Ph.D. Thesis

From Mining to Mapping and Roundtrip Transformations – A Systematic Approach to Model-based Tool Integration

Conducted for the purpose of receiving the academic title
'Doktor der Sozial- und Wirtschaftswissenschaften'

Advisors
o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Institute of Software Technology and Interactive Systems
Vienna University of Technology

a.Univ.-Prof. Mag. Dr. Werner Retschitzegger
Institute of Bioinformatics
Johannes Kepler University Linz

Submitted at the
Vienna University of Technology
Faculty of Informatics
by

Manuel Wimmer
0025221
Schanzstrasse 49/4
A-1140 Vienna

Vienna, March 2008 _____

Danksagung

Mein besonderer Dank gilt meiner Betreuerin und meinem Betreuer, Prof. Gerti Kappel und Prof. Werner Retschitzegger. Bei Gerti Kappel möchte ich mich für ihr Vertrauen und für die Chance bedanken, dass ich als wissenschaftlicher Mitarbeiter in Forschung, Lehre und Projekten gleichermaßen agieren durfte. Des Weiteren bedanke ich mich bei Gerti Kappel für die Möglichkeit beim Aufbau einer Lehrveranstaltung mit komplett neuem Forschungshintergrund und Lehrinhalten mitwirken zu können; eine Tätigkeit, die mir sehr viel Freude bereitete. Bei Werner Retschitzegger möchte ich mich besonders dafür bedanken, dass er nicht nur stets die richtigen Antworten bereit hatte, sondern auch immer die richtigen Fragen stellte. Außerdem war er mir eine große Stütze in der Finalisierungsphase.

Diese Arbeit wäre sicher nicht in dieser Breite und Tiefe entstanden ohne Mithilfe meiner InstitutskollegInnen. Besonders möchte ich bei den ModelCVS[1] Mitstreitern Thomas Reiter (jede Fahrt nach Linz war die Zeit und das Geld wert!), Horst Kargl, Gerhard Kramler, Michael Strommer und Wieland Schwinger bedanken. Vielen Dank für eure Mithilfe! Für eine sehr produktive Zusammenarbeit in Forschung und Lehre bedanke ich mich bei Andrea Schauerhuber, die mir stets mit Rat und Tat zur Seite stand.

Ein großer Dank gilt auch meiner Mutter Ingrid, ohne sie hätte ich diesen Weg nie bis hierher gehen können. Sie war für mich Vater und Mutter in einer Person und lehrte mich schon früh, dass Konsequenz und gewissenhafte Arbeit der Grundstein für jeden Erfolg darstellt. Besonderer Dank gilt auch meiner Freundin Shila Nourzad, die mir eine treue Begleiterin über die gesamte Dissertationsphase war. Im Besonderen möchte ich mich bei ihr für ihre motivierenden und Kräfte spendenden Worte und für ihr Verständnis für die zeitlichen Einschränkungen in den letzten Jahren bedanken.

[1] ModelCVS Projekt (FIT-IT Nr. 810806), www.modelcvs.org

Abstract

Model-Driven Engineering (MDE) gains momentum in academia as well as in practice. A wide variety of modeling tools is already available supporting different development tasks and advocating different modeling languages. In order to fully exploit the potential of MDE, modeling tools must work in combination, i.e., a seamless exchange of models between different modeling tools is crucial for MDE. Current best practices to achieve interoperability use model transformation languages to realize necessary mappings between the metamodels defining the modeling languages supported by different tools. However, the development of such mappings is still done in an ad-hoc and implementation-oriented manner which simply does not scale for large integration scenarios. The reason for this is twofold. First, various modeling languages are not based on metamodeling standards but instead define proprietary languages rather focused on notational aspects. And second, existing model transformation languages both do not support expressing mappings on a high-level of abstraction and lack appropriate reuse mechanisms for already existing integration knowledge.

This thesis contributes to the above mentioned problems. It proposes a comprehensive approach for realizing model-based tool integration, which is inspired from techniques originating from the field of database integration, but employed in the context of MDE. For tackling the problem of missing metamodel descriptions, a semi-automatic approach for mining metamodels and models from textual language definitions is presented, being a prerequisite for the subsequent steps which are based on metamodels and models, only. For raising the level of abstraction and for ensuring the reuse of mappings between metamodels, a framework is proposed for building, applying, and executing reusable mapping operators. To demonstrate the applicability of the framework, it is applied to the definition of mapping operators which are intended to resolve typical structural heterogeneities occurring between the core concepts of metamodels. Finally, for ensuring roundtrip capabilities of transformations, two approaches are proposed evolving non-roundtripping transformations with roundtrip capabilities.

Kurzfassung

Die *modellgetriebene Softwareentwicklung* ist nicht nur ein aktueller Trend in der Informatikforschung, sondern spielt auch bereits in der Praxis eine wichtige Rolle. Für den praktischen Einsatz steht eine Palette an Modellierungswerkzeugen zur Verfügung, wobei jedes Werkzeug einen bestimmten Bereich im Entwicklungsprozess unterstützt und daher unterschiedliche Modellierungssprachen eingesetzt werden. Aufgrund der mangelnden *Interoperabilität* ist es allerdings nicht möglich, unterschiedliche Werkzeuge in Kombination einzusetzen und so bleibt das Potential der modellgetriebenen Softwareentwicklung zum Teil ungenützt. Um Interoperabilität zwischen Werkzeugen herzustellen, werden *Modelltransformationssprachen* für die Erstellung von so genannten *Mappings* (Abbildungen) zwischen Konzepten der Modellierungssprachen – definiert in Metamodellen – eingesetzt. Die Entwicklung von Mappings wird jedoch durch eine ad-hoc und implementierungsorientierte Vorgehensweise erschwert. Die Hauptgründe dafür sind: (1) für viele Modellierungssprachen existieren keine Metamodelle, und (2) Modelltransformationssprachen bieten weder Sprachelemente, um Mappings auf einer angemessenen Abstraktionsstufe zu definieren, noch stellen sie Mechanismen für die Wiederverwendung bereits vorhandener Mappings zur Verfügung.

Die vorliegende Dissertation entwickelt Lösungen für die oben genannten Probleme. Im Rahmen der Dissertation wird eine umfassende Infrastruktur für die modellbasierte Integration von Modellierungswerkzeugen aufgebaut. Der erste Teil präsentiert einen teilautomatischen Ansatz für die Erstellung von Metamodellen und Modellen aus textuellen Definitionen. Werden Modellierungssprachen nicht durch Metamodelle repräsentiert, ist dieser Schritt Voraussetzung, um die nachfolgenden Integrationstechniken anwenden zu können. Der zweite Teil beschäftigt sich mit der Erhöhung des Abstraktionsniveaus und der Wiederverwendung von vorhandenen Mappings. Dazu bietet das vorgestellte Framework Möglichkeiten, um wiederverwendbare *Mapping Operatoren* zu definieren, anzuwenden und auszuführen. Der Einsatz des Frameworks wird durch die Entwicklung einer *Mapping Sprache* demonstriert, die Mapping Operatoren für das Auflösen von wiederkehrenden Heterogenitäten zwischen Metamodellen umfasst. Der dritte und letzte Teil dieser Arbeit beschäftigt sich mit *Roundtrip Transformationen*. Dazu werden zwei Ansätze entwickelt, um existierende, nicht roundtrip-fähige Transformationen mit Roundtrip Funktionalität zu ergänzen.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Contents

1.1 Motivation

With the advent of Model Driven Engineering (MDE) [Sch06], seamless exchange of models among different modeling tools increasingly becomes a crucial prerequisite for effective model-driven software development processes. The Model Driven Architecture (MDA) initiative [OMG03a] of the Object Management Group[1], probably the most prominent protagonist of MDE, strongly focuses on the interoperability between tools by using standards to avoid vendor lock-in which was one of the main problems of earlier CASE tools [Iiv96]. However, in practice, the implementations of these standards are not fully compliant and with trends such as using domain-specific modeling languages instead of one standardized general purpose language, interoperability based on the usage of standards, only, cannot be achieved. Due to this lack of interoperability, however, it is often difficult to use tools in combination, thus the potential of model-driven software development cannot be fully utilized.

Therefore, this thesis aims at providing suitable concepts and mechanisms for realizing model-based tool integration, a research topic which lies in the intersection of the research areas tool integration, model-driven engineering, and information integration, which are introduced in the following three subsections.

[1] www.omg.org

1.1.1 Tool Integration

Research in tool integration has been a "hot" topic since the Stoneman Model was proposed at the end of the 70's and summarized by Brown et al. [BFW92] in two categories, the conceptual level ("*what is integration?*") and the mechanical level ("*how do we provide integration?*").

Conceptual level of integration. In general, commercial of the shelf tools are meant to be integrated if they function coherently and effectively in an environment as a whole, as is the case in an integrated development environment. Wasserman [Was89] is regarded as the first author who has suggested a categorization to describe the integration of tools from a functional point of view comprising integration in terms of platforms, GUIs, data, control, and processes. Other categorizations used for characterizing tool integration comprises depth of integration, varying from exchanging byte streams to semantics-preserving integration, and the universal applicability of the integration approach. In this thesis, we do not aim at providing one integrated modeling environment, instead the modeling tools should stay lossy coupled but should be able to exchange data between them in a semantically meaningful way. Thus, the models developed in one tool should also be accessible from other tools by transparent transformations.

Mechanical level of integration. The research efforts at the mechanical level of tool integration include (1) a series of standardization efforts and middleware services like CAIS [Obe88], PCTE [BGMT88], CDIF [Fla02], CORBA [OMG08], and OMG's recent request for proposals OTIF (open tool integration framework) to support tool interoperability, (2) architecture models, infrastructures, and tool suites like the ECMA toaster model [Ear90], the ToolBus architecture [BK98], and finally (3) basic tool integration mechanisms such as data sharing, data linkage, data interchange, and message passing [SD05]. Some of these efforts were often grounded in large initiatives but have not been widely accepted. The European standardization effort PCTE, e.g., supporting data integration by providing tools with a common repository and services to store, retrieve, and manipulate data was not widely adopted in industry, not least because of its heavyweight architecture and high usage costs.

