model structure, since this information should not be explicitly defined in the metamodel. However, the utilization of independent entities to capture the relationships raises an extra issue: it is necessary to keep a reference that enables the recuperation of the models elements, which means they must be uniquely identified within a model.

Based on this diagram, we define a set of basic requirements for a generic model weaving solution with respect to the representation of relationships.

- Different kinds of relationships must be supported. In other words, the relationships must have a type. The type indicates the meaning of a given relationship.
- It must be possible to define relationships with different arities (unary, binary, ternary, etc.), i.e., a relationship has many endpoints.

- The relationship specification must be extensible to be able to reuse it in different scenarios, and to add new semantics according to new requirements. This is a very important issue that allows the creation of generic and reusable solutions.
- The relationships should be stored in specialized entities. This allows using them later in different ways: for reutilization, querying, modification, verification, or visualization.
- It is necessary to define an identification mechanism to uniquely identify the linked model elements. This is because the relationships themselves should not contain the concrete model elements, but a proxy that enables access to the elements of the original models.

2.3.2 Computation

The feature diagram in Figure 2.6 details the different features of the computation of relationships.

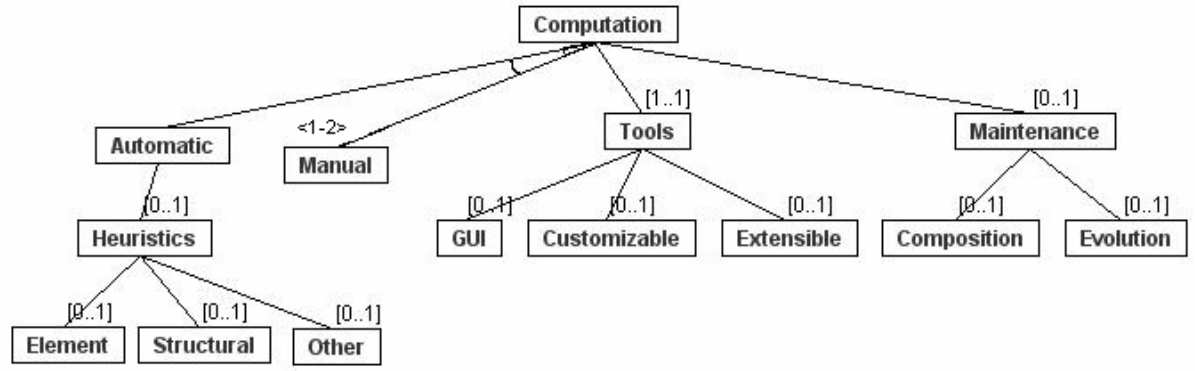


Figure 2.6 Computation feature diagram

The *automatic* and *manual* features have a grouped cardinality <1-2>. This means relationships are computed using automatic, manual, or hybrid methods.

Automatic. The automatic computation of relationships is typically processed using *heuristic techniques*. The methods are used to interpret the properties of the models (often based on the metamodels) to discover relationships between the model elements. Figure 2.7 illustrates three relationships that have been created using simple technique. These techniques calculate a numeric similarity estimation between the elements of different models, and create equivalence relationships between elements that have high similarity values. The most common kind of techniques are *element-to-element* or *structural*.

Element-to-element. Element-to-element techniques calculate similarities between pairs of elements pertaining to different models. For instance, the *String similarity* method applies string distance methods (such as Levehnstein distance [33]) to infer that *Descr* element is similar to *Description* element (the same is valid for *OpSys* and *OperatingSystem* elements). It is also possible to use dictionaries of synonyms (e.g., WordNet [57]) to discover relationships using synonyms, for example the relationships between *Car* and *Automobile*, *Professor* and *Teacher*.

Structural. Structural techniques use structural information to compute relationships. For instance, consider the elements *Bug* and *Issue* in the figure below. Both contain a *severity* attribute. A typical string comparison technique is used to create a relationship between these two attributes. The structural method consists of propagating the similarity of leaf elements into its parents. This information is used to create a new relationship between *Bug* and *Issue*.

Other. This feature subsumes the techniques that cannot be classified as element-to-element or structural. For example, relationships can be automatically created to store the execution trace of a model transformation.

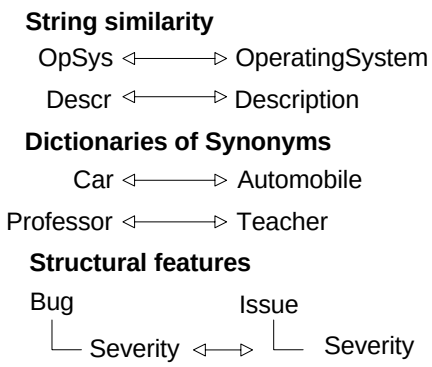


Figure 2.7 Relationships created using simple techniques

Manual. The manual methods are typically used to create complex kinds of relationships that cannot be discovered by matching techniques. We illustrate some complex kinds of relationships in Figure 2.8. Consider that the elements in the left part of the equality expression belong to one model, and that the elements in the right part belong to another model. It is necessary to create relationships with multiple cardinality (1:N), and that have different semantics (e.g., format compatibility, concatenation or data conversions).

Format compatibility
Date = Day / Month / Year
Concatenation
Name = FirstName + LastName
Data conversions
Dollar = Euro x ConversionRate

Figure 2.8 Complex relationships

Tool. A tool that supports computation of relationships has three features: *GUI*, *Customizable* and *Extensible*.

GUI. A Graphical User Interface (GUI) is very important to provide easy ways for computing relationships, either by manual creation, or by the combination of automatic methods.

Customizable. Due to the large number of existing techniques for creating relationships, a customizable tool enables creating relationships using just a subset of techniques. The automatic methods can be parameterized to be executed in a different order, or with different tuning parameters.

Extensible. An extensible tool enables integrating new automatic and manual methods for computing relationships. This feature is particularly important if the relationships have extensible semantics.

Maintenance. The maintenance of relationships concerns the *evolution* and the *reutilization* of relationships. A relationship *evolves* to adapt to any modifications on the related models. A relationship and its specification are *reusable* if they can be used in different scenarios.

Based on these considerations, we present a set of key issues that need to be considered on the computation of relationships.

- How to easily integrate different techniques into a common environment?
- How to implement matching techniques between models that conform to different metamodels?
- Which are the implications of extensible specifications of relationships when developing heuristics? For instance, how to take advantage of the kinds of relationships to ease the task of finding relationships between model elements?

- How to develop a generic tool that supports variable specification of relationships? The main challenge is to create a generic interface, which can be adapted following different semantics and formats of relationships.

2.3.3 Utilization

The feature diagram illustrated in Figure 2.9 presents the different utilizations of relationships. The different kinds of utilizations are determinant on the way the relationships are represented and computed. The key aspect about the utilization of relationships is to define relevant domain-specific relationships. We use this feature diagram to describe the different scenarios in general terms. Different scenarios are presented in Chapter 7.

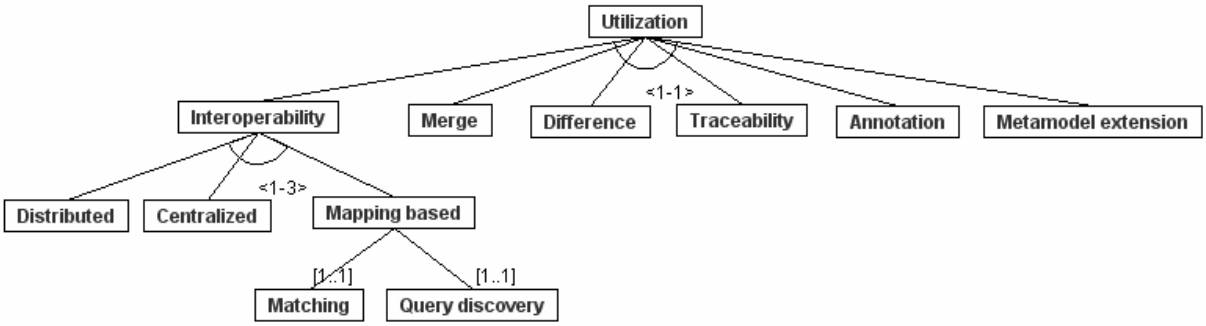


Figure 2.9 Utilization feature diagram

Interoperability. Interoperability is the problem of accessing information that is available in different data sources (database, files, tools, etc.) in a uniform way. The relationships are used to express the semantic heterogeneities between different sources. There are two major topologies of interoperability: *centralized* and *distributed*.

Centralized. The relationships are created between a centralized source and a set of distributed sources. The centralized source acts as a common access point.

Distributed. The relationships are created between every pair of sources that need to interoperate. This topology is used when it is not possible to come into a consensus about a common format.

Mapping-based. Mapping-based interoperability generalizes centralized and distributed topologies. It considers that the relationships are always created between a set of source and a set of target data, without considering if they are in a distributed or centralized topology. The mapping-based interoperability is divided in two features, *matching* and *query discovery*.

Matching. The matching feature subsumes the process of computing relationships.

Query discovery. The query discovery feature uses the relationships created by the matching feature to produce complex expressions in specific transformation metamodels.

The utilization of model weaving in interoperability scenarios encompasses typical data integration and data translation approaches. The application of model weaving as an improvement of existing interoperability techniques is one of the major contributions of this thesis.

Merge. Merge application scenarios take a set of relationships between two models as input, and produce a third, merged model, as output.

Difference. The sets of differences between two versions of a model are specified using different kinds of relationships. The relationship kind indicates if the model elements are added, deleted or removed from one version to another.

Traceability. Relationships are used to store traceability information between different models. There are different kinds of traceability scenarios. For instance, in data provenance, relationships are used to discover the origin of data after it was transformed from a source model into a target model. In

requirements traceability, the relationships keep track of all the steps of a development process: analysis, design, programming and testing.

Annotation. Relationships are used to annotate or decorate models. Annotation data usually is not conceptually relevant to be part of a model.

Metamodel extension. In metamodel extension scenarios [14], we typically have a base metamodel and fragment metamodel. A set of relationships indicates which elements of the base metamodel are extended by the fragment metamodel.

2.3.4 Major issues identified

The feature diagrams detailed in the previous section presents a general view about model weaving. Moreover, we were able to identify the major issues that must be studied to define a generic and adaptive solution. We summarize these issues below.

- A unified set of definitions and terminology for model weaving must be established.
- A generic model weaving approach must support different aspects of relationship representation, and especially the extensibility of relationships. An extensible representation allows establishing relationships adapted to different application scenarios.
- We should develop a framework that enables the creation and adaptation of the different techniques for creating relationships. This enables reuse of existing solutions, and supports the extensibility of the relationships.
- It is necessary to define several kinds of relationships that will be applied in different application scenarios. In this thesis, we focus on relationships targeted to general interoperability scenarios.
- An adaptive tool must support all these issues. The interface must support different kinds of relationships. New techniques for producing transformations should be easily developed and integrated.

2.4 Presented approach

In this thesis, we conduct a conceptual investigation on model weaving that encompasses a central work on unification. This implies the inventory of various solutions to different related problems and demonstrations that may be considered as variants of model weaving techniques. We address the issue of defining a suitable representation format by capturing the different kinds of relationships (i.e., links) in a weaving model. A weaving model conforms to a weaving metamodel. Weaving models have special characteristics. They are not self contained, i.e., a weaving model is useful only if the related models exist as well. The links have different semantics, depending on the application scenario. For instance, a data translation link has a different semantic than a traceability link. Thus, we present an extensible weaving metamodel to capture different kinds of links.

Despite having a large number of possible semantics, there is a set of common features in almost all application scenarios of model weaving. We specify a core weaving metamodel that factors out these features. This metamodel provides basic relationship management. The different kinds of links are created in separate domain-specific weaving metamodels, which are extensions to this core weaving metamodel.

Concerning the computation of weaving models, we present an adaptive approach. Weaving models can be created manually by an adaptive graphical user interface. In addition, different techniques can be used to create weaving models semi-automatically. We use *matching transformations*, which are transformations that implement different techniques that create weaving models. Thus, we reduce the matching problem to the execution of domain-specific model transformations.

The matching transformations can be modified to support different metamodel extensions. Moreover, we present an adaptation of a well-known generic structural technique: we exploit different kinds of relationships within a model to calculate similarity estimations between different model elements. This transformation is executed together with link rewriting transformations. These transformations analyze the weaving metamodel extensions to produce frequently used transformation patterns.

We investigate in detail the utilization of model weaving and model transformations in data interoperability scenarios. We classify different kinds of semantic heterogeneities according to their complexity, and we express the link semantics in a weaving metamodel. We use the weaving models as specification for producing transformations. We factor out this task into a generic model management pattern. This pattern interprets the different kinds of relationships defined in the weaving models. It takes advantage of common structures found in several existing declarative transformation languages. This pattern may be incrementally modified to handle different semantic heterogeneities. This is a frequently executed operation in model driven engineering. We encapsulate this pattern into a *TransfGen* operation. The use of weaving models and model transformations together enables a generic relationship management solution for data interoperability.

We validate our approach by implementing a generic and adaptive tool called the ATLAS Model Weaver (AMW). The AMW tool is used to implement several data interoperability use cases. We also develop uses cases in other kinds of applications that show the genericity of our approach.

2.5 Thesis outline

This thesis is organized as follows:

- **Chapter 3** contains the state of the art. It describes previous work that is considered as an application of model weaving.

- **Chapter 4** introduces the base concepts of our MDE approach. It presents the definition of model, metamodel, weaving model, weaving metamodel and extension operation. This chapter also describes our tool that uses these concepts, called the ATLAS Model Weaver.

- **Chapter 5** shows how weaving models and model transformations are used as a generic solution for data interoperability. First, we present a set of metamodel extensions for data interoperability. Then, we describe how to derive the weaving models into executable transformations. This task is encapsulated in a new model management operation.

- **Chapter 6** presents how we use model transformations to develop and to adapt different matching techniques. The overall matching process is the execution of different kinds of matching transformations and the refinement of the weaving models in the AMW tool.

- **Chapter 7** describes several use cases developed using the AMW tool. The use cases use real world models to validate our approach in different application scenarios.

- **Chapter 8** presents the general conclusion of this thesis.

3 State of the art

3.1 Introduction

We have seen in the previous chapter that model weaving encompasses all the aspects of establishing relationships between elements of different models. This differs from aspect model weaving, where the relationships are used to weave cross-cutting concepts in a principal model. The study of the relationships between model elements is closely related to metadata management research. This is a recurrent and prolific research topic that has been studied since the creation of database schemas [137]. Metadata is data that describes data, for example, different kinds of schemas, and the relationships between them. The main goal of metadata management is to provide efficient mechanisms to handle different forms of metadata. The two fundamental entities in metadata are the models and the relationships between them. There are different kinds of models; for instance a relational schema, a XML schema, or an ontology. The relationships between different models are often called *mappings*, e.g., SQL views, XSLT transformations, or ontology bridges.

There is extensive related work about the management of models and mappings. Mappings play a key role in data interoperability, because they define the relationships between different data sources. The objective of data interoperability is to be able to access heterogeneous data from different data sources. Moreover, the problem of data interoperability has become even more complex with the popularity of XML, or web-based technologies, because this increases the number of models and mappings. This has significantly increased the heterogeneity and complexity of information systems.

In this chapter, we present the state of the art about mappings, and we focus on mappings used to support data interoperability. This chapter is organized as follows. Section 3.2 introduces different kinds of representations of models. Section 3.3 describes most common mapping representations. Section 3.4 introduces a data interoperability scenario. We present three main approaches of data interoperability: centralized, distributed and mapping-based. We focus on mapping-based data interoperability. We present and compare the existing solutions. Then, we describe the model management approach, which factors out common data interoperability (and others) tasks in a set of generic operations. Finally, since this thesis has evolved in the context of MDE, we present a set of model transformation approaches. Model transformations are the central MDE solution to support interoperability. Section 3.5 presents other related work about mapping utilization not intrinsically related to data interoperability. Section 3.6 concludes.

3.2 Models

There are a multitude of different representations for models. Several propositions try to provide a standard representation. Research on a common representation starts with semantic databases (Hull et al [70]). Hull observes that semantic databases are usually defined using a small set of constructs, such as entities, entity attributes, and relationships such as *is-A*, *hasA*, *inheritsFrom*, or *contains*. The

representation depends on the application domain, varying from graphs, nested structures, schemas, ontologies, and many others. However, models typically have a set of constructs similar to the ones presented by Hull et al. Consider a simple example of a model that represents the metadata of a library. The model contains two entities *Book* and *Author*. Entity book has properties *title*, *publication year*, *nbPages*. A *Book* also contains entity *Author;* which has property *name*. This simple model would be created using similar conceptual structures, even if expressed in different languages. We can consider the models converging to a stable representation. However, there are still divergences concerning the terminology.

We describe a set of different representations below (this list focuses on representation that are widely used or that have major contributions, however, it is not exhaustive).

- SQL-DDL: SQL Data Definition Language allows defining schemas for relational databases. Typical relational schemas contain a set of flat tables. Tables have a set of columns. The different kinds of relationships between different tables are defined using foreign key columns.

- XSD: XML Schema Definition [139] is used to describe the structure of XML documents. DTD (Document Type Definition) is another language used to define the structure of XML documents. One of the major differences between XSD and DTD is that XSD has a type system. XSD schemas are based on XML. A schema has a set of nodes; nodes have attributes; and may also contain other nodes. The nesting of nodes enables to represent richer structures than relational schemas (the nested relational model is not considered in this comparison).

- OWL: Web Ontology Language [117] is a language to define ontologies over the web. Ontologies are used to define concepts and relationships with rich semantic representation. Ontologies are used to reason about the objects they represent. OWL is based on RDF (Resource Description Framework) [124] and XML. OWL adds extra vocabulary to RDF, to be able to describe more complex classes and properties, such as transitivity between properties, cardinality of properties, or restrictions over properties and classes. There are other languages to represent ontologies, such as DAML+OIL [37].

- MOF: Meta Object Facility [113] is a metametamodel developed by the Object Management Group (OMG) [115]. One of MOF's objectives is to become a standard to define metamodels. An example of metamodel described by MOF is UML (see below). MOF is formed by classes. Classes have attributes and classes may have associations between them. The expressiveness of MOF is intermediate between XSD schemas and ontology languages.

- UML: Unified Modeling Language [131] is a standard developed by OMG for the field of software engineering. UML is considered a general-purpose modeling language. It is written using MOF. UML is relatively complex if compared to other modeling languages, because it is formed by several sublanguages that are not relevant to define the structure of models, such as sequence diagrams or use cases.

- Ecore: Ecore [55] is a standard of the Eclipse Modeling Framework (EMF) to define metamodels. Ecore is similar to MOF. An Ecore metamodel has classes; classes have attributes; classes also have references (containment or aggregation references) to other classes. One of the main advantages of Ecore is its simplicity and the large number of tools available. Ecore is becoming the *de facto* standard in current Model Driven Engineering platforms.

- KM3: Kernel Metametamodel [74] is a simple language for representing metamodels. KM3 has a formal definition based on MDE concepts. The main advantage is the well-defined typing system

and the model driven architecture separated in three levels: models, metamodels and metametamodels. The KM3 definition corresponds to the metametamodel.

The proposals use graphs or tree structures according to the degree of expressiveness desired and the target application. Thus, the choice of the appropriate format is dependent on the application domain. It is not possible to say that one format/specification is better than another. Each solution tries to be the most efficient in their domain of study. For instance, OWL is more appropriate to represent ontologies than relational schemas; XSD schemas suit well to represent tree structures.

In this thesis, we do not intend to propose a new representation format. We choose KM3, for the following reasons: 1) the context of this thesis is within a MDE platform; 2) KM3 has a set of well defined concepts; 3) KM3 it is formally defined, 4) KM3 has a simple textual syntax that enable the rapid development of metamodels. KM3 is explained in details in Chapter 4.

3.3 Mappings

There are several existing representations for mappings, almost as much as for models. Mappings represent different kinds of relationships between models. In this section (and chapter) we use the more specific term mapping instead of relationship. It is a more specific term, but it is the most common terminology used in data interoperability solutions. However, this section shows that the format and terminology for mappings are still far from converging. We describe a set of key solutions below, using their specific terms. This list presents mappings with different degree of complexity, and different formats, from simple 1-to-1 relationships to complex logical axioms.

- Morphisms: morphisms are used in model management solutions [18] [99] to identify mappings between two model elements. A morphism is a pair <*l*,*r*>, where *l* and *r* store unique identifiers to the model elements. Morphisms are bidirectional relationships, defining simple equivalence semantics.

- Value correspondences: value correspondences are used in several data translation solutions [104] [135] [119] [104] [81]. A value correspondence is a pair that consists of a (1) function that defines how a source value is translated into a target value; and a (2) filter that indicates which values from the source are used in the exchange. Value correspondences are directed relationships that cannot be always inverted. Simple 1-to-1 value correspondences (called element correspondences in [135]), i.e., that relate one element of a source model and one element of a target model, are the most common format used for mappings, notably in schema and ontology matching approaches.

- Auxiliary model: the solution from [120] presents mappings as first-class entities, i.e., mapping are considered as models. The mappings are formed by an auxiliary model plus a pair of morphisms. The models are used as input for a generic merge algorithm. The models enable expressing not only equivalence semantics, but also similarity between elements. The model elements are identified by unique object IDs. Considering mappings as first-class entities enables to use the same set of primitives to manipulate mappings and models.

- First-order logic: [95] is a recent work that explicitly concentrates on the study of mappings. However, this work focuses on properties that are used to validate mappings, such as mapping composition and inference, and not on the element level representation. Based on this set of properties, it represents mappings using a variation of first-order logic.

- QVT relations: QVT relations [114] are bidirectional mappings between model elements. A QVT relation may also have a guard to restrict the elements that are mapped. QVT relations are part of QVT as a high-level definition of operational mappings. Operational mappings are executable transformations. One main drawback of this approach is the lack of available solutions and experiments.

- Ontology bridges: ontology-based approaches also consider mappings as first-class entities, i.e., mappings are ontologies. These ontologies are called ontology bridges. The ontology bridges have more complex semantics, allowing the creation of different kinds of relationships. For example, there are mappings such as *AttributeBridge* and *ConceptBridges* in [93], and *InstanceOf* and *SubclassOf* in [106]. The ontology elements are identified using RDF ids. The set of valid mappings is larger than the previous approaches.

- MOF mappings: the work from [26] proposes a mapping metamodel using MOF and UML profiles to represent ontologies. This work specifies how to map between two OWL ontologies. These mappings are used to interoperate between ontology and MDE domains.

It is not possible to say which mapping representation is the best. Each one is designed for specific application scenarios, and with precise goals in mind. However, the difference from one representation to another is much higher than between different model representations. For instance, morphisms are much less expressive than ontology bridges. An ontology bridge can be used to describe mappings equivalent to morphisms, but the opposite is not feasible. The development of a generic mapping solution that can be used in different application scenarios should support the common features of model weaving presented in Chapter 2.

These differences lead to confusion on the utilization of mappings and in the comparison between existing approaches. In typical data interoperability solutions, mappings are considered as high-level representations of relationships between model elements. The mappings are used as input to generic transformation or query engines [102] [81] [56]. The engines use the mappings for producing transformations, or for executing different operations. This enables the separation of mapping representation and transformation execution. However, some approaches consider mappings as complete transformation (or query) languages. This is often the case of ontology-based approaches [93] [111]. These solutions implement engines that natively execute complex ontological axioms.

In the following sections we detail existing solutions used in data interoperability scenarios. This enables a clear view of the advantages or drawbacks of each approach.

3.4 Data interoperability

Data interoperability is the technique of accessing information that is available in different sources (database, files, tools, etc.) in a uniform way. Most data interoperability solutions have been developed to integrate heterogeneous relational databases. However, the existing approaches may be applied to different types of data sources as well, for example text files, XML documents, models and tools.

Data interoperability solutions are organized in two main topologies: centralized and distributed. In the centralized solutions there is a unique access point to all sources of data. In the distributed solutions, the data is translated from one source to another. Both approaches need to define mappings between the different data sources. For that reason, the study of the mappings for data interoperability can be separated in a third, more generic approach, called mapping-based data interoperability.

There are several metadata tasks that are frequently executed in data interoperability solutions, for instance the creation of mappings, the management of model versions, the merge of models, and others. Model management [18] is a relatively new approach with the main goal of factoring out

frequently executed metadata tasks into a set of operations. Many model management operations are applied to data interoperability tasks.

The MDE paradigm can be considered as a branch of model management that considers every entity as a model, e.g., mappings and models. The interoperability between different models is achieved through the execution of model transformations, as well as the definition of model management operations.

This section is organized as follows. Section 3.4.1 introduces centralized data interoperability. Section 3.4.2 describes distributed data interoperability. Section 3.4.3 presents mapping-based data interoperability. We review the most relevant solutions and we provide a comparison between them. Section 3.4.4 presents the model management approach, focusing on data interoperability operations. Finally, section 3.4.5 presents a set of MDE model transformations that handle interoperability between different models.

3.4.1 Centralized data interoperability

Centralized data interoperability approaches have a common access point to a set of sources of data. This is the typical solution of data integration problems. We illustrate this scenario in Figure 3.1. Consider the three sources of data S1, S2 and S3 (called local sources). The information of each source is accessed through a common access point G (called global source). Every request is done over the global source. A request can be any kind of operation, e.g., a query, an insert, an update. There are mappings between the global source and each local source, depicted by M1, M2 and M3 (the directionality of the mappings is irrelevant here). The mappings specify how to obtain the information from the local sources based on the request done over the global source.

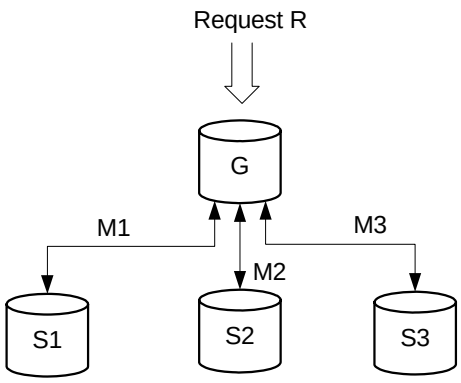


Figure 3.1 Single access point

There are different solutions following this general architecture.

- In federated databases [126], each different source is an autonomous relational database. The common access point is usually done by means of a middleware platform, called a federated database management system (FDBMS). The requests are done over the FDBMS. The federated management system may have a global representation of all local sources. In this case, the requests are translated into the format of the local sources. If there is no global representation, the requests are only redirected to the sources.
- In mediator-based systems [61], *mediators* are equivalent to the common access point to the local sources. The mappings between mediator and local sources are executed by *translators* or *wrappers*. In mediator based systems or FDBMS, the common access point is usually virtual, i.e., the information is physically stored in the local sources.

- In relational databases, it is also frequent to design virtual global sources as common access points [89]. The mappings between global and local sources are created using expressions in first-order logic. However, it may be necessary to physically store the information in the global source, for example in the case of data warehouses. In this case, the global view is materialized into permanent storage.
- In ontology integration solutions [106] [110], usually there is an alignment ontology that is the global source used to integrate a set of local ontologies.

3.4.2 Distributed data interoperability

In distributed data interoperability, it is not possible to have a common access point to all the local sources. The distributed scenario is illustrated in Figure 3.2. The set of local sources (S1, S2 and S3) are autonomous and may be distributed through different sites. There is no common access point G. The requests (R1, R2 and R3) are done over each local source. For example, a request over source S1 may read the information produced by S2 or S3. The data is obtained by using direct mappings between the sources (M12, M23 or M13). In this case, there is a distinction between source and target representations. For instance, the mapping M12 may be used to translate data from S1 to S2 (S1 is the source and S2 is the target), or in the opposite sense, to translate data from S2 to S1.

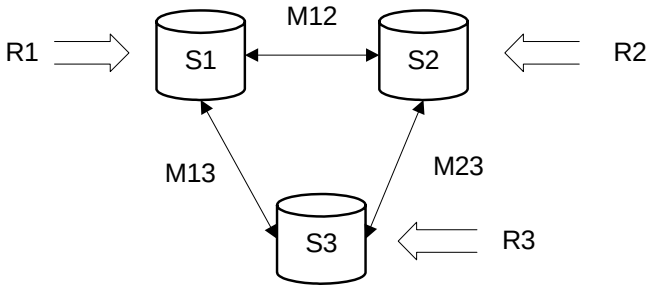


Figure 3.2 Multiple access point

The techniques of distributed data interoperability are related to data translation [8] [104] [59] and ontology mapping [56] [93] [106] [110]. These solutions have mappings between source and target representations. However, there is not a mapping between every different source. The mappings are created only if a translation is necessary. This approach is more adapted to environments with constant changes (for example, peer-to-peer systems or tool interoperability problems), when it is possible to obtain a common and integrated representation.

3.4.3 Mapping-based data interoperability

Although the two previous approaches differ in the general organization of mappings, both need to define mappings that relate two or more data sources. We illustrate this issue using the simple scenario from Figure 3.3. We do not make any initial assumption about the nature of the mapping M12 between S2 and S1.

Consider first the centralized approach. S2 is the global source, and S1 is the local source. Mappings from S1 to S2 are used to translate the data from the local source into the global source. Thus, the mappings are directed source-to-target relations. Now consider the distributed approach. We consider S1 the source data and S2 the target data. A mapping between S1 and S2 defines how to translate the data of S1 into S2. In this case there is no distinction between local or global sources.

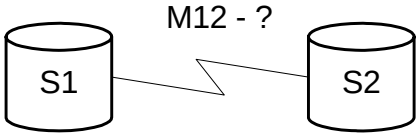


Figure 3.3 Mappings in data interoperability

Consequently, a key issue in data interoperability solutions is the creation of these source-to-target mappings. Considering mappings as a generic solution for centralized and distributed data interoperability is a relatively recent approach, called *mapping-based data interoperability*, or *mapping-based data integration*.

Current solutions define mappings as relationships at a high abstraction level, typically independent of any transformation or query language. The creation of these mappings is encapsulated in a *Match* operation (we review a set of matching approaches in the following sections). This enables to clearly separate the mappings specification and the transformation execution.

After the mappings are created, they are used to produce transformations (also called operational mappings), as illustrated in Figure 3.4.

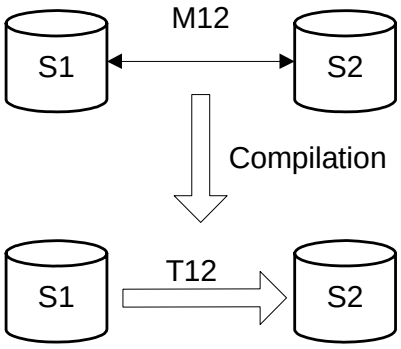


Figure 3.4 Mappings are used to produce executable programs

The mapping M12 between sources S1 and S2 is used to produce the transformation T12. This compilation process is often called *query discovery* in data exchange solutions [102]. The transformation T12 is executed in some transformation engine to translate the data from S1 to S2. These transformations are written in specific transformation languages, such as XSLT [81], ATL [41], or SQL-like queries [119]. These transformations are used to translate the source data into the target data. They are constructed in different ways, by means of model transformations, SQL queries, XSLT, SWRL or program code.

3.4.3.1 Matching

Matching is a central task in mapping-based data interoperability. Matching is the process of creating mappings (relationships) between different models. Current solutions usually encapsulate the matching task on an operation called *Match*. There have been several research efforts about the *Match* operation, in different domains. The survey from [121] considers several prototypes and approaches of schema matching. The work from [77] considers only ontology-based approaches. More recently, the work from [127] considers schema and also ontology matching approaches. The online categorizer proposed by [122] gives an idea of the large amount of solutions: it contains 138 approaches of matching and mapping (data from April 2007).

In this section, we review a set of matching approaches. We focus on solutions that describe generic frameworks that help the task of developing and combining different matching heuristics and

techniques. This list is not exhaustive: it presents solutions with significant changes and contributions to the general matching process.

We do not make an explicit distinction between schema or ontology matching solutions, as in [127] or [45]. We consider that the existing prototypes have many similarities and that they should be compared in a uniform way.

- **CUPID**

CUPID [96] presents a semi-automatic schema matching algorithm. It uses graphs to create a generic schema model. Database and XML schemas are translated into this schema model.

The schema model is a rooted graph whose nodes are elements. The elements are connected by different types of relationships, containment (i.e., every element, except the root, is contained by exactly one element), aggregation (similar to containment, but defining only references between elements) and *IsDerivedFrom* (represents generalization and typing relationships). The internal graph-based format allows matching different kinds of schemas. For instance, referential constraints are added in the graph model to match relational schemas. This increases the similarities based on structural information, since a new node is created for every referential constraint.

The algorithm uses at the same time structural and linguistic heuristics. CUPID proposes the combination of different heuristics (called matchers). The structural matcher determines the similarity between elements that have the same type. The name matcher uses synonyms and string comparison between a pair of elements.

The resulting mapping contains element level correspondences with 1:1 cardinality. The mapping as a whole has n:m cardinality. The solution concentrates on the proposition and evaluation of matching algorithms, not entering into detail about specific utilizations.

- **GLUE**

GLUE [47] [49] is a prototype for matching ontologies. It is an extension of an existing schema matching system called LSD [47]. LSD produces mappings using machine learning techniques that evaluate instance information from the input ontologies. The input ontologies are translated into a unified ontology format.

An ontology is formed by concepts, attributes and relations. The concepts provide the entities of interest of a given domain. A concept is associated with a set of attributes, and may have a set of relations with other concepts.

GLUE presents an algorithm that matches ontologies by combining different techniques. It introduces a new heuristic matcher that uses taxonomies as external input to help the matching process. The matcher returns a probability distribution of instances associated with a pair of concepts from this taxonomy. For instance, the probability P (A, B) is the probability that a given instance I is associated with both concepts A and B.

These distributions are used as input to machine-learning techniques, called learners. Two learners have been implemented: content learner (the text content of an instance) and name learner (the concatenation of all concepts in the parent taxonomy). Based on this distribution, the algorithm applies user-defined functions to obtain a similarity value for a pair of concepts. After finding the similarities, the concepts in the second ontology are labeled using the node structural features and attributes. Despite matching complex ontologies, the matching algorithm produces as a result 1 to 1 mappings. Similar to CUPID, this solution focuses on developing matching algorithms.

- **PROMPT**

PROMPT [110] is an ontology merging and alignment algorithm. Merging is the process of creating a new ontology that contains all the elements from a set of source ontologies. Alignment is the process of maintaining the consistency between ontologies without merging them.

This work presents a general knowledge model that represents different classes of ontologies. The knowledge model is formed by 4 different structures:

- Classes: collections of objects with similar properties.
- Slots: binary relations between classes and objects.
- Facets: ternary relations between a class, a slot and either a class or an object.
- Instances: the members of the classes.

The algorithm is the same regardless if the ultimate goal is to create a new ontology or to make them consistent through an alignment. The algorithm creates a list of matches (1:1 relationships) based on the class' names. A set of operations is listed based on these matches. These operations are searched over a catalog of pre-defined operations, such as merge of classes and slots, deep copy of a model or shallow copy. There is not a concrete mapping as output. The output is the merged ontology

PROMPT presents an iterative algorithm: the user chooses one of the suggested operations. Based on the results, the algorithm determines conflicts and makes new suggestions. The iterations are repeated until the user decides that the matching is accurate enough.

- **COMA/COMA++**

COMA/COMA++ [46] [45] is a generic matching prototype. The input models are relational or XML schemas translated into an internal format. The schemas are represented by rooted and directed acyclic graphs. The schema elements are graph nodes connected by directed links of different types, e.g., containment or aggregation. The mappings are simple equivalence correspondences with 1:1 cardinality.

COMA++ supports different heuristics and reuses previous matching results by composition. Different matchings are composed if there is transitivity between the 1-to-1 correspondences and the associated schemas. However, the matching result is not guaranteed to be correct.

The general procedure is divided in three phases: the algorithm execution, user refinement and the combination of matching results. The matching algorithms calculate a similarity measure (from 0 to 1) between schema elements. The users choose different algorithms that are independently executed. There is a library of matching algorithms, such as Affix (matching of common suffixes and prefixes), Name match (considering names similarities), or a structural match based on the nodes' neighborhood. The similarity results are computed as an average of the similarity values of each algorithm.

This solution provides a graphical user interface to help with phase two. The user can configure and combine different heuristics. The similarity results are combined based on the element IDs. The algorithms are part of a predefined library. (in the remaining of this thesis we will use interchangeably COMA or COMA++ to refer to the latest version of the prototype: COMA++).

- **ONION**

ONION [106] provides a formalism to support an ontology integration framework. It focuses on defining a formal platform, not a prototype. The input models are ontologies. The ontologies are represented as a directed labeled graph G = (N;E) where N is a set of labeled nodes and E is a set of labeled edges. The mappings produced are articulation ontologies that contain the rules representing the relation between the source ontologies.

The articulation ontologies are first-class entities. The triple of two source ontologies plus one articulation ontology forms a unified ontology. Since there is not a merged ontology, a query engine executes requests over the unified ontology to obtain all the source ontologies.

ONION defines operators for ontology interoperation: *Union* generates a new graph of a unified ontology; *Intersection* generates the articulation rules relating the common concepts from both ontologies; *Diff* returns the elements not expressed in an articulation rule.

- **MAFRA**

MAFRA [93] is a framework that produces mappings to relate ontologies in a distributed environment. In this work, the ontology mapping process is the set of activities required to transform instances of a source ontology into instances of a target ontology.

The ontologies are represented in RDF schemas. Different ontology formats are translated into this common representation. The mappings are ontologies called semantic bridges ontologies (SBO). Semantic bridges consider three basic types to relate: Concepts, Relations and Attribute. There is one different bridge concept for each different type of concept, for instance *ConceptBridge* or *RelationBridge*. MAFRA uses a multi-strategy process to calculate similarities between ontologies. The strategies are combination of lexical and structural methods.

This work provides an ontology mapping framework with a separation between every activity to create the ontology bridges. The framework is separated in two dimensions: the horizontal dimension contain components handling the creation and execution of semantic bridges. The vertical dimension handles the evolution and creation of the bridging ontologies in a distributed environment.