Regarding, e.g., tool suites, they are often incomplete with respect to the various development activities requiring tool support, and most often do not allow to select between "best of class" tools (apart from promising exceptions like Eclipse) [SD05]. Despite of all these important efforts, tool integration is still a challenging task, leading most often to hand-crafted bilateral integration solutions [SD05]. These "solutions" suffer from high maintenance overheads not least in case of evolutions of the underlying data or tools themselves, are often strongly technology-dependent and, most importantly, do not scale.

1.1.2 Model-Driven Engineering

With the advent of Model-Driven Engineering (MDE) and in particular the introduction of Model-Driven Architecture (MDA) by the OMG, new possibilities have been opened up to cope with the aforementioned challenges. The key idea of MDE is to focus on models instead of code as the major artefact in software development. This allows modeling tools to be integrated on basis of the metamodels describing the modeling languages supported by the tools, i.e., the *tool metamodels*, thus paving the way for another generation of (meta)model-based tool integration approaches and providing a basis to overcome the above mentioned limitations of existing integration approaches. For this, MDA includes a set of interrelated standards, comprising a language for metamodel definition (Meta Object Facility - MOF [OMG04]), and the MOF-compliant languages for constraint specification (Object Constraint Language - OCL [OMG05d]), model transformation (Query/View/-Transformations - QVT [OMG05b]), and (meta)data interchange (XML Metadata Interchange - XMI [OMG05c]).

XMI is the proposed model interchange format, however, it has only a very limited scope and some drawbacks. First, XMI import/export can only be used if two tools are based on exactly the same metamodel, and second, tools must support the same XMI format[2]. When two tools have to be integrated having two different metamodels, or two different versions of the same modeling language, or even more aggravated, the metamodels are defined in different meta-languages, XMI is no longer sufficient for model exchange. To cope with such integration scenarios, one has to bridge the tool metamodels, e.g., by using model transformation languages, in order to transform the models from one modeling tool to another.

1.1.3 Information Integration

When models are treated as data, and consequently metamodels as schemas, then the model exchange problem can be seen as a subproblem of data exchange which leads to a much broader research field, namely information integration, with database integration as its most prominent protagonist. Database integration has a long history, e.g., one of the earliest systems for realizing information integration was the EXPRESS system, developed in the 1970s by IBM [SHT+77]. Integration scenarios in the data engineering field mostly concern integrating different local schemas into one global schema, however also data exchange between a source schema and a target schema have been investigated, e.g., between relational databases and object-oriented databases, between XML schemas and relational databases, or between databases and data warehouses [DH05, Haa07].

[2] In [LLPM06] et al. it is reported that in practice, even between UML 2.0 tools, model exchange is only supported to a limited extend. The reasons for this are mainly that first, tool vendors use XMI dialects, and second, there are currently six different XMI versions proposed by the OMG.

In order to discuss communalities as well as differences between data exchange in the data engineering field and model exchange in the model engineering field, an alignment of the meta-modeling layers of both fields seems to be beneficial. Figure 2 shows this alignment by using the 4-layer meta-modeling architecture proposed by the OMG [OMG05e]. On the layer *M0* concrete objects reside which represent "real world" entities. Layer *M1* determines which objects shall be created for the domain of interest, described in a so-called *domain model*. On layer *M2*, all concepts needed for describing domain models are defined, so to say the language for describing domain models. In order to define the language specification on M2 requires another meta-layer, i.e. layer *M3*, that covers meta-concepts necessary for creating one's own language, so to say it represents the meta-language. Normally, as meta-language, a language is chosen which is able to describe itself, in order to ensure that the M3 layer is self-contained.

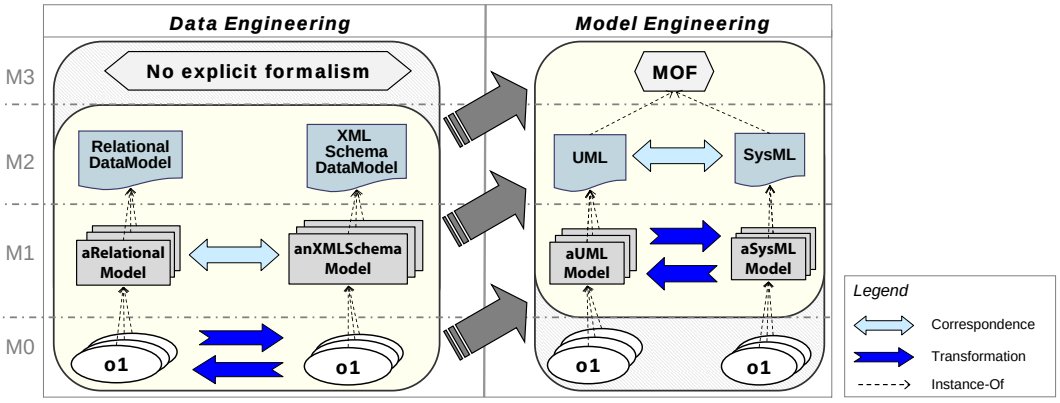

Figure 1.1: Meta-layers of Data Engineering and Model Engineering in Comparison

In the area of data engineering, integration scenarios concerning the layers M0 to M2 have been considered, but no scenarios concerning the M3 layer have been investigated, because in most cases no explicit meta-language has been applied to define the used data models[3]. Furthermore, no user-defined data models are employed as is the case in model driven engineering with user-defined modeling languages which gain more and more importance through the trend to domain-specific languages (DSL) [Fow05]. In model engineering, the integration scenarios are "lifted" one layer upwards, i.e., layer M1 to M3 are primarily concerned. Nevertheless, similar problems occur when two modeling languages are integrated on the M2 layer as when two schemas are integrated on the M1 layer. In particular, the notion of *heterogeneity* as main driver for integration problems has been established in the

[3]The main reason for this circumstance is that the relational data model and the XML schema data model represent reflexive languages, i.e., they can be described in their own terms.

data engineering field. Various heterogeneity problems and categorizations have been reported, cf., e.g., [Mot87, KS91, SPD92, KS96]. Especially in data exchange scenarios which are mostly related to model exchange scenarios, the following two kinds of heterogeneity problems have to be resolved.

Problem 1 – Heterogenous data models. Different data models (cf. meta-layer M2 of data engineering in Figure 1.1) have been proposed over the last decades in the field of data engineering. These data models offer a similar core of modeling concepts, but some details differ not least since some data models provide additional modeling concepts for enhancing expressiveness [HK87]. Consequently, schemas expressed in different data models are difficult to compare and the problem of migrating schemas to other data models arises. These problems have been tackled with a technique called *schema translation* [AT95, ACB05] which can be seen as a pre-processing step in order to reach one common integration data model, i.e., schema elements expressed in a source data model are transformed into semantic equivalent schema elements of another target data model. One important property of this kind of schema translation is that during the transformation, the represented "real world" semantics should not change or at least should be as stable as possible. A prominent example for schema translation is transforming relational schemas into XML schemas and vice versa.

Problem 2 –Structural heterogeneities between schemas. Although the same data model is used to define two schemas, various heterogeneities can occur between them, e.g., structural heterogeneities meaning that semantically equivalent schemas are modeled with different concepts. Mappings, i.e., semantic correspondences between elements of two different schemas, are the proposed mechanism for resolving those kind of heterogeneities [SPD92, BM07]. The semantics of mappings are described by the correspondences between the instances of the schemas. Thus, from mappings, transformations can be derived which are actually capable of translating instances conforming to a source schema into instances conforming to a target schema. Various classifications of mappings on the schema level have been proposed in the last three decades. One of the latest and most comprehensive classifications, which actually focus on the automatic generation of transformations out of mappings, is that of Legler and Naumann [LN07]. The authors distinguish between three main mapping situation classes: missing correspondences, single correspondences, and multiple correspondences, which are summarized in the following.

- *Missing correspondences*: This class comprises mapping situations which occur when elements of the source or target schema are not part of the mapping. Such situations are also often called *zero-to-one* or *one-to-zero* mappings. If elements of the source schema are not mapped, there is of course a loss of information during the transformation. Consequently, the source data cannot be reproduced from the target data. If

elements of the target schema are not mapped, constraints of the target schema may be violated, e.g., if an unmapped attribute of the target schema is mandatory.

- *Single correspondences*: For this class, the most important property of the correspondences is cardinality. According to the number of the participating elements in a correspondence, it can be distinguished between *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many* correspondences. On the instance level, one-to-one means that one element of the source model has exactly one corresponding element in the target model, one-to-many means that one element has to be splitted into several elements, many-to-one means that several elements have to be combined into one element, and many-to-many means a combination of one-to-many and many-to-one correspondences[4].

- *Multiple correspondences*: This class focuses on the structure of the source schema and the target schema. On the instance level this means, structurally related elements in the source model should also maintain their semantic relationship in the generated target model. Furthermore, this means, it should be possible to reproduce the original source model from the generated target model. For ensuring semantic preserving transformations, a set of single correspondences must be properly interpreted in combination with the structures of source schema and target schema.

This thesis mainly deals with the two above mentioned kinds of heterogeneities, in the area of model engineering. As a consequence, heterogeneity issues have to be considered on higher levels of abstraction leading to the notions of *meta-metamodel heterogeneity*[5] at M3-level and *structural metamodel heterogeneities* at M2-level.

1.1.4 Model-based Tool Integration in the Context of the ModelCVS Project

To tackle the previous mentioned problems of meta-metamodel heterogeneity and structural metamodel heterogeneity, we developed a system called ModelCVS aiming at providing a framework for model-based tool integration [KKK+06b, KKR+06]. ModelCVS enables transparent transformation of models between different tools' languages and exchange formats going beyond existing low-level model transformation approaches.

As can be seen in Figure 1.2, the proposed architecture of ModelCVS is organized into three major components. The red boxes represent tasks which are supported by tools and

[4] In practice, mostly all many-to-many correspondences can be reduced to a combination of one-to-many/many-to-one mappings. However, for specifying mappings, many-to-many mappings can at least provide a suitable mechanism to combine correspondences, thus reducing the size and enhancing the readability of a mapping model.

[5] Although MOF is the standardized meta-metamodel of the OMG, in practice other formats are also employed as meta-metamodels.

the green boxes represent the involved artifacts in the integration process. First, the *Technological Framework* provides the actual tool integration services by comprising tool adapters and a model transformation engine. Second, the *Metamodel Bridging Framework* provides support for defining bridges between metamodels in terms of bridiging operators which are subsumed by a dedicated metamodel bridging language. From these metamodel bridges the required model transformations executable by the model transformation engine can be automatically derived. Third, the *Ontology Framework* is an optional component and supports ontology-based metamodel bridging in terms of lifting the metamodels into ontologies and then using ontology matching techniques or mappings to domain ontologies to semi-automatically determine mappings between the generated ontologies which can be propagated back on the metamodel layer, i.e., as mappings between the metamodels. Thus, the Ontology Framework is used for automation purposes of the mapping task. The relevant parts of the ModelCVS project for this thesis are the tool adapter components and the Metamodel Bridging Framework, which are both explained in the following in more detail.

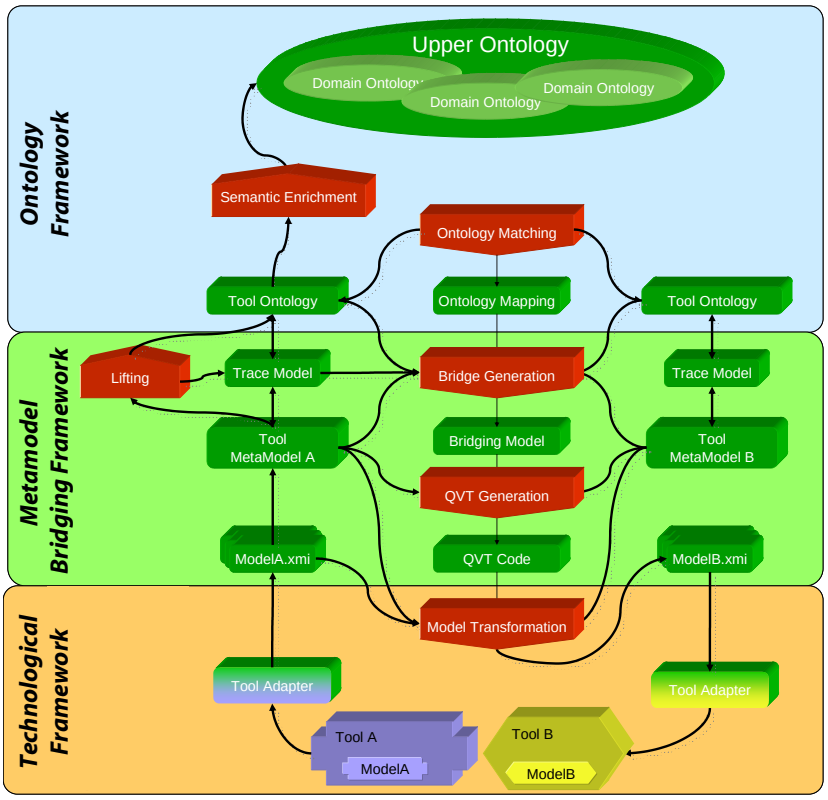

Figure 1.2: ModelCVS Technological Tree

1.1.4.1 Tool Adapter

The prerequisite for using the ModelCVS infrastructure is that the models are conform to the Eclipse Modeling Framework[6] (EMF) XMI and that an Ecore-based metamodel is available. We decided to use EMF as model repository infrastructure and Ecore as meta-modeling language, because EMF is currently the most standard conform implementation of the latest MOF version, thus the terms Ecore and MOF are often used as synonyms in this thesis. If the models are expressed in other formats, tool adapters are needed to connect ModelCVS with proprietary tools. Simple tool adapters are used to resolve incompatible XMI files, a issue which can often be solved with the construction of tool adapters based on XSLT, for instance, that finally render XMI files interchangeably. In this thesis, the focus lies not on incompatible XMI files, instead it is focused on problems coming from meta-metamodel heterogeneity.

When integrating two modeling tools, it is often the case that modeling languages are not defined with the same meta-language. Although, standardized meta-languages have been proposed in the model engineering world by the OMG, it is currently not the case that all modeling languages are defined with the same meta-language. A collection of currently used meta-languages for language engineering are shown in Figure 1.3.

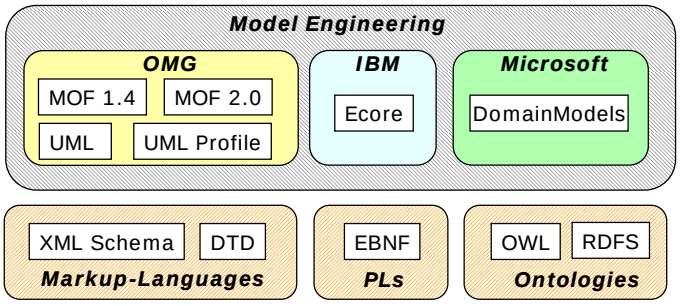

Figure 1.3: Meta-Languages Used for Language Engineering

For example, even the OMG has proposed two distinct approaches for developing modeling languages: first, the development from scratch by building a completely new metamodel, and second, the extension of standard UML by exploiting the profile mechanism [OMG05e]. For the first approach, dedicated meta-languages are necessary. The OMG has published the standardized meta-language MOF, however, various versions of MOF are available, each endowed with its own model serialization technique, i.e., XMI specification. Even within the OMG, an additional meta-language is proposed by the UML-related group called the UML Infrastructure [OMG03b], which represents the core of UML and is

[6] www.eclipse.org/emf

used for defining the UML 2 standard. Moreover, industry use their own implementation of MOF which are not completely conform to the MOF standard, such as in the Eclipse Modeling Framework[7] meta-language called Ecore that can be seen as a slightly modified MOF subset for the Java platform. Since UML 1.4, UML profiles can be created for extending standard UML with domain-specific modeling concepts. This is done by extending classes of the UML metamodel with stereotypes, which contain tagged values for expressing domain-specific values in UML models. One strong point of the profile mechanism is that standard UML tools can be reused, also for domain-specific modelling concerns. The problematic of meta-metamodel heterogeneity is, however, further aggravated, because in practice additional languages are used as meta-languages . In particular, two cases often occur in typical model engineering scenarios. First, pre-MDA modeling tools have not yet adopted OMG standards and therefore use meta-languages from other technical spaces, in particular XML-based languages. Second, mappings between modeling and programming languages defined in EBNF-like grammars or XML-based description languages defined in Document Type Definitions (DTDs) or XML Schemas, are needed for realizing the transformations between platform independent and platform dependent models as requested by the MDA initiative.

Requirements for Tool Adapters. First, tool adapters must be capable of converting models into the format supported by ModelCVS (EMF-based models) as well as generating Ecore-based metamodels from the proprietary modeling language descriptions. Second, tool adapters are nowadays mostly manually implemented for each different tool integration scenario, however, what is really needed are generic adapters which are capable of producing metamodels from any language description defined in a particular meta-language (M2-layer) and also the automatic generation of tool adapters which are capable of importing/exporting models on the M1-layer.

1.1.4.2 Metamodel Bridging Framework

For achieving interoperability between two modeling tools in terms of transparent model exchange, current best practices (cf. [Tra05]) comprise creating model transformations based on mappings between concepts of different tool metamodels. As mentioned before, the prevalent form of heterogeneity one has to cope with when creating such mappings between different metamodels is *structural heterogeneity*. Current model transformation languages, e.g., the OMG standard QVT [OMG05b], provide no appropriate abstraction mechanisms or libraries for resolving recurring kinds of structural heterogeneities. Thus, resolving structural heterogeneities requires to manually specify partly tricky model transformations again and again which simply will not scale up having also negative influence on understanding the transformation's execution and on debugging. Furthermore, having model transforma-

[7] www.elcipse.org/emf

tion code as the only integration description, it is hard to get an idea what really happens in the transformation, e.g., which information is lost in the transformation. Information loss is actually a problem when models are exported from tool A into tool B and then back from tool B into tool A, which is a quite common tool integration scenario.

Requirements for the Metamodel Bridging Framework. First, for tackling the mentioned problems, a framework should be developed for building reusable mapping operators which are used to define so-called metamodel bridges. Such metamodel bridges allow the automatic transformation of models. For this, a uniform formalism should be used not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations.

Second, the proposed framework should be applied for defining a set of mapping operators subsumed in a mapping language which is intended to resolve typical structural heterogeneities occurring between the core concepts usually used to define metamodels as provided by the OMG standard MOF [OMG04]. In case a problem is not directly solvable, new mapping operators can be defined by the user or the operational semantics of existing mapping operators can be tweaked.

Third, the defined metamodel bridges in terms of mapping operators, should allow the engineering of *roundtrip transformations*. This kind of transformations is especially needed for tool integration, namely if models are transformed from tool A to tool B and then back again to tool A. In such scenarios it is important that no information is lost during the transformation. However, practice shows that information loss often occurs in modeling tool integration, because of the high possibility of missing correspondences in mapping models. Therefore, mechanisms are needed that can support the user by developing roundtrip transformation in a systematic way where mapping models play a crucial role.