- **S-Match**

S-Match [64] [65] is an ontology matching system. It implements a complete matching tool with the same goal as CUPID and COMA/COMA++, i.e., to provide a generic matching framework. It has as input two or more ontologies that are translated into an internal logical representation. The models are translated into propositional formulas. This approach has two main distinctions from other solutions. First, the mappings have more complex logical relationships, such as *equivalence*, *more general*, *less general*, *disjointment*, *equivalence*.

Second, the matching process is not heuristic. The matching process is translated into a propositional unsatisfiability problem. It analyses the propositional formulas of the input models and it produces unique logical mappings as output.

- **An API for ontology alignment**

The work from [56] defines an API for ontology alignment with an implementation in Java. It differs from previous approaches because the main goal is not to provide a generic prototype, or matching algorithms, but to factor out common aspects of existing approaches into a generic API. It is necessary to implement a set of pre-defined interfaces. This solution also specifies interfaces for comparison of existing approaches. For that reason, the input models have a fixed format.

The input models are RDF graphs. The mappings are represented at different abstraction levels. In level zero the mappings are distant from any implementation platform, having only simple equivalence relationships between the model elements. They are specialized into level one, where calculations expressions are added, but still language-independent. In the last level the associations are refined in expressions on a particular language, such as XSLT or C-OWL and RDF. This last translation is typically classified as a query discovery task.

- **iMAP**

iMAP [38] is a matching and query discovery tool. iMAP creates mappings between relational schemas. The schemas are translated into an internal graph representation. The tool combines different matching techniques to find simple mappings between two database schemas. In addition, the tool also provides machine learning techniques to find complex kinds of mapping expressions; for instance *name = concatenation (firstname, lastname)*. It reuses the machine learning techniques from LSD [47] to exploit a corpus of existing schemas and the instances of these schemas.

iMAP also presents new techniques to create a set of specific complex mappings, for instance the concatenation of string elements. The creation of these complex expressions is often considered as part of the matching phase. However, due to the complex nature of the expressions generated, iMAP could be classified as a query discovery solution.

- **Xu et al.**

The work from Xu et al. [54] [131] presents a composite matching approach that supports simple instance level matchers, plus the discovery of more complex expressions. It uses an internal format of conceptual graphs to represent XML schemas. The mappings are created based on similarity estimations. The similarities are calculated using a set of different algorithms, called matchers, such as *Merge/SplitValues*, *Superset* and *ObjectSet*. This approach has evolved in [131] to use the knowledge of domain-specific ontologies to compare instances of the source and the target elements and to find more complex expressions, such as *address = concatenation (street, city, state)*. The input schemas are compared with the domain ontologies, and if the structure is similar, a mapping is created.

3.4.3.2 Comparison of adaptive matching solutions

In this section, we compare the matching solutions considering the genericity and the adaptability of the approaches, because adaptability and genericity are the major issues studied in this thesis. For that reason, we do not present here every existing matching technique or approach, for instance [32] [104] [90] [47] [118] [16] [17] [20] [24] [82] [96] [136] [105] [110] [69].

Table 3.1 summarizes the approaches that have been presented. This table is built based on the classification of [127] and [45]. We explain each item below.

- Input: concerns the type of input models used by the solution. The type of the input models is usually closely related to the application domain. For instance, database schemas, ontologies, XSD schemas, labelled graphs, or others. The tools typically support different input formats, which are translated into an internal format.
- Matching techniques: concerns the techniques euristics used to create the mappings. For instance, the comparison of every pair of elements (element-to-element) or the use of structural information. So far, most solutions focus on developing different heuristics or on combining a set of heuristics.
- Mapping nature: concerns the kind of the mappings produced. For instance, 1:1 value correspondences or ontology bridges.
- Application scenario: depicts the target application scenario of the solution and where the result mapping is used; for instance, for data interoperability, merging or ontology integration.

	Input (internal and external representation)	**Matching techniques**	**Mapping nature**	**Application scenario**
CUPID	DB and XML schemas (rooted graphs)	Structural and linguistic	1:1 correspondences	Generic matching tool
GLUE	Unified ontology (rooted graph)	Data instances, probability distribution	1:1 mappings	Generic matching tool
PROMPT	Ontologies (general knowledge model)	Set of iterative operations	None: merges the ontologies	Ontology merging and alignment
COMA / COMA++	SQL, XML and OWL schemas (rooted directed graphs)	Library of heuristics	Equivalence 1:1 correspondences	Generic matching tool
ONION	Ontologies (directed graphs)	Interoperation operators	Articulation ontologies	Ontology integration
MAFRA	RDF schemas	Multi strategies (lexical and structural)	Semantic bridging ontology (SBO)	Alignment of distributed ontologies
S-Match	Ontologies (propositional formulas)	Propositional unsatisfiability problem	Logical relations	Generic framework
API for ontology alignment	RDF graphs	Provides an API	Simple 1:1 correspondences translated into XSLT, C-OWL,RDF	Generic ontology matching
iMAP	Database schemas (graphs)	Different machine learn searchers. Use of domain knowledge	1:1 mappings and complex functions	Data integration
Xu et al.	Database schemas	Different matchers and domain ontologies	Complex mapping expressions	Data integration

Table 3.1 Summary of the matching tools

Most of the solutions need to translate different kinds of models into an internal format. The most common internal format is some variation of directed labeled and rooted graphs, such as in CUPID, GLUE, iMAP, or PROMPT. This enables the creation of generic solutions that can be used in different application scenarios. An essential requirement is to create different application-oriented import methods. However, such generic solutions may have difficulties to in taking advantage of specific relationships between models to produce performing matching procedures; for instance foreign keys or nested relationships. Ontology-based approaches such as MAFRA, the API from Euzenat and Xu et al. have some kind of ontology as an internal or external format. This proximity between internal and external representations eases the construction of the import procedures. On the other side, it is more difficult to use these solutions for different application scenarios.

Concerning the matching techniques, CUPID, GLUE, COMA++, MAFRA, iMAP, Xu et al. propose the combination of a set of element to element and structural techniques. This is the most common approach used in current matching solutions. iMAP, COMA, Xu et al. also use taxonomies, or corpus of schemas as extra input parameters. This enables the creation of more accurate matching techniques. However, the size of the extra input may affect the overall performance. Two different approaches are S-Match and the API of Euzenat. S-Match does not use similarity estimations as all the other solutions. S-Match reduces the matching problem into a propositional unsatisfiability problem.

The number of correct mappings found is usually smaller, but they are considered to be correct. Euzenat proposes a standard API to implement matching techniques. This API could be used as an implementation base for the existing platforms, increasing the reusability of existing solutions.

Most part of approaches produces essentially mappings with 1:1 cardinality, such as CUPID, COMA, or GLUE. This means a mapping relates one source element with one target element. The set of all mappings have n:m cardinality. In COMA, a new mapping is generated for every input element that relates with more than one output element. MAFRA and ONION produce ontology bridges. These bridges can express more complex mappings. However, the matching heuristics find 1-to-1 mappings. The exceptions are the work from S-Match, because the mappings are propositional formulas, and iMAP and Xu et al. The two later solutions propose the creation of complex mapping expressions, based on extra input data.

These solutions are generic matching solutions, i.e., that present generic mechanisms to match different models. The solutions that concentrate on specific application domains, such as PROMPT (merging), iMAP and Xu et al. (data integration) provide more performing procedures for their specific cases.

The adaptability and extensibility are very important features when the solutions are intended to be used in different application scenarios. The remainder of this section presents a comparison of existing prototypes with respect to their adaptability and extensibility. Thus, we only compare the tools that provide adaptability facilities. The comparison criteria are explained below (see Table 3.2).

- Graphical user interface (GUI): indicates if the solution provides a graphical interface to create mappings.
- Extensibility: indicates if different mapping techniques can be easily plugged.
- Customization: concerns how the techniques are reused or modified, and if the tools can handle different kinds of mappings.

	Graphical interface	**Extensibility**	**Customization**
CUPID	No	-	Combination of pre-defined heuristics. Import of different kinds of schemas. Fixed mapping representation
GLUE	No	Plugging of new learners.	Combination of heuristics. Import of different kinds of schemas. Fixed mapping representation
COMA / COMA++	Yes	Implementation of new matchers in Java	Reuse by composition. Import of different schemas. Fixed mapping representation
MAFRA	Yes	-	Combination of existing methods. Fixed mapping representation
API for ontology alignment	No	Plugging of new classes extending the API	New methods should be compatible with the API. Fixed mapping representation
iMAP	No	Plugging of searchers and learners	Implementation of different searchers. Fixed mapping representation

Table 3.2 Adaptability of tools

Among the adaptive solutions, only COMA++ and MAFRA provide simple graphical user interfaces. S-Match and PROMPT also provide a user interface for matching, but they cannot be extended or adapted with different matching techniques. The graphical interface is a central component in matching tools. The GUI handles the user interactions needed to create mappings. MAFRA, despite implementing a user interface, does not provide extension mechanisms to plug different techniques, only to combine the ones that are delivered with the tool. GLUE, COMA++ and iMAP enable to plug different matching techniques implemented in Java. The solution of Euzenat is

also extensible, because new techniques can be encapsulated in classes that implement a set of interfaces, or can extend existing classes of the API provided. However, new methods are usually implemented in Java, or in some other programming language. The distance between the conceptual definition (graph models) and the implementation (Java objects) is too big. The developers must implement from scratch most of the navigating structures, being difficult to reuse or to adapt previous implementations.

The lack of explicit extension mechanisms usually limits users to combine or to customize existing methods. All the tools of Table 3.2 enable the combination of techniques, either using the GUI (COMA++ or MAFRA), or by calling specific methods (CUPID, iMAP and the API from Euzenat). However, the combinations usually follow a strict set of parameters. The creation of new parameters and the customization of existing techniques are not natively supported.

Finally, all the tools are implemented following a fixed mapping representation, which enables developing highly tuned and performing heuristics. However, a fixed mapping representation narrows the application scenarios of the matching prototypes, for instance, for ontology alignment scenarios (MAFRA) or for schema matching scenarios (COMA++). This constraint may be very limiting in mind the multitude of application scenarios of mappings (annotations, data interoperability, model difference, merge, etc.).

3.4.3.3 Query discovery

In this section, we present different query discovery approaches. Query discovery is the data interoperability task of finding complex expressions in specific transformation languages. The expressions are created based on mappings produced by a *Match* operation. However, the query discovery task is not always completely dissociated from the match operation. Most part of the query discovery solutions also presents matching facilities, which makes it difficult to classify them. For example, the complex expressions created in [38] may be language oriented, in a way this solution could be also classified as query discovery approach. We describe a set of most relevant solutions below.

- **Clio**

Clio [102] [103] is a generic matching and query discovery tool. Clio provides a graphical user interface where different matching techniques can be plugged. However, the main contribution of this solution is the production of complex transformations based on value correspondences.

Clio presents a semi-automatic query discovery algorithm that produces SQL views: the algorithm first groups all the value correspondences into sets. The sets are formed based on the possibility of creating joins between the schema elements. From these different sets, it selects the ones that will generate the smaller queries. The queries are generated by a union between all joined elements.

The work from [119] is an improvement of previous versions of Clio. The models are represented in an internal nested model. It translates DTDs and XML schemas into this nested model. This solution uses a simplified form of value correspondences, called element correspondences. Element correspondences are attribute to attribute mappings, without the possibility of adding any source-to-target functions.

This simple correspondence format facilitates the generation of queries, since it generates only equality expressions. The algorithm has an intermediate level of representation between the element correspondences and the generated queries. This new representation is called logical mappings. Logical mappings are a platform-independent representation used to “understand” the mappings.

The logical mappings are the basis for generating the set of queries. The developer chooses some queries from this subset to create a final query, called an operational mapping.

Clio has a recent evolution described in [60]. This work uses the same nested representation for models. However, it provides nested mapping representations as well. The authors claim that nested mappings enable the representation of more complex relationships in a simpler way.

- **Kedad et al.**

Kedad et al. [81] presents a query discovery approach. Similar to Clio, this solution produces operational mappings based on a set of element correspondences. The input models are XML Schemas. The element correspondences are used to generate XQuery's. This work focuses on finding mappings between XML schemas, and not relational schemas as in Clio, or in the work presented in [80].

The algorithm decomposes a XML schema into smaller parts. First, it interprets the relationships between indivisible parts. Indivisible parts do not have any child element. These elements are used to create equality expressions. Second, the algorithm searches for navigation expressions using the composite parts. The goal of this solution is to provide a performing process to generate XQuery, and not to implement a complete matching and query discovery solution as Clio.

- **An et al.**

An et al. [5] [6] [7] describe a solution that discovers semantic mapping expressions between different schemas. It differs from Clio and Kedad et al because it does not rely on referential integrity constraints or nested relationships.

This solution proposes deriving declarative mappings from a set of simple correspondences. The correspondences are relationships between columns of different tables. A conceptual model (CM) is created and associated with each input schema. This conceptual model contains additional semantic information about the intra schema relationships. The CM is represented by a CM graph. The key phase in this work is the creation of the conceptual models. The CM contains enough information to create relational mappings between two input schemas.

- **SMART**

SMART [107] is a prototype that produces data transformations between different data sources. This solution uses XML schemas as input, and produces XSLT as output. SMART proposes the incorporation of reverse engineering approaches to find data transformations. The main contribution is the production of conceptual schemas based on the input XML schemas. The conceptual schemas are richer than XML, for instance by adding hierarchy information between classes. The conceptual schemas can be extracted by some automatic procedure, or manually created by the application developers. According to the authors, the conceptual schemas enable finding more accurate mappings. The mappings are 1:1 value correspondences with inclusion labels (e.g., subset of). These mappings are derived into XML transformations between the original XML schemas. If the system cannot produce an output transformation, it asks for the user for additional mappings, as part of an interactive process.

3.4.3.4 Comparison of query discovery prototypes

The number of query discovery approaches is much smaller than matching approaches. However, these solutions have gained more interest recently (for instance, through the work of An et al.). We compare four solutions of query discovery below (see Table 3.3). We focus on three main aspects for

comparison: the input models; the nature of the input mappings and the kind of transformations that are generated.

	Input	**Mapping nature**	**Transformations**
Clio	A pair of relational and XML nested schemas (internal nested format)	1:1/n:m value correspondences	Produces logical operational mappings that are translated in SQL or XSLT
Kedad et al.	Two or more XML schemas	1:1 value correspondences	XQuery
An et al.	Relational schemas plus a conceptual model	1:1 value correspondences and the mappings between a schema and its conceptual model	Relational mappings
SMART	XML schemas and conceptual schemas	1:1 value correspondences with inclusion labels	XML transformations

Table 3.3 Query discovery approaches

Clio is one of the solutions most used in comparisons. Clio translates XML and database schemas into an internal nested format. Kedad et al. focuses on using XML schemas as input. An et al. and SMART have additional models (conceptual model and conceptual schemas) that enable the decoration of the input schemas (relational or XML) with richer semantic information. The algorithms are implemented based on the decoration models, which may ease the task of query discovery. All solutions first create value correspondences between the input schemas. An et al. also needs to map the input schemas with the conceptual model. This is not necessary in SMART because the conceptual model is considered as an extension of the input schemas. Clio is the only solution that provides details about its graphical interface. The interface enables to plug-in different matching algorithms.

Clio proposes a list with SQL and XSLT transformations. The list is quite extensive. Kedad et al. and An et al. already produce the final mappings. This may cause problems if the transformations are not correctly created. The user must correct them by hand, while in Clio it is more probable to find a correct transformation from a list. SMART differs from Clio because it does not creates a list, but asks the user for additional input information if it finds more than one possible transformation, or if the transformation is not found. This is an advantage if the set of transformations is too extensive. From the best of our knowledge, none of the existing approaches proposes any generalization of the query discovery process. The algorithms are specific to the kind of input schemas.

3.4.4 Model management

Model management is a relatively recent research field in metadata management, introduced in [19]. The two key concepts in model management are models and mappings. Model management aims to factor out common metadata tasks into a set of generic operations over the models and mappings. Model management operations can be applied to a variety of metadata applications. Frequently used data interoperability operations can be defined as a subset of operations in a model management platform.

The work in [18] describes a set of model management operations, with different levels of complexity. We cite some operations below.

- *Match*: creates mappings between two models.
- *Diff*: returns the difference between two models.
- *Copy:* creates a new model copying all the elements of an input model.
- *Merge:* merges two models given an input mapping.

- *Compose:* composes two mappings.
- *Invert*: inverts a mapping.

Several metadata management approaches can be considered as applications of model management. The existing solutions are separated into two categories. First, generic solutions that propose complete model management platforms. Second, solutions that focuses on specific application scenarios and model management operations.

In this section, we present solutions that are generic and that focus on the *ModelGen* operation. *ModelGen* is very important in data interoperability solutions, because *ModelGen* enables the translation of a model represented in one language into a model represented in another language, for example RDBMS to XML, or RDBMS to OWL. This is a preliminary step before being capable of translating data between different sources. Consequently, this is a very important issue in data interoperability scenarios. Other model management approaches focusing on other operations are briefly introduced in Section 3.5.

- **Rondo**

Rondo [99] is the first generic prototype of model management. It implements the model management operations introduced in [18]. Rondo has a script language used to execute a sequence of model management operations. The scripts can be applied in change propagation scenarios.

Rondo translates different model formats into its internal format. The models of Rondo are directed labeled graphs. The nodes of the graphs are model elements. The model elements are uniquely identified. A directed labeled graph is a set of edges <s, p, o> where s is the source node, p is the edge label, and o is the target node.

The mappings are simple binary relationships, called morphisms. A morphism establishes 1:1 correspondences between the elements of two models. Morphisms are syntactic structures, i.e., without semantics. Morphisms are used by model management operations to produce other models or morphisms. Some operations over morphisms are very simple, such as *Id*, or *Apply*. *Merge* and *Compose* are more complex; thus studied separately. The execution of these operations as scripts enables the management and maintenance of mappings and models.

One of the operations studied in more detail in Rondo is the *Match* operation. This solution presents a generic matching algorithm named Similarity Flooding [101]. Similarity flooding is a structural matching algorithm that propagates the similarity between neighbor nodes. For instance, if two database tables have a high similarity value and thus match, it is probable that the containing columns also match. The algorithm takes advantage of these relationships based on the name of the graph labels, and it propagates the similarity through the model elements.

- **Moda**

Moda [100] is an evolution of Rondo that provides semantics to the mappings. Moda produces transformations that translate instances of one model into instances of another model. Models are defined as sets of instances. A model can be described by an expression in a concrete language, such as SQL DDL, or XML Schema.

Morphisms are extended to path-morphisms. A path-morphism is a relation on instances. Let *map* be a morphism connecting tree schemas *m1* and *m2*. If *map* connects each tree of one schema to at most one tree of the other schema and *map* connects the root keys of every pair of connected trees, then *map* is a path-morphism.

However, later in this work the authors state that path-morphisms cannot handle rich mapping semantics. Moda is modified to support more expressive mappings. The mappings are logical

dependencies between relational schemas. These logical dependencies are formulas that can be translated into executable transformations, differently from morphisms or path-morphisms.

The models are not a generic graph representation as in Rondo, but concrete relational schemas. This enables the creation of more complex mappings as well. The mappings are used as input to other model management operations, and also to produce model transformations into a particular language (XSLT). However, this solution does not provide the implementation of all the model management operations of Rondo. This is essentially because the mappings are more complex. Consequently, the operations are more complex, and need to be studied more deeply.

- **MIDST**

MIDST [11] [12] is a tool that implements *ModelGen* operations. MIDST is based on the notion of a unique multi-level dictionary [10]. This dictionary is used to represent both metamodels and models. This dictionary is based on the Hull et al observation that all models have similar kind of constructs (cf. section 4.2). MIDST takes advantage of the generic dictionary to create at the same time schema and data translation procedures. The translation between different schema languages is defined using DataLog rules. Then, a *Down* procedure is used to generate the data translation rules. The *Down* procedure is generic only if both the schemas and the data are translated. It cannot be used to directly execute data translations.

- **GeRoMe**

GeRoMe [83] proposes a generic metamodel (GMM) to define models with more complex kinds of relationships between them. The GMM is extended by the addition of *decorations*. These decorations are specific roles over the model elements. A same model can be decorated with different roles, such as *Aggregation* or *Association*. The model management operations check if a certain model is decorated with a set of roles. The operations are executed only if the model is decorated with the correct roles.

Every model element must have at least one role associated to it. This work defines roles for different metamodels, such as OWL, relational model, or entity relationship model, and explains how the standard elements of these metamodels are decorated. The specification of the role metamodel is a very important phase, since it must contain every possible role that could be decorated.

This work explains how to execute *ModelGen* operations to translate between two models decorated with the generic roles. It presents a set of smaller operations to translate between different roles, for example, the transformation of *IsA* relationships. The authors also claim that the same approach may be used to implement other model management operations. However, there is no experimental evidence of such an implementation.

- **MOMENT**

MOMENT [25] proposes a model management platform that implements model merging and *ModelGen* operations for MOF metamodels. The *ModelGen* operations are specified between a UML model and a RDBMS model using QVT relations. The operation translates one model into another and generates a traceability model with the execution trace of the translation operation. The merge operation is defined using QVT relations (the correspondences between the model elements) and QVT mappings (to apply the merge operation). The merge can be defined between models conforming to the same metamodel, or to models conforming to different metamodels.

3.4.4.1 Comparison of model management prototypes

Rondo is the unique solution that provides an implementation of a large set of the model management operations described in [18]. Moda, GeRoMe and MOMENT are also generic platforms. This means several operations could also be implemented, event if the solutions focus on specific operations, i.e., *ModelGen*. The simple morphisms used in Rondo enable the development of several operations in a relatively simple manner. For instance, an invert over a morphism *(a, b)* is implemented only by changing the domain and the range of this morphism, i.e., *(b, a)*. Consequently, Rondo is limited when it is necessary to implement complex data interoperability operations, such as the translation between two different schemas.

The limited expressiveness of mappings is one of the major drawbacks of model management solutions. Moda identifies this issue and thus proposes more complex mappings. The mappings are logical dependencies between relational schemas. The mappings are used to produce transformations in particular transformation languages. Due to the rise in the complexity level, the authors focus on developing only data translation operations. This is also the case of MIDST and GeRoMe. MIDST is one of the first solutions that studied *ModelGen* operations. The unified representation for models and metamodels enables the creation, in a single step, of both schema and data translation operations. However, this is only possible if the schemas are not yet available, restricting the approach only to more specific scenarios.

GeRoMe can be used to implement different model management operations as well, but the authors also focus on *ModelGen*. Differently from MIDST, the *ModelGen* operation only handles schema translation. The generic metamodel annotates the input schemas with roles. The decorations allow working with the original schemas. However, the generic metamodel is very difficult to develop, and should be accepted and used by the user community. This restricts the possibility of having highly heterogeneous environments and of executing translations between them.

MOMENT provides a framework for model management, but similar to GeRoMe and MIDST, it focuses on the *ModelGen*, plus the *Merge* operation. MOMENT uses OMG standards (MOF and QVT) to implement the operations. The main difference from other approaches is the possibility of defining operations between models conforming to different metamodels. This is a new requirement that has emerged with the development of MDE solutions and the production of different kinds of models.

3.4.5 MDE tools for interoperability

After having presented different mapping-based data interoperability solutions, in this section we introduce the key solutions for interoperability in MDE platforms. MDE platforms have many different kinds of models, conforming to different metamodels. It is very important to provide ways to interoperate between these different models. In general MDE platforms support interoperability through model transformations, i.e., by transforming a set of input models into a set of output models. This is one main reason why model transformations are one of the most important kinds of models in MDE.

There are several model transformation approaches. In this section, we briefly introduce a set of the most representative model transformation solutions. We do not intend to compare in detail every solution, for instance comparing their syntax, cardinality, or execution semantics. Comparisons of such solutions are found at [35] and [73]. We give an overview of what they do and their target application domain.

- **QVT**

Query/Views/Transformation [114] is a specification of OMG for a language capable of expressing queries, views and transformations over models. Several proposals have been submitted since 2001, since the final recommendation has been adopted in 2005.

QVT is divided in three sublanguages. These three languages together make the QVT recommendation a hybrid transformation language, i.e., it contains imperative and declarative constructs.

- QVT relation: defines transformations as a set of high-level relationships between models. QVT relation is a completely declarative language. QVT relations support traceability between the related models.
- QVT core: it is also declarative. It enables the definition of rules with more complex structures than QVT relations. The traceability between models is not automatically handled. Every transformation action should be explicitly defined when using QVT core. QVT core provides a basis to specify the semantics of QVT relations. Transformations written in QVT relations are transformed into QVT core.
- QVT operational mappings: it is not always possible to define a transformation using only declarative rules. Operational mappings extend QVT core and QVT relations with imperative constructs.

QVT also enables the invocation of primitives defined in different languages or platforms (e.g., a Java method) by providing a black box mechanism.

- **ATL**

ATL (Atlas Transformation Language) [73] is a solution for model transformation. Transformations are specified in transformation models. ATL has a hybrid language. These models can be specified using declarative and/or imperative rules. However, the recommended style by the authors is declarative. ATL is a simple language that enables to easily develop model transformations. ATL is a QVT-like language, i.e., it handles most of the QVT recommendations. However, ATL does not intend to be 100% compliant to QVT. Its main goal is to provide a simple and easy to use transformation language that can be used in the majority of cases.

ATL provides a set of formal MDE definitions that can be used as a base for other model transformation languages. It has an integrated development environment as a plug-in for the Eclipse platform [51] (debugger, execution engine, and graphical interface).

- **Graph transformations**

Graph transformations operate over graphs by applying graph rewriting rules. Graph transformations are executed in several steps. Each step executes a rule over an input graph. A rule defines an input and an output pattern. The rule searches for an occurrence of the input pattern in the input graph. This pattern is replaced by the output pattern defined in the rule. The rules can be executed according to different criteria, for instance priorities or some special control flow variable. Graph transformation approaches can be used to transform models as well, because models are usually represented as graphs. These languages often have a graphical representation. Among the existing solution, we cite VIATRA2 [133] [130], GReAT [79], AToM3 [132]or AGG [130].

3.4.5.1 Analysis

The analysis of model transformation solutions for data interoperability follows a set of practical criteria. The transformations must have a simple (though powerful) language, capable of expressing complex expressions in relatively simple ways. The transformation solution must follow well defined standards. It must be supported by a set of tools, such as debuggers, textual and/or graphical editors, exception handling facilities, and the possibility to reuse code.

Current transformation solutions have several similarities that satisfy many of these criteria. Most approaches have declarative rules. This enables the creation of simple transformations by specifying only what to do, and not how to do it. This is particularly important when developing complex data interoperability transformations. However, declarative rules are not always enough, or at least not practical, to define some complex kinds of transformations. Consequently, imperative rules are usually part of the transformation language, even if they are not the recommended development style. This is the case of ATL and QVT. Other approaches not introduced in this section, such as UMLX [138], Tefkat [88], C-SAW [67] or MOLA [78], follow similar paradigms.

The official industry standard for model transformations is QVT. However, QVT is relatively complex, which makes very difficult (and sometimes impractical) to develop tools that are fully QVT-compliant. This goes in the opposite direction of simplicity of design, which is often a very important factor to the establishment of standards. The preferred approach should be to develop a transformation platform based on simpler concepts. In this case, experiments can be rapidly developed, and the language and its environment can be evaluated rapidly. This is the case of ATL, which is becoming an important model transformation solution. ATL has been promoted from research prototype in the Eclipse GMT [66] subproject, into a standard component in the Eclipse M2M project [92].

A very important feature when choosing a transformation solution is the tool support. A transformation language should have an editor, debugger, compilation facilities, and the possibility to interoperate with other platforms. Most solutions are being developed as Eclipse plug-ins. This allows a solid base workbench, where new features can be included in a relatively easy way. Tool support and interoperability with other solutions are the main drawbacks of graph-based solutions.

3.5 Other application scenarios of mappings

Data interoperability is one of the most common application scenarios for mappings; however, there are several other possible application scenarios. Although we focus on data interoperability solutions, in this section we give an overview of other utilizations for mappings. We do not compare them one by one, because they focus on heterogeneous application scenarios.

3.5.1 Merge

Merging of different models is a typical application of schema integration. Considering the merge problem as a generic model management operation, we informally define merge as an operation that takes two models as input, a mapping between them, and that produces a new output model that contains all the elements of the input models, but no additional information. We detail a set of merge approaches in the following.

- **Buneman et al**

The work from [29] defines a set of theoretical aspects of schema merging and proposes a generic merging algorithm. The merge problem is defined as follows: "*given schemas A and B, and a mapping Mab, produce a schema C*". The created schema has information preservation constraints. This means

C presents all the information of the schemas being merged, but no additional information. Schemas are represented in a special type of graph, containing attributes and inheritance edges.

This work shows that a merge operation does not always produce a valid schema conforming to its specification. To solve this problem, the merge algorithm first produces a “weak” schema, which contains extra implicit classes that cannot always be represented in the target schema. These extra classes are further translated into a format compatible with the target schema.

- **Generic *Merge* operation**

The work from [120] provides a formal specification of a *Merge* operation for a model management platform. The operation takes as input two models and a mapping between them. The operation produces a model as output that contains a duplicate-free union of the two input models, with respect to the input mapping.

The *Merge* operation assumes that a mapping between the input models is previously created by a *Match* operation. The mappings are a triple containing a pair of morphisms and an auxiliary model. This auxiliary model contains more complex structures, such as equivalence relationships. This solution defines a representation format for models in three meta levels: models, metamodels, and metametamodels. Informally, a model can be a database schema definition; a metamodel consists of the type definitions of the objects of the models, for instance the tables and columns; a metametamodel, which is the representation language in which models and metamodels are expressed.

- **EML**

Epsilon Merge Language (EML) [85] is a rule-based language to merge different models. EML is built on top of Epsilon [84], which is a framework for developing specific model management tasks, such as model merging or model transformation. Epsilon provides uniform access to different kinds of models. It has a set of components that are reused to implement new task-specific languages.

EML is divided in four phases: matching, conformance, merge and restructuring. The match and conformance phases produce mappings between the two models that are going to be merged. This architecture follows the one presented in [15]. The merge phase uses these mappings as input to merge two or more models. The restructuring phase defines how to reorganize the merged models based on a set of constraints. This is because the result of a merge is not always the desired model, as shown in [29].

One on the main features of EML is the possibility to create built-in features that abstract some common merging tasks, for example simple matching strategies (*MofIdMatchingStrategy* and *EmfIdMatchingStrategy)*. The objective of these strategies is to relieve developers of defining trivial rules that are frequently executed. The strategies can be extended with extra functionalities.

- **Merging of statecharts**

The work from [109] presents an approach for matching and merging state charts. A state chart is a design and implementation language that is widely used for specifying dynamic behaviors of software systems [68]. A state chart is formed basically by a set of states and transitions between these states. This solution is part of a broader approach to generic model management [27], which presents a formal description of a set of model management operations, such as *match*, *merge*, *slice*, or *diff*.

The main contribution of this work is the matching and merging heuristics that take into account the behavior of the state charts. The behavior is specified in the transitions. A transition may have a firing event, and may execute some actions. The matching heuristic analyzes the properties of every

pair of transitions and calculates a *behavioral similarity value*, to obtain more accurate mappings between the states of two input state charts. These mappings are used as input to the merge algorithm.

3.5.2 Traceability

Traceability information is used for several reasons: for data provenance [134], requirements traceability [123], and traceability of model transformations [72]. In this section, we present three different application scenarios that use mappings to handle traceability between different models.

- **Data provenance**

In [134], data provenance is the problem of discovering the origin of data after it was transformed from a source schema into a target schema. This work proposes inserting traceability information in the transformed data to obtain the source schema.

The schemas and mappings are represented in the same nested representation of [135]. The target data is annotated with information about the schema elements and also the mappings that produced a given instance. The relation *elementOf* indicates that an instance conforms to a schema element.

The query language is an extension of SQL to fetch the annotations. They define an operator *@map* that returns the set of mappings that generated a given instance. The language also provides an operator *@elem* to verify if an instance data conforms to a schema element.

- **Requirements traceability**

Requirements traceability [123] keeps track of all the steps of a development process: analysis, design, programming, testing. Some possible kinds of links are *developed_by*, *allocated_to*, *performed*, *based_on*, *modify*. The key processes are the identification of the possible kinds of links and the development of new traceability reference models.

This solution defines several reference models of traceability. Each reference model is used to record traceability information in different abstraction levels. For instance, a decision maker uses more general provenance and traceability information, while a developer uses traceability information in a smaller and more specific context.

- **Traceability of model transformations**

Similar to data provenance scenarios, it is often necessary to store the execution trace of model transformations. The execution trace of a transformation indicates, for a set of generated elements, which transformation rules are executed, and which input elements are used. This traceability information is recorded in specific traceability mappings. The work from [72] proposes to save the execution trace of ATL transformation in domain-specific traceability models. These models depict traceability relationships between a set of input model elements, a set of output model elements, and a set of transformation rules. The stored traceability information allows having

3.5.3 Mapping composition

Mapping composition is the problem of creating a new mapping given two mappings as input. Two application scenarios are optimization in peer-to-peer systems and mapping reusability. Reusability of mappings avoids repeating the matching phase, which is considered a very difficult problem (AI-

complete). This is not a typical data interoperability operation, but it is very important to support the reuse and evolution of mappings. We present three approaches below.

- **Model management composition**

The work from [18] presents a set of model management operators, and composition is one of them. It is a generic definition used to implement composition operators in a generic model management platform. The composition operation creates a mapping by combining two other mappings.

Models are a set of objects with properties, has-as relationships, and associations. The models are identified by a root object. The mappings are defined using morphisms, as in [99].

- **Composing semantic mappings**

The work from [97] defines mapping composition based on the answer obtained by a query that is executed over the mapped models. The models are represented in the relational data model. The queries are conjunctive queries (select, join, where), but only equality joins are allowed.

- **Composing mappings given embedded dependencies**

The composition definition from [108] takes into account the relationships between the models and mappings. Mappings are GLAV formulas used to associate the models. It defines mappings as *a schema mapping describes the relationship between the data instances of two schemas*. Mapping composition refers to combining two mappings into a single one.

3.6 Conclusions

In this chapter, we have presented the state of the art about mappings used in different application scenarios, and especially in data interoperability problems. We have seen that the mapping-based approach may be considered a generic solution that can be applied to the two basic topologies of data interoperability, i.e., centralized and distributed.

Due to the large number of formats of models and mappings, we have concluded that it is very hard to develop a generic platform that can be used in several distinct application scenarios. Typically, the format of the input models is determined by the choice of the mapping format, as well as in the implementation of tools and algorithms for mapping management. The development of such platforms is a trade-off between simplicity and expressiveness. Simpler mapping formats (e.g., element correspondences or equivalence relations), allow implementing simple platforms based on a set of well-defined concepts. However, this simplicity restricts the possible application scenarios. On the other side, general-purpose approaches based on rich mappings formats can be used in a larger number of use cases. However, it increases the complexity of development of all phases of mapping management. A more generic solution would be the definition of a simple platform based on strong extensible mechanisms. This allows the customization of each solution based on different requirements and degrees of complexity.

We have seen that the creation of mappings and the production of data interoperability operations are usually divided in two phases, matching and query discovery. The majority of existing solutions focus on the development of matching techniques, using different algorithms or heuristics. More recent solutions combine different matching techniques, enabling a better customization. This is a very important feature, because the user should be able to choose between a set of techniques adapted to

different situations. We have provided a comparison based on generic aspects and on the adaptability of each platform. Based on that comparison, we believe it is still difficult to modify and to integrate methods developed by one solution into another solution. Concerning the query discovery solutions, they are all targeted to generate some specific kind of mappings. None of the proposals provide a generalization of this process.

Model management is a more recent approach based on the principle of reutilization, because their objective is to define generic operations for frequently executed metadata tasks. This approach can be applied in different domains. Regarding data interoperability, there is almost a consensus about the isolation of the matching process in a *Match* operation. This is becoming true also to *ModelGen* operations; however, the number of solutions is still small. In contrast, to the best of our knowledge, there is no proposition for a model management operation for producing transformations.

We have also presented a set of MDE model transformation solutions. These approaches have concepts similar with the model management solutions. However, they do not focus on developing specific operations, but on defining general purpose transformation languages that enables to achieve data interoperability. These MDE approaches provide practical answers to many issues, such as language implementation and tool support. The development of such platforms enables creating model management operations as well.

Mappings can be used in applications other than data interoperability. For that reason, we have given a brief overview of some other applications for mappings. Mappings can be used, for instance, to support data provenance and traceability, to merge models, and others. Each one of these applications could be studied in more detail. However, this is out of the scope of this thesis.

Finally, based on the solutions presented in this chapter, we conclude that one of the major challenges not yet achieved is to provide a generic solution that can be easily adapted and extended, for instance, by plugging different heuristics or query discovery methods. This solution could be used in different application scenarios. In the remainder of this thesis, we present a solution that makes progress in that direction.

4 Model weaving[1]

4.1 Introduction

In this chapter we present the base concepts and solutions of this thesis. We build our solution on top of the AMMA (Atlas Model Management Architecture) platform [22]. The AMMA platform is a Model Driven Engineering (MDE) platform based on precise definitions of models and on the operations between these models. AMMA allows the specification of domain-specific languages (DSLs) to manipulate different kinds of models. So far, the AMMA platform has focused on the construction of such DSLs and in the study of model transformations.

We contribute to AMMA by providing the basis for generic relationship (i.e., mapping) management. Our approach is called *model weaving*, which uses weaving models to capture the relationships between different model elements. Model weaving covers the different aspects of relationship management: *representation*, *computation* and *utilization*[2]. A weaving model conforms to a weaving metamodel. We follow the main idea of AMMA, which is the creation of smaller and well targeted metamodels. Thus, we create a core weaving metamodel that is extensible. The basic requirements for relationship management are supported by this core metamodel. The requirements are identified based on the feature diagrams that have been presented. The core weaving metamodel is extended with domain-specific kinds of links (a set of metamodel extensions and use cases of model weaving is presented in Chapter 7). The extensibility of weaving metamodels enables achieving a better trade-off between simplicity and expressive power of relationships.

In this chapter, we formally define weaving models and metamodels. We also describe an extension mechanism that enables the creation of different weaving metamodel extensions. In addition, we present an adaptive and extensible tool called AMW (Atlas Model Weaver) for manipulating weaving models and metamodels.

This chapter is organized as follows. Section 4.2 presents the base modeling concepts of our approach. Section 4.3 presents model weaving, with formal definitions for weaving models, metamodels and extensibility. Section 4.4 describes our adaptive prototype. Section 4.5 concludes.

4.2 Models

The basic assumption in MDE is to consider models as first-class entities. We abstract the implementation details and representation issues of different solutions by using a unifying modeling

[1] This chapter is an adapted version of [21] [23], [41] and [42].

[2] Note that model weaving presented here differs from the term used by aspect modeling community, when model weaving is typically the process of weaving an aspect model with a business model.

platform. This enables handling the generic relationship management problem in a uniform way. We present the definitions below (following [41] and [74]).

Definition 4.1 (Directed Multigraph). A directed labeled multi-graph $G = (N_G, E_G, \Gamma_G)$ consists of a finite set of nodes N_G and a finite set of edges E_G, a mapping function $\Gamma_G : E_G \rightarrow N_G \times N_G$.

Definition 4.2 (Model). A model $M = (G, \omega, \mu)$ is a triple where:

- $G = (N_G, E_G, \Gamma_G)$ is a directed multigraph,
- ω is itself a model associated to a multigraph $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$,
- $\mu : N_G \cup E_G \rightarrow N_\omega$ is a function associating elements (nodes and edges) of G to nodes of G_ω. The function μ associates every node and edge of G ($N_G \cup E_G$) with one element in ω (N_ω).

Definition 4.3 (Reference model).Given a model $M_1 = (G_1, \omega_1, \mu_1)$, and a model $M_2 = (G_2, \omega_2, \mu_2)$, if $\omega_1 = M_2$, M_2 is called the reference model of M_1.

Some models are their own reference model ($\omega = M$). This allows stopping the recursion introduced in this definition. The relation between a model and its reference model is called *conformance*.

Definition 4.4 (Conformance relation). The relation between a model and its reference model is called *conformance relation*.

The conformance relation is denoted by *conformsTo* (or *c2*). A reference model may be considered as the type of a given model, because it defines a set of constraints for creating the model. These definitions allow an indefinite number of levels. However, we observed from different domains (XML, RDBMS, ontologies) that typically only three levels are needed (cf. Chapter 2). We call these three levels metametamodel (M3), metamodel (M2) and terminal model (M1).

Definition 4.5 (Metametamodel). A metametamodel is a model that is its own reference model.

Definition 4.6 (Metamodel). A metamodel is a model such that its reference model is a metametamodel.

Definition 4.7 (Terminal model). A terminal model is a model such that its reference model is a metamodel.

The illustration in Figure 4.1 presents the relations between the different kinds of models and the modeling levels. The *conformsTo* arrow shows the conformance relation. A model conforms to only one reference model. A reference model may have several models conforming to it. The *isA* relation indicates that a terminal model is a model, and that metametamodels and metamodels are reference models (note that *isA* is not an inheritance relation in the object oriented sense).

The metametamodel is the base representation of all metamodels and terminal models of a given domain. The metametamodel expresses the set of basic concepts, such as model elements and the relationships between these elements. Consequently, the choice of the metametamodel is a very important issue when developing a generic MDE solution.

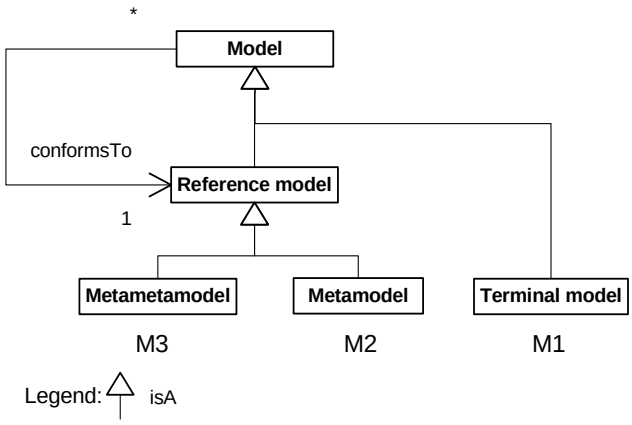


Figure 4.1 Modeling relations

4.2.1 Kernel MetaMetaModel (KM3)

The metametamodel (M3 level) chosen to develop our solution is KM3. A complete formal version of KM3 is available in [74].

```
package KM3 {
class Package extends ModelElement {
      reference contents[*] ordered container : ModelElement;
}
abstract class ModelElement {
      attribute name : String;
}
class Classifier extends ModelElement {
}
class Class extends Classifier {
      attribute isAbstract : Boolean;
      reference supertypes[*] : Class;
      reference structuralFeatures[*] ordered container :
            StructuralFeature oppositeOf owner;
}
class StructuralFeature extends TypedElement {
      reference owner : Class oppositeOf structuralFeatures;
}
class TypedElement extends ModelElement {
      attribute lower : Integer;
      attribute upper : Integer;
      attribute isOrdered : Boolean;
      attribute isUnique : Boolean;
      reference type : Classifier;
}
class Attribute extends StructuralFeature {
}
class Reference extends StructuralFeature {
      attribute isContainer : Boolean;
      reference opposite[0-1] : Reference;
}}
```

Figure 4.2 Self-definition of KM3

We choose KM3 for practical reasons. KM3 is self defined, thus the same set of primitives are used to manipulate KM3 metametamodels, metamodels and models. The KM3 metametamodel has constructs such as inheritance, composition, references and typed elements. All of them are not natively supported by relational schemas or XML documents; for instance, the inheritance between elements (the other practical alternatives). KM3 has a simple textual syntax that enables rapidly creating metamodels. This is particularly important in environments with frequent changes. We show a simplified version of KM3 in Figure 4.2. We used the KM3 syntax to define the KM3 metametamodel. This shows that it is self-defined.

A KM3 model has one or more *Package*. A *Package* is a container for *ModelElement* (with a *name* attribute), from which all other classes inherit. A *Class* is the base first class element, capturing the concepts of a model. A KM3 model is basically formed by a set of classes and the relations between them. A *Class* may inherit from other classes (*supertypes* reference). A *Class* contains structural features, which are *Attributes* or *References*. *Attributes* or *References* inherit from *TypedElement*, so they have cardinality (*lower* and *upper*) and a *type*. An *Attribute* contains values expressed in primitive data types or in other classes. A *Reference* creates relationships with other classes in the model. It may be a containment reference or association (*isContainer*). *Boolean*, *String* and *Integer* are primitive datatypes.

4.3 Model weaving

In this section, we present the base concepts of our approach for model weaving. Model weaving encompasses the different facets of relationship management. First, we define weaving metamodel and models. Weaving models are our solution to cope with the basic features of relationship management introduced in Chapter 2. Then, we define an extension mechanism to specify domain-specific weaving metamodels.

4.3.1 Definitions

We capture the relationships (i.e., links) between model elements in a weaving model. A weaving model conforms to a weaving metamodel. The weaving metamodel defines the kinds of links that may be created. In this section, we start defining weaving metamodel and model. Then we present a core weaving metamodel.

Definition 4.8 (Weaving metamodel). A weaving metamodel is a model $MM_W = (G_M, \omega_M, \mu_M)$, that defines link types, such that:

- $G_M = (N_M, E_M, \Gamma_M)$,
- $N_M = N_L \cup N_{LE} \cup N_O$, N_L denotes the *link types*; N_{LE} denotes the *link endpoint types* and N_O denote other auxiliary nodes,
- $\Gamma_M : E_M \rightarrow (N_L \times N_{LE}) \cup (N_O \times N_M)$, i.e., a *link type* refers to multiple *link endpoint types* and the auxiliary nodes refer to any kind of node.

Definition 4.9 (Weaving model). A weaving model is a model $M_W = (G_W, \omega_W, \mu_W)$, a graph $G_W = (N_W, E_W, \Gamma_W)$, such that its reference model is a weaving metamodel ($\omega_W = MM_W$).

A weaving model is a terminal model. In the remaining of this thesis, we will use only weaving models. A weaving model contains typed links that enable linking elements of different models. The elements of the weaving model are called *weaving elements*. The weaving elements that conform to the *link endpoint types* ($\mu_W (N_W) = N_{LE}$) are pointers to the elements of the linked models. To obtain the *real value* of the linked elements, the *link endpoints* are associated with a dereferencing function ρ.

Definition 4.10 (Dereferencing function). Given a weaving model $M_W = (G_W, MM_W, \mu_W)$, a graph $G_W = (N_W, E_W, \Gamma_W)$ and a linked model $M = (G, \omega, \mu)$, $G = (N_G, E_G, \Gamma_G)$, a dereferencing function ρ returns the elements of the linked model:

$\rho : N_{WL} \rightarrow N_{G,}\ N_{WL} \subset N_W$, such that $\mu_W(N_{WL}) = N_{LE}$.

We illustrate weaving metamodel and models using the mapping expression $t = s_1 + s_2 + s_3 + s_4 / 4$. The mapping language contains the addition and subtraction operators, plus the tokens (the model elements). The language does not explicitly specify that it is possible to create expressions that calculate the average between a number of elements. The semantics is only known if we analyze the expression itself. In our solution, we create a link type *average* that abstracts the semantics provided by the combination of operations "+" and "/". This process is the *promotion* of the mapping semantics into the weaving metamodel. The link type refers to a link endpoint with cardinality N (the source elements), and to a link endpoint with cardinality 1 (the target element). The mapping expression (the link between the elements) is created in a weaving model conforming to the weaving metamodel.

A set of linked models, and the weaving model between them is called a *weaving*.

Definition 4.11 (Weaving). A weaving is a tuple $<M_W, S_{WM}>$, where:

- $M_W = (G_W, \omega_W, \mu_W)$ is a weaving model,
- $S_{WM} = \{M_i = (G_i, \omega_i, \mu_i), i = [1..n]\}$ is a set of models linked by $M_{W.}$

4.3.1.1 Core weaving metamodel

We present a core weaving metamodel based on the previous definitions. The metamodel is illustrated in Figure 4.3. The core metamodel has elements with information about link types, link endpoints and element identifications. Element identification is a practical solution for saving unique identifiers for the linked elements. These values are used by dereferencing function to access the elements of the linked model elements.

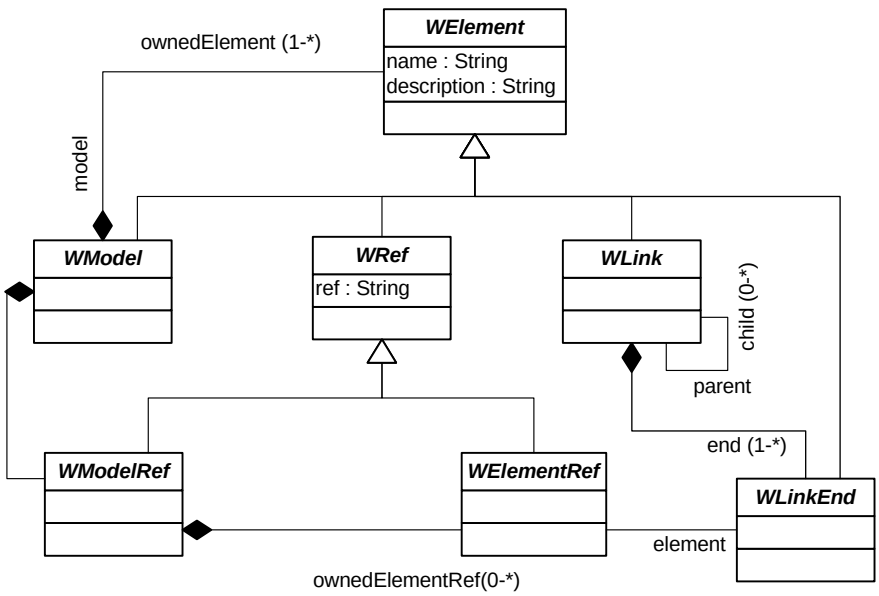


Figure 4.3 The core weaving metamodel

- *WElement* is the base element from which all other elements inherit. It has a name and a description.
- *WModel* represents the root element that contains all model elements. It is composed by the weaving elements and the references to woven models.
- *WLink* expresses a link between model elements, i.e., it has a simple linking semantics. To be able to express different link types and semantics, this element is extended by different metamodel elements (we explain how to add different link types in the following section).
- *WLinkEnd* defines the link endpoint types. Every link endpoint represents a linked model element. It allows creating N-ary links.
- *WElementRef* elements are associated with a dereferencing function. This function takes as parameter the value of the *ref* attribute and it returns the linked element. For practical reasons, we define a string attribute. There is also the inverse identification function that takes the linked element as parameter and that returns a unique identifier.

It is possible to associate the dereferencing/identification functions directly with the link endpoints. However, we create separated *WElementRef* because it enables referencing the same model element by several link endpoints.

This metamodel has only abstract types. We illustrate a simple weaving model in Figure 4.4, to show the minimal set of elements needed to link two model elements, i.e., one link, two link endpoints and two identification elements. This weaving model links the elements of *LeftMM* and *RightMM* metamodels. The weaving model contains one link (*WLink*); the link contains two endpoints (*WLinkEnd*), i.e., one refers to an element in *LeftMM* and the other to an element in *RightMM*. Each *WLinkEnd* refers to one *WElementRef*.

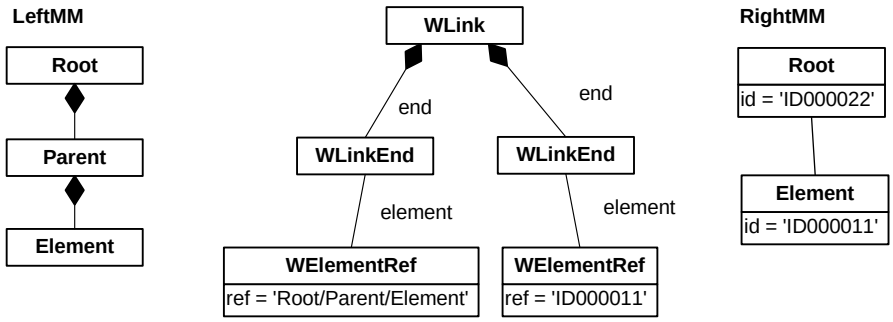


Figure 4.4 A simple weaving model

The left *WElementRef* element has the identification (ID) of *Element* from *LeftMM*. The ID is calculated taking the element name (*Element*) and the name of the parents (*Root/Parent*). The right *WElementRef* refers to *Element* from *RightMM*. The ID is a string that is automatically generated. In this example, the number of endpoints and linked elements are the same. Different link types and link endpoint types are added using metamodel extensions. The link element must be extended to create different link types, for example equality, equivalence, dependency, and so on.

4.3.2 Extension operation

A weaving metamodel must support different kinds of links. We define different kinds of links by extending the core weaving metamodel to form domain-specific weaving metamodels. This is done using the metamodel extension operation.

Definition 4.12 (Metamodel extension operation). The metamodel extension is an operation MM_R = Extend (MM_W, MM_E, M_{WD}), that takes metamodels MM_W, MM_E and the extension definition weaving

model M_{WD} as input, and produces a new metamodel MM_R. Metamodel MM_R is MM_W extended by MM_E, following the specification of the extension definition M_{WD}.

The weaving model M_{WD} conforms to a metamodel that is an extension of the core weaving metamodel. This extension is bootstrapped to be able to extend the core weaving metamodel for the first time. It defines inheritance links, as shown below. The reference *child* refers to the elements of the extension MM_E. The reference *parents* refers to the elements of the core weaving metamodel MM_W. Any other extension semantics can be added in subsequent extension operations.

```
class InheritanceLink extends WLink {
      reference parents[1-*] container : WLinkEnd;
      reference child container : WLinkEnd;
}
```

We illustrate the algorithm that implements the extension operation in Figure 4.5. The operation's main requirement is to create at least one new element in the resulting metamodel. This element must link an element of MM_W and an element of MM_E. The function *addLink()* interprets the extension definition weaving model to add the new elements.

```
MM_R = Extend (MM_W, MM_E, M_WD)
Input:
      MM_W : the metamodel to be extended
      MM_E : the metamodel extension
      M_WD : a weaving model between the elements of MMw and MMe
Output:
      MM_R : an extended MMw
/* add all elements and edges from MM_E into MM_W, if they do not already exist*/
for each mme ∈ MM_E and not mme ∈ MM_W
     MM_W ← MM_W ∪ mme
/* addLink gets the elements represented by M_WD and create a link between them*/
MM_W ← MM_W  addLink (M_WD)
return MMw
```

Figure 4.5 The extension operation

We illustrate the result of an extension operation by extending the core weaving metamodel with an equivalence link. MM_W is the core weaving metamodel. MM_E is illustrated below. The *Equivalence* class contains two references, *source* and *target*, that refer to *LinkEnd*. The code between angle brackets is the result after the execution of the extension operation (MM_R).

The extension operation adds the *Equivalent* and the *LinkEnd* classes into the core weaving metamodel, plus the edges and elements defined in M_{WD}. The extension definition weaving model M_{WD} has two inheritance links. The first link indicates that an inheritance element is added between *Equivalent* and *WLink*. The second link indicates that an inheritance element is added between *LinkEnd* and *WLinkEnd*. The definition of inheritance links using weaving models allows having an approach that is independent of the input metamodels. These links are translated into the *extends* keyword when the operation is executed.

```
class Equivalent extends WLink {
      reference source container : LinkEnd;
      reference target container : LinkEnd;
}
class LinkEnd extends WLinkEnd {
}
```

As already stated, the core weaving metamodel is not designed to support all existing kinds of links for every existing application scenario. A complete weaving metamodel covering every case is not a practical solution. The set of possible links is very extensive, adding much complexity and many kinds of links. The extension operation is used to define different subsets of domain-specific weaving metamodels (DSWMs).

The definition of different kinds of links is not a trivial task. It is an application-oriented task that often requires in-depth knowledge of the underlying domain. We envisage different DSWMs with different kinds of links (a general view is shown in Figure 4.6):

- Interoperability: links such as *Equality*, *SourceToTarget*.
- Data integration: *Concatenation*, *Equality*, *IntToStr*.
- Traceability: *Origin*, *Source*, *Evolution*, *Modified*, *Added*.
- Composition: links such as *Override*, *Merge*, *Delete*.
- Ontology alignment: *Equivalent*, *Equality*, *Resemblance*, *Proximity*.

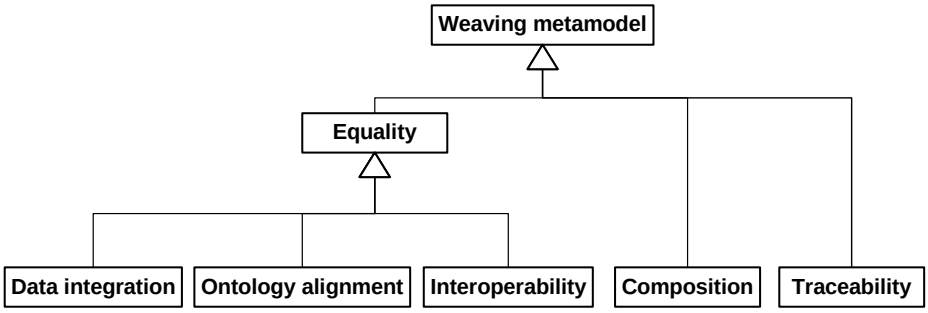


Figure 4.6 A set of DSWMs extensions

From this list (which is not exhaustive), we can see that some kinds of links overlap between different domains. For example, equality links are available in almost every scenario. This motivates the creation of different sets of modular extensions to the core weaving metamodel. The extensions are reused in different applications to finally create the desired DSWMs.

The existence of different extensions adds complexity to the design and the creation of tool support. This is because a generic tool should be capable of adapting to different metamodel extensions. We explain how we implemented a generic and adaptive prototype in the following sections.

4.4 ATLAS Model Weaver tool

In this section we present the ATLAS Model Weaver (AMW) tool. AMW provides a generic and adaptive workbench to manipulate weaving models that conform to different metamodel extensions. The tool implements the concepts presented in the previous sections. The extensibility of the core weaving metamodel has several implications on the design of AMW. The main challenge is to develop a workbench that can be easily adapted and extended. Moreover, the implementation may vary according to the metamodel extensions and on the models that are woven.

First, we give a general description of the tool. Then, we present the different points where the tool can be extended.

4.4.1 General description

The three notions on which we based the design of AMW are: metamodel extensions, tool extensions and generic model manipulation. The tool borrows engineering concepts from the Eclipse

Platform [51]: to build a solid base workbench that is extensible to a wide range of applications. The Eclipse architecture is based on contributions: we contribute to the platform with a new plugin (component) and we also define extension points (an entry point for plugging new contributions). This type of architecture has proven to be effective and widely approved by the software development community. We apply the same principles to create an extensible AMW workbench.

The main idea of the implementation is to have a simple user interface of the tool that might be partially generated, without having to build a specific tool for each weaving task or use case. The tool architecture is illustrated in Figure 4.7.

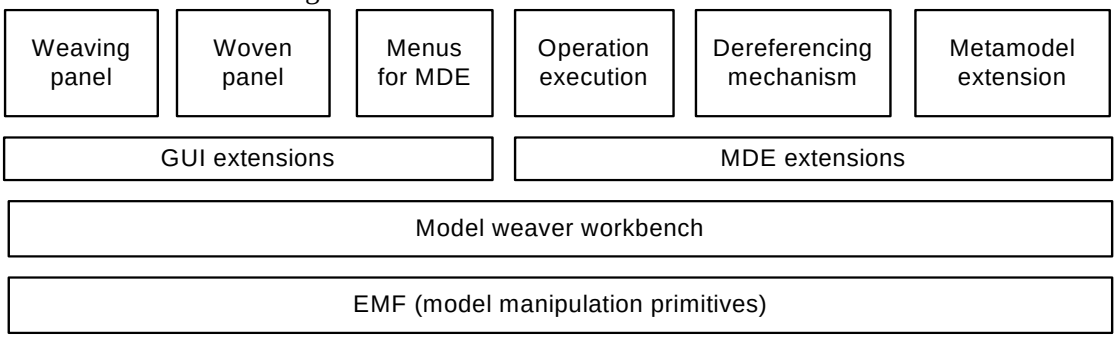


Figure 4.7 Model weaver workbench

The model weaver workbench provides a set of standard facilities for management of weaving models and metamodels. It is built as a contribution to the Eclipse EMF (Eclipse Modeling Framework) [55] plug-in. EMF provides an API for model manipulation, i.e., persistency, inclusion, deletion or update of elements. The API accesses models that are based on the Ecore metametamodel.

EMF relies on the notion of *adapters*. Every model element is associated with one or more adapters. An adapter provides a set of interfaces that must be implemented to support different functionalities. There are different types of adapters, for example label adapters, content adapter, etc. The implementation of an adapter is called *item provider*. Consider for instance the standard three-based EMF interface: one label adapter is responsible to show the correct label for a given node (e.g., the element name); the content adapter contains the model element itself.

The common process of EMF is to automatically generate Java code (with a set of adapters) based on the Ecore metamodel. The generated code accesses model elements conforming to Ecore. This way each adapter may be modified as needed to change the application behavior. However, we do not generate Java code to handle with this metamodel. We use the reflective API of EMF. This means the implementation acts over the Ecore metametamodel elements (using an introspection mechanism), so it supports all weaving metamodel extensions.

The workbench is implemented based on the core weaving metamodel. Since the weaving metamodel extensions always extend elements such as *WLinkEnd*, *WLink* or *WElementRef*, the workbench provides a standard interface that handles different metamodel extensions. The interface auto-generates a set of menus for each different metamodel element. This core provides the standard behavior of the tool.

The AMW workbench defines itself by different extension points, where different components are plugged. There are two main categories of extensions, MDE extensions and GUI extensions. The workbench controls the interactions between these two categories of extensions. We present these two kinds of extensions in the following sections.

4.4.1.1 MDE extensions

The MDE extensions provide additional functionalities with model management facilities, i.e., that operate over the weaving models and metamodels. The workbench can be extended with three kinds of MDE extensions.

- **Metamodel extension**

The AMW tool provides a practical implementation of the metamodel extension operation. The input format of the core weaving metamodel and of the metamodel extensions is KM3. The tool takes the core metamodel and a set of metamodel extensions as input and it generates an extended weaving metamodel that is used by the tool. The KM3 metamodel is translated into Ecore, since it is the implementation format used by AMW. To extend different metamodels (not only KM3), the tool supports creating weaving models with different kinds of extension links.

The XML excerpt shown below illustrates how to define a metamodel extension. First, we indicate that the plug-in is contributing to the extension point "org.eclipse.weaver.metamodelExtensionID", which is a unique identifier for the different kinds of extensions supported by the workbench. The *filename* and *relativePath* properties indicate where the KM3 file is located (relative to the plug-in path). The workbench uses the Eclipse resource mechanism to search for the metamodel extension and to load it in the tool.

```
<extension
         point="org.eclipse.weaver.metamodelExtensionID"
         id="DefaultMetamodelExtension">
         <extensionFile
             name="Base extension for the model weaver"
             fileName ="mw_base_extension.km3"
             relativePath="metamodels/"/>
</extension>
```

- **Dereferencing mechanism**

The metamodel elements that extend *WElementRef* and *WModelRef* are associated with different dereferencing components. These components read the value of the *ref* property and return the corresponding element. This way the tool supports creating relationships between different kinds of metamodels, e.g., SQL-DDL or XML. Consider for example an extension to *WElementRef* called *SQLRef*. This element contains an identifier that enables the identification of the tables and columns in a relational database. Another extension called *XMLRef* can be used to weave XML documents. The elements are identified by the concatenation of the element name and the name of all the parent elements; for instance, an attribute *name* of a class *Table* can be identified by the value "*Table/name*".

```
<extension
            point="org.eclipse.weaver.itemProviderID"
            id="ItemProviderExtension">
            <itemProviderAdapter
                  name="Base Item provider extension"
  class="org.eclipse.weaver.extension.providers.ElementRefItemProvider"
                  adaptedClassName="ElementRef"
                  icon="icons/link_end.gif"
                  isChildrenProvider="false"/>
</extension>
```

The dereferencing mechanism is implemented in a specific adapter, which is associated with a given metamodel element. We define an extension point that contributes with new adapters to each metamodel element. The adapters use the extension point called "org.eclipse.weaver.itemProviderID", as shown below.

The attribute *class* has the name of the Java class with the provided implementation. It must extend *IWeaverItemProvider* interface. The attribute *adapterClassName* contains the name of the type of the model element that is adapted. This means that every time any operation over an element with type *ElementRef* is executed, the tool calls the wrapped methods from the *ElementRefItemProvider* class. The element can be associated with a icon. The class may be an adapter of all the children classes of the given class. This is specified in the *isChildrenProvider* property.

The implementation must also implement the *IIdentifierAdapter* interface (see below). The *setId(Object obj)* method sets a unique identifier to a given model element. The *getId()* method returns the identifier of the element, if it is already set. In the model weaver workbench, the default extension implements a standard adapter that sets an XMI-ID for every created object.

```
public interface IIdentifierAdapter extends Adapter {
      public void setID(Object obj);
      public Object getID();
}
```

- **Operation execution**

These extensions enable the execution of additional model management operations in the AMW workbench. There are different kinds of model management operations, for instance, merging two models, transforming one model into another, or even automatically creating the weaving models. The model management operations can be executed using different mechanisms. The AMW tool uses transformations to execute model management operations and more specifically, the ATL transformation engine [9].

The extension specification below was created to execute a transformation that creates weaving models (the different model management operations executed by the tool are explained in Chapters 5 and 6). The *transformation* attribute has the name of the operation that is executed (*CreateLinks.asm*). The *description* and *category* attributes are shown in the menus created by the user interface. The *binding* element defines which woven model is assigned to a specific parameter of the operation; for instance, the woven model referred by the *leftM* reference is assigned to the parameter *left* of this operation (in this case, the signature of the operation is *CreateLinks (left : model, right : model)*).

```
<matching transformation="transformations/CreateLinks.asm"
    description="Create weaving model"
    category="Metamodel match">
    <binding weavingReference="leftM" header="left"/>
    <binding weavingReference="rightM" header="right" />
</matching>
```

4.4.1.2 GUI extensions

The GUI extensions improve the standard graphical interface of the tool. In general, the GUI extensions call the primitives provided by the MDE extensions. The messages that are exchanged between the GUI extensions and the MDE extensions are controlled by the AMW workbench. We describe the GUI extensions in the following:

- **Woven panel**

The woven panel extension enables creating different user interfaces for manipulating the woven models, for instance tree-like panels or graphical interfaces (e.g., with boxes to represent elements and lines to represent references). The only constraint is that the components might implement a previously defined interface to return all model elements. This way they can be accessed by the weaving panel extensions. These panels correspond to the left and right panels in Figure 4.8.

The workbench provides a standard implementation for manipulating the woven models. The implementation is itself plugged into the workbench by contributing to the "org.eclipse.weaver.wovenPanelID" extension point, in a class named org.eclipse.weaver.extension.panel.DefaultWovenModelPanel". This means that the base implementation that is provided is itself a plug-in that extends the core. This shows the feasibility of such approach.

```
<extension
         id="DefaultWovenPanelExtension"
         point="org.eclipse.weaver.wovenPanelID">
            <modelPanel
                  name="Base woven panel extension"
      class="org.eclipse.weaver.extension.panel.DefaultWovenModelPanel"/>
</extension>
```

- **Weaving panel**

The weaving panel extensions enable plugging different panels for manipulating the weaving models. The weaving panels invoke MDE primitives over the weaving model, such as the dereferencing components and the creation of weaving elements. The weaving panels provide standard functionalities for creating and modifying weaving elements, such as a property editor, creation of compositions and references. Thus, the weaving panels are tightly coupled with the metamodel extension plugins, since this panel should adapt its interface to handle different weaving metamodels. This panel corresponds to the middle panel in Figure 4.8 ("Weaving model").

The definition of the extension is similar as the one of the woven panels. It uses the "org.eclipse.weaver.weavingPanelID" extension point, and the panel is implemented by the class "org.eclipse.weaver.extension.panel.DefaultWeavingPanel".

```
<extension
            point="org.eclipse.weaver.weavingPanelID"
            id="DefaultWeavingPanelExtension">
            <weavingPanel
                  name="Base weaving panel extension"

      class="org.eclipse.weaver.extension.panel.DefaultWeavingPanel"/>
</extension>
```

- **Menus for MDE**

These menus are wrappers that invoke the operation extensions, the MDE primitives to manipulate the weaving models and all the graphical methods wrapped by the different *adapters*. There are two kinds of menus: menus that execute complex model management operations (the transformations), and menus that provide basic model manipulation primitives. The associations between a given menu and a model management operation are specified directly in the implementation. In this case, there is no

XML definition required. The menus that manipulate the weaving models (create, update or delete elements) are available to any kind of metamodel, using the methods provided by the *adapter* classes.

These menus are automatically generated according to the metamodel extension that is loaded. This is possible because the tool is implemented using the EMF reflective API. This adds flexibility because the tool adapts to different metamodel extensions without modifying a single line of code. These menus are available in the "Weaver menu" from Figure 4.8.

In Figure 4.8, we illustrate a weaving model loaded in AMW. This weaving model is used in a tool interoperability scenario (this application scenario is explained in details in Chapter 7). The panels from left and right (*mantisModel* and *bugzillaModel*) contributes to the *woven panel* extensions. These panels are implemented using a standard tree interface provided by EMF. The panel in the middle (*WeavingModel*) contributes to the *weaving panel* extension. The weaving model conforms to a weaving metamodel. The extension is a KM3 file that is itself a contribution to the *metamodel extension*. We show an excerpt of the metamodel extension in the following:

```
class Equivalent extends WLink {
      reference source container: Element;
      reference target container: Element;
}
class Equal extends Equivalent {
}
class AttributeEqual extends Equal {
}
class Element extends WLinkEnd {
}
```

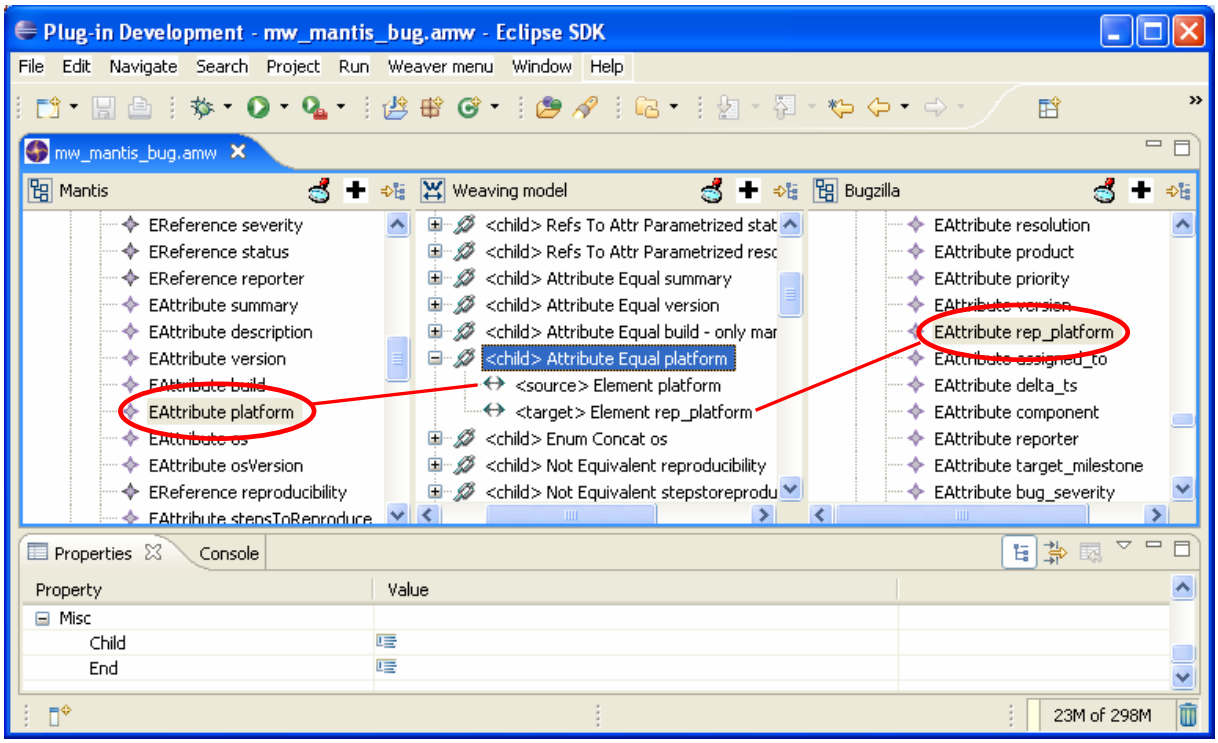


Figure 4.8 The AMW tool

The model element "*Attribute Equal platform*" from the weaving panel links *platform* in the left panel with *rep_platform* in the right panel. This element conforms to the *AttributeEqual* element. The weaving model is created using a set of menus provided by the user interface. Only the *Bug Model* element is automatically created by calling a model management operation that initializes the weaving model.

To summarize, the AMW tool is a generic workbench that supports the basic aspects of generic relationship management. The extension mechanisms enable the adaptation of the interface for different application scenarios. These adaptability facilities are validated through the development of several use cases (presented in Chapter 7). In particular, we stress the possibility of executing different model management operations. This is an important feature that enables using AMW in different data interoperability scenarios.

The tool is available for download as an open source sub-component of the Eclipse GMT (Generative Modeling Technologies) project [4]. The standard workbench and the tool extensions have more than 15.000 lines of Java code, and more than 4.800 lines of transformation code (the integration of AMW with transformations are explained in the following chapters). The site provides extensive documentation, with Wiki, FAQ, code, a set of metamodel extensions, a set use cases. These links are detailed in Appendix A.

4.5 Conclusions

In this chapter we have presented the central concepts used in this thesis: weaving models, weaving metamodels and the metamodel extension operation. These concepts are part of a generic model driven solution for relationship management, called model weaving. The relationships between the elements are captured by weaving models. The weaving models conform to a weaving metamodel. Our approach improves the AMMA platform with the support of generic relationship management.

We have presented a core weaving metamodel that is used as a basis for defining a generic model weaving tool. We defined a metamodel extension operation that uses weaving models to extend different metamodels. The weaving metamodels and models are independent of any implementation. We have presented specifying different dereferencing mechanisms to create relationships between different kinds of models.

We have designed a generic tool called ATLAS Model Weaver, which implements the concepts presented in this chapter. AMW uses standard components of a well-known modeling platform. This allowed having an implementation with a minimal gap between the conceptual definitions.

One major characteristic of the tool is the strong extension mechanisms. The tool handles different metamodel extensions in a straightforward way. The user interface is auto-generated to support different metamodel extensions. The tool also provides extension points to execute model management operations over weaving models. This is an important feature to be able to use the tool in data interoperability scenarios.

Although the implementation has been done under Eclipse, the ideas presented in this chapter could be implemented in different modeling platforms, such as the Microsoft DSL tools. The Microsoft DSL However, the AMMA platform should also ported to DSL tools. Tools provide a set of facilities to develop domain-specific languages. However, at the time AMW has started to be developed, Eclipse was the only solution that provided the basic model management facilities.

In the remainder of this thesis we present the utilization of weaving models in different application scenarios. First, we concentrate on the use of weaving models to improve existing data interoperability approaches. Then, we present different use cases that validate the genericity of our solution.

5 Model-driven data interoperability[1]

5.1 Introduction

In this chapter, we present the use of weaving models and transformation models as a practical solution for data interoperability. There are many different data sources available, with different formats and semantics. As a result of increased collaboration between organizations and rapidly changing environments, it is often necessary to use the data coming from different sources. However, the data produced by distinct organizations are often heterogeneous with very different data formats, thus making data interoperability difficult.