1.2 Contribution of This Thesis

In regard of these crucial problems and requirements for Tool Adapters and the Metamodel Bridging Framework, the contribution of this thesis is threefold.

Contribution 1. To ease the burden of developing tool adapters for each tool combination again and again, we propose a mining pattern for metamodels and models which can be used for implementing bridges on the M3-layer between two different meta-languages. With the term mining, we are not referring to stochastic methods such as used in the area of data mining [HK00], instead, we use the term mining as a name for the process of generating model-based representations out of text-based descriptions. In particular, we have implemented this mining pattern in the DTD2Ecore framework, a framework for producing Ecore-based metamodels out of Document Type Definitions (DTD). Therefore, we present a semi-automatic process how a language definition described in a language with limited expressiveness, e.g., DTD, is transformed into a language definition described in a language

with much more language features, e.g., Ecore. This (meta)model mining pattern opens the door to the model engineering technical space, which can be also applied in Architecture Driven Modernization (ADM) [OMG05a, Ulr05] scenarios where the aim is to extract models for different purposes, e.g., documentation or introspection, from legacy systems.

Contribution 2. In order to raise the abstraction level of metamodel bridges, we propose a framework for defining mapping operators and metamodel bridges in a declarative manner. Nevertheless, such metamodel bridges allow the automatic transformation of models since for each mapping operator the operational semantics is specified on basis of Colored Petri Nets [Jen92]. Colored Petri Nets provide a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations, thus facilitating understanding and debugging. To demonstrate the applicability of our approach, we apply the proposed framework for defining a set of mapping operators subsumed in our mapping language called CAR. This mapping language is intended to resolve typical structural heterogeneities occurring between the core concepts usually used to define metamodels, i.e., class, attribute, and reference, as provided by the OMG standard MOF.

Contribution 3. To facilitate the task of building roundtrip transformations, first we discuss how information loss, which can be seen as a cross-cutting concern, can be tackled with aspect-oriented concepts. Therefore, we propose an aspect-oriented extension for the CAR mapping language called AspectCAR. Second, we present an approach for a particular integration scenario which is quite often occurring in the field of model driven engineering, namely bridging domain-specific modeling tools with UML tools. This scenario is selected, because it has a high possibility for information loss due to the fact that DSLs have features which cannot be directly represented in a general purpose language such as UML. Nevertheless, these features can be indirectly expressed with the help of UML profiles representing the inherent language extension mechanism of UML [FFVM04]. In our proposed approach called ProfileGen, UML profiles are automatically derivable from CAR mapping models between the DSM metamodel and the UML metamodel. This approach can be also seen as a framework for developing tool adapters between domain-specific modeling tools and UML tools.

Big Picture. According to this contribution, this thesis is structured into three parts, considering mining of metamodels and models, mapping of metamodels to derive executable transformations, and engineering of roundtrip transformations, respectively. Figure 1.4 provides an overview of the considered integration scenarios, the thesis' contributions, and the relationship to the structure of this thesis.

Integration Example. In the following, we will discuss how the individual concepts, frameworks, and tools proposed in this thesis can be combined to solve a larger integration prob-

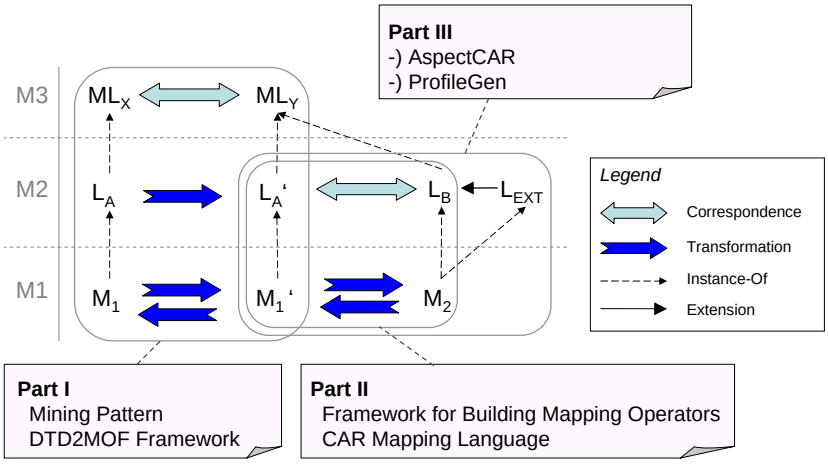

Figure 1.4: Contribution and Supported Integration Scenarios of this Thesis

lem in the area of integrating modeling tools for web applications. For this, we present an integration scenario from the MDWEnet project [VKC+07], the integration of *WebRatio*[8] which is a CASE tool for the web modeling language WebML [CFB+03] developed at the Politecnico de Milano with VisualWade[9] another prominent web modeling tool supporting the language OO-H [GCP01] as well as an UML tool, the Rational Software Modeler[10] from IBM.

The first problem which has to be solved is that WebRatio employs XML technologies such as Document Type Definitions (DTD) for defining its supported language WebML and XML documents for storing WebML models persistently. Therefore, the *DTD2Ecore Framework* is needed to generate, first a metamodel for WebML out of the DTD, and second, to generate a WebRatio tool adapter which is capable of transforming WebML documents into WebML models. In particular, the WebRatio tool adapter is able to bridge the XML technical space with the model technical space [KAB02]. As soon as the WebML metamodel is generated, the user can define a mapping model between the WebML and OO-H metamodel. This mapping model can be executed within the *Metamodel Bridging Framework* which is capable of generating an OO-H model out of the WebML model. Finally, a mapping model can be also defined between the WebML metamodel and the UML metamodel. This mapping model is used as input for the *ProfileGen* component to produce, first an UML profile for WebML, and second, a transformation expressed in the ATLAS Transformation Language (ATL) [JK06] which is capable of producing from the WebML model an UML model which

[8] www.webratio.com
[9] www.visualwade.com
[10] www-306.ibm.com/software/awdtools/modeler/swmodeler

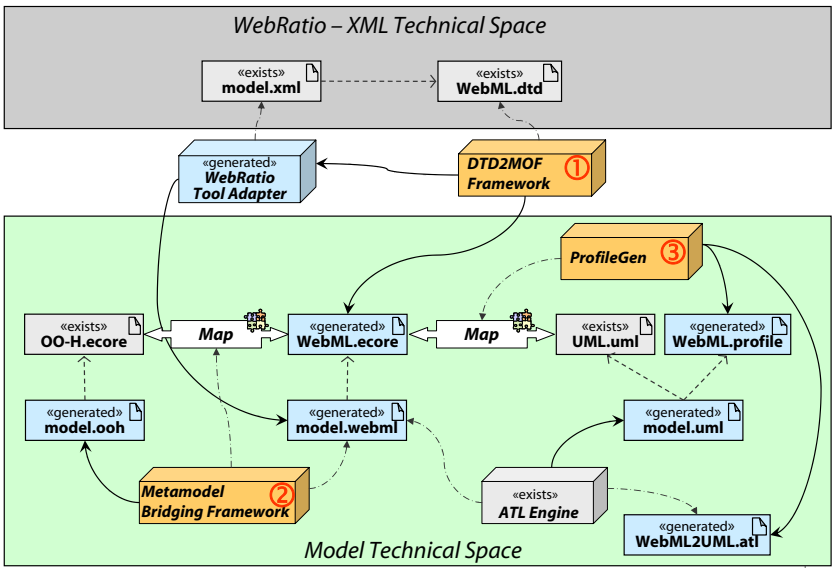

Figure 1.5: Combination of Proposed Frameworks – MDWEnet Example

also employs the WebML profile.

1.3 Thesis Outline

Part I. The subsequent Chapter 2 presents a framework which implements the (meta)model mining pattern for transforming DTDs into semantically enriched Ecore-based metamodels in a three-step process. In Chapter 3, the proposed framework is evaluated by employing it to produce a metamodel for WebML and discussing its resulting properties. Chapter 4 summarizes Part I and discusses related work.

Part II. Chapter 5 introduces the Metamodel Bridging Framework, in particular how the syntax and operational semantics of mapping operators can be defined as well as how the mapping operators can be applied to define bridges between metamodels. In Chapter 6, the proposed framework is employed for defining mapping operators for resolving structural heterogeneities between metamodels which are subsumed under the mapping language CAR. Chapter 7 summarizes Part II and related work concerning reusable model transformations is given.

Part III. Chapter 8 introduces the term roundtrip transformations and two integration scenarios exemplifying the need for roundtrip transformations in the context of tool integration. In Chapter 9, an aspect-oriented approach for engineering roundtrip transformations is presented, which allows to separate the definition of existing correspondences

and missing correspondences. Chapter 10 presents an alternative approach for engineering roundtrip transformations aiming at the integration of DSLs with UML. Chapter 11 summarizes Part III and discusses related work.

Finally, Chapter 12 concludes the thesis by means of a short summary and a discussion of future work is given.

Part I

Mining

Chapter 2

From DTDs to Ecore-based Metamodels

Contents

In this part of the thesis, the mining pattern is applied to define a bridge between the XML technical space and the model technical space. In particular, the mining pattern exploits the fact that most technical spaces use at least three meta-layers as shown in Figure 2.1. This figure also highlights the focussed integration scenario of this part of the thesis. On the M3 meta-layer, meta-languages reside, whereas each technical space employs its own meta-language. In order to bridge two technical spaces, the correspondences between concepts of the meta-languages have to be defined. These correspondences are the basis for establishing mining techniques. First, the correspondences can be used to generate from the source language description (cf. L_A) conforming to a source meta-language (cf. ML_X), a target language description (cf. $L_A{}'$) conforming to the target meta-language (cf. ML_Y). Second, in the generation process of the target language description, a transformation can be automatically produced which is capable of generating from models conforming to the source language (cf. M_1), models conforming to the target language (cf. $L_A{}'$).