The interoperability of heterogeneous data sources has been studied for a long time in data integration and data translation applications (cf. Chapter 3). In order to integrate the data of different sources, it is necessary to identify the semantic heterogeneities. The format and the semantics of data are typically specified as metadata. Semantic heterogeneities can be expressed as mappings that specify the relationships between elements of metadata.

Many solutions have proposed different kinds of mappings, ranging from 1-to-1 correspondences to ontology bridges. However, we have seen in Chapter 3 that existing mappings typically provide a limited set of semantic relationships, e.g., equality and equivalence. They do not provide support to explicitly define complex kinds of mappings such as mapping expressions. Mapping expressions are manipulations over elements that involve *1:m*, *n:1* or *n:m* relationships, e.g., splitting an element *Address* into *Street* and *Number*. Most solutions implement complex mappings directly in executable transformations using generic arithmetic expressions, e.g., *project_duration = end_date – start_date*, *name = first_name + last_name*. In this case, the semantics of the entire mapping (e.g., *"name concatenation"*) is not defined in the mapping specification, but in the mapping expression itself. Therefore, it is difficult to create and reuse these expressions. The lack of explicit representation also hardens the task of deriving these mappings into executable transformations.

Our approach allows specifying and capturing different kinds of heterogeneities, and to automatically produce executable transformations. In our approach, the data manipulated is a terminal model. A terminal model conforms to a metamodel. We classify different kinds of heterogeneities according to their complexity, and we present a solution to express different kinds of mappings in a weaving metamodel, i.e., at the specification level. The metamodel elements are created with a vocabulary close to their semantic meanings, e.g., *override*, *concatenate*, *split*. A weaving model conforming to this metamodel contains the mappings between a set of input metamodels.

[1] This chapter is an adapted version of the works published at [40] and [41].

The weaving models are used to generate executable transformations. Since we consider transformations as models, the heterogeneities (e.g., mapping expressions) are translated into constructs of specific transformation models. We generalize the process of producing transformations into a pattern that is automatically executed. This pattern may be incrementally modified to handle different semantic heterogeneities. This is a frequently executed operation in model driven engineering. We encapsulate this pattern in a *TransfGen* operation.

The main contributions of this chapter are the following. First, we develop weaving metamodel extensions that capture different kinds of semantic heterogeneities. We emphasize the creation of complex mapping expressions. Second, we provide a generic pattern to automatically generate transformations based on weaving models. Third, considering all entities as models allows applying the same principles to manipulate every involved model.

This chapter is organized as follows. Section 5.2 describes a motivating example that we use as a guide for presenting our approach. Section 5.3 presents weaving metamodel extensions for data interoperability. These extensions enable creating relationships between different models. Section 5.4 explains how these relationships are used to produce model transformations. Section 5.5 concludes.

5.2 Motivating example

We illustrate the data interoperability issue using two bug tracking tools. We use a bug tracking scenario to show the existence of different kinds of semantic heterogeneities. Bug tracking tools manage the bugs (reporting, fixing) of a given application. Many bug tracking tools are available, e.g., GNATS, Mantis, Bugzilla, and many others [58]. Consider two autonomous software development companies, C_A and C_B, and a set of N bug tracking tools. Company C_A uses tool T_i and company C_B uses tool T_j. They need to collaborate without aligning their software development practices. This is due to pragmatic reasons, e.g., the companies already participate in other cooperative projects.

We illustrate this situation using two bug tracking tools, Bugzilla [28] and Mantis [98]. Bugzilla is a general purpose, open source bug tracking tool. It provides features such as error tracking and quality assurance management. The Bugzilla metamodel is illustrated in Figure 5.1.

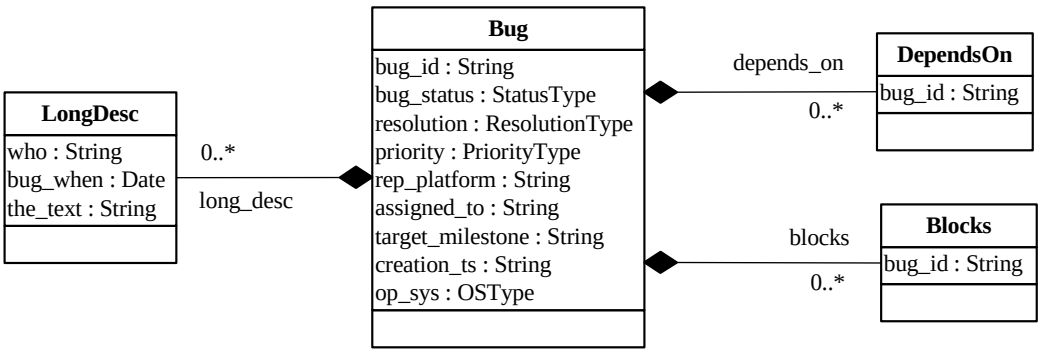


Figure 5.1 Bugzilla metamodel

Mantis is another bug tracking tool. It differs from Bugzilla as a light weight tool which allows adding new modules. The metamodel of Mantis is illustrated in Figure 5.2.

We observe that it is possible to establish different kind of mappings between the elements of the tools metamodels. The most common kind of mappings is equality, where two concepts are said to be equal. For example, a software bug is represented by *Bug* in Bugzilla and *Issue* in Mantis. As another example, the date a bug is created is represented by *creation_ts* and *date_submitted*, respectively. There are also elements representing equivalent data, but not the same, e.g., *target_milestone* is the version where a bug will be fixed, and *fixed_in_version* is the version where a bug was fixed.

There are also more complex kinds of mappings. For example, Bugzilla has two kinds of relationships between bugs: *depends_on* and *blocks*. In *Mantis*, bugs are related to each other using the element *relationships*, which points to *Relationship*. The relationship type is stored in the element *type*. As another example, *assigned_to* contains the responsible to solve a given bug in Bugzilla. In Mantis the relationship *assigned* points to element *Person* (that contains elements *login*, *value* and *id*).

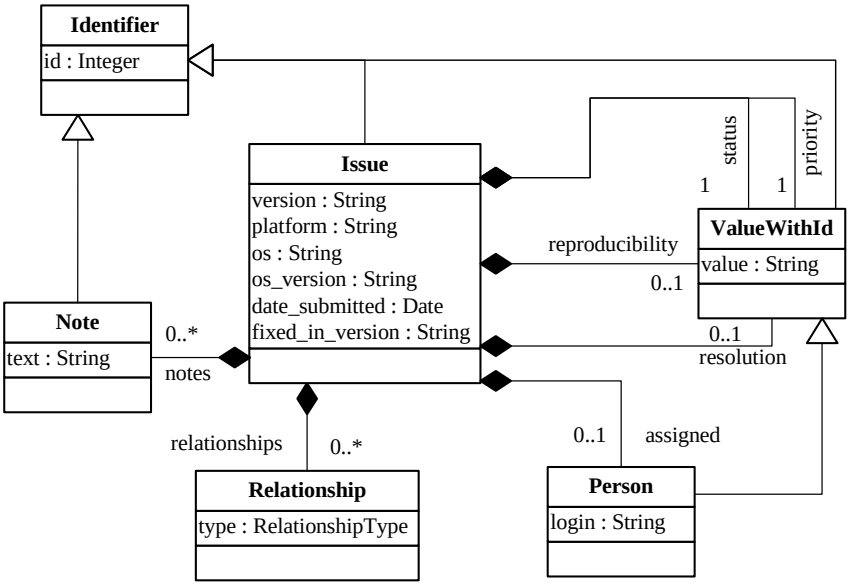


Figure 5.2 Mantis metamodel

In addition, there are semantic heterogeneities at the data level. For instance, the element *bug_status* in Bugzilla and relationship *status* in Mantis (that points to *ValueWithId*) defines the bug state (e.g., a bug was included in the database, a bug was solved, etc.), and the element *priority* defines the priority to solve a given bug (e.g., immediate, urgent). Each tool has its own set of status and priorities. For example, it is necessary to take into account that the *priority* with value *"P_1"* in *Bugzilla* is translated into the value *"urgent"* in Mantis. The same analogy applies to the element *status*. Different kinds of heterogeneities and the other elements not explained here are discussed later in section 5.3.

Traditional data interoperability applications usually create mappings to capture similarity heterogeneities (e.g., equality, equivalence). These mappings can be used to produce transformations that execute the translations from Bugzilla to Mantis. However, complex mapping expressions and data-level heterogeneities are coded either in some element in the mappings, or in the produced transformations. For example, the developer must code how to translate between the enumeration values in one specific language. The lack of explicit structures for complex expressions hardens the creation of mappings because there is no domain information about the possible mappings. The possible mappings are virtually unlimited when using generic arithmetic expressions. This way is not possible to understand all the mappings without analyzing the entire expression in the produced transformations. This also reduces the reusability of these expressions. In addition, there is not enough information to automatically produce the transformations, which is a frequently executed operation in model management. The mappings and produced transformations must be kept synchronized.

In order to efficiently achieve data (and tool) interoperability, all these kinds of mappings must be explicitly specified. These mappings must be derived into executable transformations. This process must be efficient, such that new transformations between other tools can be rapidly developed.

We capture the semantic heterogeneities between the set of tools in a weaving model. Many different transformation patterns are applied in these transformations, such as equality, equivalence and concatenation. These transformation patterns depict translation links between the source and target metamodels. Instead of directly creating a transformation between the models, the typed links of the weaving model capture these semantic heterogeneities in a more abstract representation. These links are finally translated into an executable transformation language.

5.3 Data interoperability metamodel extensions

In this section we present a set of weaving metamodel extensions used for data interoperability. We illustrate our extensions using the motivating example (tool interoperability) from section 5.2. The tool data and metadata are represented as terminal models and metamodels. Thus, the tool heterogeneities are expressed as mappings between tool metamodels. The terminal models and metamodels can be generalized for other data interoperability problems. The mappings types are specified in a weaving metamodel. We define tool and mappings below.

Definition 5.1 (Tool). A tool T is a tuple $<M_t, S_t>$, where:

- $M_t = (G, MM_t, \Gamma_t)$ is the tool terminal model. M_t is the data that is manipulated by T,
- MM_t is the reference model (metamodel) that represents the tool metadata,
- $S_t = \{s_i; i = [1..n]\}$ is the set of services (querying, updating, inserting, etc.) provided by T. Every service $s \in S_t$ must respect the constraints specified in MM_t.

Consider a bug tracking tool $T_a = <M_{ta}, S_{ta}>$. The metamodel MM_{ta} specifies how the bugs are organized, the properties of a bug and the possible states of a bug during its life cycle. The model M_{ta} has the value of the bugs, e.g., that a given bug "B" has a status of "in correction" to a developer called "Joseph". The set S_{ta} contains miscellaneous services: the inclusion of a new bug in the database, the update of a bug status and the query of a set of bugs.

Consider another bug tracking tool, $T_b = <M_{tb}, S_{tb}>$ with a different model, reference model and set of services. The semantic heterogeneities between metamodels MM_{ta} and MM_{tb} are expressed as mappings. The mappings between tool metamodels have different types, structures and semantics. However, intuitively, mappings depict the notion of typed-links between (meta) model elements.

Definition 5.2 (Mapping). Given two models M_{ta} and M_{tb}, a mapping *M* is a tuple $<S_a, S_b, T>$, where:

- S_a is a set of elements from the model M_{ta},
- S_b is a set of elements from the model M_{tb},
- T is the type of mapping between the sets S_a and S_b.

There are many different kinds of mappings, for instance *equality*, *equivalence* or, *generalization*. These are simple kinds of mappings that express element similarity, usually denoting 1-to-1 links. Complex mappings have multiple cardinalities and semantic meaning. These kinds of mappings abstract commonly used mapping expressions, e.g., the average between a set of elements or the concatenation of strings. We specify the different mapping types as weaving metamodel extensions. The mapping types are expressed as extensions of the *WLink* element from the core weaving metamodel.

However, it is not possible to create a weaving metamodel extension containing all kinds of links for data interoperability. We present the creation of different metamodel extensions to capture different types of links. We classify them in three major groups according to the complexity of the links semantics. We assume that the mappings are directed, i.e., that there is one *source* metamodel and one *target* metamodel. The link kinds are defined in KM3.

5.3.1 Similarity expressions

Similarity expressions represent resemblance links between metamodel elements. These expressions are the link types encountered in most semantic-based mapping solutions. There are different kinds of similarity expressions. We describe them below.

Equality: elements that represent exactly the same information are connected by equality links. The link type *Equal* is a binary relation. It indicates that a source element is equal to a target element. The link type does not specify the exact data type (*String*, *Class*). The data type is specified when deploying the solution (as extensions of *WLinkEnd*).

```
class Equal extends WLink {
      reference source container : WLinkEnd;
      reference target container : WLinkEnd;
}
```

Illustration: There is an element *priority* in both Mantis and Bugzilla metamodels. It contains the priority to solve a given bug, i.e., a bug with a higher priority is corrected before a bug with lower priority.

Equivalence: the linked elements represent similar information, but not exactly the same. However, the translation semantics may be the same as in equality links, i.e., one target element receives the value of a source element. We add a description attribute to provide additional information about the equivalence, and a similarity measure.

```
class Equivalent extends WLink {
      reference source container: WLinkEnd;
      reference target container: WLinkEnd;
      attribute description : String;
      attribute similarity : Double;
}
```

Illustration: the equivalence links could be created between the same elements used to create equality links, though, with the additional similarity estimation. This is often the case when the links are created with the help of some semi-automatic method.

Typed correspondences: the equality and equivalence definitions do not differ between element types. The addition of type constraints avoids generating invalid equalities, for example a link between a *Class* and an *Attribute* is not always possible, or requires specific conversions. We define a *<Type1><Type2>Equal* class. The type information is used for converting elements. The two *<Type>* templates are replaced by the data types, for example *String*, *Integer*, *Reference*, *Class*, *Attribute*.

```
class <Type1><Type2>Equal extends Equal {
      reference source container : WLinkEnd<Type1>;
      reference target container : WLinkEnd<Type2>;
}
```

Illustration: The element *dateSubmitted* in Mantis has the date the bug was created and it has *Date* type. In Bugzilla the date the bug was created is represented by the *creation_ts* element, which has string type.

Disjointness[2]**.:** two elements represent incompatible data. The link type also contains a description

```
class Disjoint extends WLink {
      reference source: WLinkEnd;
      reference target : WLinkEnd;
      attribute description : String
}
```

Generality[2]**:** the elements have a relation of inheritance.

```
class Inherit extends WLink {
      reference parent : WLinkEnd;
      reference child : WLinkEnd;
}
```

Non equivalence: it is not always possible to represent all the information produced by one tool in another tool. Some elements from the tool metamodels do not have any semantic relationship, or are not relevant for a given translation and do not need to be generated. The element may be simply ignored.

We define an extension to keep track of the elements that do not have any equivalences. The class *NotEquivalent* has source and target references, which are mutually exclusive. The attribute *note* has a description about the elements.

```
class NotEquivalent extends WLink {
      reference source container : WLinkEnd;
      reference target container : WLinkEnd;
      attribute note : String;
}
```

Illustration: reproducibility in Mantis contains the frequency of reproduction of a given issue. This information is not mandatory, and it is not available in Bugzilla. The element *urlbase* from Bugzilla contains the base URL of a given bug (since Bugzilla is web based). This element does not have equivalents in Mantis.

5.3.2 Mapping expressions

Mapping expressions are mappings that involve a set of source elements and a set of target elements. The weaving metamodel encapsulates mapping expressions in metamodel elements. The underlying formalism of how the mapping expressions are executed is hidden from the weaving metamodel. The navigational and calculations expressions over source and target models are defined in a subsequent step.

However, it is not possible to define metamodel extensions for every type of mapping expression. Mapping expressions vary from application to application. In addition, mapping expressions are often created manually, because the relationships between model elements are typically complicated and cannot be created by automatic algorithms because they involve semantic reasoning. We define an abstract class *Expression*. This class is not directly created in a weaving model; it is an initial point for any other expressions. We describe the type of other mapping expressions below.

```
abstract class Expression extends Equal {
}
```

[2] This kind of link is not present in the tool interoperability example

Many-to-one: Many-to-one expressions link a set of elements of the source metamodel and a single element in the target metamodel. For instance the addition of numbers and concatenation of strings attributes. The abstract class *ManyToOne* must be extended with a metamodel element that indicates the required operation. It refers to a set of *source* elements and to one *target* element.

```
abstract class ManyToOne extends Expression {
      reference source[*] ordered container : WLinkEnd;
      reference target container : WLinkEnd;
}
```

Illustration: the attributes *os* and *osVersion* from Mantis contains the operating system and the operating system version. In Bugzilla, this information is available in one single attribute *op_sys*. This means elements from Mantis must be concatenated. We create a class *Concatenation* that extends the *ManyToOne* class.

The weaving metamodel has two *source* references, one pointing to *os* and the other to *osVersion*, and the target reference points to *op_sys*. The *ordered* keyword means that the elements are concatenated in the order the *source* references are created. The *separator* attribute contains a separator between elements (e.g., “;”), if necessary.

```
class Concatenation extends ManyToOne {
      attribute separator : String;
}
```

Split or one-to-many: Split (or one-to-many) expressions are the opposite of many-to-one expressions, i.e., they link more than one target element with a single source element.

```
abstract class Split extends Expression {
      reference source ordered container : WLinkEnd;
      reference target[*] container : WLinkEnd;
}
```

Illustration: this is the opposite scenario of many-to-one expressions, for example to split *osVersion* into *os* and *op_sys*. The elements are parsed according to a given criteria, for example the separator character. This case is a typical string parsing. Thus, we define a class *SplitStr* with the separator character.

```
class SplitStr extends OnetoMany {
      attribute separator : String;
}
```

Many-to-many: Many-to-many expressions relate a set of elements of source metamodels with a set of elements of target metamodels. One may argue that a many-to-many expression can be created in terms of many-to-one and one-to-many expressions. However, it would have reduced expressiveness. The class *ManyToMany* contains two references to a set of *source* and *target* elements.

```
abstract class ManyToMany extends Expression {
      reference source[*] ordered container : WLinkEnd;
      reference target[*] ordered container : WLinkEnd;
}
```

Illustration: In Mantis, a bug may have dependencies with other bugs. The reference *relationships* points to a *Relationship* class. The type of the relationship is saved in a *type* attribute. The domain of valid relationships is defined in an enumeration. The values are: *related_to*, *parent_of*, *duplicate_of*, *has_duplicate*. These dependencies must be taken into account when fixing a bug.

There is a similar concept of bug dependencies in Bugzilla. However, for each different type of dependency there is a reference to a different element. There are only two types of bug dependency: *dependson* and *blocks*. *Blocks* means that the related bug can be fixed only after the current bug is

fixed. *Dependson* is used for all the other types of dependencies. Thus, it is necessary to reorganize one reference (*relationships*) and one literal (*type*) in Mantis into different types of references and elements in Bugzilla.

New values on the target: New value expressions are used to generate values in the target model that do not have a correspondence in the source model. These values may be automatically generated or may take a predefined value from user input. An example of automatic generation is the production of element identification.

The class *AutoSetValue* is extended into *AutomaticGenInt* and *ManualInput*. The class *AutomaticGenInt* reads the element that is referred by the *target* reference and generates a random number for it. The class *ManualInput* sets the the *target* reference attribute with the value of *sourceValue*.

```
abstract class AutoSetValue extends Expression {
}
class AutomaticGenInt extends AutoSetValue {
}
abstract class ManualInput extends AutoSetValue {
      attribute sourceValue : WLinkEnd;
}
```

Illustration: The developer responsible for fixing a bug is represented by the *assignedTo* reference. The data is stored in the element *Person* with attributes *login* (the user login) and *id* (and unique user identifier). Both are mandatory. In Bugzilla *assigned_to* has the same meaning, but it contains only the login in a text field, without any *id*. This information must be generated in the target model when a transformation is executed.

5.3.3 Data value expressions

Data value expressions differ from mapping expressions because they also evaluate the terminal model elements, not only the metamodel elements. Data value expressions compare the source terminal model elements (not only the metamodel elements) and modify them to make compatible with the target model.

The class *DataExpression* refers to a set of equivalences. The *source* element is evaluated, and if it is equal to one *sourceValue* from the set of equivalencies, it sets the target element with the corresponding *targetValue*. The equivalencies may be of any data type.

```
abstract class DataExpression extends Expression {
      reference equivalences[*] container : Equivalence;
}
abstract class Equivalence extends WLinkEnd {
      attribute sourceValue : WLinkEnd;
      attribute targetValue : WLinkEnd;
}
```

Illustration: In Mantis the *resolution* reference contains the correction status of a bug (for example if it was fixed or not). It may have the following values: *OPEN*, *FIXED*, *REOPENED*, *UNABLE_TO_DUPLICATE*, *NOT_FIXABLE*, *DUPLICATE*, *NOT_A_BUG*, *SUSPENDED*, *WONT_FIX*. We must set the equivalencies with the *resolution* element (with the same meaning) in Bugzilla. The possible values are: *null*, *FIXED*, *INVALID*, *WONTFIX*, *LATER*, *REMIND*, *DUPLICATE*, *WORKSFORME*, *MOVED*.

We illustrate these extensions in Figure 5.3. We show only the inheritance relations between the elements.

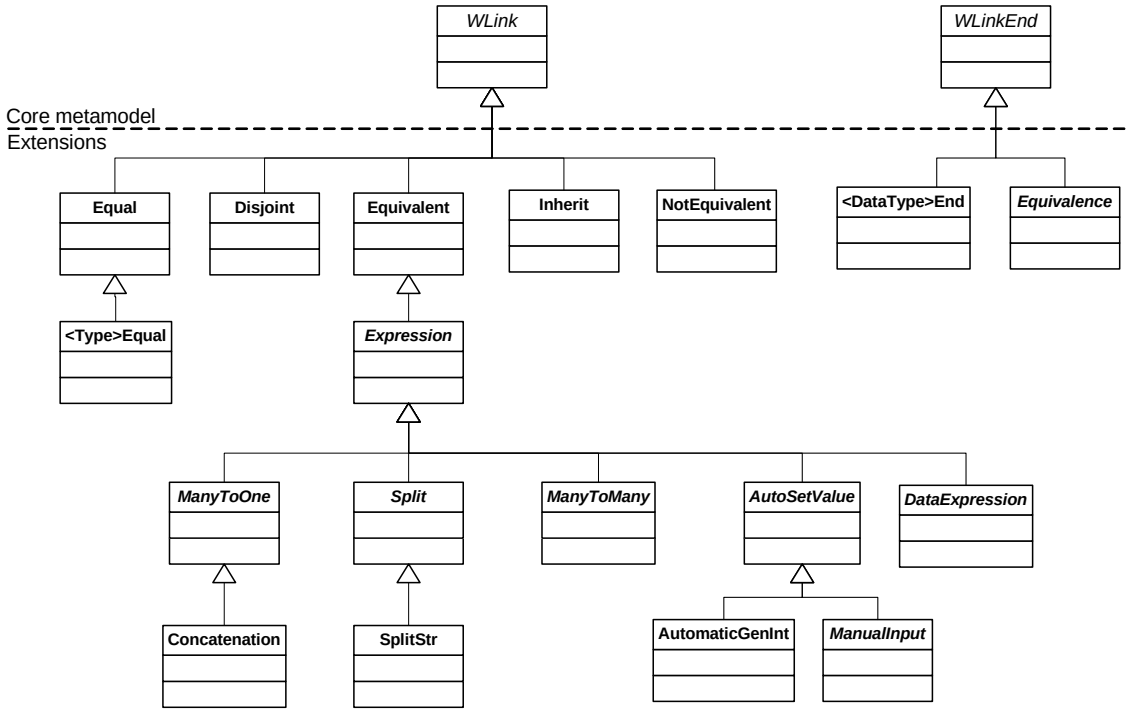


Figure 5.3 Metamodel extensions for data interoperability

The upper side of the Figure shows two classes of the core weaving metamodel, *WLink* and *WLinkEnd*. We can see that the class *WLink* is the parent class of all classes that define different link semantics. The *WLinkEnd* class has fewer extensions, since it typically defines the elements that are linked. All these extensions describe the basic kinds of links of most data interoperability solutions. They are abstract elements that should be extended in turn with concrete links such as *Concatenation* or *Split*.

5.4 Interpreting data heterogeneity

In the previous section, we have described a set of metamodel extensions to capture different semantic heterogeneities. The next step is to create a weaving model conforming to these extensions and to derive this model into executable transformations. The produced transformations translate the set of input models into the set of output models.

The weaving models are created using the adaptive graphical interface of AMW (cf. Chapter 4). The weaving models can also be created using semi-automatic methods. We do not specify in which way the weaving model is created in this chapter. These methods are explained in detail in Chapter 6.

Based on the weaving metamodel extensions, we present a generic pattern used to automatically produce model transformations, which are responsible to translate a set of source models into a set of target models.

First, we introduce model transformations, which is the central mechanism to perform operations over models. We present an overview of the common structures of declarative transformation languages. Then, we describe a generic pattern used to automatically produce model transformations. This pattern allows encapsulating the transformation production task in a generic model management operation.

5.4.1 Model transformations

Model transformations enable executing model management operations over models. We define model transformation below.

Definition 5.3 (Model transformation). A model transformation is an operation that given as input a set of models, evaluates their elements and produces as output a set of models.

A model transformation has the following signature:

$$< OUT_1 : MM_{OUT1}, \dots , OUT_m : MM_{OUTm} > = T (< IN_1 : MM_{IN1}, \dots , IN_n : MM_{INn} >)$$

T is the operation name; $<IN_1 - IN_n>$ is the set of input models ($n \geq 1$); the input models conform to the input metamodels $<MM_{IN1} - MM_{INn}>$; the input metamodels may be equal; $OUT_1 - OUT_m$ is the set of output models ($m \geq 1$); the output models conform to the output metamodels $<MM_{OUT1} – MM_{OUTn}>$; the output metamodels may be equal.

Our approach considers transformations as models. Thus, the operation *T* is specified in a transformation model $T = (G_T, MM_T, \mu_T)$. Transformation models are terminal models. *T* conforms to a transformation metamodel MM_T. This means that all operations on models may be applied to transformations, including transformations of transformations (the advantages of considering transformations as models are explained later in this section).

However, when producing transformations for data interoperability, there are many engines and languages that could be used (e.g., ATL, XSLT, SQL-like languages). Thus, we produce different transformation models as output based on the same weaving model. This means the same weaving may be used to produce a transformation T_1 that conforms to a transformation metamodel MM_{T1} (e.g., ATL), or to produce a transformation T_2 that conforms to a transformation metamodel MM_{T2} (e.g., XSLT). This is possible because, despite their different syntax and expressive power, several transformation languages are typically declarative and have similar structures. We describe these common structures below:

- *input* and *output models* and their *metamodels*: are the source and target models, e.g., an XML document, an ontology, a relational table;
- *rules*: are self-contained commands containing all the necessary constructs to translate source elements into target elements, e.g., an SQL view, an XSLT template or an ATL rule;
- *input elements*: define which elements from the input model are transformed. Input patterns usually relate elements formed by sub-elements or attributes, e.g., ATL input patterns, XSLT matched templates or SQL select from clauses;
- *output elements*: define the target elements, strictly related with the input elements, e.g., ATL output patterns, XSLT elements or SQL create view statements;
- *selection expressions*: define filters in the input patterns to produce subsets of elements, e.g., ATL filters, XPath expressions or SQL where clauses;
- *equivalence expressions*: define the relationships between the attributes of a given input element and the attributes of the output elements, e.g., ATL bindings, XSLT *value-of*. The weaving elements indicating relationships and their semantics should be translated as equivalence expressions;
- *computing expressions*: return a new value after executing computations over input elements that are used in equivalence expressions, e.g., OCL expressions, XPath or SQL functions.

We define a transformation metamodel subsuming these common structures of transformation languages. The metamodel is illustrated in Figure 5.4 (in KM3). It is a metamodel for a declarative transformation language. When implementing an application of the transformation pattern, the metamodel must be replaced by a complete transformation metamodel.

The transformation *Module* is the main element that contains one or more transformation *Rule*. A *Rule* defines how to translate one input element into a set of output elements. The declarative nature of the language abstracts the need to translate a set of input elements. The input and output elements are represented by a *ReferredElement*. A *ReferredElement* points to the elements of the input models. These elements may be classes, attributes or a root element.

A *Rule* contains one *InputElement*. The reference *element* points to the element of the input metamodel. This element is said to be matched in the transformation rule (a rule matches an element when its type is the one of the input element). The reference *condition* contains an expression to filter a subset of the model elements. A *Rule* has many output elements (reference *output)*. An *OutputElement* is the model element that is created in the output model. The output element type is captured in the *element* reference.

An output element has a set of *bindings*. A *Binding* specifies the values of the attributes for the related output element. A *Binding* contains a *target* element. The target element is either an *Attribute*, or a *Reference* to a class. The *source* attribute is a model element from the input models or a mapping expression (*MappingExpression*) over input elements. Expressions depend on the output transformation model. We do not specify a complete expression language here. For instance, it is possible to define expressions in XQuery or XPath. This transformation model is used to define a generic transformation pattern.

```
package Transformation {

class Module {
      reference inputModels [1-*] container : ReferredElement;
      reference outputModels [1-*] container : ReferredElement;
      reference rules [1-*] container : Rule;
}
class Rule {
      attribute name : DataType;
      reference input container : InputElement;
      reference output [*] container : OutputElement;
}
class InputElement {
      reference element container : ReferredElement;
      reference condition [0-1] container : Expression;
}
class OutputElement {
      reference element container: ReferredElement;
      reference bindings [*] container : Binding;
}
class Binding {
      reference target container : ReferredElement;
      reference source container : Expression;
}
abstract class Expression {
}
class ReferredElement extends Expression {
}
class MappingExpression extends Expression {
}
}
```

Figure 5.4 Generic transformation metamodel

5.4.2 Generic pattern of transformation

We encapsulate the query discovery phase in a generic pattern of transformation. The definition of the generic pattern of transformation relies on three facts. First, the core weaving metamodel is formed by links, link endpoints and extensions of these elements. Second, declarative transformation languages have similar structure. Third, we use declarative patterns of transformation that specify only what to transform, and not how to transform. The pattern of transformation expresses the execution semantics of the weaving model, because it transforms the different kinds of links into executable mapping expressions in some transformation language.

The generic pattern is specified using higher-order transformations (HOT). A HOT takes as input a weaving model conforming to an extension of the weaving metamodel and transforms it into a transformation model.

Definition 5.4 (Higher-order transformation). A higher-order transformation is a transformation $T_{OUT} : MM_T = T_{HOT} (T_{IN} : MM_T)$, such that the input and/or the output models are transformation models. Higher-order transformations either take a transformation model as input, either produce a transformation model as output, or both.

We create a simple syntax for a transformation metamodel to define the generic patterns. This pattern is the basis to define a model management operation called *TransfGen*. We define this operation below.

Definition 5.5 (TransfGen operation). *TransfGen* is a higher-order transformation that takes a weaving model M_W as input and that produces a transformation model M_T as output. The weaving model conforms to a data interoperability metamodel extension MM_W.

$M_T : MM_T = TransfGen\ (M_W : MM_W)$.

Figure 5.5 illustrates the conformance relations (denoted by *c2*) of the models involved in the *TransfGen* operation: *Mw* is the input weaving model; *TransfGen* is the higher-order transformation. It produces a transformation model M_T. M_T and *TransfGen* conform to the same transformation metamodel MM_T. However, it is also possible that the output transformation conforms to a different transformation metamodel. This enables producing different transformation models as output.

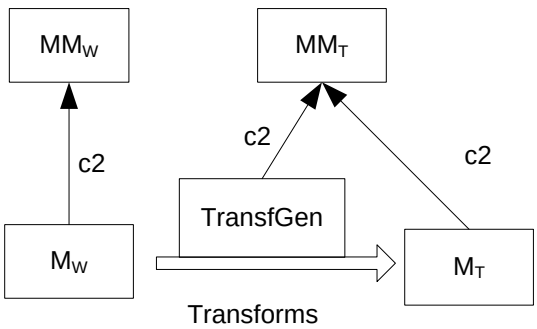


Figure 5.5 Models in the *TransfGen* operation

Figure 5.6 describes the semantics of the *TransfGen* transformation using transformation rules. These transformation rules conform to the generic transformation metamodel. Note that the syntax is based on the ATL syntax, but it is not ATL. The keywords are in bold font. The transformation has a set of declarative rules. The *input* element matches the input weaving metamodels. The *output* element creates a new element in the output model. The output element has *bindings* to assign the source values to the target elements. The weaving metamodel has one extension of *WLink* (as shown below)

to denote source and target elements. The pattern can also be used with different metamodel extensions.

The transformation has a set of declarative rules. The *input* element matches the input weaving metamodels. The *output* element creates a new element in the output model. The output element has *bindings* to assign the source values to the target elements. To denote source and target elements, the weaving metamodel has one extension of *WLink* as follows:

```
class WLinkST extends WLink {
      reference source container : WLinkEnd;
      reference target container : WLinkEnd;
}
```

1 **Module TransfGen (C: ω_C)**
2
3 **inputModel:** C /* a correspondence model conforming to a correspondence metamodel ω_C*/
4 **outputModel:** T /* a transformation model conforming to ω_T */
5
6 **rule** newModule
7 **input** WModel
8 **output** Module
9 rules ← ownedElement (ownedElement isA WLinkST)
10
11 **rule** newRule
12 **input** WLinkST (parent isA WModel) /*classifiers (classes, references, attributes)*/
13 **output** Rule
14 input ← source
15 output ← target
16
17 **rule** newInput
18 **input** WLinkEnd (link.source = self)
19 **output** InputElement
20 element ← ρ (element.ref)
21 condition ← /*depends on the WLinkST and WLinkEnd types*/
22
23 **rule** newOutput
24 **input** WLinkEnd (link.target = self)
25 **output** OutputElement
26 element ← ρ (element.ref)
27 bindings ← link.child /*get the sibling WLinkEnd*/
28
29 **rule** newExpression
30 **input** WLinkST (parent isA WLinkST)
31 **output** Binding
32 source ← MapExp (ρ (source.element.ref)) /*mapping expressions here,*/
33 target ← ρ (target.element.ref) /*according to the WLinkST type*/

Figure 5.6 Higher-order transformation pattern

The pattern can also be used with different metamodel extensions. The rule *newModule* (line 6) creates a new transformation module (line 8) matching the root element *WModel* (line 7). The rule creates a new rule for all the elements of type *WLinkST* owned by *WModel* (line 9) (the references are

relative to the current input element, in this case *WModel.ownedElement*). The values of the rules are set in *newRule*.

The rule *newRule* (line 11) matches the *WLinkST*'s children of *WModel* (line 12) and creates a transformation rule (*Rule* element) for each of them (line 13). These *WLinkST* refer to classes, attributes and references. The created rule has one input element that matches the element referred by the *source* reference (line 14). The output element corresponds to the *target* reference of the current *WLinkST* (line 15).

The rule *newInput* (line 17) matches *WLinkEnd*s used as *source* of a *WLinkST* (line 18), i.e., it returns the source element of line 14. The *newInput* rule creates an *InputElement* (line 19). The input element has a filter condition (line 21) that varies according to the *WLinkEnd* type. The reference *element* (line 20) is bound with an identification function (ρ). A unique identification function is associated with each different extension of *WElementRef*. The function returns a *ReferredElement*. A *ReferredElement* contains the value used to identify the elements of the input metamodels. Each different extension of *WLinkEnd* has a new rule according to this pattern.

The rule *newOutput* (line 23) matches *WLinkEnd*s used as *target* of a *WLinkST* (line 24), i.e., it contains the value that is returned to the output reference in line 15. This rule creates an *OutputElement* (line 25). The output element is fetched from the output tool metamodel using a specific identification function as in the previous rule. This rule creates bindings (line 27) for the children elements of the current target element (such as attributes or references). The children bindings allow having different containment levels between model elements. The bindings contain the different implementations of mapping expressions.

The *newExpression* rule (line 29) matches all the *WLinkST* that are not a child of *WModel*. The rule creates a binding (line 31) setting the value of the target element (line 33) with the mapping expression over the source (or set of source) elements (line 32). The created *Binding* is the return value for the *bindings* reference from line 27. Each different mapping expression (denoted by *MapExp*) must implement a different calculation expression.

The *TransfGen* operation encapsulates the task of producing transformations. This way it is possible to separate the overall data interoperability process into distinct operations. The weaving model is created by a *Match* operation (cf. Chapter 6). The weaving model is translated into a transformation model using *TransfGen*. The translations between the source and target models are encapsulated in the generated transformations, which are specific data transformation operations. We validate our approach by applying these techniques to an extended version of the motivating example. We present these experiments separately in Chapter 7.

5.5 Conclusions

In this chapter, we have presented an approach that uses data integration techniques applied to data interoperability problems. We based our solution on MDE principles to capture the semantic heterogeneities and to produce transformations between models.

After having provided a classification of heterogeneities, we have shown how this classification may be translated in various kinds of links defined in a weaving metamodel. Furthermore, the weaving metamodel may be seen as an extension of the core weaving metamodel that provides basic support for link management. The main original aspect of our approach is to offer maximum extensibility to capture the semantics of different kinds of mappings and data value expressions.

We have shown that metamodel extensions allow expressing the different kinds of heterogeneities with a dedicated vocabulary and in a declarative way. Every domain-specific metamodel prevents from developing a generic language (and not well-adapted) without the capability to explicitly express the heterogeneities.

The weaving model can be interpreted following a generic and declarative pattern. The semantics of this pattern is the basis for a novel model management operation called *TransfGen*. Based on this

pattern, we have developed higher-order transformations that automatically produce output transformation models. This operation encapsulates the transformation production task of typical data interoperability solutions. The transformations are generated automatically because we leave all the human intervention to the process of creating weaving models. Different use cases using these techniques are presented later in Chapter 7. We assumed in this chapter that the weaving models are created using the AMW tool. We present in Chapter 6 how to use semi-automatic techniques to ease the task of creating these weaving models.

Finally, considering all entities as models enables manipulating all of them using the same set of principles. The main principle is to define different types of domain models and to apply transformations between them. This is particularly useful when specifying the semantic heterogeneities and when translating a weaving model into executable transformation models.