In this chapter, the mining pattern is applied to define a bridge between the meta-languages XML Document Type Definitions (DTDs) and Ecore. This bridge is needed, because the WebML language has been partly specified in terms of DTDs and partly hard-coded within the tool accompanying the language. Consequently, in order to support model-driven

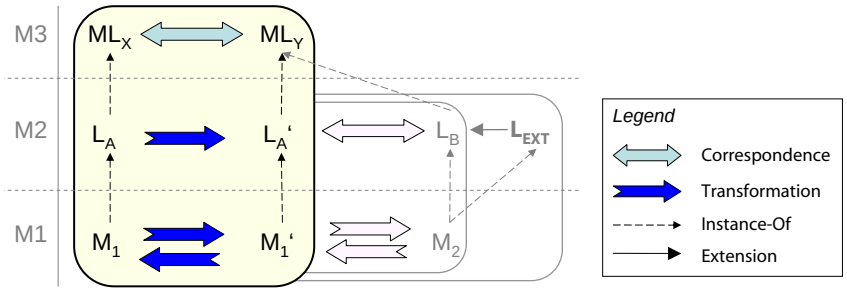

Figure 2.1: Integration Scenarios Revisited – Focus of Part I

development of web applications in the sense of Model-Driven Engineering (MDE), the WebML language needs to be specified in a MDE-suitable way, e.g., in terms of a metamodel. Considering the language's size, however, we refrain from manually re-modeling WebML from scratch, since this would be a cumbersome and error-prone process. Instead, the existing DTD-based language specification shall be reused in a semi-automatic process for metamodel generation from DTDs which implements the previously discussed mining pattern. In this respect, constraints hard-coded within the language's modeling tool WebRatio shall be extracted as well. After an introduction to this chapter's particular motivation in Section 2.1, Section 2.2 is dedicated to the explanation of the concepts of DTDs and metamodels as well as certain deficiencies of DTDs when used as a mechanism for defining modeling languages. Section 2.3 then describes the transformation process, including a set of transformation rules, and a set of heuristics giving indication for a manual refactoring, as well as a presentation of the implementation of the semi-automatic transformation approach in the form of the so-called *MetaModelGenerator* (MMG).

2.1 Motivation

Metamodels are a prerequisite for MDE in general and consequently for Model-Driven Web Engineering (MDWE) in particular. As already stated in previous chapters, various modeling languages, just as in the web engineering field, however, are not based on metamodels and standards, like OMG's prominent Meta Object Facility (MOF) [OMG04]. While web modeling approaches originally were based on proprietary languages and rather focused on notational aspects, today more and more approaches do provide language specifications based on standards though not always from the model technical space (cf. Chapter 1). Consequently, MDE techniques and tools cannot be deployed for such languages, which prevents exploiting the full potential of MDE in terms of standardized storage, exchange, and transformation of models.

Amongst the approaches not yet in line with MDE, WebML [CFB+03] is one of the most elaborated web modeling languages stemming from academia and is supported already over several years by the commercial tool WebRatio as already explained in Chapter 1. WebML's language concepts are partly defined in terms of XML document type definitions (DTDs) [W3C06], i.e., a grammar-like textual definition, specifying an XML document's structure, and partly hard-coded within the corresponding modeling tool. In contrast to MOF, DTDs represent a rather restricted mechanism for describing modeling languages, e.g., with respect to expressiveness, extensibility as well as readability and understandability for humans. Furthermore, since WebRatio internally represents models in XML [W3C06], it uses XSLT for generating code directly from WebML models. In contrast to dedicated MDE code generation technologies, e.g., MOFScript[1], writing XSLT programs for code generation, however, is difficult and error-prone. Concerning these problems, a metamodel-based approach allows expressing transformation rules in a more compact and readable way by using existing MDE-conform code generation techniques or model transformation languages such as QVT [OMG05b] and ATL [JK06] in order to produce platform-specific models in an additional step before generating code.

In order to define WebML's language concepts in an MDE-suitable way and thus to bridge WebML to MDE, a MOF-based metamodel for WebML is a fundamental prerequisite. Considering the language's size, however, we refrain from re-modeling WebML from scratch, since this would be a cumbersome and error-prone process. Instead, the existing DTD-based language specification as well as constraints hard-coded within WebRatio shall be reused in a semi-automatic process for MOF-based metamodel generation from DTDs. This process is illustrated in Figure 2.2 and encompasses three phases, whereby the first two phases concern the semi-automatic generation of the basic WebML language concepts defined within the WebML DTD. During the first phase a preliminary version of the metamodel is automatically generated from the available DTD, while in the second phase this preliminary version is manually validated and refactored according to constraints captured within the WebRatio tool support.

The following section explains the concepts of DTDs and MOF and points out the aforementioned DTD deficiencies.

2.2 DTDs and Ecore at a Glance

As a first step towards bridging WebML to MDE, this section elaborates on the expressiveness of DTDs, i.e., the concepts used to describe the WebML language, with respect to MOF. In the context of OMG's meta-level architecture [OMG05e], this means that a WebML model, which is represented by an XML document, relates to the model level (M1) (cf. Figure 2.3).

[1] www.eclipse.org/gmt/mofscript

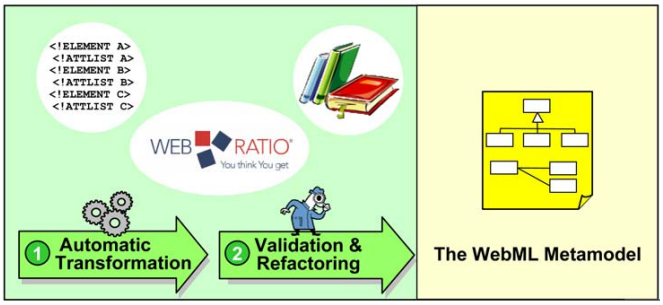

Figure 2.2: Process of Designing the WebML Metamodel

Such a model has to conform to the WebML DTD describing the WebML language concepts at the metamodel-level (M2). The WebML DTD in turn is based on the DTD-grammar [W3C06] defined at the meta-metamodel-level (M3). Note that, while in case of WebML, a DTD is used to define a modeling language and therefore can be assigned to the M2 level, DTDs typically are used at M1 in order to describe the structure of data stored in XML documents. Analogously, MOF concepts defined at M3 are used to describe metamodels in the sense of MDE at M2. In the present case, this is the targeted WebML metamodel of which instances in terms of WebML models can be formulated at M1. This discussion shows that the two M3 level formalisms, in terms of the DTD-grammar and MOF respectively, represent the concepts on which to identify correspondences. In turn, these correspondences serve as a basis for the M2-level done by the framework presented in Section 2.3.

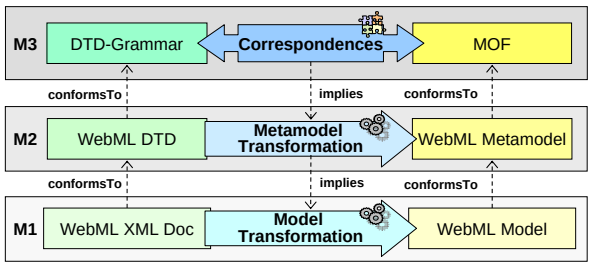

Figure 2.3: Interrelationships Between the Language Layers of DTD and MOF

In the following, UML class diagrams [OMG05e] are used as a common formalism to explain and to illustrate the major concepts of the DTD-grammar (cf. Subsection 2.2.1) and MOF (cf. Subsection 2.2.2). This explanation serves as the basis for identifying differences in the expressiveness of the two meta-meta-languages (cf. Subsection 2.2.3).

2.2.1 Document Type Definition (DTD) Concepts

The UML class diagram given in Figure 2.4 is based on previous work [KKR04] and depicts the DTD concepts present in the WebML DTD. These concepts need to be considered for finding correspondences to MOF concepts and consequently are reviewed briefly in the following.

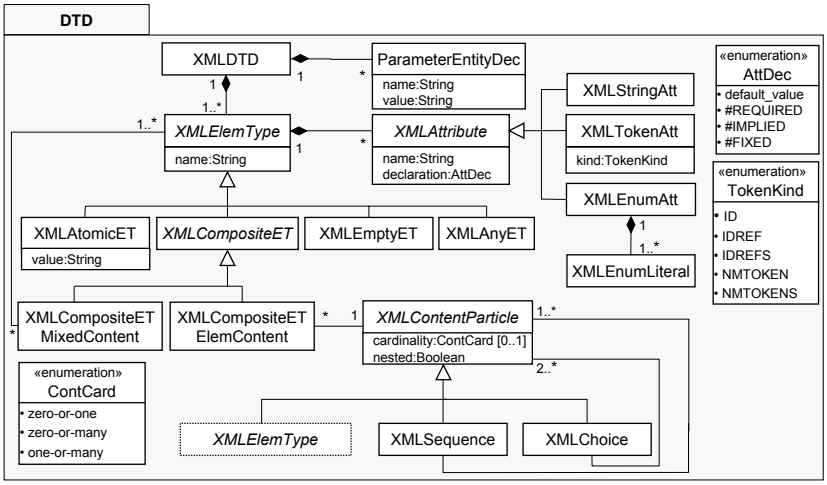

Figure 2.4: Overview of Relevant DTD Language Concepts

In general, DTD's contain markup declarations comprising element types (`XMLElemType`) and attributes (`XMLAttributes`) for defining the logical structure as well as primarily entity declarations (e.g., `ParameterEntityDec`), as a reuse mechanism for certain reoccurring markup declarations, defining the physical structure.

Element types, being first-class citizens in DTDs, have a name and are specialized into `XMLAtomicET` (contains no other element types but character data), `XMLEmptyET` (no content is allowed), `XMLAnyET` (the content is not constrained), `XMLCompositeETMixedContent` (a mix of character data and child element types), and `XMLCompositeETElemContent` (consists of an `XMLContentParticle`). An `XMLContentParticle` either is an `XMLSequence`, an `XMLChoice`, or an `XMLElemType`. An `XMLChoice` or an `XMLSequence` can be enclosed in parentheses for grouping purposes and suffixed with a '?' (zero or one occurrences), '*' (zero or more occurrences), or '+' (one or more occurrences), whereas the absence of a particular symbol denotes a cardinality of exactly one.