There are two major issues that are subject for future work. First, different metamodel extensions should be designed and applied to different application scenarios. The extensive utilisation and the refinement of metamodel extensions by domain experts is the best way to come into an agreement and to disseminate these extensions. Second, the *TransfGen* operation should be implemented and validated in different platforms, for instance, relational databases.

6 Matching transformations[1]

6.1 Introduction

We have seen in the previous chapters that transformations models are a very important kind of models in data interoperability. As a consequence, there are an increasing number of transformations models that are being developed for different application scenarios. For instance, there are transformations to provide data interoperability, to translate from textual to graphical representations, or to merge models.

However, the development of transformations involves many repetitive tasks. Consider for example a generic data interoperability scenario that transforms one source model into one target model. The transformation development consists of creating rules that transform a set of elements of the source model into a set of elements of the target model. The properties of these elements are transformed using a set of transformation expressions.

We have seen in Chapter 5 how weaving models are used as specifications to produce transformation models by capturing different kinds of links. The links are used as specification to frequently used transformation patterns. The process of establishing links between different model elements is called *matching*. However, the matching process can be partially automated. A semi-automatic process based on well-defined patterns brings many advantages: it accelerates the development time of transformations; it diminishes the errors that may occur in manual coding; it increases the quality of transformational code. To the best of our knowledge, there is not a MDE approach that provides enough generic mechanisms to semi-automate the development of transformations.

The discovery of transformation patterns to integrate models is related to schema and ontology matching approaches (see the approaches presented in Chapter 3). These approaches aim at discovering relationships between elements of different models. These relationships are used for different purposes, such as ontology alignment or data translation. However, these approaches have some drawbacks. Most solutions cannot be applied to models conforming to different metamodels. The distance between the conceptual basis (models) and the implementation is too important. This makes it difficult to decompose and to customize different algorithms. There is no support for different kinds of relationships between models. Hence, native constructs of transformations are not supported, such as rule inheritance or nested relationships.

In this chapter, we present a novel solution to semi-automate the creation of weaving models, called *matching transformations*. Matching transformations are transformations used to implement

[1] This chapter is an extended version of the work published at [43] and [71].

different matching techniques. This means that, based on the elements of a set of input models, they produce a weaving model with links between these elements. The weaving model conforms to extensions to the core weaving metamodel.

Matching transformations enable the implementation of new or the adaptation of existing techniques to create weaving models. This is an important feature to be able to deploy an adaptive tool. In addition, we present different ways to express a well-known generic algorithm. We exploit different kinds of relationships between the model elements to calculate similarity estimations between different model elements. The matching transformations are executed together with link rewriting methods that analyze the weaving metamodel extensions to produce frequently used transformation patterns. Finally, we create extensions to the AMW tool to handle the execution and combination of different matching transformations.

This chapter is organized as follows. Section 6.2 presents a motivating example that we use as a guide for presenting our approach. Section 6.3 presents the general overview of our approach. Section 6.4 presents the matching transformations in more detail. Section 6.5 presents how we extended the AMW tool to support the execution of matching transformations. Section 6.6 presents a general discussion. Section 6.7 concludes.

6.2 Motivating example

We motivate the necessity of using semi-automatic methods to create model transformations using two simple metamodels *MM1* and *MM2*. Both metamodels are illustrated in Figure 6.1. They describe the teachers and the students of different educational institutions. These metamodels have similar attributes and references, but they are organized differently.

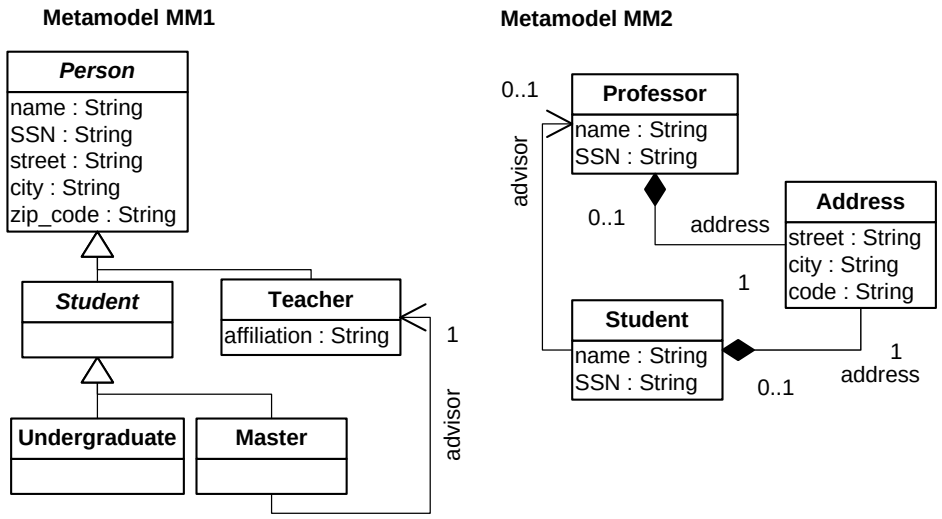


Figure 6.1 Two simple metamodels

Metamodel *MM1* contains an abstract class *Person*, with attributes *name*, *SSN (Social Security Number)*, *street*, *city* and *zip_code*. The class *Teacher* inherits from *Person*, and it has the *affiliation* of the teacher. *MM1* has two types of students: undergraduate students (*Undergraduate*) and master students (*Master*). Only master students have an *advisor*. Metamodel *MM2* does not support inheritance. *MM2* contains a class *Professor* and only one class *Student*. The presence of an advisor indicates if the student is undergraduate or master. The address of the professors and the students is factored out on the class *Address*.

In Figure 6.2 we show an ATL transformation used to transform models conforming to *MM1* (i.e., source model) into models conforming to *MM2* (i.e., target model)[2]. This transformation has 3 rules; each rule matches one element of the source model and creates elements in the target model. The transformation developer must know that *Teacher* is transformed into *Professor* and that *Master* and *Undergraduate* are transformed into *Student*. After that, all the attributes and references of each class must be translated as well (*name*, *SSN*, *address*, *advisor*, *street*, *code*, etc.).

```
rule CreateProfessor {
  from source : MM1!Teacher
  to target : MM2!Professor (
     name <- source.name,
     SSN <- source.SSN,
     address <- address ),
  address : MM2!Address (
     street <- source.street,
     city <- source.city,
     code <- source.zip_code )
}
```

```
rule CreateStudent1 {
  from source : MM1!Undergraduate
  to target : MM2!Student (
-- copy bindings from CreateProfessor
  )
}
rule CreateStudent2 {
  from source : MM1!Master
  to target : MM2!Student (
    advisor <- source.advisor
-- copy bindings from CreateProfessor
  )
}
```

Figure 6.2 ATL transformation

This transformation has basically two kinds of expressions: transformations between self-contained elements (i.e., classes), and the setup of their properties (i.e., attributes and references). Thus, in the three rules, the transformation has a source class and a target class. The rule *CreateProfessor* assigns the attributes of *Teacher* to *Professor*. These attributes are inherited from *Person*. The attributes from both classes have similar properties, such as name and type. These attributes are transformed in the containing class, or in a newly created class (*Address*). The same set of expressions must be rewritten in *CreateStudent1* and in *CreateStudent2* rules, because *Undergraduate* and *Master* inherit from *Student*, that inherits from *Person*. The transformation developer has two choices: to copy and paste the code, or to apply rule inheritance predicates.

These expressions are common patterns in transformations that involve similar metamodels, for example in data interoperability or in model evolution scenarios. These transformations can be very large depending on the source and target metamodels. The automatic discovery of these transformation patterns can increase the development speed of model transformations. The intervention of qualified transformation developers is left essentially to more complex expressions that do not occur frequently and that cannot be created automatically.

In order to automate the development of transformations, it is necessary to create the different kinds of relationships (links) between metamodel (or model) elements. These links must be saved in a weaving model. A weaving model can be validated or modified by the transformation developer.

Techniques similar to ontology and schema matching can be used to discover these links; for instance, to assign a similarity value to a link between elements with the same name. However, model transformations can be executed over several different source and target metamodels, with different attributes, relations and properties. The patterns applied vary from case to case. Consequently, it is very important to have efficient ways to implement new algorithms or heuristics and to adapt existing ones. As a final step, these links must be translated into the correct transformation expressions, for instance links between attributes of abstract classes must be translated into bindings (a binding is denoted by the "←" symbol) in the inherited classes. This should be done by implementing a *TransfGen* transformation.

[2] The target and source models are terminal models

6.3 General overview

This section presents an overview of the components of the general matching process). The main goal is to semi-automate the matching process, and consequently, the production of transformations. The components are illustrated in Figure 6.3.

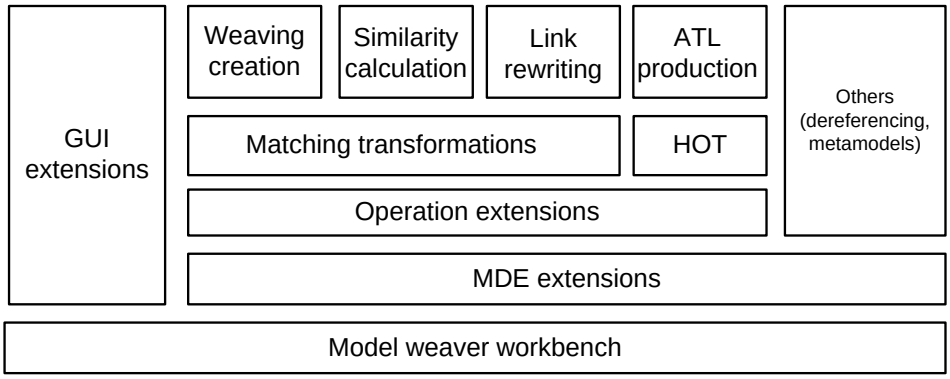


Figure 6.3 General overview of the matching components

These components are built on top of the model weaver workbench (AMW) presented in Chapter 4. The workbench provides the base weaving platform. There are two kinds of extensions, GUI and MDE extensions. Initially, we concentrate on the MDE extensions, and, more specifically, on the operation extensions. We implement different model management operations using model transformations. Thus, the operation extensions wrap a model transformation engine to be able to execute model transformations. Each model transformation corresponds to a different model management operation. There are two main kinds of transformations: matching transformations and higher-order transformations.

The matching transformation extension implements different methods that interpret the structure of the input model elements to create weaving models. There are three kinds of matching transformations. The first kind creates a weaving model with links between the elements of the input models. However, it is not possible to create a weaving model with only correct links between the model elements in a single transformation. For instance, we create links between *name-name* attributes or even *name-SSN*. These links are refined by other matching transformations. The second kind of matching transformations calculates a similarity value between every linked element. These transformations implement different matching techniques (we explain them in the subsequent sections). In this case, the *name-name* link may have a higher similarity value than *name-SSN* link. The third kind of matching transformation selects the links with higher similarity values to produce a weaving model with only a subset of links. For instance, we select only the *name-name* links. After the execution of these transformations, the weaving model can be manually modified in the weaving tool.

The ATL production extension enables the execution of higher-order transformations that produce the data interoperability transformations. These HOT's are encapsulated in the *TransfGen* model management operation. In other words, the weaving models are transformed into transformation models. The transformation model can be extracted into a textual language, for instance ATL or XSLT.

This pluggable architecture allows adding different matching or HOT transformations. These transformations can use different techniques, and can support different extensions to the core weaving metamodel.

6.4 Matching transformations

In this section we present in detail our solution for establishing links between model elements.

Definition 6.1 (Matching). Matching is the process of establishing relationships between elements belonging to different models.

The matching process uses different techniques to create links between a set of model elements. We define one generic model management operation for each different heuristic or algorithm. The goal is to discover the relationships between a set of input models and to create a weaving model. The whole process is encapsulated in an operation called *Match*. The *Match* operation takes two models M_a and M_b as input and produces a weaving model M_w as output. M_a and M_b conform to MM_a and MM_b; M_w conforms to MM_w.

$$M_w : MM_w = Match\ (M_a : MM_a, M_b : MM_b).$$

We implement the match operation using model transformations. This means that the matching techniques are implemented as domain-specific model transformations. These domain-specific transformations are called *matching transformations.*

Definition 6.2 (Matching transformation). A matching transformation is a domain-specific transformation T that takes two or more models as input, and that transform them into a new weaving model M_W.

$$< OUT_1 : MM_{OUT1}, \ldots , OUT_n : MM_{OUTn} > = T\ (< IN_1 : MM_{IN1}, \ldots , IN_m : MM_{INm} >)$$

A matching transformation implements different methods that produce weaving models. We may consider that the set of input models are *transformed* in a weaving model.

The whole process of creating weaving models is semi-automatic, i.e., it is an interactive process that alternates between the automatic execution of matching transformations and the manual refinement of weaving models in the weaving engine. We explain the different kinds of matching transformations in the following sections. First, we describe a simple metamodel extension that is used by these transformations. Then, we describe the matching transformations using the ATL language.

6.4.1 Metamodel extensions

The weaving metamodel specifies the different kinds of links that are generated by the matching transformations. Each kind of link corresponds to one transformation pattern. The weaving metamodels are created as extensions of the core weaving metamodel. It is necessary to define specific matching extensions to be able to execute the matching transformations. For instance, one of the most common patterns of declarative transformation rules is to select an element of a given type in a source model and to create a new element in a target model. These metamodel extensions are shown in Figure 6.4.

The class *Element* is a concrete extension of *WLinkEnd*. It can refer to any kind of (meta)model element. The class *Equivalent* contains two references to save the *source* and *target* elements. The class *Equivalent* has a similarity value that is calculated in the matching transformations. This value is a numeric value that measures the semantic proximity of the linked elements. The other classes capture five different transformation patterns:

- **Generic equality**: the class *Equal* indicates that the linked elements represent the same information. The <*Type*> tag must be replaced by the element type, for example *AttributeEqual* or *ReferenceEqual*.

- **Attribute to references**: the class *AttributeToRef* captures links between attributes in the source model and references in the target model. The *targetAttribute* contains an attribute of the element referred by the *target* reference.
- **Element inheritance**: the class *ElementInheritance* relates elements that inherit from others. The reference *super* points to the parent element of a given element.

```
class Element extends WLinkEnd {
}
class Equivalent extends WLink {
    attribute similarity : Double;
    reference source container : Element;
    reference target container : Element;
}
class <Type>Equal extends Equivalent {
}
class AttributeToRef extends Equivalent {
    reference targetAttribute container : Element
}
class ElementInheritance extends Equivalent {
    reference super container : WLink;
}
```

Figure 6.4 Matching extensions

6.4.2 Creating weaving models

Transformations that create weaving models are the first kind of matching transformations that are executed. The transformation that creates weaving models is called *CreateWeaving*. The transformation takes two models M_a and M_b as input and transforms them into a weaving model M_w. M_a conforms to MM_a, M_b conforms to MM_b and M_w conforms to MM_w.

$M_w : MM_w = CreateWeaving\ (M_a : MM_a, M_b : MM_b)$.

This transformation matches a set of elements of a given type of M_a with a set of elements of a given type of M_b. It creates a restricted Cartesian product $M_a \times M_b$, i.e., it creates a link between every pair of elements. However, the execution of a Cartesian product can create too many elements if the input models are large. Consider for example two input models with 100 elements each. The Cartesian product would create a weaving model with at least 100 x 100 = 10.000 elements. These elements are captured in extensions of *WLinkEnd*. Moreover, there is one additional element containing the linking semantics (an extension of *WLink*). For that reason we use restricted versions of the Cartesian product that take into account the type of the elements.

Figure 6.5 illustrates how the transformation can be implemented using a generic transformation rule. MM_a and MM_b denote the input metamodels. MM_w denotes the output weaving metamodel. This rule matches all elements of type <*TypeA*> with all elements of type <*TypeB*> and it produces an equivalence link (*Equivalent*) between the source and target elements.

```
rule CreateLink {
    from
          aSource : MMa!<TypeA>, aTarget : MMb!<TypeB>
    to
          alink : MMw!Equivalent (
                source <- aSource ,
                target <- aTarget
          )
}
```

Figure 6.5 Creation of equivalence links

Consider the case of a matching transformation between the metamodels from the Professor/Student example from section 6.2. The transformation depicted in Figure 6.6 contains only the guard, the input and the output patterns. Rule *CPClass* is matched only for every pair of *EClass*. It creates *ClassEqual* links. The rule *CPAttr* matches for every pair of *EAttribute*, and creates *AttributeEqual* links. The guard of *CPAttr* can also be more restrictive to match only the pair of attributes that have links between the containing classes. These different restrictions allow creating more performing methods, according to the application requirements and resources.

```
rule CPClass  {
   from
        left : Ecore!EClass, right : Ecore!EClass
    to
        AMW!ClassEqual
}
rule CPAttr {
   from
        left : Ecore!EAttribute, right : Ecore!EAttribute
    to
        AMW!AttributeEqual
}
```

Figure 6.6 Rule that creates links between classes and/or attributes

The operation can also be modified to update weaving models (to create or to remove other links). In this case it has a weaving model as an extra input parameter.

$M_w : MM_w = CreateWeaving\ (M_a : MM_a, M_b : MM_b, M_w' : MM_w)$.

The use of matching transformations enables the customization of the implementation. It is possible to change the types of the left or of the right elements. This allows establishing correspondences between elements of terminal models conforming to different metamodels (i.e., not only between elements of metamodels). This is a typical requirement when it is necessary to produce weaving models as input for *merge* or *diff* operations. For instance, consider a matching transformation using *KM3* and *SQL-DDL* metamodels. The ATL guard shown in Figure 6.7 enables the creation of equivalence (or other) links between a KM3 *Class* and a SQL-DDL *Table*.

```
rule CPClassTable  {
    from
        left : KM3!Class, right : SQLDDL!Table
      ...
}
```

Figure 6.7 Rule that create links between models conforming to different metamodels

6.4.3 Calculating element similarity

The second kind of matching transformation calculates a similarity value between the elements referred by the *source* and *target* references of the equivalence links. This similarity value is used to evaluate the semantic proximity between the linked elements. A link with a high similarity value indicates that there is a good probability that the source element must be translated into the target element.

We define a transformation called *AssignSimilarity*. The transformation takes a weaving model M_w' and a *weight* as input, and it produces a weaving model M_w as output. The input and the output models conform to the same weaving metamodel MM_w. The output weaving model has the new similarity values. However, there are many different methods to calculate similarity values. The tag *<method>* indicates the method that is implemented.

$M_w : MM_w = AssignSimilarity<method> (M_w': MM_w, weight: double)$.

The *weight* parameter is used to restrict the similarity values between [0-weight]. This parameter enables to adjust the impact of a given similarity method. For instance, a similarity method that compares element's names may have weight 0.8, and a similarity method that compares element's types may have weight 0.2. This means that a set of elements is considered more similar if they have the same name than the same type. Different matching transformations can be executed to obtain a more accurate similarity value. We implement element-to-element and structural methods. We explain them below.

6.4.3.1 Element-to-element similarities

Element-to-element similarities are calculated taking the *source* and *target* elements of an *Equivalent* link and comparing the element properties in different ways. We develop different matching transformations, each one implementing a different method.

- *String similarity*: the names of the model elements are considered strings. The names are compared using string comparison methods such as Levenshtein distance, n-grams and edit distance [33].
- *Dictionary of synonyms*: the names are compared using a dictionary of synonyms (we use *WordNet* [57]). This dictionary provides a tree of synonyms. The similarity between two terms (element names) is calculated according to the distance between these terms in the synonym tree. This way it is possible, for example, to increase the similarity value between elements such as *Teacher* and *Professor*, which does not yield good results if using string comparison methods.

Several element-to-element techniques are already implemented and available in public APIs. Thus, we extend the ATL transformation engine to be able to call methods from external APIs. The transformation engine provides wrapper methods that can be applied to every model element. This way we use APIs such as the SimMetrics API [128], which contains string similarity methods, and the JWNL API [76], which accesses the WordNet database.

The ATL rule shown in Figure 6.8 calculates the name similarity between two attributes. This rule considers that the input model is a weaving model that contains *AttributeEqual* links. It updates the similarity value by executing the *similarityName* helper. This helper calls the *Levenstein* string comparison method of the JWNL API. The final result is multiplied by the weight parameter. To compare the code complexity, we implemented this same rule using only Java. The Java class has approximately 250 lines. This is essentially due to all the navigation code used to find the correct model elements.

```
rule AttributeSimilarity {
    from
          mmw : AMW!AttributeEqual
    to
          alink : AMW!AttributeEqual (
                similarity <- (mmw.similarity +
                            mmw.left.similarityName(mmw.right)) * weight
                )
```

Figure 6.8 Simple element-to-element similarity rule

6.4.3.2 Structural similarity

Structural similarities are calculated using the internal properties of the model elements (e.g., types and cardinality) and the relationships between model elements (e.g., containment or inheritance trees).

We implement a structural method called *metamodel-based similarity*. The metamodel-based similarity method is executed after an element-to-element method to improve the accuracy of these methods. The metamodel-based method calculates the similarity using the internal properties and the relationships between model elements.

- **Internal properties**

Model elements have a set of properties, such as type, cardinality, order and length, etc. Consider two model elements $a \in M_a$ and $b \in M_b$; M_a and M_b are different models, but conform to the same metamodel. A matching transformation compares the properties of a with the properties of b. If a given property has the same value, it adds 1(one) to a temporary similarity value. This temporary value is multiplied by the *weight* parameter and added to the initial similarity value. However, this generic comparison is valid only if M_a and M_b conform to the same metamodel. When the metamodels are different, the operation is adapted for every different property.

Consider two different metamodels, KM3 and SQL-DDL (the complete metamodels can be found in the AM3 Zoo [3]). We consider two elements from these metamodels, *Attribute* from KM3 and *Column* from SQL-DDL. An *Attribute* has properties such as *type, lower, upper, isOrdered, or isUnique*. A *Column* has the following properties: *default, type, keys, canBeNull*. These properties cannot be directly compared if using a generic implementation, because their values are not compatible and there is no name equivalence. For example, the transformation must take into account that *canBeNull* is a *Boolean*. The same information is captured analyzing the value of *lower* property. We illustrate the transformation rule for this case in Figure 6.9.

This rule calculates the similarity between KM3 and SQL-DDL elements. It selects an *Equal* link that satisfies the following condition: the *source* reference points to an *Attribute* of KM3, and the *target* reference points to a *Column* of SQL-DDL. The helper *requiredSim* compares the *required* property with the *CanBeNull* property, and returns one (1) if they satisfy the equality criteria.

```
rule UpdateStructuralSim {
    from
          mmw : MMw!Equal mmw.source.isTypeOf(KM3!Attribute)
                and mmw.target.isTypeOf(SQLDDL!Column))
    to
          alink : MMw!Equal (
                similarity <- ( mmw.similarity +
                mmw.source.requiredSim( mmw.target )) * weight
          )
}

helper context KM3!Attribute def: requiredSim
    (column : SQLDDL!Column) : Real =
           if (self.lower = 0 and column.canBeNull) then
                1
          else
                0
          endif;
```

Figure 6.9 Structural similarity rule

- **Element relationships**

There are different kinds of relationships between elements of the same model, for instance, containment or inheritance relationships. Most existing structural methods that exploit the element relationships rely on the following assumption: if two model elements are similar, the neighbors of these elements are likely to be similar as well. For example, if a link between two attributes of two different models has a high similarity value, the containing classes of these attributes have a good probability to be similar.

We create a transformation inspired by the Similarity Flooding (SF) algorithm [101]. We first explain the key idea of SF, and then how we change it. Consider two input metamodels M_a and M_b, and the model elements $a, a' \in M_a$ and $b, b' \in M_b$. Elements a and a' are connected by a labeled edge *(a, "containment", a')*. Elements b and b' are connected by a labeled edge *(b, "containment", b')*. Initially, the algorithm executes a Cartesian product $M_a \times M_b$ and assigns a similarity value for every pair of elements. Consider the pairs *(a, b)* and *(a', b')*, with similarities x and y, respectively. The key idea of SF is to propagate the similarity value between the pair of elements that are connected by edges with the same label. In other words, it propagates x to *(b, b')* and it updates the similarity value y. The propagation is done by the formula: $y = y + (p * x)$. The value p is calculated based on the number of edges connecting a given pair of elements (i.e., the number of neighbor elements). For instance, if *(a, a')* has 10 neighbors, then $p = 1/10$. This propagation information is encoded in a propagation graph. The propagation can be done in both directions.

We explain how we implemented and adapted this solution using matching transformations. The main advantage is the possibility of having different forms of propagation based on different structural or semantic relationships between the elements of the input metamodels, and not based uniquely on the value of the label of the edges. This assumption is too restrictive, because it cannot capture different relationships between the elements. In contrast, it is also too generic, because we cannot create application-specific propagation models.

The propagation graph is encoded in a weaving model, called the weaving propagation model. The weaving model conforms to the metamodel extension shown in Figure 6.10. The class *WAssociation* is an abstract class that depicts relationships between extensions to *WLink*s within the same weaving model. The class *PropagationElement* has two references: *outgoingLink* refers to the link with the

source similarity value, and *incomingLink* refers to the link with the target similarity value. The *propagation* attributes contains a value that is multiplied with the similarity value of the *outgoingLink*.

```
package mw_core {
      class WAssociation extends WElement {
      }
}
package mmw_propagation {
      class PropagationElement extends WAssociation {
            reference incomingLink : Equivalent;
            reference outgoingLink : Equivalent;
            attribute propagation : Double;
      }
}
```

Figure 6.10 Weaving propagation metamodel extension

Our approach allows constructing different propagation models according to the application scenario, and also to propagate similarities between elements conforming to different metamodels. An important issue is the creation of relevant propagation elements and values between a set of links. We show a generic transformation rule in Figure 6.11. This rule assumes that the input model of the transformation is a weaving model that contains a set of links and similarity values.

```
rule CreatePropagationElement {
      from
        source_link : AMW!Equivalent,
        target_link : AMW!Equivalent (
          <semantic guard>
        )
      to
          out : AMW!PropagationElement (
            propagation <- 1 / <propagation_value>,
            outgoingLink <- source_link,
            incomingLink <- target_link
          )
    }
```

Figure 6.11 Creation of propagation edges

The rule input pattern matches two links. These links are extensions of *Equivalent* links. The source link contains the similarity value that is propagated. The target link contains the similarity that is updated. This means the similarity is propagated from the source link to the target link. The *<semantic guard>* determines the condition that must be filled to create a propagation element (we show different semantic guards later). The rule creates a propagation element, and it assigns the source and target links to the corresponding references. The propagation value is calculated in this rule.

We develop three different kinds of propagation based on this generic rule. We illustrate our approach assuming that the input metamodels conform to KM3. However, the rules can be adapted to match different metamodels or models.

Containment-tree propagation: the containment-tree propagation method enables propagating the similarity between elements that have containment relationships, for instance classes and attributes or classes and references (note that this is not the containment between classes, but between classes and its members). Consider for example a KM3 *Class*. The reference *structuralFeatures* points to classes *Reference* and/or *Attribute*. We create propagation elements from the links between classes

(*ClassEqual*) to the links between its attributes. The guard of the rule is shown below. The *getReferredLeft/Right* is a helper that returns the element of the input metamodel.

```
source_link : AMW!ClassEqual,
target_link : AMW!AttributeEqual (
  target_link.getReferredLeft.owner = source_link.getReferredLeft
  and
  target_link.getReferredRight.owner = source_link.getReferredRight
)
```

The link between classes is assigned to the *outgoingLink*, and the link between the attributes is assigned to the *incomingLink*. In the same way as the SF algorithm, we consider that a given method can contribute to a maximum similarity value of 1. Consequently, the propagation is one (1) divided by the multiplication of the total number of attributes of the two input classes. It is also possible to propagate the similarity from the attribute's links to the class' links. To do that, we create a propagation element with inverted incoming/outgoing links and a new propagation value. The *getAttributeCount()* method returns the number of attributes of a given class.

```
outgoingLink <- source_link,
incomingLink <- target_link
propagation <-
   1 / ( source_link.getReferredLeft.getAttributeCount()->size() *
         source_link.getReferredRight.getAttributeCount()->size()
 )
```

Relationship-graph propagation: this propagation method takes into account the type of the references of two given classes. For instance, consider the links between classes *(a,b)* and *(c,d)*; *a* has a reference to *c* and *b* has a reference to *d*. The relationship-graph is used to propagate the similarity between these two links. The ATL guard for this method is shown below. The *getReferences()* method returns a set with all the references of the class.

```
source_link : AMW!ClassEqual,
target_link : AMW!ClassEqual (
   target_link.getReferredRight.getReferences()->
      exists( e | e.type = source_link.getReferredRight) and
  target_link.getReferredLeft.getReferences()->
      exists( e | e.type = source_link.getReferredLeft)
)
```

The maximum propagation value (1) is divided by the multiplication of the number of references of these two classes, in a similar way as for the containment tree propagation.

Inheritance-tree propagation: this method enables propagating the similarity value from the link between two source classes to links between the parent classes of the source classes, if any. It can be considered as an extension to the relationship tree propagation method. However, it takes into account only the references that represent inheritance relationships. In KM3, this reference is called *supertypes*.

```
source_link : AMW!ClassEqual,
target_link : AMW!ClassEqual (
  target_link.getReferredRight.supertypes->
      exists( e | e = source_link.getReferredRight) and
  target_link.getReferredLeft.supertypes->
      exists( e | e = source_link.getReferredLeft)
)
```

The source link is assigned to the reference *outgoingLink*, and the target link is assigned to the reference *incomingLink*. The propagation value is calculated based on the multiplication of the number of super-types of the source classes.

```
propagation <- 1 /
   (source_link.getReferredLeft.supertypes->size() *
    source_link.getReferredRight.supertypes->size())
```

These propagation elements are created in the same weaving model. However, it is also possible to have separate weaving models that are used with specific input models. For example, the inheritance tree propagation is not relevant when creating a weaving model between SQL-DDL models that do not have native inheritance relationships. Thus, this propagation method is not used in this particular matching scenario.

These structures can be used to propagate the similarity between elements of different metamodels as well. Consider again the SQL-DDL and KM3 metamodels. The containment trees from both metamodels are different. However, the containment relationship between a *Table* and a *Column* is equivalent to the relationship between a *Class* and an *Attribute*. The matching transformations enable to build a containment tree of these two metamodels.

Once the propagation model is created, the similarities are propagated. The SF-based propagation is implemented with an ATL rule, as shown in Figure 6.12.

```
rule PropagationClass {
from
  mmw : AMW!Equivalent
to
  alink : AMW!Equivalent()
do {
  thisModule.aTuple <- AMW!PropagationElement.allInstances()->
    select ( e | e.incomingLink  = mmw)->
        iterate (e1; acc : TupleType(value : Real, count : Integer) =
          Tuple {value = 0, count = 0} |
            Tuple {
              value = acc.value +
                      (e1.outgoingLink.similarity * e1.propagation),
              count = acc.count + 1
            }
      );
  alink.similarity <- mmw.similarity +
            thisModule.aTuple.value / thisModule.aTuple.count;
}
}
```

Figure 6.12 Propagation in ATL

The goal is to update the similarity value of every link in the weaving model. Thus, this rule matches every link in the model. Then, in the *do* block, it selects all *PropagationElement* (through the *allInstances()* method) that have the *incomingLink* equals to the current link. For every

PropagationElement, it multiplies the similarity value of the *outgoingLink* by the propagation value, and adds it into an accumulator (the *Tuple*). The accumulator also counts the number of propagation elements that refer to the current link. The accumulated similarity value is divided by the number of outgoing links and then added with the current similarity value of the link. The division enables more coherent global similarity estimation. For instance, without this division, elements that are connected by several propagation edges may have a similarity value that is too high.

6.4.4 Selecting best links

The third kind of matching transformations selects only the links that satisfy a set of conditions; for instance, a given similarity threshold. The selected links are included in the final weaving model or rewritten into different kinds of links. These matching transformations are generalized by the operation *Select<method>*.

$M_w : MM_w = Select<condition> (M_w' : MM_w)$.

The operation takes a weaving model M_w' as input and produces another weaving model M_w as output. Both weaving models conform to the same weaving metamodel MM_w. The *condition* tag denotes the selection criteria. Links are selected using two methods: link filtering and link rewriting. These methods are explained below.

6.4.4.1 Link filtering

There are different kinds of link filtering methods. The most simple method (and also most used) is to set up a minimum threshold value and to select only the links that have a similarity value higher than this threshold. The biggest drawback of this method is the choice of a correct threshold method. Creating a new weaving model based on low threshold values may yield too many false links, i.e., that should not be created. In contrast, too high threshold values may filter relevant links.

In typical data interoperability scenarios, a common method is the selection of links with the highest similarity values for every source element. This method usually yields good results because data interoperability transformations need to translate all the elements of a source model (or as most as possible) into a target model. Thus, it is necessary to obtain a link between every element of a source metamodel with the elements of a target metamodel.

We illustrate a matching transformation rule in Figure 6.13.

```
rule getMaxLink (aSource : MMa!ModelElement) {
    using {
          newLink : MMw!Equivalent = null;
          maxSim : Real = 0;
    }
    do {
       for(e in MMw!Equivalent.allInstaces()->select(e.source=aSource)){
       if (e.similarity > maxSim) {
          maxSim <- e.similarity;
          newLink <- e;
       }
    }
    return newLink;
}
```

Figure 6.13 Link filtering method

This rule is executed for all the source elements. It loops over all the equivalence links (and inherited links) of a given source element and it selects the link that has the highest similarity value.

The *using* part declares variables to store an auxiliary similarity value and the selected link. The *allInstances()* method returns all the instances of links conforming to the *Equivalent* link. The *for* block selects the links that have the *source* reference equal to the *aSource* parameter. The similarity values are compared with a current similarity value. The maximum value and the corresponding link are stored in the auxiliary variables.

The output weaving model contains one link for each element of the source model. This rule selects all the elements from the source metamodel, but the same target element may be selected several times. The last adjustments are done by link rewriting methods.

6.4.4.2 Link rewriting

Link rewriting methods are executed after the execution of selection or filtering methods. These relationships are used to transform simple links (e.g., *Equivalent*, *Equal*) into complex kinds of links that capture different transformation patterns. Common patterns are nesting, inheritance, data conversions, concatenation and splitting. For instance, if more than one source element is linked with the same target element through *Equal* links, this link can be rewritten as a *Concatenation* link. The most common form of link rewriting is the nesting between elements with containment relationships, for example classes and attributes, or tables and columns.

Consider a weaving model that links two KM3 metamodels, MM_a and MM_b. After the execution of a link filtering transformation, it contains a set of links between classes (*ClassEqual*) and attributes (*AttributeEqual*). However, they are children of the root element. Now consider classes $A \in MM_a$ and $B \in MM_b$, attributes $a \in A$, $b \in B$, links *ClassEqual (A, B)* and *AttributeEqual (a, b)*. Since *a* is an attribute of *A* and *b* is an attribute of *B*, the *AttributeEqual* link is rewritten as a link child of *ClassEqual*. Note that the rewriting is not based on the similarity values.

We illustrate the rewriting of nested links in Figure 6.14. This rule matches *AttributeEqual* and *ClassEqual* links at the same time and it checks if the *owner* of the attribute is the current element. If the result is true, it executes the rule and assigns the *class_link* element to the *attr_link.parent* reference. The output weaving model preserves the containment relationships between classes and attributes.

```
rule NestedRewriting {
    from
          attr_link : MMw!AttributeEqual,
          class_link : MMw!ClassEqual (
                attr_link.source.owner = class_link.source and
                attr_link.target.owner = class_link.target
          )
    to
          link : MMw!AttributeEqual (
                parent <- class_link
    )
}
```

Figure 6.14 Rewriting of attribute-equal links

These transformations are executed always after the calculation of some similarity estimation. The different guards in the transformation rules match the existing links. The *to* part recreate the same links (to copy them) or create new kinds of complex links. Link rewriting transformations are closely related to the application scenario. This means these methods are less generic than similarity calculation methods.

In addition to the creation of complex links, link rewriting transformations can create links that record different kind of information about the overall matching process. After the execution of a set of matching transformations, it is normal that some elements of the source metamodel are not linked with any element of the target metamodel, and vice-versa. We create a link rewriting transformation that enables to record the source and/or target elements that are not referenced by any link. This kind of link can be used for different purposes: to verify if the resulting weaving model is correct, to record which elements cannot be translated from one model to another, or to use them as input to model difference algorithms.

Figure 6.15 depicts an extension to the core weaving metamodel that record elements that are not linked. The class *NotFound* (extension to *WLink*) has two references, *left* and *right*. These references points to a class that contains a list of elements from the source (*left*) or target (*right*) models.

```
class NotFound extends WLink {
      reference left container : ListNotFound;
      reference right container : ListNotFound;
}
class ListNotFound extends WLink {
      -- @subsets end
      reference element [*] container : ReferredElement;
}
```

Figure 6.15 Metamodel extension for elements not linked

The *NotFound* links are created by the matching transformation rule shown in Figure 6.16. The guard of this rule checks if the matched left element *mmw* is referenced by some link endpoint *LeftElement* from the left woven model *leftM*. If it is not referenced, a new link endpoint *LeftElement* is created and it is added into a global list of the elements not linked.

```
rule NotLinked {
  from
      mmw : AMW!ElementRef (
      if mmw.modelRef = mmw.modelRef.refImmediateComposite().leftM then
        not AMW!LeftElement.allInstancesFrom('IN')->
          exists(e | e.element.ref = mmw.ref)
      else
         false
      endif
      )
  to
      left : AMW!LeftElement (
            element <- mmw
      )
      do
         {
            thisModule.notFound.left->
                  first().element <- thisModule.notFound.left->
                  first().element->union(Sequence{left});
      }
}
```

Figure 6.16 Link rewriting method

6.5 Extending the AMW tool to support matching transformations

In this section, we present the extensions to the AMW tool that enable the integration and the customization of different matching transformations. The MDE extensions enable executing different kinds of transformations, matching or higher-order transformations. The matching transformations have a pre-defined signature. These transformations take a weaving model, a source terminal model (or metamodel), and a target terminal model (or metamodel) as input, and they produce a new weaving model as output. The matching transformations can be executed using two different settings.