Attribute declarations define one or multiple `XMLAttributes` (i.e., name-value pairs) for a single element type. Each `XMLAttribute` has a name, a data type, and a default declaration. The most commonly used data types for attributes are: `CDATA` (`XMLStringAtt`), `ID`, `IDREF` (refers to one `ID`-typed element), `IDREFS` (refers to multiple `ID`-typed elements),

and Enumeration (`XMLEnumAtt`). For default declarations there are four possibilities: `#IMPLIED` (zero or one), `#REQUIRED` (exactly one), `#FIXED` (the attribute value is constant and immutable), and `Literal` (the default value is a quoted string).

Please note that, the order constraints imposed by DTDs and the majority of physical structures of the DTD-grammar (i.e., general entity declarations, notation declarations as well as `XMLAttributes` of type `ENTITY`, `ENTITIES`, and `NOTATION`) are ignored, since they are actually more relevant to XML documents than to DTDs and the purpose of finding correspondences to MOF concepts is rather questionable [LM05].

2.2.2 MOF Concepts in Terms of Ecore

In the following, a brief overview of the most important concepts of MOF with respect to finding correspondences to DTD concepts is given. Note that, by the time of writing there is no standardized implementation of MOF 2.0 available. Therefore, in this thesis, Ecore - a slightly modified Essential MOF (EMOF) implementation in Java - which is provided by the widespread Eclipse Modeling Framework (EMF) [BSM+04], is used. In Figure 2.5, the most important concepts of Ecore with respect to finding correspondences to DTD concepts are summarized.

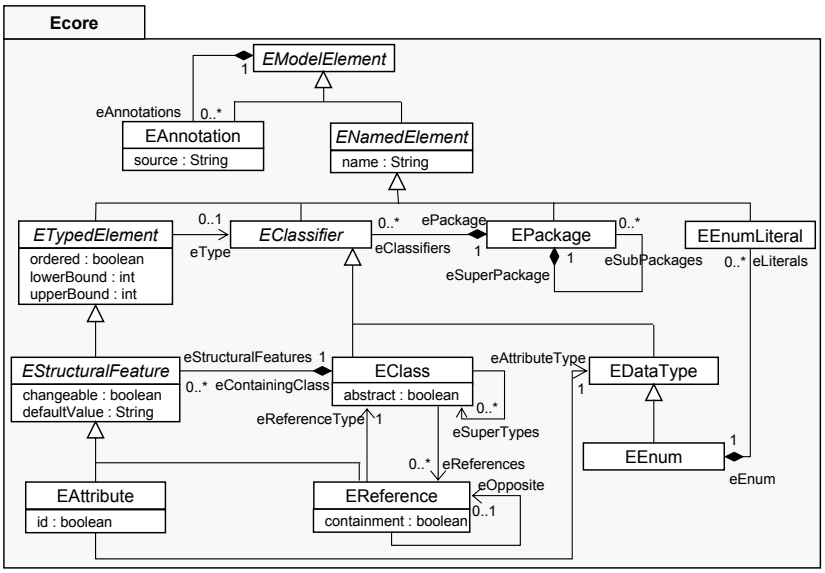

Figure 2.5: Overview of Relevant Ecore Language Concepts

In Ecore there is a single root concept (`EModelElement`) being the base class for all modeling elements. Its sub-class `EAnnotation` is used for describing additional information

which cannot be presented directly in Ecore-based metamodels. `ENamedElement` is the base for the remaining Ecore modeling elements, because it provides for a 'name' meta-attribute. An `EClassifier` represents a type in a model and as such, is the base class for `EClass` and `EDataType`, whereas an `ETypedElement` serves as the base for other modeling concepts having a type such as `EStructuralFeature`, which in turn represents a structural feature of an `EClass`. `EClasses` are the first-class citizens in Ecore-based metamodels. An `EClass` may have multiple `EReferences` and `EAttributes` for defining its structural features as well as multiple super-`EClasses`. An `EAttribute` is part of a specific `EClass` and can have, as any `ETypedElement`, a lower and an upper bound multiplicity. Additionally, it can be specified as being able to uniquely identify the instance of its containing `EClass` (cf. 'id' meta-attribute) and as being ordered. The type of an `EAttribute` is either a simple `EDataType` or an enumeration. `EString`, `EBoolean`, `EInt`, and `EFloat` are part of Ecore's default data type set. `EEnum` allows to model enumerations defined by an explicit list of possible values, i.e., its literals (cf. `EEnumLiterals`). Analogous to `EAttribute`, an `EReference` is part of a specific `EClass` and can have a lower and an upper bound multiplicity. An `EReference` refers to an `EClass` and optionally to an opposite `EReference` for expressing bi-directional associations. Besides, an `EReference` can be declared as a being ordered and as a containment reference. `EPackages` group related `EClasses`, `EEnums`, as well as related `EPackages`. Each element is directly owned by an `EPackage` and each `EPackage` can contain multiple model elements.

2.2.3 DTD Deficiencies

When comparing a language specified in Ecore to one specified on the basis of DTDs, it is obvious that DTDs considerably lack extensibility, readability, and understandability for humans, and above all expressiveness [LM05]. In the following, the major deficiencies of DTDs when used as a mechanism for defining modeling languages are described. Note that some of these deficiencies have been resolved with the introduction of XMLSchema [W3C04], such as limited set of data types, awkward cardinalities, missing inheritance concept, and lack of an explicit grouping mechanism. A profound comparison between DTD and XMLSchema can be found in Lee et al. [LC00]. Nevertheless, in the context of WebML, which is based on DTDs, the following shortcomings need to be addressed.

Limited Set of Data Types

In contrast to Ecore, DTDs have a limited set of data types that cannot be extended to support, e.g., Integer or Float data types. While the provided data types generally are based on Strings, some other data type may be simulated by defining an enumeration with specific literals. In this way, a Boolean attribute can be simulated by an attribute of type Enumeration having two literals, e.g., 'true' and 'false'. Enumerations, however, cannot capture

numeric data types such as Integer or Float, which are naturally infinite.

Unknown Referenced Element Type(s)

DTDs referencing mechanism is based on IDREF(S)-typed attributes, which are able to reference any element type having an ID-typed attribute. Unlike Ecore, which provides typed references, it is not possible to identify the element type that may be referenced from an IDREF(S)-typed attribute based on the information given in DTDs. DTDs even allow to reference different element types. These referenced element types potentially have a common super-type, which, however, cannot be specified in the DTD. Due to this peculiarity of DTDs, it is neither possible to determine which element type(s) may be referenced based on the information given in the DTD nor if a certain set of element types may be referenced, only.

No Bi-directional Associations

While Ecore offers bi-directional associations, in DTDs only uni-directional references can be specified. There is no way to specify that two uni-directional references in combination form a bi-directional association either.

Awkward Cardinalities

DTDs offer a restricted mechanism to specify cardinalities. More specifically, in contrast to Ecore there are no explicit concepts for defining cardinalities having a lower bound greater than '1' and for defining cardinalities having an upper bound other than '1' or '*'. This can only be simulated in an awkward way by redundantly specifying a certain element type within the content specification of its related (parent) element type according to the required cardinality.

Missing Role Concept

In DTDs, there is no explicit concept to express that an element type can be deployed in different contexts, i.e., a role concept such as in Ecore is missing. Thus, in DTDs this is sometimes bypassed by defining each role as a separate element type each named after the specific role they represent, and redundantly defining the same content and attribute specifications.

Missing Inheritance Concept

DTDs are not able to express inheritance relationships between element types as provided for Ecore. Hence, DTDs cannot profit from the typical benefits of inheritance such as reuse.

No Explicit Grouping Mechanism

There is no explicit mechanism to group parts of a DTD that semantically belong together as it is supported in Ecore. Nevertheless, this deficiency can be bypassed by encapsulating parts of a DTD in separate files and employing parameter entities to import these separated definitions where appropriate.

Missing Constraint Mechanism

A mechanism for defining complex constraints, as it is supported in Ecore by using the OCL standard [OMG05d], is not provided for DTDs. Thus, even simple XOR constraints, which are often required in metamodels, cannot be specified. This deficiency is specifically problematic, since possible ambiguities in DTDs cannot be resolved and XML documents, while valid according to their DTD, might still not represent the domain data correctly.

2.3 A DTD to Ecore Transformation Framework

On the basis of the discussion of DTD and Ecore concepts as done in the previous section, it is now possible to give more insight into the semi-automatic transformation approach, which is based on previous work [SWK06]. Generally, the transformation approach consists of an automatic phase and a manual phase (cf. Figure 2.6). The first phase is responsible for automatically generating a first version of the WebML metamodel and is performed by a component called *MetaModelGenerator* (MMG). The metamodel generator employs, in a first step, a set of transformation rules expressing all identified non-ambiguous correspondences between DTD concepts and Ecore concepts (cf. Subsection 2.3.1). In a second step of that phase, a set of heuristics is applied, dealing mainly with the aforementioned deficiencies by proposing possible correspondences (cf. Subsection 2.3.2). On the basis of these suggestions, in the second phase, the user needs to manually validate the generated metamodel and refactor it accordingly (cf. Subsection 2.3.3). The implementation architecture of the transformation framework is presented in Subsection 2.3.4.