- First, the transformations are executed one by one using a context menu in the weaving panel. This menu is automatically generated based on the declaration of an operation extension. The extension is specified in XML files. The XML file specifies the transformation path, the headers (input and output parameters), and the text that appears on the menu.
- Second, we implement a GUI extension that enables customizing the available matching transformations and executing them in a single step. This allows combining different transformations and setting up different execution parameters.

The parameterization of the matching transformations is defined in a configuration model. This configuration model contains parameters such as *weight* or *threshold*. This model conforms to a match parameter metamodel. This metamodel specifies the transformations that are executed, the execution order, and a set of tuning parameters. The match parameter metamodel is illustrated in Figure 6.17.

```
package match_parameter {
      abstract class NamedElement {
            attribute name : String;
      }
      class ParameterSet extends NamedElement {
            reference transformations[*] ordered container: Transformation;
            reference metamodels [*] container : Metamodel;
      }
      abstract class Transformation extends NamedElement {
            reference metamodels [*] : Metamodel;
            attribute description : String;
            attribute selected : Boolean;
            reference depends [*] : Transformation;
      }
      class LinkGeneration extends Transformation {
      }
      class ElementToElement extends Transformation {
            attribute weight : Double;
      }
      class Structural extends Transformation {
      }
      class Filter extends Transformation {
            attribute threshold : Double;
      }
      class Metamodel extends NamedElement {
            attribute description : String;
      }
}
```

Figure 6.17 Match parameter metamodel

The class *ParameterSet* contains a set of transformations (ordered) and the set of metamodel extensions. The class *Transformation* defines the standard attributes for every transformation. A transformation is executed if the *selected* attribute is set to *true*. The reference *metamodels* contains

the metamodel extensions that need to be loaded to be able to execute a transformation. The reference *depends* indicates that one transformation can be executed only if a depending transformation is previously executed. For instance, the similarity flooding transformation cannot be executed if the propagation model is not previously created. There are four types of transformations. They correspond to the different kinds of matching transformations presented in this chapter: *LinkGeneration*, *ElementToElement*, *Structural*, and *Filter*.

The GUI extension interprets the models conforming to this metamodel and it produces a generic configuration window, as illustrated in Figure 6.18.

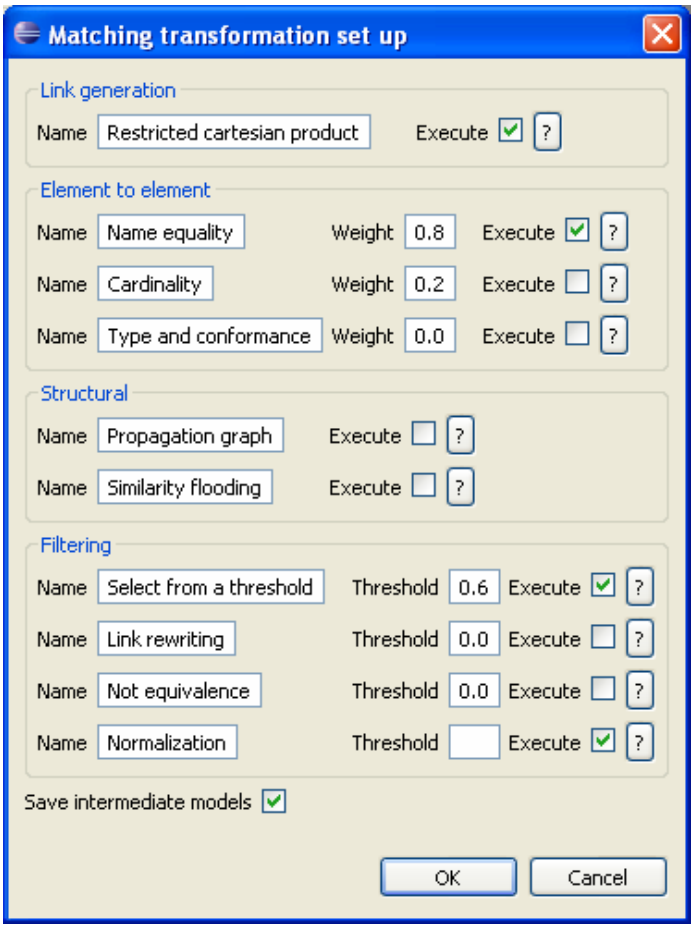


Figure 6.18 Matching transformation configuration

The configuration window has one group for each different kind of transformation. Each group shows the set of available parameters. The "? (question mark)" button shows the dependencies between the transformations and the metamodels. The "Save intermediate models" button saves a new weaving model after the execution of each matching transformation. This enables the comparison of the intermediate results. The configuration model loaded in this window is only illustrative. It executes the following transformations: a restricted Cartesian product based on the type of elements; a comparison over the elements names, with a weight of 0.8; a comparison over the elements cardinality, with weight 0.2; a selection of the links with similarity value higher than 0.6; and the normalization of the results.

The configuration metamodel and model brings some advantages when combining the execution of different matching transformations. The correct tuning of matching transformations requires a lot of experience on matching. This configuration model enables recording these parameters, and can be

reused later by other developers. The tuning of matching transformations is a subject of study by itself. Each application scenario can have different parameters with the objective to obtain the best matching results. The variation of a single parameter can modify the final generated weaving model. Consequently, the exchange of these configuration models among the developers may help to better tune their heuristics and environments.

6.6 General discussions

We use two variations of the Professor/Student example to execute a set of matching transformations. In this section, we concentrate on this illustrative example to show the effectiveness of our proposition[3]. In the first example, *MM1* and *MM2* conform to KM3. In the second example, *MM1* conforms to KM3 and *MM2* conforms to SQL-DDL. The weaving models are translated into model transformations. The goal is to verify if the transformations are generated correctly, and to verify if the matching transformations can be easily adapted in both examples.

The Professor/Student example has different transformation patterns, such as class inheritance, nesting of elements, or classes with different names. *MM1* contains 17 elements and *MM2* contains 18. The creation of links between every model element without any type restriction yields a weaving model with 950 elements: 306 links, plus one right and one left element for each link, i.e., 3 x 306, plus additional control elements. It is important to reduce the number of initial links as early as possible in the process, to be able to scale up the approach to match larger models.

The weaving model with the type-restricted Cartesian product contains 273 elements, with 78 links. The name similarity transformation enables the creation of links between elements such as *SSN-SSN*, *name-name*, or *zip_code-code*. The dictionary of synonyms increases the similarity of elements such as *Professor* and *Teacher*, *Master* and *Student*. The containment tree and inheritance relationships enable the propagation of the similarity of the attributes of *Master* and *Student*.

We execute the propagation of similarities two times. The propagation of the similarities more than two times increases the similarity between the classes (e.g., *Teacher* and *Professor*), but the values are not significantly different in our example. Several propagation steps may be more useful in the case of model comparison, where more accurate values are necessary.

The creation of links and the computation of similarities can be applied for more generic examples, not only to generate integration transformations. On the other side, the link filtering and rewriting methods are more specific to the type of the output.

Consequently, link selection methods are very important to obtain the final integration transformations. For example, the similarity between the abstract class *Person* and class *Professor* is high. This would produce a rule that transforms *Person* into *Professor*. The filtering method does not select links with abstract classes. Then, the link rewriting method copies the bindings of the attributes of the class *Person* to the rules that transform the inherited classes, i.e., *Master*, *Teacher*, *Undergraduate*.

The only link that is not generated correctly is the one between *Undergraduate-Student*. This is because none of the initial similarity methods can find high similarity values. The values are not propagated, because the inheritance relationships exist only in the source model. We thus use the weaving engine to modify the weaving model. After applying all the transformations and using the weaving engine, the weaving model is reduced to 78 elements, with 12 links.

Finally, the weaving model is used as input to higher-order transformations, following the pattern presented in Chapter 5. We created a HOT with 250 lines. It is relatively complex compared to the generated transformation, with only four ATL rules. However, this HOT and the matching transformations are implemented to be used many times in different applications.

[3] The execution of matching transformations using bigger metamodels is presented in the use cases of Chapter 7.

In the second example, we evaluate if the matching transformations can create weavings between terminal models conforming to different metamodels. The base algorithms of the matching transformations are the same, leading to similar results. However, we adjust the implementation of the containment tree, as well as the metamodel-based technique. For example, we compare data types such as *String* (in KM3) and *char* (in SQL-DDL). Thus, the generic matching transformation can be rapidly modified to match terminal models conforming to different metamodels. The weaving models generated in both examples are equivalent.

To summarize, the use of matching transformations and weaving models enables the semi-automation of the production of model transformations. Matching transformations enables the implementation of different heuristics or algorithms that produce a weaving model. These transformations can be adapted to different metamodels. The weaving model captures different transformation patterns specified in a weaving metamodel. The weaving model is translated into a model transformation language.

6.7 Conclusions

In this chapter, we have presented a solution to semi-automate the production of model transformations. We have presented to use matching transformations to create weaving models. These transformations use different matching heuristics. The weaving models capture common transformation patterns between model elements. The weaving model is translated into executable model transformations.

We have shown that matching transformations are a practical solution to implement new or to adapt matching heuristics or algorithms. We consider the matching process as a model transformation that takes two models as input, and that creates a weaving model as output. The use of declarative transformation languages abstracted several implementation details of these techniques. The separation of the whole matching process into different kinds of matching transformations enabled the combination of different methods in a straightforward way. The extensions to the GUI of the AMW tool enabled customizing the execution of a set of matching transformations. We stored a set of execution parameters in a configuration model. This model can be reused later by different developers. This is important to be able to reuse specific tuning parameters.

The matching transformations created weaving models between terminal models conforming to different metamodels, and also the creation of links between a restricted set of elements. We presented an improvement of a generic structural technique: our solution specifies different propagation models and it stores the propagation information in a weaving model. This opens the opportunity of developing new propagation methods in a relatively simple way. Moreover, we have presented a new link rewriting operation that analyzes the relationships between the links of a weaving model. These links are transformed into other complex kind of links. This operation is particularly important to capture different transformation patterns.

As general a conclusion, we have seen that matching transformations are a practical way to create and to adapt different matching heuristics or algorithms. This is essentially because the use of matching transformations diminishes the gap between the conceptual definition and the implementation. We develop the transformations reasoning about models, and not over low level structures of some general-purpose programming language. This motivates the creation of specialized matching DSLs. However, this is the subject for future work.

The execution of several matching transformations sequentially can cause performance problems when generating weaving models between large models. Thus, the optimization of these operations is becoming important and is also subject for future work. For instance, after choosing a set of operations to create a weaving model, these operations could be merged by a transformation engine to be executed in a single rule.

7 Case studies

In this chapter, a set of use cases of model weaving is presented. These use cases show that weaving models are convenient to be used in several application scenarios, with variable complexity and scope. The diversity of the scenarios validates the choices selected when developing our generic approach, which is adapted according to different application requirements.

7.1 Tool interoperability

There are a large number of different tools that can be used to solve similar problems. It is often necessary to use the data produced by one tool in another tool. However, the tools have different data format and semantics. To support interoperability between different tools, it is necessary to represent the semantic heterogeneities between the tools elements. This use case shows how weaving models are used to capture the semantic heterogeneities between two metamodels. The weaving model acts as high-level specifications for producing model transformations.

This use case describes our experiments using the tool interoperability motivating example of Chapter 4 of two different bug tracking tools. First, we show the creation of a weaving model based on the weaving metamodel extensions for tool interoperability. Then, we demonstrate how we use the generic transformation pattern to interpret the weaving model and to automatically produce model transformations. We end with a discussion about our results.

7.1.1 Capturing the semantic heterogeneities

Consider the metamodels of two bug-tracking tools, Mantis and Bugzilla. The Bugzilla metamodel has 146 elements. The Mantis metamodel has 62 elements. We need to discover the semantic heterogeneities between them. The metamodels of both tools may be stored in different data sources. The tool metamodels are originally in SQL-DDL. They are translated into Ecore. The semantics of Ecore is very close to KM3. This allows us to write metamodels using KM3 textual syntax.

We implement an extension of the core weaving metamodel for tool interoperability. This extension supports typical mapping expressions, varying from simple mappings (e.g., 1-to-1 equivalence links), to complex kinds of mappings (e.g., concatenation, data conversions). These mappings are extended as well to obtain expressions specific for tool interoperability. We show below an excerpt of the weaving metamodel. It specifies a data value expression used to translate enumeration values. It compares the value of given *source* element with the set of *sourceValue*, and sets the *target* element with the corresponding *targetValue*.

```
class EnumerationEquiv extends DataExpression {
    reference equiv [*] : EnumEqual;
}
class EnumEqual extends Equivalence {
    reference sourceValue: String;
    reference targetValue: String;
}
```

We create the weaving model by executing a sequence of matching transformations, which refine the initial input (the cross-product of elements) and generate a weaving model. Our AMW plugin is used to generate the interoperability weaving metamodel based on a set of extensions and to refine the weaving model during the manual phase.

An excerpt of the weaving model is shown in Figure 7.1. In this use case we use a human readable syntax to represent information models, similar to HUTN [112].

```
EnumerationEquiv = {
    source.ref = Left.priority.id;
    target.ref = Right.priority.id;
    equivalence = { source = "NONE";  target = "pt_null"};
    equivalence = { source = "low";  target = "pt_P1"};
};
Left = {
    name ="Mantis";
    ref = "c:\Tool_interoperability\Mantis.ecore";
    priority {  id = "EAttribute_priority";   }
};
Right = {
   name ="Bugzilla";
   ref = "c:\Tool_interoperability\Bugzilla.ecore";
    priority {   id = "EAttribute_priority";   }
}
```

Figure 7.1 A weaving model in HUTN

The model contains the equivalencies between the *priority* values. Note that both tool models have a priority property and both have the same ID “EAttribute_priority”. This does not cause problems because it is relative to the containing model.

The complete weaving model has 312 elements. This difference in the number of elements is due to the structure of the weaving metamodel, because for every couple of referred elements there is at least one element indicating the link type, plus the source and target elements. In addition, the source and target elements refer to an element that contains their identifiers (in the *Left* and *Right* elements).

7.1.2 Interpreting the weaving model

The execution semantics of the weaving model is specified through in a transformation that takes the weaving model as input and produces a transformation model as output. The transformation (485 lines) is implemented based on the generic transformation pattern. The ATL transformation rules are divided in three parts: the from block filters the appropriated model elements by their type; the *to* block contains the declarative code; the *do* block contains imperative code. We show in Figure 7.2 the rules that interpret the metamodel extension to translate the enumeration values. The “AMW” identifier denotes the weaving metamodel. The “ATL” identifier denotes the transformation metamodel.

```
rule EnumDataTranslation {
  from amw : AMW!EnumerationEquiv
  to atl : ATL!Binding (
        propertyName <- MOF!EClassifier.getInstanceById(amw.target.element.ref).name
    )
  do { atl.value <- thisModule.CreateEnum(amw, amw.enumEqual);}
}
rule CreateEnum(amw: AMW!EnumerationEquiv, attrEnum: Sequence (AMW!EnumEqual)){
  to ifExp : ATL!IfExp (
          thenExpression <- targetEnum,
          condition <- operation
        ),
        operation : ATL!OperatorCallExp (
          operationName <- '=',
          arguments <- sourceEnum
        ),
        endExp : ATL!StringExp (),
        sourceEnum: ATL!StringExp (
          stringSymbol <- attrEnum->first().sourceValue.toString()
        ),
        targetEnum : ATL!StringExp (
          stringSymbol <- attrEnum->first().targetValue.toString()
        )
  do { operation.source <- amw, amw.source->collect(e | e.element.ref),true);
          if ( attrEnum->size() = 1 ) {
                ifExp.elseExpression <- endExp;
          } else {
                ifExp.elseExpression <- thisModule.CreateIfEnum(amw,
                                        attrEnum->subSequence(2,attrEnum->size()));
          }
  }
}
```

Figure 7.2 Higher-order transformation

The rule *EnumDataTranslation* matches the element *EnumerationEquiv* from the correspondence model. It produces a *Binding* element conforming to the ATL metamodel. A binding has a *propertyName* that corresponds to the target element. The target element is obtained by *getInstanceById* function. The property *value* calls the rule *CreateIfEnum.* It receives the set of enumerations as parameters and produces a model with a set of nested *IfExp* (conditional expressions).

The *IfExp* contains a *condition* expression, which is formed by an equality operator (*OperatorCallExp*). This operation compares the source value of the enumerations and sets the correct target value specified at *thenExpression*. The *StringExp* elements return the *sourceValue* and *targetValue* (an empty *String* if there is no equivalence). The complete transformation produces a transformation model with a set of rules. This model is extracted into a text representation that is executed in the ATL engine.

7.1.3 Discussion

The metamodel extensions enable producing a domain-specific (tool interoperability) weaving metamodel. Among the different metamodel extensions that are created, the most used are the concatenation of elements (e.g., *os* concatenated with *os_version*), data type conversions (e.g., *Integer* to *String*, references to attributes, etc.) and conversion of enumeration values.

One interesting observation is that the values of the enumerations from Mantis are not described in the metamodel, only in a Php file. Since the tool metamodels cannot be modified (otherwise the services provided might not work properly), the enumerations are added in one metamodel extension. This is a very specific extension, which is probably not useful outside the bug-tracking example, but it is still necessary to be able to create the output transformation.

The weaving model has composite elements that conform to a combination of metamodel extensions. For instance, we combine the conversion of “references to attributes” extension with the “concatenation” extension. This way, it is possible to create more complex output transformation models with the same set of extensions.

The metamodel extensions ease the task of repeatedly creating complex mapping and data value expressions between tool metamodels. The adaptive user interface is used together with semi-automatic matching algorithms. The extensibility of the weaving metamodel enables human intervention essentially at the matching process, because all the necessary information to produce transformations is available in the weaving model. This is different from traditional approaches that have an extra step of mapping discovery. However, it is still possible that a weaving metamodel covers most semantic interoperability cases, but not all. Complex expressions that are not often used can be coded manually in the final generated transformation.

The declarative structure of the weaving metamodel allows a clear separation of the input model (the weaving model) from the output model (a transformation model). Thus, it is relatively straightforward to modify only the output expressions and produce different transformation models. This also enables generating different expressions in the output transformation. For instance, the translation of enumeration values may be implemented as nested ifs (our final choice), or using *case*-like statements. This opens the possibility of optimizations of the output transformations (however, this is not the focus in this work). On the negative side, transformation languages may have complicated metamodels, in particular for querying and navigation expressions (e.g., OCL, XPath).

To summarize, this use case demonstrates that the use of weaving models and transformation models enables to improve two data integration phases (matching and transformation production) to solve tool interoperability problems in a practical and efficient manner. We are able to define different extensions of the core weaving metamodel to cope with distinct kinds of semantic heterogeneity. We create a weaving model using some matching algorithms and a user interface. Finally, we implement the transformation pattern that automatically generates a transformation to transform the tool models.

This use case has been published at [41]. It is available for download at (http://www.eclipse.org/gmt/amw/usecases/interoperability/). This page contains a fully implemented example, with the metamodels, models, the generated transformation, a *HowTo* and the sources.

7.2 Bridge between SQL-DDL and KM3

A modeling platform is not an isolated world. There are several other “worlds” that are based on different metametamodels, set of principles, representation format, etc. These different “worlds” are called technical spaces. Examples of technical space are: XML, relational databases, ontologies, MDE, grammarware. It is of major importance to provide mechanisms to interoperate between these technical spaces.

This use case shows how weaving models and model transformations are used to bridge between two different concrete syntaxes: SQL-DDL (Data Definition Language) and KM3. We define two bridges: one from KM3 to SQL-DDL, and from SQL-DDL to KM3. We use our model driven platform as a pivot between these two representations. In model management platforms, this use case is considered an application of *ModelGen* operations [11]. This is a complex process divided in several steps: first, we create a SQL-DDL metamodel conforming to Ecore. The SQL-DDL metamodel has 48 elements. We briefly describe this metamodel here: it contains a root element *Database*, which contains a set of *Table*; a *Table* contains a set of *Column* and *ForeignKey* (the complete metamodel can be found at the Atlantic Zoo [2] [3]).

The next step is to inject an SQL-DDL file into the modeling technical space. In other words, we translate this file into a model conforming to the SQL-DDL metamodel, as shown in Figure 7.3. The SQL-DDL file conforms to a SQL grammar, which conforms to EBNF.

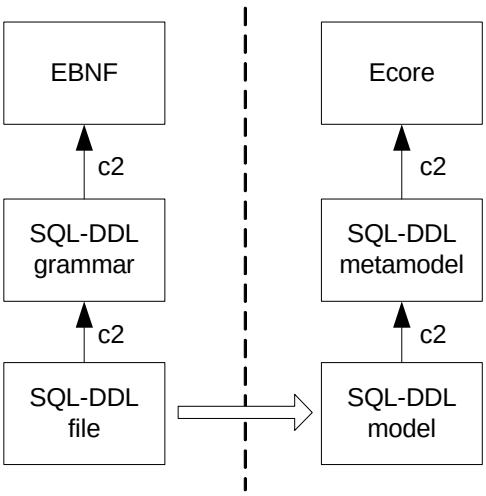


Figure 7.3 Injection of a SQL-DDL file into an SQL-DDL model

The injection is implemented using the TCS (Textual Concrete Syntax) tool. TCS provides a practical way to inject (and extract) textual syntax into a modeling platform. It defines how to translate each element of the textual file into specific model elements. Figure 7.4 illustrates an excerpt of the input SQL-DDL file. It represents a table of the Bugzilla tool. The injection of the SQL-DDL file into a model enables a homogeneous platform to create weavings models and transformations models.

```
CREATE TABLE mantis_bug_relationship_table (
  id int (7) unsigned NOT NULL,
  source_bug_id int (7) unsigned NOT NULL default '0',
  destination_bug_id int (7) unsigned NOT NULL default '0',
  relationship_type int (2) NOT NULL default '0',
  FOREIGN KEY (source_bug_id) REFERENCES mantis_bug_file_table (id),
  FOREIGN KEY (destination_bug_id) REFERENCES mantis_bug_file_table (id)
)
```

Figure 7.4 SQL-DDL textual syntax

A weaving model (*Mw*) is created between the SQL-DDL and KM3 metamodels, as illustrated in Figure 7.5. This weaving model contains 132 elements. *Mw* conforms to *MMw*, which is an extension to the core weaving metamodel. This extension contains different kinds of links that define the equivalences between the elements of SQL-DDL and the elements of KM3. This extension reuses part of the metamodel extensions for tool interoperability.

However, these two metamodels have different expressiveness. This means it is not always possible to link all the elements of SQL and KM3. For instance, a KM3 *Class* does not have a "default value" property; a SQL-DDL *Table* does not have references.

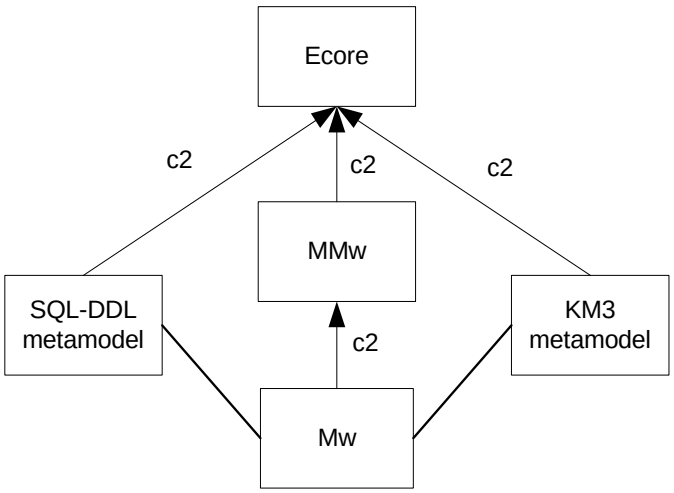


Figure 7.5 Weaving between SQL-DDL and KM3

The weaving model is used as a specification to produce a model transformation in ATL. We implement a higher-order transformation (HOT) based on the generic transformation pattern. This HOT takes the weaving model as input and produces a transformation model as output. The output transformation has 83 lines. This transformation translates the SQL-DDL terminal model into a KM3 terminal model. In this case, extensions to generate default values are constantly used, because KM3 models have attributes such as *lower*, *upper* (for cardinality), *isAbstract*, that are not present in the SQL-DDL metamodel.

The final step of the bridging process is the extraction of the KM3 model (with the KM3 concepts such as *Class*, *Attribute*, *Reference*) into the textual concrete syntax of KM3 (see the final result in Figure 7.6). We use the same TCS definition used to inject the SQL-DDL file, because it is bidirectional (i.e., it supports injectors and extractors).

```
class mantis_bug_relationship_table {
      attribute id : int;
      attribute source_bug_id : int;
      attribute destination_bug_id : int;
      attribute relationship_type : int;
      reference source_bug_id : mantis_bug_file_table;
      reference destination_bug_id : mantis_bug_file_table;
}
  datatype int;
```

Figure 7.6 Resulting KM3

The same process is also executed in the opposite direction: a KM3 file is injected into a KM3 model; a weaving model is created between the KM3 and SQL-DDL models; this weaving model is used to produce an ATL transformation; this transformation translates the KM3 model into the SQL-DDL model; this model is extracted into its textual syntax.

To summarize, this use case demonstrates the use of weaving models and transformations as a practical approach to ease the task of developing bridges between different concrete syntaxes. The overall process is divided in smaller steps that are implemented based on generic concepts. This allows reusing several components, thus avoiding the implementation of complete ad-hoc programs every time such a bridge is needed. An important result is that we are capable to reuse part of the weaving metamodel extensions defined for the tool interoperability use case. This shows that generic weaving metamodel extensions can be reused in different application scenarios. This use case is available for

download at (http://www.eclipse.org/gmt/amw/usecases/modelgen/). This page has fully implemented bridges, with general documentation, a *HowTo* and the sources.

7.3 Data mapping between relational database and XML documents

The data mapping between relational databases and XML documents is a common problem in many organizations. This is a typical data exchange problem. However, existing techniques are not generic enough to support different kinds of models and mappings.

In this use case, we develop a weaving model that captures the relationships between a metamodel with flat structures and foreign keys relationships (representing a relational database) and a metamodel that contains nested structures (representing an XML base).

We illustrate this data mapping problem using two simple library metamodels. The objective is to execute the data mapping between models conforming to these two metamodels. Libraries typically exchange data to have a standard catalogue format, both for standardization and interoperability purposes. Let us consider the two data sources in Figure 7.7. One library has its own relational schema as defined by *Relational schema R1*. But it also agrees to use an XML format as defined by *XML schema X1*. Schema *R1* has two tables: *Books (ISBN [International Standard Book Number], Title, Author, SID)* and *Subjects (SID, Description)*, with the foreign key *SID* on *Books* referencing the subjects of a book. Schema *X1* has the same basic structure except for the foreign key in books since this correspondence is represented by the nested structure between *Books* and *Subjects*.

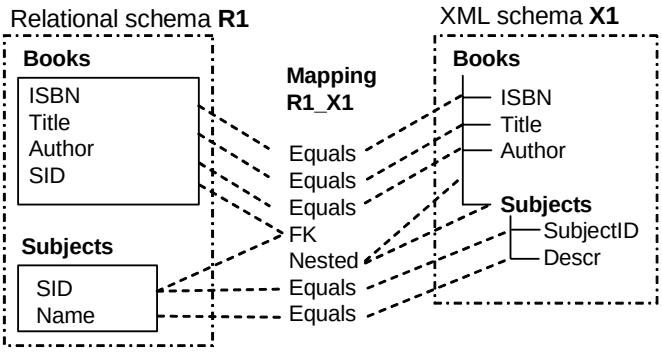


Figure 7.7 Relational to XML mapping

The translation from *R1* to *X1* is represented by the mapping *R1_X1*. It has three mapping structures: *Equals*, an inter-schema correspondence that indicates equalities such *as R1.Books.ISBN = X1.Books.ISBN, R1.Books.Title = X1.Books.Title*, and so on; *FK*, an intra-schema correspondence that indicates the foreign key constraint between *R1.Books.SID* and *R1.Subjects.SID*; and *Nested*, another intra-schema correspondence that represents the nesting relationship between *X1.Books* and *X1.Books.Subjects*. These intra-schema correspondences guarantee the generation of a valid output model. Analyzing this scenario, we observe that all inter- and intra- model relationships need a structure to represent links between elements, independently of the mapping semantics. This also shows the importance to have an expressive representation allowing to reason about links between complex models, like the *Nested* relationship.

We define a metamodel extension that enables the creation of declarative links between these metamodels. We first define an extension of the core weaving metamodel, and we create a weaving model to represent mappings *R1_X1* and *X1_O1*, first without specific semantics. We incrementally extend the existing weaving metamodel (represented by *MMw* in Figure 7.8) until obtaining a weaving metamodel with all necessary structures. Thus, we have dedicated mapping specifications with variable expressive power: we represent from simple element links such as *Equals;* then *Nested* and *FK* constraints.

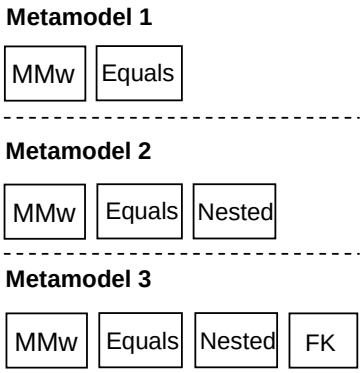


Figure 7.8 Extended weaving metamodels

We use simplified versions of the terminal models and metamodels, capable of representing only the desired structures. The weaving model (*Mw*) is used as specification to produce transformation models. The weaving model is completely independent of the output transformation metamodel. This enables to produce transformations in different target languages, such as ATL, XSLT, or SQL-like languages. We implement two higher-order transformations based on the generic transformation pattern. These transformations produce an ATL and a XSLT model.

The transformation models are extracted to the corresponding concrete syntax. The resulting ATL and XSLT are actually used to transform the source models into the target models. We show in Figure 7.9 an excerpt of the generated transformations, with the rules to handle nested and foreign key semantics.

XSLT rule

```
<xsl:template match="bookRcds">
    <xsl:element name="books">
        <xsl:attribute name="ISBN">
            <xsl:value-of select="@ISBN"/>
        </xsl:attribute>
        <xsl:variable name="sid" select="@SID"/>
        <xsl:apply-templates select="/descendant-or-
            self::subjectRcd[@SID=$sid]">
        </xsl:apply-templates>
    </xsl:element>

</xsl:template>
<xsl:template match="subjectRcd">
    <xsl:element name="subjects">
        <xsl:attribute name="SubjectID">
            <xsl:value-of select="@SID"/>
        </xsl:attribute>
    </xsl:element>
</xsl:template>
```

ATL rule

```
rule Books {
    from
        db : RDBMS!BookRcd
    to
        xml : XML!Book (
            ISBN <- db.ISBN,
            subjects <- RDBMS!SubjectRcd.
            allInstances()->select (e | e.SID = db.SID)
        )
}
rule Subjects {
    from
        db : RDBMS!SubjectRcd (RDBMS!BookRCD.
            allInstances()->exists(e | e.SID = db.SID))
    to
        xml : XML!Subject (
            SubjectID <- db.SID
        )
}
```

Figure 7.9 Generated XSLT and ATL

We illustrate in Figure 7.10 a screenshot of the AMW plug-in. On the left side is the source relational database schema, on the right side is the target XML schema, and in the middle the weaving model created conforms to the *Metamodel extension 3*.

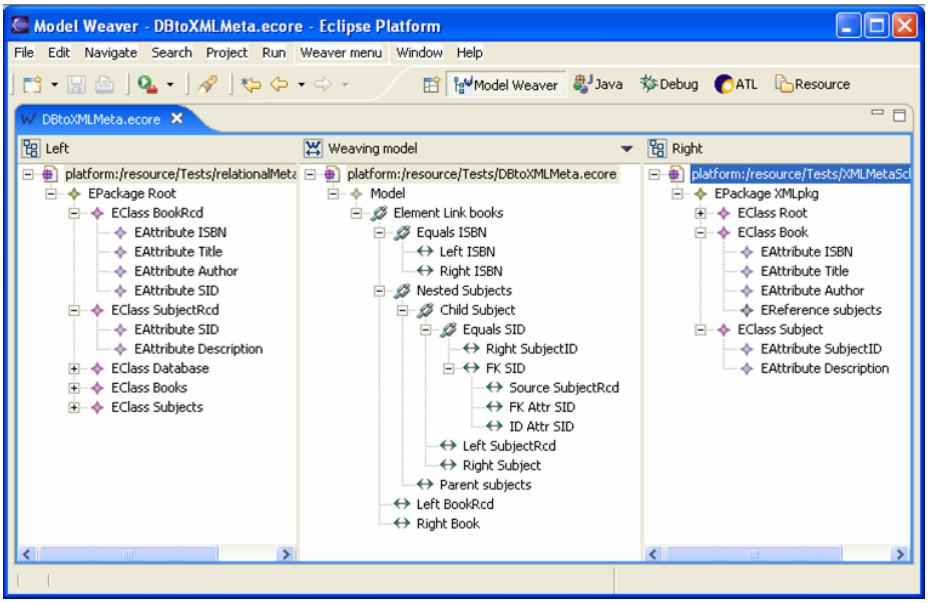


Figure 7.10 Weaving in the AMW prototype

This use case demonstrates how weaving models and transformations models are used to create model management operations for data mapping. The links of the weaving model abstract widely used structures for this kind of scenarios, such as nested relationships and foreign keys. The weaving models can be easily created, due to the declarative nature of the metamodel extensions. An advantage of developing declarative operations is the abstraction of expressions that are typically hard to develop, such as navigation expressions. Finally, the weaving models are language independent. Thus, we implement a generic transformation pattern that enables the production of transformations in different languages, such as ATL and XSLT.

This use case has been published at [40]. It is available for download at (http://www.eclipse.org/gmt/amw/usecases/RDBMSXML/). This page has fully implemented bridges, with general documentation, a *HowTo* and the sources.

7.4 Metamodel comparison and model migration

Metamodels need to be compared for several reasons. One important reason is to discover the equivalent elements between two versions of a metamodel. The result of a comparison is used to migrate between the terminal models conforming to these metamodels. This use case presents how weaving models created with the help of the AMW tool are used to compare two different metamodels. The weaving model is used to produce model transformations between the terminal models conforming to these metamodels.

Consider two versions of a *Scade* metamodel, *Scade (v1)* and *Scade (v2)*. Scade is a standard for development of embedded software for the Avionics Industry [44]. The *v2* of the metamodel is derived from *v1*[11]. An organization using Scade decides to migrate from a model conforming to *Scade (v1)* into a model conforming to *Scade (v2)*. To correctly migrate from one version into another, it is necessary to know the equivalences between the metamodel elements. Based on this information, we produce a model transformation from *v1* to *v2*.

[11] We do not describe the metamodel elements in details because each metamodel has an average of 400 elements (these metamodels are published at [2]).

We produce a comparison weaving model that contains links with equivalence semantics between the metamodels' elements. The comparison weaving model conforms to a weaving metamodel that is an extension of the core weaving metamodel (see Figure 7.11). This metamodel extension contains an *Equivalent* link. This link contains a similarity attribute that contains similarity estimation between a left (*v1*) and a right (*v2*) element. Links with high similarity values are considered to be equivalent. This link is extended by different kinds of links, depending on the type of elements that are being compared, for example *AttributeEqual* (for attributes) and *ElementEqual* (for classes). The references (*child*) between *ElementEqual* and *AttributeEqual* links are created according to the containment relations between classes and attributes. *NotEquivalent* links are used to store the elements that do not have any equivalence in the two versions. The *ReferredElement* class is similar to the *ElementRef* class from the traceability use case: it acts like a proxy to the real linked elements.

```
abstract class Equivalent extends WLink {
   attribute similarity : Double;
   reference left container : ReferredElement;
   reference right container : ReferredElement;
}
abstract class Equal extends Equivalent {
}
class ElementEqual extends Equal {
}
class AttributeEqual extends Equal {
}
class ReferenceEqual extends Equal {
}
class NotEquivalent extends WLink {
   reference left container : ReferredElement;
   reference right container : ReferredElement;
}
class ReferredElement extends WLinkEnd {
}
```

Figure 7.11 Metamodel extension for comparison

The equivalence links are created semi-automatically by executing a sequence of matching transformations. In the case of metamodel comparison, rather simple matching techniques yield good results, because the elements have several similarities. We execute five matching transformations. First, a restricted Cartesian product operation creates links between the pairs of elements with the same type. Second, a similarity value is assigned to each link. This value is based on the name, type and cardinality of elements. Third, the links with best values are selected to create a refined weaving model. Fourth, the links are rewritten to represent the nested relationships between classes and attributes/references, and the inheritance between classes. Finally, *NotEquivalent* links are created for all the elements that do no have equivalence links.

The resulting weaving model should be analyzed by a domain expert, and can be refined using the graphical facilities of AMW. An easy-to-use user interface is very important to analyze the results of the matching transformations, especially when the input models are large. The creation of different kinds of links enables an efficient type-based search through the weaving model.

The weaving models are interpreted by a higher-order transformation that transforms the declarative links of the weaving model into an executable ATL transformation, as show in Figure 7.12.