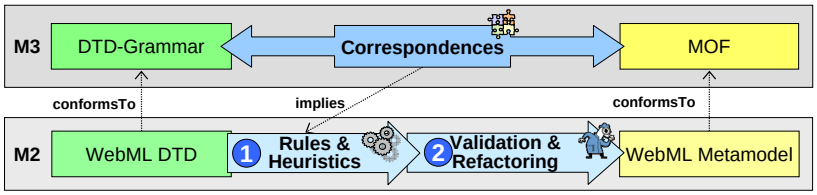

Figure 2.6: Two Phase Semi-Automatic Transformation Approach

To illustrate the transformation approach, a small sub-set of the WebML DTD is used to

show the effects of applying transformation rules, heuristics and refactoring steps in terms of the resulting WebML metamodel (M2). In particular, this small sub-set consists of part of the concepts provided by WebML to represent a web application's content layer which in fact resembles the well-known ER-model [Che76]. It has to be emphasized that the focus in this section is on the illustration of the transformation approach, i.e., the consecutive application of (some) transformation rules, heuristics, and refactoring steps. As a consequence, using an example requiring concepts for modeling a web application's hypertext has been avoided, since the concepts are too numerous and often relate to concepts defined for the content layer. In this respect, the WebML content layer serves as a self-contained and small example. For those rules and heuristics that cannot be illustrated in the context of WebML's content model, an abstract example is provided in this section as well as a reference to an illustration of their concrete application in the context of WebML's hypertext model in Section 3.1.

2.3.1 Transformation Rules

A couple of rules for transforming concepts of DTDs into Ecore concepts have been designed. Table 2.1 summarizes these rules by differentiating between rules for `XMLElement-Types`, `XMLAttributes` and XOR-Constraints, denoting DTD concepts on the left-hand side and their Ecore counterparts on the right-hand side. Rules are marked using a decimal numbering schema and may contain sub-rules, further specializing the correspondences between DTD concepts and Ecore concepts. Finally, alternative correspondences depending on the concrete DTD concept are depicted by a distinction of cases. Following, the application of some of these rules are illustrated using the running example introduced above.

Rule 1 - Element Type

For each `XMLElemType`, an `EClass` is created and the name of the `EClass` is set to the element type's name. Depending on the particular sub-class of `XMLElemType`, additional metamodel elements have to be created in the transformation process, which is outlined in Table 2.1.

Example. In Figure 2.7(a), the WebML DTD specifies amongst others element types for *ENTITY* and *RELATIONSHIP*, since WebML's content model is based on the ER-model. According to Rule 1, two `EClasses` are generated and named after the element types (cf. Figure 2.7(b)). In addition, the ENTITY `XMLCompositeETElemContent` contains the *RELATIONSHIP* `XMLEmptyET`, and with respect to case (4) in Table 2.1 an `EReference` is produced, specifying *RELATIONSHIP* as the contained `EClass`.

Table 2.1: Transformation Rules from DTD to Ecore

	Rule	DTD Concept		Ecore Concept
XML Element Type	**R 1**	*XMLElemType (ET)*		*EClass*
		XMLElemType. Name		*EClass.name*
	(1)	XMLEmptyET		no additional elements required
	(2)	XMLAnyET		no additional elements required
	(3)	XMLAtomicET		add EAttribute EAttribute.name="PCDATA", EAttribute.eAttributeType=EString, EAttribute.defaultValue=XMLAtomicET.value
	(4)	XMLCompositeET ElemContent	If XMLSequence with cardinality=1 and nested=false	add EReference for each XMLElementType in XMLSequence EReference.name=XMLElementType.name, EReference.containment=true
			If XMLChoice with cardinality=1 and nested=false	add EReference for each XMLElementType in XMLChoice EReference.name=XMLElementType.name, EReference.containment=true add OCL constraints restricting the alternative EReferences
			If XMLContentParticle with cardinality>1 or nested=true	add helper EClasses for each XMLSequence or XMLChoice serving as containers for nested XMLContentParticles
	(5)	XMLCompositeETMixedContent		add EReference for each XMLElementType EReference.name=XMLElementType.name, EReference.containment=true add EAttribute EAttribute.name="PCDATA", EAttribute.eAttributeType=EString, EAttribute.defaultValue= XMLCompositeETMixedContent.value
	R1.1	*XMLContentParticle.cardinality*		*EReference.multiplicity*
	(1)	? (Zero-or-one)		EReference.lowerBound=0, EReference.upperBound=1
	(2)	* (Zero-or-more)		EReference.lowerBound=0, EReference.upperBound=-1
	(3)	+ (One-or-more)		EReference.lowerBound=1, EReference.upperBound=-1
	(4)	no symbol		EReference.lowerBound=1, EReference.upperBound=1
XML Attribute	**R2**	*XMLAttribute*		*EAttribute*
		XMLAttribute.name		*EAttribute.name*
	(1)	XMLStringAtt, NMTOKEN(S), IDREF(S)		EAttribute.eAttributeType=EString
	(2)	ID		EAttribute.eAttributeType=EString, EAttribute.id=true
	(3)	XMLEnumAtt		add EEnum EEnum.name= XMLEnumAtt.name+"_ENUM" for each XMLEnumLiteral add EEnumLiteral EAttribute.eAttributeType=EEnum
	R2.1	*XMLAttribute.cardinality*		*EAttribute.multiplicity*
	(1)	default_value	Single-valued	EAttribute.lowerBound=1, EAttribute.upperBound=1, EAttribute.defaultValue=XMLAttribute.default_value
			Multi-valued	EAttribute.lowerBound=1, EAttribute.upperBound=-1, EAttribute.defaultValue=XMLAttribute.default_value
	(2)	#FIXED	Single-valued	EAttribute.lowerBound=1, EAttribute.upperBound=1, EAttribute.defaultValue=default_value, EAttribute.unchangeable=true
			Multi-valued	EAttribute.lowerBound=1, EAttribute.upperBound=-1, EAttribute.defaultValue=default_value, EAttribute.unchangeable=true
	(3)	#REQUIRED	Single-valued	EAttribute.lowerBound=1, EAttribute.upperBound=1
			Multi-valued	EAttribute.lowerBound=1, EAttribute.upperBound=-1
	(4)	#IMPLIED	Single-valued	EAttribute.lowerBound=0, EAttribute.upperBound=1
			Multi-valued	EAttribute.lowerBound=0, EAttribute.upperBound=1
XOR	R3	If XMLElemType is part of several XMLCompositeETElemContent		then add OCL constraint to contained EClass specifying that the produced EReferences are exclusive

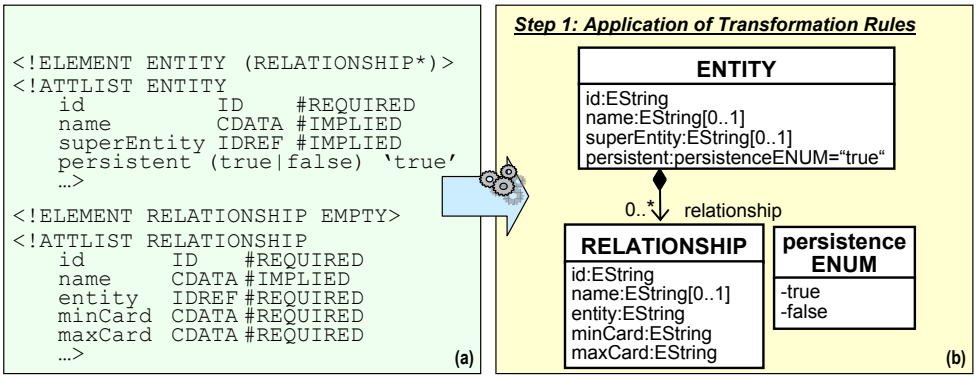

Figure 2.7: Example of Applying the Transformation Rules (Step 1)

Rule 1.1 - Content Particle Cardinality

Each `XMLContentParticle` may have a certain cardinality, which is represented in metamodels through the `EReference`'s multiplicity in terms of lower and upper bound.

Example. Considering the running example in Figure 2.7(b), according to this rule the cardinality of the relationship from *ENTITY* to *RELATIONSHIP* is set to '0..*'.

Rule 2 - Attribute

For each `XMLAttribute` an `EAttribute` is created and attached to the `EClass` representing the `XMLElemType`, which in turn owns the `XMLAttribute`. The name of the `EAttribute` is set to the name of the `XMLAttribute`. The data type of `XMLAttribute` is one of the following: `CDATA`, `NMTOKEN`, `NMTOKENS`, `ID`, `IDREF`, `IDREFS`, and `Enumeration`. Each of these possibilities requires an appropriate transformation as is outlined in Table 2.1.

Example. The example in Figure 2.7(b), shows that all `XMLAttributes` of type `ID`, `CDATA`, and `IDREF` have been transformed into `EAttributes` of type `EString`, while the `XMLEnumAtt` *persistent* has been transformed to an `EEnum` having two `EEnumLiterals`.

Rule 2.1 - Attribute Cardinality

Attributes in both, DTDs and Ecore have a certain kind of cardinality. In DTDs, the cardinality of an `XMLAttribute` is determined on the one hand by the differentiation between single-valued (e.g., `ID`, `CDATA`, `IDREF`, `NMTOKEN`, and `XMLEnumAtt`) and multi-valued (e.g., `IDREFS`, `NMTOKENS`) attributes and on the other hand by the `XMLAttribute` declaration (`#REQUIRED`, `#IMPLIED`, `#FIXED`, and `Literal`). Table 2.1 illustrates how `XMLAttribute` cardinalities are transformed into `EAttribute` multiplicities.

Example. In Figure 2.7(b), all XMLAttributes are single-valued meaning that the upper bound is set to one. Only the EAttributes *name* and *superEntity* of EClass *ENTITY* as well as *name* of EClass *RELATIONSHIP* have a multiplicity of '0..1' since their corresponding XMLAttributes have been defined #IMPLIED. The default value of the EAttribute *persistent* is set to one of the EEnumLiterals, i.e., 'true'.