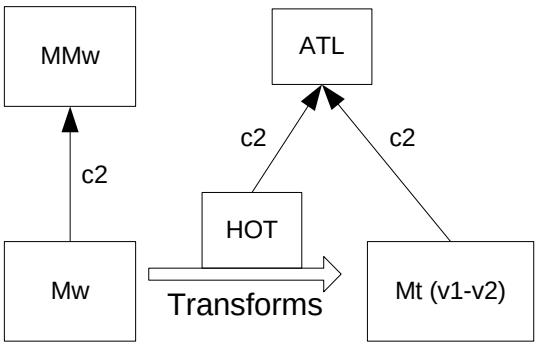


Figure 7.12 Transformation of a comparison weaving model into ATL

Each *ElementEquivalent* link is transformed into an ATL rule. The left and right references (*ReferredElement*) are transformed into the input and output elements. The *AttributeEqual* and *ReferenceEqual* links are transformed into bindings. Figure 7.13 shows an excerpt of the transformation that is generated. This transformation contains essentially binding expressions between the equivalent elements. The identifiers after the rule names are automatically generated. The transformation developer can add new expressions based on the information of the *NotEquivalent* links. This model transformation is responsible to translate the model conforming to *Scade (v1)* into the model conforming to *Scade (v2)*.

```
abstract rule Object_2_Object26 {
  from
    v_left : Scadev1!Object
  to
    v_right : Scadev2!Object (
      name <- v_left.name,
      runLine <- v_left.runLine
    )
}
rule Label_2_Label21 extends Object_2_Object26 {
  from
    v_left : Scadev1!Label
  to
    v_right : Scadev2!Label (
      expression <- Set {v_left.expression}
    )
}
```

Figure 7.13 A part of the transformation generated automatically

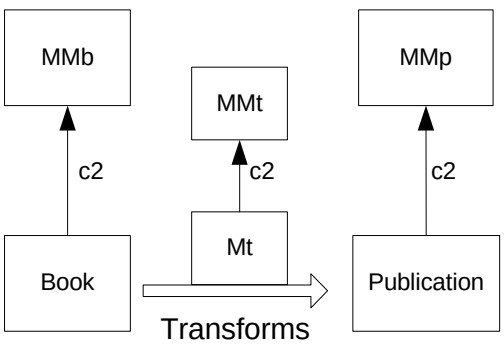
This use case demonstrates how weaving models are used to compare different metamodels and to migrate between the terminal models conforming to these metamodels. The different kinds of links enable the identification of the relationships between the elements in a clear way. The user interface helps on the creation of weaving models by not experts. A non-expert, or less experienced developer needs an easy to use graphical tool, together with automatic facilities. The easy configuration of the matching transformations allows the use of methods adapted to the comparison use case, demonstrating the advantage of defining generic data interoperability methods. In this use case it is not necessary to use complicated methods that would not bring much improvement on the final result.

The weaving extensions are quite simple, and can be generalized for different scenarios, which is shown in other use cases in this chapter. Finally, the weaving model is used to successfully generate an executable ATL transformation to perform the model migration. This use case is available for download at (http://www.eclipse.org/gmt/amw/usecases/compare/). This page contains a fully implemented example, with general documentation, a *HowTo* and the sources.

7.5 Traceability of model transformations

This use case presents how weaving models are used to store the execution trace of model transformations, i.e., to support traceability of model transformations. A model transformation takes a set of models as input and produces a set of models as output. The elements of the input models are visited and then transformed into elements of the output models. After the execution of a transformation, we need to discover which set of elements of the source models are visited and transformed into a set of target model elements. This is, for instance, a typical application of data provenance.

We present in Figure 7.14 a concise example of a transformation between two models. Although simple, this example clearly presents the challenge of traceability of model transformations.




```
rule Book2Publication {
   from
      s : MMb!Book
   to
      t : MMp!Publication (
         title <- s.title + s.subtitle,
         pubYear <- s.year,
         authors <- author
      ),
     author : MMp!Author (
         name <- s.author
     )
}
```

Figure 7.14 Book to publication transformation

The input model contains information about books (*Book*). It conforms to the metamodel *MMb*. *MMb* has one class *Book*, which has attributes *title*, *author*, *year* (the content of the columns is self-explanatory). The output model contains information about generic publications (*Publication*). It conforms to a publication metamodel *MMp*. *MMp* has two classes: *Publication*, which has attributes *title*, *authors*, *pubYear* and reference *authors [multiple cardinality]*; *Author*, with attribute *name*. The transformation *Mt* is an ATL transformation, thus it conforms to the ATL metamodel (denoted by *MMt*). A part of the code of the ATL transformation is shown in the right side of the figure. For every *Book* of the source model, the transformation creates a new *Publication* in the target model, and it assigns the values of the source attributes to the target attributes.

We illustrate the traceability between model elements in Figure 7.15. It shows one element from the source model (*001 : Book*), and two elements from the target model (*002 : Publication* and *003 : Author*). The traceability information is represented by the lines between the elements. Without this information, it is not possible to directly discover which elements of the source model are used to generate a given author. The model transformation is created using the metamodel elements, and it is executed over the model elements. Thus, the relationships between the input and the output model elements (as well as the transformation rules), are accessible only in the moment of its execution. Thus, without any traceability information, it would be necessary to apply an inverse procedure to transform the *Author* class into the *author* attribute, and to compare the result with the source model

elements. This may be an expensive operation if the input and the output models are large. These relationships must be saved to be able to exploit the execution trace information afterwards.

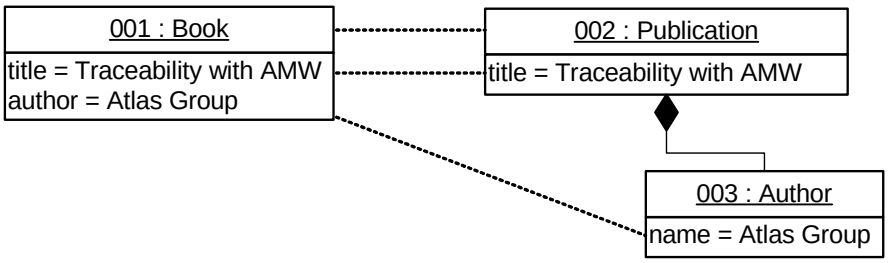


Figure 7.15 Traceability between two model elements

A weaving model is used to capture this traceability information. This weaving model conforms to an extension to the core weaving metamodel. The traceability metamodel extension is depicted in Figure 7.16.

```
class TraceLink  extends WLink{
    attribute ruleName : String;
    reference sourceElements[*] ordered container : WLinkEnd;
    reference targetElements[*] ordered container : WLinkEnd;
}
class TraceLinkEnd extends WLinkEnd {
}
class ElementRef extends WElementRef {
}
```

Figure 7.16 Traceability metamodel extension in KM3

The central element of this extension is the *TraceLink* element. Every time a transformation visits a source element (e.g., the *001 : Book* in our case), it creates a new *TraceLink* in the weaving model. The reference *sourceElements* refers to the source elements (the *title* and *author* values, not the metaelements). The reference *targetElements* refers to the generated target elements (the multiple cardinality enables having more than one target element). The attribute *ruleName* has the name of the rule that is executed (e.g., *Book2Publication*). This attribute enables storing the name of the transformation rule that is executed, not only to the source elements.

The class *TraceLinkEnd* represents each source and target elements. The reference *element* (from the core weaving metamodel) refers to class *ElementRef*. This element is a proxy to the real linked elements. The format of the identifier is specified by an annotation --*@wmodelRefType* (e.g., XMI IDs and XPointers). This allows having weaving models that do not modify the source and target models, for instance by adding some traceability meta-information.

However, the original model transformation *Book2Publication* does not specify how to create the traceability weaving model, only how to transform a *Book* into a *Publication*. Hence, the original transformation is modified into *Mt'*. *Mt'* has additional rules to create the elements of the traceability weaving model. The new setup is shown in Figure 7.17: the modified transformation *Mt'* takes the *Book* model as input and produces a *Publication* model and a traceability weaving model *Mw* as output (the metamodels are omitted for better visualization). The weaving model has the traceability links between *Book* and *Publication*.

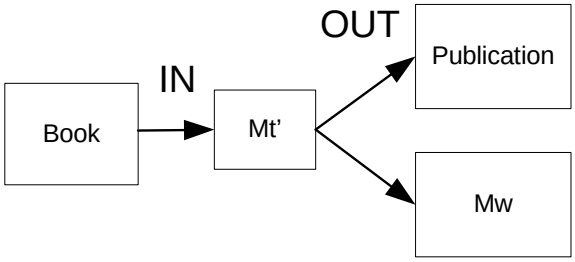


Figure 7.17 *Mt'* generates a *Publication* model and a weaving model

We focus on traceability of model transformations. However, there are different use cases of traceability, for instance requirements traceability. Requirements traceability keeps track of all the steps of a development process, analysis, design, programming, testing, etc. The kinds of links are *developed_by*, *allocated_to*, *performed*, *based_on* or *modify*. The key processes are the identification of the possible kinds of links and the development of new weaving metamodel extensions.

To summarize, this use case shows that weaving models are an appropriate solution to provide traceability of model transformations. The traceability metamodel extension enables creating traceability links (domain specific links). The weaving model is automatically created when the ATL transformation is executed. The traceability weaving model can be visualized and modified on the AMW tool, without any modification on the code of the tool. This shows the advantage of developing the tool using a generic and reflective API. This use case has been published at [13]. It is available for download at (http://www.eclipse.org/gmt/amw/usecases/traceability/). This page contains a fully implemented example, with general documentation, a *HowTo* and the sources.

7.6 Merge in a Geographical Information System (GIS)

Geographical Information Systems (GIS) [62] are used to relate geographical information with descriptive data. This use case presents a weaving model used to define a merge of a metamodel with geographical information and a metamodel with statistical data, into a graphical representation.

Let us consider the two XML schemas *Gs* and *Es* shown in Figure 7.18. Schema *Gs* describes only geographical information about election precincts within a district. A *District* is formed by a set of precincts. Each *Precinct* contains elements *Number* (precinct identification), *Address*, *City* and *Geometry*; *Geometry* contains a set of *Coordinate*, which are points in the form *(x, y)* defining the precinct limits. Schema *Es* contains data about the election results. The *Election* element is formed by a set of *Precinct* and a set of *CandidateDescr* (candidate's description). Each election precinct contains elements *PID*, *Voters* (the number of electors that voted), *Absentees* and a set of *Candidate; Candidate* contain *Votes* (the number of votes received) and *CID*, which is a foreign key for obtaining candidates' *Name* and *Party* from *CandidateDescr*.

Assume that we want to publish the results in a graphical form as shown in visualization schema *Vs*. The illustration shows the graphical interface, but there is an underlying schema to represent the data. The map is divided into precincts.

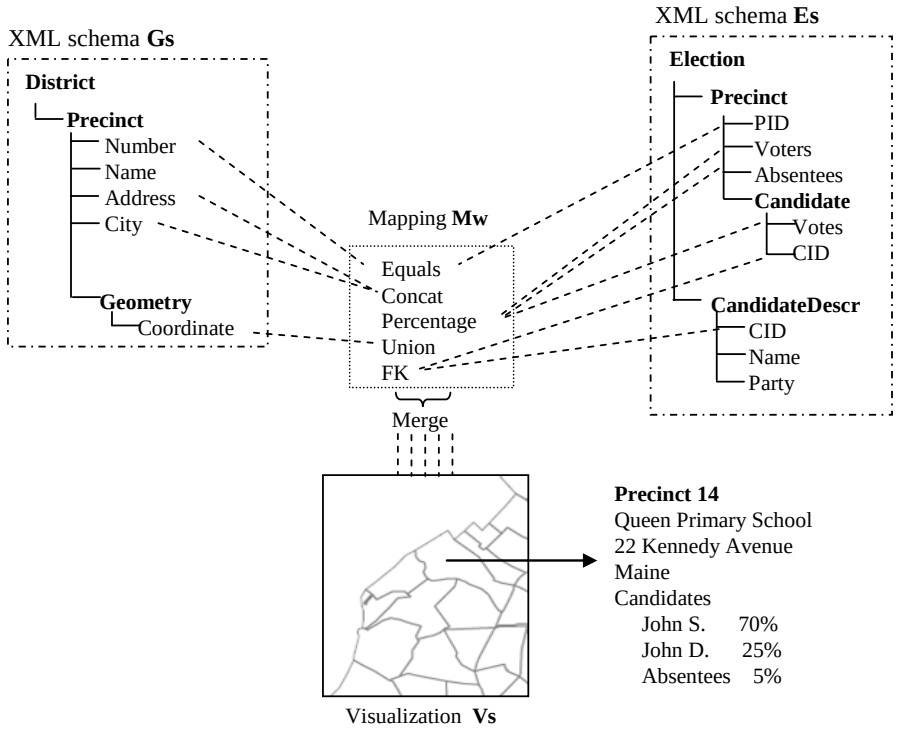


Figure 7.18 Data mapping of election results

When the user clicks over a precinct, the application shows its location and the voting percentage of each candidate and absentee's percentage, e.g., the statistical data. The graphical representation is not saved in persistent storage. It is used to visualize data over the web. There is a mapping *Mw* between three schemas: two input schemas and one output schema. It specifies different semantics: the equality between *Number* and *PID* in the element *Equals;* the concatenation of A*ddress* and *City* in *Concat;* the vote percentage of each *Candidate* and *Absentees* in *Percentage*; the foreign key between *Precinct.Candidate.CID* and *CandidateDescr.CID* in the element *FK;* and the union of all geometry coordinates in *Union*. We have one *Merge* element between every mapping element from *Gs* and *Es* into the output elements in *Vs*, specifying that we want to merge *Gs* and *Es* elements into *Vs.* They are subsumed in the vertical dashed lines coming from mapping *Mw* to schema *Vs*.

This use case has a particularity: the two input models and the output model conform to different metamodels. This constraint does not enable the utilization of generic algorithms that assume that all models conform to the same metamodel. Thus, we specify the merge operations through declarative weaving models. We develop a weaving metamodel that extends the core weaving metamodel. The geographical schema *Gs* is represented using a GML subset. GML [116] is the standard format for representing geographical information. The schema *Es* with the election results conforms to an XML schema. The visualization output format is SVG [125].

The base weaving metamodel is incrementally adapted in order to obtain dedicated mapping specifications. It represents elements of variable complexity: element links and associations such as *Equals;* foreign key semantics in *FK; Concat* to indicate concatenation. The metamodel also contains elements that represent complex semantics, such as *Percentage*, obtained by a computation over *TotalVotes* and *Voters; Union* of coordinates and *Merge*.

In Figure 7.19, we illustrate the weaving model created with the AMW prototype. There are four panels which show, respectively: (1) the GML schema *Gs*; (2) the weaving model *Mw*; (3) the election schema *Es;* (4) the extended SVG schema *Vs.* The third panel is added without any modification on the plug-in code. This is possible because the UI adapts to support several woven models, according to

the weaving metamodel extensions. We highlight in the figure the equality links: it refers to elements *Left* and *Right*. They represent the correspondences to *GMLPrecinct.number* and *Precinct.PID*, respectively. These elements are merged into the element *Target* (*SVG.Precinct.Number)*.

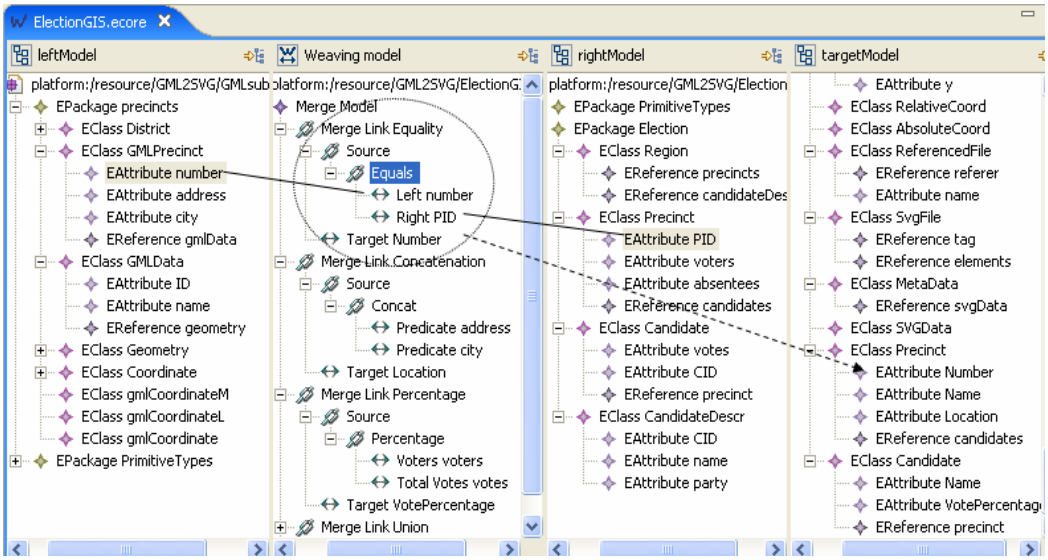


Figure 7.19 Weaving GML and Election metamodels into SVG

This weaving model is used as specification for producing transformations in two different languages: ATL and XSLT (see an excerpt in Figure 7.20). We implement another HOT according to generic transformation pattern. This HOT supports the different kinds of links from this weaving metamodel. Thus, it produces different types of ATL and XSLT expressions as output. The output transformation models are serialized in their text format. The resulting ATL and XSLT are actually used to merge the GML and XML data into a SVG document. However the standard SVG schema definition is designed focusing on graphical representations without transparent mechanisms to associate data with it. The transformation thus embeds the statistical data into the SVG document. It conforms to an extended SVG schema.

```
rule GMLprecincts {
  from
   gml : GML!GMLPrecinct
  to
   svg : SVG!G (
       metaData <- metadata,
       groupContent <- gml.gmlData.geometry    ),
   metadata : SVG!MetaData (
       svgData <- svgdata    ),
    svgdata : SVG!Precinct (
       Number <- gml.gmlData.ID,
       candidates <- ElectionMM!Precinct->
         allInstances()->select( e | e.PID =
         gml.gmlData.ID)->first().candidates   )
}
rule Candidate {
  from
     election : ElectionMM!Candidate
  to
    svg : SVG!Candidate (
        Name <- ElectionMM!CandidateDescr.
          allInstances()->select ( e | e.CID =
          election.CID)->first().name,
        VotePercentage <- election.votes /
          election.precinct.voters * 100     )
}
```

```
<xsl:template match="GMLprecincts">
  <xsl:element name="g">
    <xsl:element name="metaData">
      <xsl:element name="precinct">
        <xsl:attribute name="Number">
        <xsl:value-of select="gmlData/@ID"/>
      </xsl:attribute>
      <xsl:variable
              name="local_pid" select="gmlData/@ID"/>
      <xsl:apply-templates select="document
                ('ElectionData.xml')/Region/precincts
          [  @PID=$local_pid]/candidates"/>
     </xsl:element></xsl:element>
    <xsl:apply-templates select="gmlData/geometry"/>
  </xsl:element>
</xsl:template>
<xsl:template match="candidates">
  <xsl:element name="candidates">
    <xsl:variable name="local_cid" select="@CID"/>
      <xsl:attribute name="Name">
        <xsl:value-of select="document(
            'ElectionData.xml')/Region/candidateDescr
             [@CID=$local_cid]/@name"/>
      </xsl:attribute>
      <xsl:attribute name="VotePercentage">
         <xsl:value-of
              select="@votes div  (../@voters ) * 100 "/>
      </xsl:attribute></xsl:element>
</xsl:template>
```

Figure 7.20 Generated ATL and XSLT

This scenario shows that the creation of different metamodel extensions enables the utilization of weaving models to quite distinct applications scenarios. This scenario has a particularity that usually is not supported by generic merge approaches: the two input models and the output model conform to three different metamodels. This constraint does not directly apply to generic merge algorithms, because these algorithms typically assume that all models conform to the same metamodel. Thus, the specification of merge operations through declarative weaving models is an interesting achievement. The extensions are easy to understand, because they are created using a domain-specific vocabulary. The AMW interface proves to be efficient when weaving three models. The prototype interprets the references to woven models (extensions of *WModel*) and it adds an extra panel. Finally, the rather simple metamodel extensions enable to generate complex transformation code, and in two different languages. This use case will be published at [23]. It is available for download at (http://www.eclipse.org/gmt/amw/usecases/mergeSVG/). This page contains general documentation, an example, a *HowTo* and the sources.

7.7 Model annotation

Models are annotated or decorated to insert information that is not defined in the metamodel. Annotation data usually is not conceptually relevant to be part of the metamodel. For example, annotations are often meta-information used for pre-processing, testing, logging, versioning, or parameterization.

This use case shows how a weaving model is used to annotate terminal models. Consider a *Java* terminal model that conforms to the *Java 1.4* metamodel, as shown in Figure 7.21. The metamodel defines the basic elements of Java, e.g., classes, fields, methods and packages. However, this

metamodel does not support annotations, since annotations are first included in the Java 5 metamodel. The Java 1.4 metamodel cannot be extended to keep the compatibility.

The model annotations are defined in a weaving model. This scenario differs from typical scenarios where two models are woven. In this case only one model is woven, i.e., the annotated terminal model. The annotations are created as a pair key-value. These annotations are linked with the Java elements. These links and annotations are defined in the weaving model *Mw*.

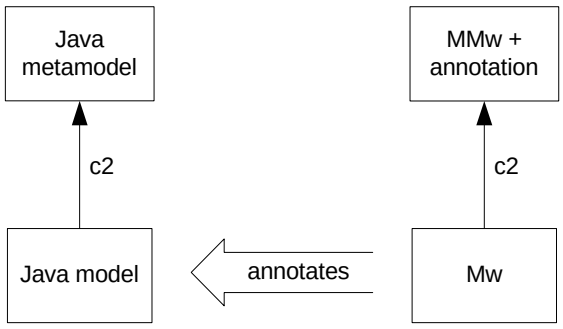


Figure 7.21 Annotating a Java model with a weaving model

The weaving model conforms to an annotation extension to the core weaving metamodel. The annotation extension is illustrated in Figure 7.22.

```
class AnnotationModel extends WModel {
   reference contents[*] ordered container : Annotation
   reference referencedModel container : AnnotatedModel;
}
class AnnotatedModel extends WModelRef {
}
class AnnotatedModelElementRef extends WElementRef{
}
class Annotation extends WLink {
   reference properties[*] ordered container : Property
   reference annotatedModelElement container : AnnotatedModelElement;
}
class AnnotatedModelElement extends WLinkEnd {
}
class Property {
   attribute key : String;
   attribute value : String;
}
```

Figure 7.22 Metamodel extension for annotation

It is important to note that the *AnnotationModel* class has a single-valued reference to *AnnotatedModel*. This means the annotations are defined only for one model. The same analogy is true for the Annotation class, which contains a single-valued reference to the model elements, plus a list of properties. The properties have an identification key and the corresponding value. The *AnnotatedModelElement* class is the proxy for the linked elements.

The annotations are created and modified using the AMW graphical interface. The interface automatically adapts to the number of woven models, and creates only two panels, one with the weaving model and another with the annotated model.

Figure 7.23 shows a screenshot of the annotated model. The left panel has the Java terminal model. This model is formed by class *Person*, with fields *name* and *address*, methods *getName ()* and *setName (String name)*; class *Address*, with fields *number*, *street* and *city*. The right panel has the annotation weaving model. It annotates classes *Person* and *Address*, and method *getName*. This shows that it is possible to annotate different types of elements using the same mechanism. The annotations can have one or more properties. For instance, class *Person* is annotated with a property release, value 1.0; method *getName* is annotated with two properties, *override* and *canBeNull*.

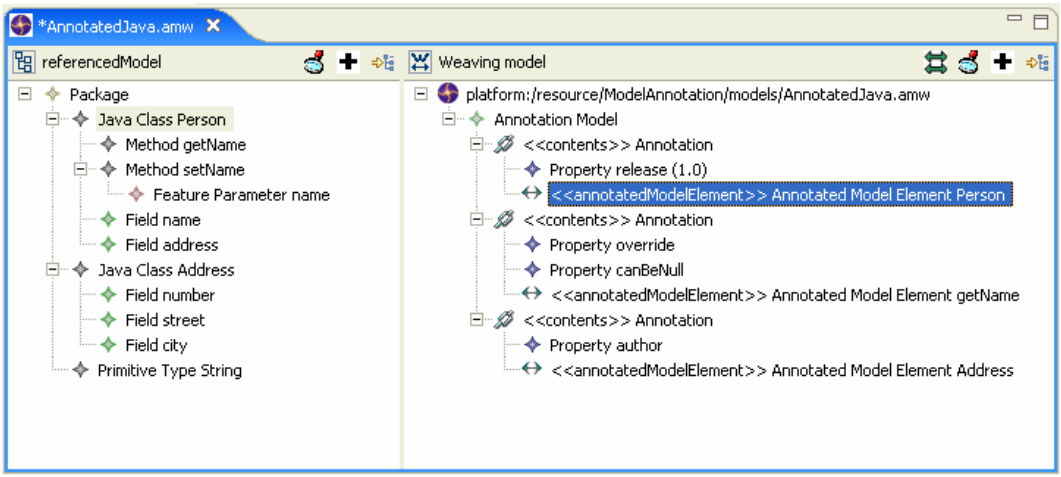


Figure 7.23 Annotations in the Atlas Model Weaver

This use case shows how weaving models are used to annotate models. We develop a generic metamodel extension for annotations. The extension supports any annotation in the form *key-value*. This approach has many advantages. The creation of the annotations in a separated weaving model avoids polluting the Java model with additional information, keeping the design clean. This is particularly useful when using different types of annotations. The annotation metamodel can be extended to add specific types of annotations, not only our generic representation. This scenario also shows the flexibility of the extensible metamodels and of the AMW adaptive user interface. The model weaver showed it adapts well even in cases where only one model is woven. This use case is available for download at (http://www.eclipse.org/gmt/amw/usecases/annotation/).

7.8 Calculating the difference between models

This use case demonstrates how weaving models are used to save the differences between two KM3 models. Calculating the difference between two models is an essential process to control the changes and evolutions of models. The result of the difference is used to apply a patch in one of the models.

Consider a distributed development environment in which a KM3 model can be modified by different persons. There is one centralized repository that contains the "official" version *M (v1)*. A developer recuperates *M (v1)*, creates a personal working copy *M (v2)*, and modifies it. Both versions must be synchronized (i.e., a new up-to-date version is created), as shown in Figure 7.24. The synchronization (*Patch* operation) is based on a difference weaving model. This model is calculated using a *Diff* operation.

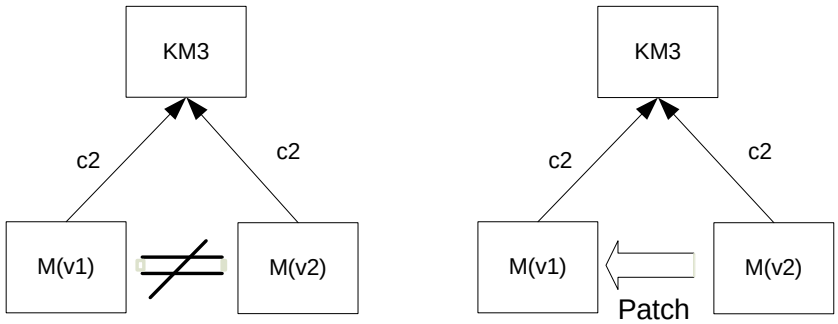


Figure 7.24 Difference and patch

The general difference process is divided in two main phases:

Matching: the matching phase identifies the elements that did not change between the two models. The result of a matching is saved in a weaving model, which contains equivalence links between the elements that were not modified in both KM3 models. This phase is encapsulated in the *Match* operation.

Difference calculation: the result weaving model is used to compute which elements were added, removed or modified. The difference algorithm is implemented using ATL transformations. It produces another weaving model that represents the difference between the two versions. This weaving model conforms to the metamodel extension for difference shown in Figure 7.25.

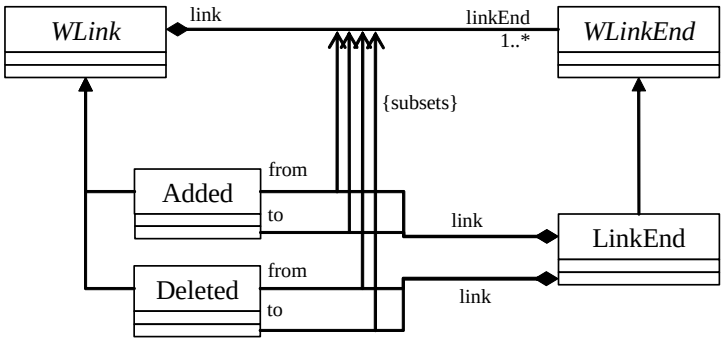


Figure 7.25 Metamodel extension for difference

This extension supports deletion or inclusion of elements. *Added* links are created for the elements that were added in the repository model. *Deleted* links are created for the elements that were removed. For instance, if a Class A is added in the Package B of the repository model, the addition element links the Package B (*from* reference) with Class A (*to* reference).

The screenshot in Figure 7.26 shows the difference between the two KM3 models. The models represent two versions of a relational database. The left panel contains the repository version. The right panel contains the working version, and the middle panel contains the different weaving model. The selected element is an *Added* link that indicates the attribute *value* is added into the class *Named* in the new version.

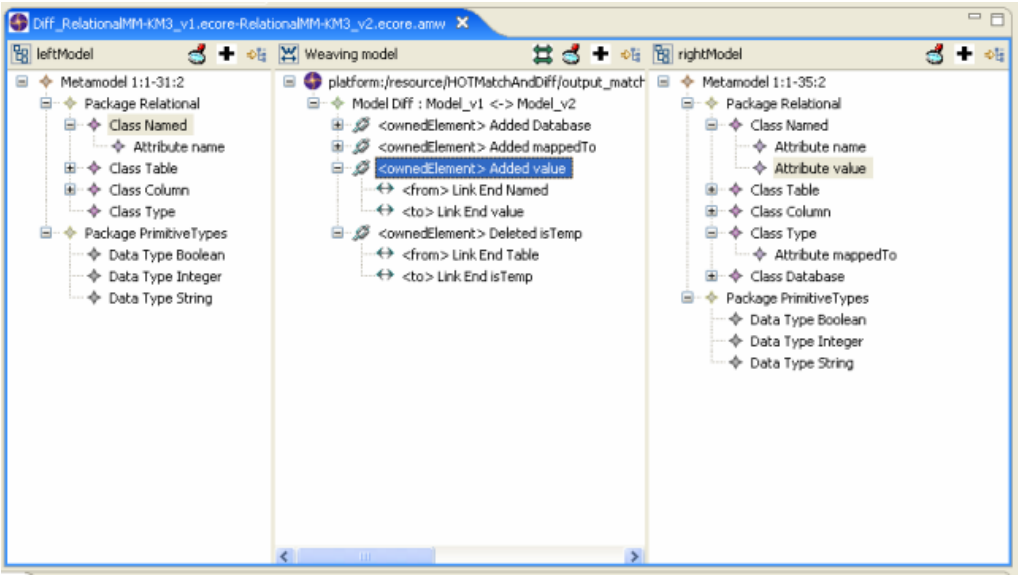


Figure 7.26 Difference weaving model between KM3 models

The difference weaving model is used as input to a *Patch* transformation. This transformation analyses the different kinds of links and executes the patch operation, i.e., addition or deletion of elements.

We generalize our solution to perform the difference and patch independently of the metamodel. In this case higher-order transformations are used to analyze the input metamodels (e.g., KM3, SQL) and to produce ATL transformations that execute the metamodel-specific *Match*, *Diff* and *Patch* operations.

To summarize, this use case shows how a weaving model can be used to deploy a model difference and patch solution. The possibility to save the result of a difference algorithm in a weaving model is an important result. This weaving model can be visualized in the generic interface, and it is used to apply a *Patch* operation afterwards. The difference weaving model can be stored to maintain a registry of the differences. Finally, we generalize our solution to be able to obtain the difference between models conforming to different metamodels. This use case is available for download at (http://www.eclipse.org/gmt/amw/usecases/diff/). It has been developed as part of a master thesis. This page contains general documentation, an example, a *HowTo* and the sources.

7.9 Conclusions

In this chapter we have presented eight use cases of model weaving. We have developed different extensions to the core weaving metamodel. The diversity of the use cases demonstrated that it is not possible to efficiently handle every application requirement by using general purpose mechanisms, such as transformation languages, or fixed mapping models. We created domain-specific metamodel extensions for all the use cases. We used the AMW tool to implement all the presented use cases, showing the genericity and the adaptability of our solution. Most of the use cases are based on real-world scenarios, with models of reasonable size and complexity. This also shows that our solution has achieved a reasonable maturity level to be used in industrial scenarios. Moreover, other use cases not presented in this thesis are publicly available for download at (http://www.eclipse.org/gmt/amw/usecases/). These use cases have been developed in the context on this thesis and also in collaboration with different organizations.

8 Conclusions

In this chapter we present the general conclusions of this thesis. First, we revisit the major issues for creating a generic model weaving solution. Then, we present the contributions of our approach. Finally, we present the future work, and the list of publications and the extra activities that have been carried out during this thesis.

8.1 Issues on model weaving

A model weaving solution should support the representation, computation and utilization of various kinds of relationships between model elements. We have identified a set of issues that have been investigated during this thesis.

- First, it is necessary to establish a coherent conceptual foundation (definitions and terminology).
- None of the existing approaches supported relationships with extensible representations. An extensible representation allows establishing relationships adapted to different application scenarios.
- It is necessary to define several kinds of relationships for different application scenarios. And, particularly, complex kinds of relationships targeted to data interoperability.
- It is necessary to define mechanisms to easily create, to reuse and to adapt matching techniques. They should support different relationships' specifications.
- The task of producing transformations based on a set of relationships should be factorized in a generic process.
- Finally, there is a lack of adaptive tools that support model weaving and related solutions.

8.2 Contributions of this thesis

In this thesis, we have presented model weaving. Model weaving is a new approach that encompasses the representation, computation and utilizations of relationships, providing a generic MDE solution for relationship (i.e., link) management. The major conclusion after studying extensive related work is that existing solutions lack adaptability and genericity in practically every aspect of relationship management.

We have identified the major aspects for relationship management and presented an inventory of existing solutions. This enabled us to present a set of definitions that unify different applications of model weaving. The basic requirements for relationship management are supported by a *weaving metamodel*. Weaving models conforming to this metamodel enable the creation of links between different model elements. However, through several experiments, we have shown that there are several different kinds of links. Thus, we presented a core weaving metamodel that is based on a strong extension mechanism. This is one of the central contributions of this thesis. The extension of

metamodels is performed using a metamodel extension operation. This operation uses weaving models to define the extension links between two metamodels. We have shown how to define the extension operation using a version of the core weaving metamodel that is bootstrapped with inheritance links. We were able to define different kinds of links adapted to different application scenarios; for instance, data interoperability, model merging, or traceability of model transformations.

However, we have seen that extensible metamodels have implications in every aspect of model weaving: representation, computation and utilization. Every time we create a new weaving metamodel extension, the set of tools and procedures to create and to use the weaving models must adapt to support these extensions.

Thus, we have presented a set of techniques to handle different weaving metamodels. With respect to the computation of weaving models, we have designed an adaptive tool named ATLAS Model Weaver (AMW). The tool has strong extension mechanisms. The user interface is auto-generated according to the metamodel extensions that are loaded. This means we were able to create weaving models conforming to any metamodel that is an extension to the core weaving metamodel.

In addition to the adaptive capabilities of AMW, we have presented matching transformations as a solution to semi-automate the creation of weaving models. The matching transformations enabled the implementation of matching techniques and the support of different metamodel extensions. We separated the overall matching process in different kinds of matching transformations. We have developed several existing techniques, and we improved a well-known technique. We take advantage of the relationships between metamodels to create different kinds of propagation of similarities. The complex kinds of links are created using link rewriting transformations. Moreover, we have extended the AMW tool to combine and to customize the execution of matching transformations. We create a configuration model with a set of customization parameters. The configuration model conforms to a configuration metamodel. The configuration model stores the parameters of a particular execution. This enables the fine-tuning of the execution of matching transformation, and the utilization of this tuning in further executions.

We have used model weaving and model transformations to improve existing data interoperability approaches. We have defined several kinds of links with variable complexity. These links were all expressed in terms of weaving metamodel extensions. This allows creating a hierarchy that organizes different kinds of links according to their semantic relations. The weaving models were used as specification for producing transformations. We encapsulated this task in a *TransfGen* model management operation. To the best of our knowledge, this is the first solution that factors out this task in a model management operation.

This operation is based on a generic pattern of transformation that specifies how to interpret the different kinds of links defined in the weaving models. The links are translated into elements of transformation models. The pattern may be incrementally modified to handle different semantic heterogeneities. This enables isolating the query discovery task in a single operation. Moreover, since we considered transformations as models, we were able to define this operation using higher-order transformations.

We validated the genericity of model weaving by developing several use cases with the AMW tool. The use cases showed the genericity and the adaptability of our solution. Most of the use cases are based on real-world scenarios, with models of reasonable size and complexity level. This also shows that our solution has achieved a reasonable maturity level to be used in industrial scenarios.

8.3 Summary of contributions

The major contributions of this thesis are the following:

- We have presented model weaving as a novel approach for relationship management. Model weaving is a unifying solution that subsumes the different aspects of representation, computation and utilization of relationships between different model elements.
- We have defined the conceptual foundations of model weaving. We presented a set of definitions for weaving models, metamodels, and metamodel extension operation. The extensible weaving metamodels can express different kinds of links.
- We have defined a set of weaving metamodel extensions and weaving models for different application scenarios. We validated these extensions by applying them in a set of case studies.
- We have factored out the process of producing transformations in a generic model management operation called *TransfGen*. This is a first step to generalize this task.
- We have presented matching transformations as a practical solution to semi-automatically create weaving models. This approach enabled to easily develop new or to adapt existing matching techniques.
- We have implemented an adaptive model weaving tool. The interface adapts to different metamodel extensions, and it is completely integrated with an existing transformation engine. The tool is available for download in the official site of Eclipse, and it has extensive documentation, examples, as well as a growing user community.

8.4 Future work

There are different issues that are subject for future work. We separate possible research efforts in three main topics: metamodel extensions, production of transformations and matching transformations.

8.4.1 Metamodel extensions

In Chapter 7, we have presented several use cases, using different metamodel extensions. The extensions covered different domains, such as model annotation or tool interoperability. The set of metamodel extensions can constitute a library of metamodel extensions. Such a library is already available in the site of the AMW tool. However, it can be enriched with different extensions. Once the number of extensions will be greater, the library should be reorganized by application-domain, to be able to easily find specific extensions.

There are different improvements that may be done in the case studies that have been presented, which are listed below:

- The extensions for interoperability may be refined with different kinds of links to capture a broader set of transformation expressions. The AMW tool can be deployed with different subsets of extensions according to the user needs.
- We have presented an extension to capture the difference between two models. However, it supports only the inclusion or exclusion of model elements. This approach may be extended to identify also the elements that are updated. It can take stock of existing approaches about difference. This information enables a better understanding of the changes executed in the model elements.
- The graphical user interface can be adapted with new functionalities; for instance, with a new weaving panel specific for model or metamodel comparison. A new implementation of the weaving panel can have an interface similar with the existing tools used for comparing Java files (e.g.,

CVS). The implementation of a similar interface can diminish the training time, since such tools are already widely used by developers.

- The composition of different models requires the utilization of precise operators, such as *override*, *merge*, or *inherit*. A composition operation specifies how to compose different models. There are different ways to compose models. The operation would be constructed using a set of primitive operators, which are not natively supported by general-purpose transformation languages. The set of primitive operators should be first identified. Then, a weaving metamodel extension could capture this set of operators [87]. The composition operation can be specified in two ways. First, by using the standard AMW interface. Second, a textual concrete syntax may be created, enabling the creation of the operation using an adapted textual editor.

The definition of concrete textual syntaxes for creating weaving models can be used in different scenarios different of model composition; for instance, model or metamodel annotations. In the AMMA architecture, the annotation of KM3 metamodels is done directly in the KM3 file, using a specific kind of comment followed by keyword (e.g., "--@version 1.0" or "--@author didonet"). These annotations should not be created directly in the metamodel. A concrete syntax for defining these annotations enables the storage of this additional information in a separated file. This file can be injected in the form of a weaving model, to be processed by a model transformation engine.

8.4.2 Production of transformations

The *TransfGen* pattern presented in Chapter 5 enabled the production of transformation models based on a weaving model that captures different kinds of links. The weaving models conform to a metamodel extension for interoperability scenarios. Once new kinds of links are added into this extension, the higher-order transformation that implements the operation should be extended as well. There are different issues that can be studied about the *TransfGen* operation.

The transformations that are generated are not guaranteed to be correct. They should be verified by domain-experts to validate the result. The manual verification of the correctness of these transformations should be minimized as much as possible, in order to automate the whole process. In addition, the *TransfGen* operation may be implemented in different ways, to produce transformations that are optimized for translating large models. The pattern has also been implemented to produce a bridge between KM3 and SQL-DDL. This case study should be compared with existing solutions of *ModelGen* operations.

8.4.3 Matching transformations

The matching transformations enabled the implementation of several matching techniques. To produce weaving models more accurate as possible, it is necessary to create new matching transformations implementing different techniques. Every time that new metamodel extensions are created, the existing transformations should be adapted as well. The solutions presented in this thesis should be tested with different kinds of models, for instance, ontologies or XML documents. The matching transformations should be evaluated with respect to their performance. These evaluations are necessary to be able to scale up to very large models. It is very important to have specialized mechanisms to verify the result of the matching transformations, especially for large models.

We have implemented matching transformations between a pair of models. However, it is technically possible to implement matching transformations between more than two models. This can be a typical scenario in complex systems (e.g., avionics systems) where several versions of different models may coexist. There are two solutions to be explored. First, the transformations can take extra models as input parameters, and the transformations rules can be modified to have more than two

elements in the input pattern. Second, the models can be linked pair-wise, and the result would be used as input to another matching transformation. The matching transformations implemented are based on lexical or structural properties. However, there are metamodels that provide less useful information that can be used in the matching than others. These metamodels could be enriched with specific annotations that could be used as additional input for the matching transformations.

The AMW tool stored the execution parameters of a set of transformation in a configuration model. We have defined a specific configuration model to be used in the metamodel comparison and model migration scenario. While it yielded good results, new combinations of transformations and execution parameters should be tested, especially in application scenarios that have not been explored. The parameterization of the matching transformations is a difficult problem, which requires the realization of several tests. We propose the creation of application-specific configuration models, with a set of optimized parameters and combination of transformations. These configuration models should be easily accessed by the user community, in order to reuse or to optimize existing parameters.

We have seen that the development of matching transformations helped on the implementation of different matching techniques because it diminishes the gap between the conceptual structures and the implementation. This motivates the creation of a domain-specific language (DSL) for developing matching techniques. The language may provide a set of built-in techniques with the most common matching techniques and with easy ways of parameterization. The language would have specialized keywords. A textual editor with syntax coloring can ease the task of finding errors and of developing new algorithms or heuristics. The language may also have a debugger, an outline, and navigation facilities. The matching language can be developed in the top of the AMMA platform, using KM3 to create the metamodel and TCS for injecting the concrete syntax into a model conforming to this metamodel. The language would be directly transformed into the byte-code executed by the virtual machine of ATL. This transformation should be optimized to produce a performing byte-code. This means the language would have the support of a complete development environment.

References

[1] Abiteboul, S, Cluet, S, Milo, T. Correspondence and Translation for Heterogeneous Data. In proc. of ICDT 1997, Dephi, Greece, pp 351-363

[2] AM3. ATLAS Megamodel Management. Ref. site: http://www.eclipse.org/gmt/am3, 03/2007

[3] AM3 Atlantic Zoo. Reference site: http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/, 10/2006

[4] AMW. The ATLAS Model Weaver. Reference site: http://www.eclipse.org/gmt/amw, 05/2007

[5] An, Y, Borgida, A, Miller, R, Mylopoulos, J. A Semantic Approach to Discovering Schema Mapping Expressions. In proc. of ICDE, Istanbul, Turkey, 2007 (to appear)

[6] An, Y, Borgida, A, Mylopoulos, J. Inferring Complex Semantic Mappings between Relational Tables and Ontologies from Simple Correspondences. In proc. of ODBASE'05, 2005, Agia Napa, Cyprus, pp 1152-1169

[7] An, Y, Borgida, A, Mylopoulos, J. Constructing Complex Semantic Mappings between XML Data and Ontologies. In proc. of ISWC'05, 2005, Galway, Ireland, pp 6-20

[8] Arenas, M, Libkin, L. XML data exchange: consistency and query answering. In proc. of the Symposium on Principles of Database Systems (PODS), 2005, Baltimore, USA, pp 13-24

[9] ATL: ATLAS Transformation Language. Reference site: http://www.eclipse.org/m2m/atl, 05/2007

[10] Atzeni, P, Cappellari, P, Bernstein, P A. A Multilevel Dictionary for Model Management. In proc. of ER 2005, Klagenfurt, Austria, pp 160-175

[11] Atzeni, P, Cappellari, P, Bernstein, P A. ModelGen: Model Independent Schema Translation. In proc. of ICDE 2005, Tokyo, Japan, pp 1111-1112

[12] Atzeni, P, Cappellari, P, Bernstein, P A. Model independent schema and data translation. In proc. of EDBT 2006, Munich, Germany, pp 368-385

[13] Barbero, M, Didonet Del Fabro, M, Bézivin, J. Traceability and Provenance Issues in Global Model Management. In: 3rd ECMDA-Traceability Workshop, Haifa, Israel 2007

[14] Barbero, M, Jouault, F, Gray, J, Bézivin, J. A Practical Approach to Model Extension. ECMDA-FA 2007, Haifa, Israel, pp 32-42

[15] Batini, C, Lenzerini, M, Navathe, S B. A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys, 18(4):323–364, 1986

[16] Bergamaschi, S, Castano, S, Vincini, M, Beneventano, D. Semantic Integration of Heterogeneous Information Sources. Data and Knowledge Engineering, 36(3), 215-249, 2001

[17] Berlin, J, Motro, A. Database Schema Matching Using Machine Learning with Feature Selection. In proc. of 14th Intl. Conf. Advanced Information Systems Engineering (CAiSE) 2002, Toronto, Canada, pp 452-466

[18] Bernstein, P A. Applying Model Management to Classical Meta Data Problems. In proc. of the 1st CIDR 2003, Asilomar, USA, pp 209-220

[19] Bernstein, PA, Haas, L M, Jarke, M, Rahm, E, Wiederhold, G. Panel: Is Generic Metadata Management Feasible? In proc. of VLDB 2000, Cairo Egypt, pp 660-662

[20] Bernstein, P A, Melnik, S, Petropoulos, M, Quix, C. Industrial-strength Schema Matching. ACM SIGMOD Record 33(4), 2004

[21] Bézivin, J, Bouzitouna, S, Didonet Del Fabro, M, Gervais, M, Jouault, F, Kolovos, D, Kurtev, I, Paige, R. A Canonical Scheme for Model Composition. In proc. of Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, LNCS 4066, edited by Arend Rensink and Jos Warmer. Springer-Verlag, Bilbao, Spain, pp 346-360

[22] Bézivin, J, Brunelière, H, Allilaire, F, Barbero, M, Didonet Del Fabro, M, Jouault, F. OMG MDA Tool Capabilities RFI. Response of ATLAS Group – INRIA, Response date: 2007-03-05. Reference site: http://www.omg.org/docs/mda-user/07-03-03.pdf

[23] Bézivin, J, Didonet Del Fabro, M, Jouault, F, Kurtev, I, Valduriez, P. Model Engineering: From Principles to Applications, Springer - 2007 (to appear)

[24] Bilke, A, Naumann, F. Schema Matching using Duplicates. In proc. Intl. Conf. Data Engineering (ICDE), 2005, Tokyo, Japan, pp 69-80

[25] Boronat, A, Carsi, J, Ramos, I. Exogenous Model Merging by means of Model Management Operators. In proc. of the Third Workshop on Software Evolution through Transformations: Embracing the Change, Recife, Brazil, 2006

[26] Brockmans, S, Haase, P, Stuckenschmidt, H. Formalism-Independent Specification of Ontology Mappings - A Metamodeling Approach. In proc. of ODBASE 06, Montpellier, France, pp 901-908

[27] Brunet, G, Chechik, M, Easterbrook, S, Nejati, S, Niu, N, Sabetzadeh, M. A manifesto for model merging. In proc. of Workshop. on Global Integrated Model Management (in ICSE'06), Shangai, China, 2006

[28] Bugzilla Bug Tracking Tool. Reference site: http://www.bugzilla.org, 06/ 2006

[29] Buneman, P, Davidson, S B, Kosky, A. Theoretical Aspects of Schema Merging. In proc. of Intl. Conf. on Extending Database Technology (EDBT). LNCS; vol. 580. Springer, Vienna, Austria, pp 152-167, 1992

[30] CBOP, DSTC, IBM (2003) MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-03

[31] Cicchetti, A, Di Ruscio, D, Pierantonio, A. A Metamodel-independent approach to difference representation. Journal of Object Technology (Special Issue from *TOOLS Europe 2007*), June 2007, 20 pages

[32] Clifton, C, Housman, E, Rosenthal, A. Experience with a Combined Approach to Attribute-Matching Across Heterogeneous Databases. In proc. of IFIP 2.6 Working Conf. Database Semantics, 1996

[33] Cohen, W, Ravikumar, P, Fienberg, S E. A Comparison of String Distance Metrics for Name-Matching Tasks. In proc. of IIWeb 2003, Acapulco, Mexico, pp 73-78

[34] Czarnecki, K, Eisenecker, U. Generative Programming. Methods, Tools and Applications. Addison-Wesley, 2000, ISBN 0-201-30977-77

[35] Czarnecki, K, Helsen, S. Classification of Model Transformation Approaches. In proc. of OOPSLA'03 Workshop on the Generative Techniques in the Context of Model-Driven Architecture, Anaheim, California, USA, 2003

[36] Czarnecki, K, Helsen, S. Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, special issue on Model-Driven Software Development. 45(3), 2006, pp 621-645

[37] DAML+OIL: Reference Description (W3C) (12/18/2001). Reference site: http://www.w3.org/TR/daml+oil-reference

[38] Dhamanka, R, Lee, Y, Doan, A H, Halevy, A, Domingos P. iMAP: Discovering Complex Semantic Matches between Database Schemas. In proc. of SIGMOD 2004, Paris, France, pp 383-394

[39] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Breton, E, Gueltas, G. AMW: a generic model weaver. In proc. of 1ères Journées sur l'Ingénierie Dirigée par les Modèles, 2005, Paris, France, pp 105-114

[40] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Valduriez, P. Applying Generic Model Management to Data Mapping. In proc. of BDA 2005, Saint-Malo, France, pp 343-355

[41] Didonet Del Fabro, M, Bézivin, J, Valduriez, P. Model-Driven Tool Interoperability: An Application in Bug Tracking. In proc. of The 5th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE'06), LNCS 4275, edited by R. Meersman and Z. Tari et al. Springer-Verlag Berlin Heidelberg 2006, Montpellier, France, pp 863-881

[42] Didonet Del Fabro, M, Bézivin, J, Valduriez, P. Weaving Models with the Eclipse AMW plugin. In proc. of Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany

[43] Didonet Del Fabro, M, Valduriez, P. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In proc. of The 22nd Annual ACM SAC, MT 2007 - Model Transformation Track, Seoul (Korea), pp 963-970

[44] Dion, B. Efficient Development of Safe Railway Applications Software with EN 50128 Objectives using SCADE Suite, Innotrans 2006

[45] Do, H H. Schema Matching and Mapping-based Data Integration. Verlag Dr. Müller (VDM), ISBN 3-86550-997-5, Phd thesis, Univesity of Leipzig, Germany, 2005

[46] Do, H H, Rahm, E. COMA - A System for Flexible Combination of Schema Matching Approaches. In proc. of VLDB 2002, Hong Kong, China, pp 610-621

[47] Doan, A H. Learning to Map between Structured Representations of Data. Phd Thesis. University of Washington. 2002

[48] Doan, A H, Domingos, P, Levy, A. Learning source descriptions for data integration. In proc. of WebDB Workshop 2000, Dallas, USA, pp 81-92

[49] Doan, A, Madhavan, J, Domingos, P, Halevy, A Y. Learning to map between ontologies on the semantic web. In proc. of WWW 2002, Honolulu, Hawai, USA, pp 662-673

[50] Doan A, Halevy A. Semantic Integration Research in the Database Community: A Brief Survey. AI Magazine, Special Issue on Semantic Integration, Spring 2005, pp 83-94

[51] Eclipse Foundation. Reference site: http://www.eclipse.org.

[52] Ehrig, M, Haase, P, Hefke, M, Stojanovic, N. Similarity for Ontologies - A Comprehensive Framework. In proc. of ECIS 2005, Regensburg, Germany

[53] Ehrig, M, Staab, S. QOM - Quick Ontology Mapping. In proc. of Intl. Semantic Web Conf. (ISWC), 2004, Hiroshima, Japan, pp 683-697

[54] Embley, DW, Jackmann, D, Xu, L: Multifaceted Exploitation of Metadata for Attribute Match Discovery in Information Integration. In proc. of Intl. Workshop Information Integration on the Web (WIIW), 2001, Rio de Janeiro, Brazil

[55] EMF. Eclipse Modeling Framework. Reference site: http://www.eclipse.org/emf, 04/2007

[56] Euzenat, J. An API for Ontology Alignment. In proc. of ISWC 2004, Hiroshima, Japan, pp 698-712

[57] Fellbaum, C. WordNet, an Electronic Lexical Database. MIT Press, 1998. Reference site: http://wordnet.princeton.edu/

[58] Flanakin, M. Web Log. Comments and complaints on software and technology in general. Comparison: Web-based Tracker. 08/08/2005. http://geekswithblogs.net/flanakin/articles/CompareWebTrackers.aspx

[59] Fletcher, G, Wyss, C M. Relational data mapping in MIQIS. SIGMOD (Demo), 2005, Baltimore, Maryland, USA

[60] Fuxman, A, Hernandez, M A, Ho, H, Miller, R J, Papotti, P, Popa, L. Nested Mappings: Schema Mapping Reloaded. In proc. of VLDB 2006, Chicago, Illinois, USA, pp 67-78

[61] Garcia-Molina, H, Hammer, J, Ireland, K, Papakonstantinou, Y, Ullman, J, Widom, J. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In proc. of the AAAI Symposium on Information Gathering, Stanford, California, USA, March 1995

[62] Geographic matters. An ESRI White Paper - September 2002

[63] Georg, G, France, R, Ray, I. Composing Aspect Models. In proc. of The 4th AOSD Modeling With UML Workshop UML 2003. San Francisco, CA, USA

[64] Giunchiglia, F, Shvaiko, P, Yatskevich, M. S-Match: an Algorithm and an Implementation of Semantic Matching. In proc. of 1st European Semantic Web Symposium (ESWS), 2004, Crete, Greece, pp 61-75

[65] Giunchiglia, F, Yatskevich, M. Efficient Semantic Matching. In proc. of 2nd European Semantic Web Conf. (ESWC), 2005, Crete, Greece, pp 272-289

[66] GMT Project. Generative Modeling Technologies. Reference site: http://www.eclipse.org/gmt/, 05/2007

[67] Gray, J, Lin, Y, Zhang, J. Automating Change Evolution in Model-Driven Engineering, IEEE Computer (Special Issue on Model-Driven Engineering - Doug Schmidt, ed.), vol. 39, no. 2, February 2006, pp 51-58

[68] Harel, D, Politi, M. Modeling Reactive Systems With Statecharts. The Statemate Approach. McGraw Hill, 1998

[69] Hoshiai, T, Yamane, Y, Nakamura, D, Tsuda, H. A Semantic Category Matching Approach to Ontology Alignment. In proc. of 3rd Intl. Workshop Evaluation of Ontology based Tools, 2004

[70] Hull, R, King, R. Semantic Database Modeling: Survey, Applications, and Research Issues. ACM Comput. Surv. 19(3): 201-260 (1987)

[71] Jossic, A, Didonet Del Fabro, M, Lerat, J, Bézivin, J, Jouault, F. Model Integration with Model Weaving: a Case Study in System Architecture. In proc. of International Conference on Systems Engineering and Modeling – ICSEM 2007, Haifa, Israel

[72] Jouault, F. Loosely Coupled Traceability for ATL. In proc. of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany. 2005

[73] Jouault, F. Contribution à l'étude des langages de transformation de modèles. Ph.D. thesis, Université de Nantes. 2006

[74] Jouault, F, Bézivin, J. KM3: a DSL for Metamodel Specification. In proc. of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp 171-185

[75] Jouault, F, Kurtev, I. Transforming Models with ATL. In proc. of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, pp 128-138

[76] JWNL (Java WordNet Library). Ref. site: http://sourceforge.net/projects/jwordnet. August 2006

[77] Kalfoglou, Y, Schorlemmer, W M. Ontology Mapping: The State of the Art. Semantic Interoperability and Integration, 2005

[78] Kalnins, A, Barzdins, J, Celms, E. Model Transformation Language MOLA. Edited by Uwe ASSMANN, Mehmet AKSIT et Arend RENSINK, réds., Model Driven Architecture, European MDAWorkshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The

Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers, LNCS 3599, pp 62-76

[79] Karsai, G, Agrawal, A, Shi, F, Sprinkle, J. On the Use of Graph Transformation in the Formal Specification of Model Interpreters.J. UCS, 9(11):1296–1321, 2003

[80] Kedad, Z, Bouzeghoub, M. Discovering View Expressions from a Multi-Source Information System. In proc. of CoopIS 1999. Edinburgh, Scotland, pp 57-68

[81] Kedad, Z, Xue, X. Mapping discovery for XML data integration. In proc. of CoopIS 2005, Agia Napa, Cyprus, November 2005, pp 166-182

[82] Kementsietsidis, A, Arenas, M, Miller, R J. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In proc. of ACM SIGMOD Intl. Conf. Management of Data, 2003, San Diego, USA, pp 325-336

[83] Kensche, D, Quix, C, Chatti, M A, Jarke, M. GeRoMe: A Generic Role Based Metamodel for Model Management. OTM Conferences (2) 2005, Agia Napa, Cyprus, pp 1206-1224

[84] Kolovos, D S. Extensible Platform for Specification of Integrated. Languages for mOdel maNagement (Epsilon), Official Web-Site. http://www.cs.york.ac.uk/~dkolovos/epsilon. 04/2007

[85] Kolovos, D, Paige, R F, Polack, F A C. Merging Models with the Epsilon Merging Language (EML), In proc. of ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006), Genova, Italy, pp 215-229

[86] Kurtev, I, Bézivin, J, Aksit, M. Technological Spaces: An Initial Appraisal. In proc. of CoopIS, DOA'2002 Federated Conferences, Industrial track, 2002, Irvine, California, USA

[87] Kurtev, I, Didonet Del Fabro, M. A DSL for Definition of Model Composition Operators. In proc. of 2nd Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, ECOOP 2006, Nantes, France. July 2006

[88] Lawley, M, Steel, J. Practical Declarative Model Transformation with Tefkat. In proc. of Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844, pp 139-150. Springer Berlin / Heidelberg, January 2006

[89] Lenzerini, M. Data Integration: A Theoretical Perspective. In proc. of PODS 2002, Madison, Wiscosin, USA, pp 233-246

[90] Li, W S, Clifton, C. SemInt - A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Network. Data and Knowledge Engineering 33(1), 49-84, 2000, 85

[91] Lin, Y, Gray, J, Jouault, F. DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems (Special Issue on Model-Driven Systems Development), Fall 2007

[92] M2M. Model-to-Model Transformation project. Ref. site: http://www.eclipse.org/m2m/, 05/2007

[93] Maedche, A, Motik, B, Silva, N, Volz, R. MAFRA - a mapping framework for distributed ontologies. In proc. of EKAW 2002, Siguenza, Spain, pp 235-250

[94] Madhavan, J, Bernstein, P A, Chen, K, Halevy, A, Shenoy, P. Corpus-based Schema Matching. In proc. of Workshop Information Integration on the Web at IJCAI, 2003, Acapulco, Mexico, pp 1567-1572

[95] Madhavan, J, Bernstein, P A, Domingos, P, Halevy, A. Representing and Reasoning about Mappings between Domain Models. In proc. of AAAI/IAAI 2002, Edmonton, Canada, pp 80-86

[96] Madhavan, J, Bernstein, P A, Rahm, E. Generic Schema Matching Using Cupid, In proc. of VLDB 2001, Roma, Italy, pp 49-58

[97] Madhavan, J, Halevy, A Y. Composing Mappings Among Data Sources. In proc. of VLDB 2003, Berlin, Germany, pp 572-583

[98] Mantis Bug Tracking System. Reference site: http://www.mantisbt.org/, 06/2006

[99] Melnik, S. Generic Model Management: Concepts and Algorithms, Ph.D. Dissertation, University of Leipzig, Springer LNCS 2967, 2004

[100] Melnik, S, Bernstein, P A, Halevy, A, Rahm, E. Supporting Executable Mappings in Model Management. In proc. of SIGMOD 2005, Maryland, USA, pp 167-178

[101] Melnik, S, Molina, H G, Rahm, E. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In proc. of ICDE 2002, San Jose, California, USA, pp 117-128

[102] Miller, R, Haas, L, Hernández, M. Schema Mapping as Query Discovery. In proc. of VLDB 2000, Cairo, Egypt, pp 77-88

[103] Miller, R J, Hernandez, M A, Haas, L M, Yan, L-L, Ho, C T H, Fagin, R, Popa, L. The Clio Project: Managing Heterogeneity. In proc. of SIGMOD Record 30, 1, 2001, pp 78–83

[104] Milo, T, Zohar, S. Using Schema Matching to Simplify Heterogeneous Data Translation. In proc. of VLDB 1998, New York City, USA, pp 122-133

[105] Mitra, P, Wiederhold, G. Resolving Terminological Heterogeneity in Ontologies. In proc. of Workshop on Ontologies and Semantic Interoperability at the 15th European Conference on Artificial Intelligence (ECAI), 2002, Lyon, France

[106] Mitra, P, Wiederhold, G, Kersten, M L. A Graph-Oriented Model for Articulation of Ontology Interdependencies. In proc. of EDBT 2000. Konstanz, Germany, pp 86-100

[107] Morishima, A, Okawara, T, Tanaka, J, Ishikawa, K. SMART: a tool for semantic-driven creation of complex XML mappings. In proc. of SIGMOD 2005 (demo), Maryland, USA, pp 909-911

[108] Nash, A, Bernstein, P A, Melnik, S. Composition of mappings given by embedded dependencies. In proc. of PODS 2005, Maryland, USA, pp 172-183

[109] Nejati, S, Sabetzadeh, M, Chechik, M, Easterbrook, S, Zave, P. Matching and Merging of Statecharts Specifications, In proc. of 29th ICSE, Minneapolis, USA, May 2007

[110] Noy, N, Musen, M. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In proc. of AAAI/IAAI 2000, Austin, Texas, USA, pp 450-455

[111] Omelayenko, B. RDFT: A Mapping Meta-Ontology for Web Service Integration. Knowledge Transformation for the Semantic Web 2003. pp 137-153

[112] OMG. Human Usable Textual Notation (HUTN) Specification, Final Adopted Specification. (ptc-02-12-01)

[113] OMG. Meta Object Facility (MOF) Specification, version 1.4, OMG Document formal/2002-04-03, 2002. Reference site : http://www.omg.org/technology/documents/formal/mof.htm

[114] OMG. MOF QVT Final Adopted Specification, OMG Document ptc/2005-11-01, 2005. Reference site http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[115] OMG. Object Management Group. Reference site: http://www.omg.org. 04/2007

[116] OpenGIS. Geographic Markup Language (GML) Implementation Specification, January, 29th, 2003. https://portal.opengeospatial.org/files/?artifact_id=7174

[117] OWL. Web Ontology Language. 10/02/2004. Reference site: http://www.w3.org/2004/OWL/

[118] Palopoli, L, Terracina, G, Ursino, D. The System DIKE - Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses. In proc. of ADBIS-DASFAA 2000, Prague, Czech Republic, pp 108-117

[119] Popa, L, Velegrakis, Y, Hernandez, M, Miller, R J, Fagin, R. Translating Web Data. In proc. of 28th International Conference for Very Large Databases (VLDB 2002), August 2002, Hong Kong, China, pp 598-609

[120] Pottinger, R A, Bernstein, P A. Merging Models Based on Given Correspondences. In proc. of VLDB 2003. Berlin, Germany, pp 862-873

[121] Rahm, E, Bernstein, P A. A Survey of Approaches to Automatic Schema Matching, VLDB Journal 10, 4 (Dec. 2001), pp 334-350

[122] Rahm, E, Bernstein, P A. An Online Bibliography on Schema Evolution. SIGMOD Record 35(4): 30-31 (2006)

[123] Ramesh, B, Jarke, M. Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering. V27 , Issue 1 (January 2001), pp 58-93

[124] RDF – Resource Description Framework. W3C Specification. Reference site: http://www.w3.org/RDF/

[125] Scalable Vector Graphics Specification (1.1) W3C Recommendation. 14 January 2003. http://www.w3.org/TR/SVG/

[126] Sheth, A P, Larson, J A. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Comput. Surv. 22(3): 183-236 (1990)

[127] Shvaiko, P, Euzenat, J. A Survey of Schema-Based Matching Approaches. Journal of Data Semantics IV: 146-171 (2005)

[128] SimMetrics. Developed by Sam Chapman. Reference site: http://sourceforge.net/ projects/simmetrics/, 08/2006

[129] Stirewalt, R E K, Rugaber, S. Automating user interface generation by model composition. In proc. of The IEEE International Conference on Automated Software Engineering, 1999

[130] Taentzer, G, Ehrig, K, Guerra, E, Lara, J, Lengyel, L, Levendovszky, T, Prange, U, Varro, D, Varro-Gyapay, S. Model Transformation by Graph Transformation: A Comparative Study. In proc. of the Model Transformations in Practice (MTiP) Workshop at MoDELS 2005, 2005

[131] UML. Unified Modeling Language. OMG Specification. Version 2.1.1. 01/04/2005. Reference site: http://www.omg.org/technology/documents/formal/uml.htm

[132] Vangheluwe, H, de Lara, J. Domain-Specific Modelling with AToM3. In proc. of 4th OOPSLA Workshop on Domain-Specific Modeling, Vancouver, Canada, October 2004

[133] Varro, D, Varro, G, Pataricza, A. Designing the Automatic Transformation of Visual Languages. Science of Computer Programming, 44(2):205–227, August 2002

[134] Velegrakis, Y, Miller, R J, Mylopoulos, J. Representing and Querying Data Transformations. In International Conference on Data Engineering (ICDE) 2005, Tokyo, Japan, pp 81-92

[135] Velegrakis, Y, Miller, R J, Popa, L. Mapping Adaptation under Evolving Schemas. In proc. of VLDB 2003, Berlin, Germany, pp 584-595

[136] Wang, J, Wen, J, Lochovsky, F, Ma, W. Instance-based Schema Matching for Web Databases by Domain-specific Query Probing. In proc. of VLDB, 2004, Toronto, Canada, pp 408-419

[137] Wiederhold, G. Database Design (2nd edition). New York: McGraw-Hill, 1982

[138] Willink, E. UMLX: A graphical transformation language for MDA. In proc. of Arend RENSINK, ed., CTIT Technical Report TR-CTIT-03-27, pp 13–24, Enschede, The Netherlands, june 2003. University of Twente

[139] XSD: XML Schema. W3C Specification. 28/10/2004. Reference site: http://www.w3.org/XML/Schema

[140] Xu, L, Embley, D. Discovering Direct and Indirect Matches for Schema Elements. In proc. of Intl. Conf. Database Systems for Advanced Applications DASFAA 2003, Kyoto, Japan, pp 39-46

List of tables

Table 3.1 Summary of the matching tools 56
Table 3.2 Adaptability of tools 57
Table 3.3 Query discovery approaches 60

List of figures

Figure 2.1 A model representing a system ... 35
Figure 2.2 Three-level modeling architecture ... 35
Figure 2.3 Model transformation base schema ... 36
Figure 2.4 Model weaving feature diagram ... 37
Figure 2.5 Representation feature diagram ... 38
Figure 2.6 Computation feature diagram ... 39
Figure 2.7 Relationships created using simple techniques ... 40
Figure 2.8 Complex relationships ... 40
Figure 2.9 Utilization feature diagram ... 41
Figure 3.1 Single access point ... 49
Figure 3.2 Multiple access point ... 50
Figure 3.3 Mappings in data interoperability ... 51
Figure 3.4 Mappings are used to produce executable programs ... 51
Figure 4.1 Modeling relations ... 73
Figure 4.2 Self-definition of KM3 ... 73
Figure 4.3 The core weaving metamodel ... 75
Figure 4.4 A simple weaving model ... 76
Figure 4.5 The extension operation ... 77
Figure 4.6 A set of DSWMs extensions ... 78
Figure 4.7 Model weaver workbench ... 79
Figure 4.8 The AMW tool ... 83
Figure 5.1 Bugzilla metamodel ... 86
Figure 5.2 Mantis metamodel ... 87
Figure 5.3 Metamodel extensions for data interoperability ... 93
Figure 5.4 Generic transformation metamodel ... 95
Figure 5.5 Models in the *TransfGen* operation ... 96
Figure 5.6 Higher-order transformation pattern ... 97
Figure 6.1 Two simple metamodels ... 102
Figure 6.2 ATL transformation ... 103
Figure 6.3 General overview of the matching components ... 104
Figure 6.4 Matching extensions ... 106
Figure 6.5 Creation of equivalence links ... 107
Figure 6.6 Rule that creates links between classes and/or attributes ... 107
Figure 6.7 Rule that create links between models conforming to different metamodels ... 107
Figure 6.8 Simple element-to-element similarity rule ... 109
Figure 6.9 Structural similarity rule ... 110
Figure 6.10 Weaving propagation metamodel extension ... 111
Figure 6.11 Creation of propagation edges ... 111

Figure 6.12 Propagation in ATL ... 113
Figure 6.13 Link filtering method ... 114
Figure 6.14 Rewriting of attribute-equal links ... 115
Figure 6.15 Metamodel extension for elements not linked ... 116
Figure 6.16 Link rewriting method ... 116
Figure 6.17 Match parameter metamodel ... 117
Figure 6.18 Matching transformation configuration ... 118
Figure 7.1 A weaving model in HUTN ... 122
Figure 7.2 Higher-order transformation ... 123
Figure 7.3 Injection of a SQL-DDL file into an SQL-DDL model ... 125
Figure 7.4 SQL-DDL textual syntax ... 125
Figure 7.5 Weaving between SQL-DDL and KM3 ... 126
Figure 7.6 Resulting KM3 ... 126
Figure 7.7 Relational to XML mapping ... 127
Figure 7.8 Extended weaving metamodels ... 128
Figure 7.9 Generated XSLT and ATL ... 128
Figure 7.10 Weaving in the AMW prototype ... 129
Figure 7.11 Metamodel extension for comparison ... 130
Figure 7.12 Transformation of a comparison weaving model into ATL ... 131
Figure 7.13 A part of the transformation generated automatically ... 131
Figure 7.14 Book to publication transformation ... 132
Figure 7.15 Traceability between two model elements ... 133
Figure 7.16 Traceability metamodel extension in KM3 ... 133
Figure 7.17 *Mt'* generates a *Publication* model and a weaving model ... 134
Figure 7.18 Data mapping of election results ... 135
Figure 7.19 Weaving GML and Election metamodels into SVG ... 136
Figure 7.20 Generated ATL and XSLT ... 137
Figure 7.21 Annotating a Java model with a weaving model ... 138
Figure 7.22 Metamodel extension for annotation ... 138
Figure 7.23 Annotations in the Atlas Model Weaver ... 139
Figure 7.24 Difference and patch ... 140
Figure 7.25 Metamodel extension for difference ... 140
Figure 7.26 Difference weaving model between KM3 models ... 141

Appendix A

This appendix lists a set of key web resources about the ATLAS Model Weaver tool and model weaving.

Official site
http://www.eclipse.org/gmt/amw/
The official site of the AMW tool contains extensive information about the tool: it explains the base concepts; it contains a set of use cases and it provides user documentation. The site is divided in different sections.

Use Cases
http://www.eclipse.org/gmt/amw/usecases/
The use cases section contains a list of use cases of AMW. All the use cases presented in this thesis are available for download in this section. They are presented through a short overview. The use cases contain the sources (metamodel extensions, model transformations, weaving models) and additional user documentation. This section also contains additional use cases not introduced in this thesis, which cover a variety of application scenarios.

Metamodel extensions Zoo
http://www.eclipse.org/gmt/amw/zoo/
This section contains the metamodel extensions to the core weaving metamodel presented in this thesis and also used in the different application scenarios.

Wiki
http://wiki.eclipse.org/index.php/AMW
The AMW Wiki contains general information about the tool. It informally introduces model weaving and the core weaving metamodel. The AMW Wiki describes how to use the tool using the standard interface, how to extend the tool for different applications and how to use a set of advanced features.

Download
http://www.eclipse.org/gmt/amw/download/
The AMW tool can be downloaded in this section. This section describes the steps to download and install the tool.

Gestion de Métadonnées Utilisant Tissage et Transformation de Modèles

Résumé

L'interaction et l'interopérabilité entre différentes sources de données sont une préoccupation majeure dans plusieurs organisations. Ce problème devient plus important encore avec la multitude de formats de données, APIs et architectures existants. L'ingénierie dirigée par modèles (IDM) est un paradigme relativement nouveau qui permet de diminuer ces problèmes d'interopérabilité. L'IDM considère toutes les entités d'un système comme un modèle. Les plateformes IDM sont composées par des types de modèles différents. Les modèles de transformation sont des acteurs majeurs de cette approche. Ils sont utilisés pour définir des opérations entre modèles. Par contre, il y existe d'autres types d'interactions qui sont définies sur la base des liens. Une solution d'IDM complète doit supporter des différents types de liens. Les recherches en IDM se sont centrées dans l'étude des transformations de modèles. Par conséquence, il y a beaucoup de travail concernant différents types des liens, ainsi que leurs implications dans une plateforme IDM.

Cette thèse étudie des formes différentes de liens entre les éléments de modèles différents. Je montre, à partir d'une étude des nombreux travaux existants, que le point le plus critique de ces solutions est le manque de généricité, extensibilité et adaptabilité. Ensuite, je présente une solution d'IDM générique pour la gestion des liens entre les éléments de modèles. La solution s'appelle le tissage de modèles. Le tissage de modèles propose l'utilisation de modèles de tissage pour capturer des types différents de liens. Un modèle de tissage est conforme à un métamodèle noyau de tissage. J'introduis un ensemble des définitions pour les modèles de tissage et concepts liés. Ensuite, je montre comment les modèles de tissage et modèles de transformations sont une solution générique pour différents problèmes d'interopérabilité des données. Les modèles de tissage sont utilisés pour générer des modèles de transformations. Ensuite, je présente un outil adaptive et générique pour la création de modèles de tissage. L'approche sera validée en implémentant un outil de tissage appelé AMW (ATLAS Model Weaver). Cet outil sera utilisé comme solution de base pour différents cas d'applications.

Mots-clés: tissage de modèles, transformation de modèles, interopérabilité des données, ingénierie des modèles

Metadata Management Using Model Weaving and Model Transformation

Abstract

The interaction and interoperability between different data sources is a major concern in many organizations. The different formats of data, APIs, and architectures increases the incompatibilities, in a way that interoperability and interaction between components becomes a very difficult task. Model driven engineering (MDE) is a paradigm that enables diminishing interoperability problems by considering every entity as a model. MDE platforms are composed of different kinds of models. Some of the most important kinds of models are transformation models, which are used to define fixed operations between different models. In addition to fixed transformation operations, there are other kinds of interactions and relationships between models. A complete MDE solution must be capable of handling different kinds of relationships. Until now, most research has concentrated on studying transformation languages. This means additional efforts must be undertaken to study these relationships and their implications on a MDE platform.

This thesis studies different forms of relationships between models elements. We show through extensive related work that the major limitation of current solutions is the lack of genericity, extensibility and adaptability. We present a generic MDE solution for relationship management called model weaving. Model weaving proposes to capture different kinds of relationships between model elements in a weaving model. A weaving model conforms to extensions of a core weaving metamodel that supports basic relationship management. After proposing the unification of the conceptual foundations related to model weaving, we show how weaving models and transformation models are used as a generic approach for data interoperability. The weaving models are used to produce model transformations. Moreover, we present an adaptive framework for creating weaving models in a semi-automatic way. We validate our approach by developing a generic and adaptive tool called ATLAS Model Weaver (AMW), and by implementing several use cases from different application scenarios.

Keywords: model weaving, model transformations, data interoperability, model driven engineering
Discipline: Informatique