Rule 3 - XOR Containment References

An XMLElemType can be part of an XMLContentParticle of different instances of XML-CompositeETElemContent. In the corresponding Ecore-based metamodel an EClass can participate as the contained element in an arbitrary number of containment references. At instance level, the contained object, however, can be contained by an instance of only one of the container EClasses at the same time. Hence, this rule adds an OCL constraint to the contained EClass specifying this restriction.

Example. In the abstract example in Figure 2.8(a), the XMLElemType *C* is an XML-ContentParticle of XMLElemType *A* and XMLElemType *B*. The corresponding metamodel in Figure 2.8(b) must ensure that an instance of EClass *C* is contained either by an instance of EClass *A* or by an instance of EClass *B*. Therefore, an XOR constraint is introduced between the relationship *c* from *A* to *C* and the relationship *c* from *B* to *C*. For an example application of Rule 3 in the context of WebML the reader is referred to Listing 3.3 in Subsection 3.1.3.

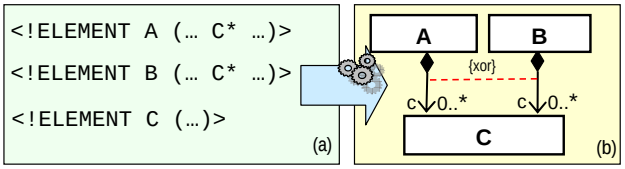

Figure 2.8: Rule 3 - XOR Containment References

2.3.2 Heuristics

As mentioned before, transformation rules are not enough to obtain an Ecore-based metamodel from a specific DTD, due to the deficiencies of DTDs described in Subsection 2.2.3. Thus, for resolving most of these deficiencies, a set of six heuristics (cf. Table 2.2) is proposed, exploiting the assumption that design patterns and naming conventions have been used by DTD designers that have also been found when analyzing the WebML DTDs. This means that the effectiveness of the heuristics, however, is strongly correlated with the quality of the design of the DTDs. For example, the heuristics operate more effectively if naming conventions are used, e.g., for IDREF(S)-typed XMLAttributes, (cf. Heuristic 1 -

IDREF(S) Resolution) or a common DTD design pattern [VIG05] for grouping related element types by splitting up a DTD into several external DTDs is employed (cf. Heuristic 3 - Grouping Mechanism).

In any case, these heuristics propose possible correspondences along with annotations guiding the validation and refactoring step in phase 2. In this way, semantically rich language concepts of Ecore such as typed references, data types, and packages can be used to equalize the DTD deficiencies. Following, the application of some of the heuristics is shown in using the running example in Figure 2.9.

Table 2.2: Heuristics from DTD to Ecore

Heuristic	DTD Concept	Ecore Concept	DTD Deficiency Resolved
H1	If (XMLTokenAtt.kind=IDREF) AND (XMLElemType.name=XMLAttribute.name)	then add EReference to EClass with name= XMLElemType.name, EReference.name=XMLAttribute.name annotate with «Validate IDREF(S)»	Unknown Referenced Element Type(s)
	else	then annotate EAttribute with «Resolve IDREF(S) manually»	
H2	If XMLEnumAtt has two XMLEnumLiterals and XMLEnumAtt is one of {true, false}, {1, 0}, {on, off}, {yes, no}	then EAttribute.eAttributeType=EBoolean annotate with «Validate Boolean»	Limited Set Of Data Types
	else if XMLEnumAtt has two XMLEnumLiterals	then annotate EEnum with two EEnumLiterals with «Resolve possible Boolean type manually»	
H3	If DTD imports external DTDs	then add EPackages for each external DTDs to the root EPackage	No Explicit Grouping Mechanism)
H4	If the name of two or more XMLElemTypes in a XMLSequence are equal	then annotate container EClass with «Resolve multiplicity manually»	Awkward Cardinalities
H5	If XMLElemType has two or more XMLTokenAtts with declaration=#IMPLIED and (kind=IDREF or kind=IDREFS)	then annotate each EAttribute or EReference (cf. Heuristic 1) with «Resolve XOR constraint manually»	Missing Constraint Mechanism
H6	If XMLElemType is of type XMLAnyET	then annotate EClass with «Resolve XMLAnyET manually»	Missing Inheritance Concept

Heuristic 1 - IDREF(S) Resolution

The first heuristic is based on the assumption that an `IDREF(S)`-typed `XMLAttribute` might be named after the `XMLElemType` it is intended to reference. Thus, although DTDs lack the possibility to explicitly specify the referenced element types (cf. Subsection 2.2.3 Unknown referenced element type(s)), it is possible to find them relying on naming conventions of element types and attributes. Heuristic 1 is intended to find such name matches in DTDs. If a match is found, an `EReference` is generated pointing to the identified `EClass`. In addition, the `EReference` is annotated with ≪*Validate IDREF(S)*≫. Furthermore, the

multiplicity of the EReference is set to the multiplicity of the XMLAttribute.

It has to be emphasized that, since this heuristic is based on name matches, two problematic cases can occur. On the one hand, it may falsely resolve references in case IDREF(S) attributes are incidentally named like XMLElemTypes but in fact do not reference them. On the other hand, it may not be possible to resolve a reference in case IDREF(S) attributes are not named according to the XMLElemType they shall refer to. Consequently, the user has to validate if the resolution of the IDREF(S) is correct or if another EClass should be referenced.

Example. In the running example, the XMLAttribute *entity* in Figure 2.9(a) is resolved to an EReference in Figure 2.9(b). In case no name match is found, the IDREF(S)-typed XMLAttribute is transformed into an EAttribute of type EString annotated with ≪*Resolve IDREF(S) manually*≫, such as the *superEntity* EAttribute in Figure 2.9(b).

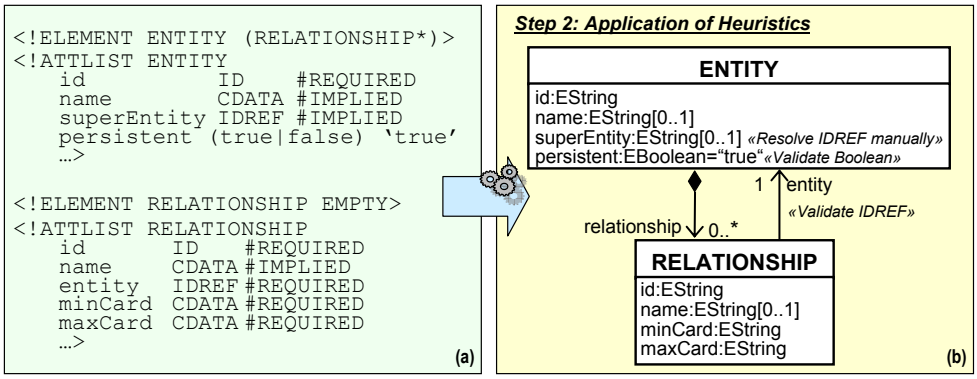

Figure 2.9: Example of Applying the Heuristics (Step 2)

Heuristic 2 - Boolean Identification

Heuristic 2 is based on the assumption that an XMLEnumAtt consisting of two XMLEnum-Literals might represent an attribute of type Boolean (cf. 2.2.3 Limited set of data types). It recognizes such optimization possibilities and, instead of generating an EEnum, produces an EAttribute of type EBoolean for the following sets of enumeration literals: {true, false}, {1, 0}, {on, off}, and {yes, no}. Furthermore, an annotation ≪*Validate EBoolean*≫ is added to the attribute. In case the two XMLEnumLiterals are not one of the aforementioned sets, the produced EEnum is annotated with ≪*Resolve possible EBoolean type manually*≫, indicating the possibility of replacing the EEnum by the EBoolean data type.

Example. In the running example, the XMLEnumAtt *persistent* is identified to be of type Boolean (cf. Figure 2.9(a)). Thus, in the metamodel, the EAttribute *persistent* is of type EBoolean and no EEnum is generated (cf. Figure 2.9(b)).

Heuristic 3 - Grouping Mechanism

In Heuristic 3, a parameter entity declaration that points to a further DTD file is interpreted as representing a group of related markup declarations (cf. Subsection 2.2.3 No explicit grouping mechanism), that can be referenced from within a so called root DTD. A root DTD is equivalent to a root package in a metamodel and external DTDs are equivalent to sub-packages of the root package. Thus, a package for each external DTD and one root package for the root DTD needs to be generated.

Example. In the abstract example of Figure 2.10(a) a DTD named *Root* is shown which defines two `ParameterEntityDec` *PartA* and *PartB*, both referencing an external DTD, *A.dtd* and *B.dtd*. The DTD *Root* is transformed into the `EPackage` *Root* which contains an `EPackage` *A* and `EPackage` *B* for the external DTDs (cf. Figure 2.10(b)). An example application of Heuristic 3 in the context of WebML may be found in Listing 3.1 in Subsection 3.1.1.

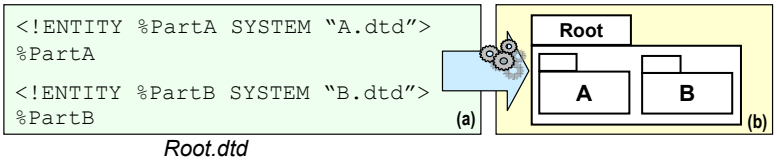

Figure 2.10: Heuristic 3 - Grouping Mechanism

Heuristic 4 - Cardinalities Identification

This heuristic is based on the assumption that an `XMLSequence` containing element types of the same name has to be interpreted as "one piece of information" (cf. 2.2.3 Awkward cardinalities). This means that instead of producing an `EReference` for each single element type, only one `EReference` should be generated and the cardinality has to be inferred from all element type cardinalities within the sequence. Heuristic 4 adds an annotation ≪*Resolve cardinality manually*≫ to the `EClass` containing the `EReferences` to indicate that a specific sequence of element types transformed into a set of `EReferences` possibly has to be remodeled into one `EReference` and the appropriate multiplicity has to be assigned.

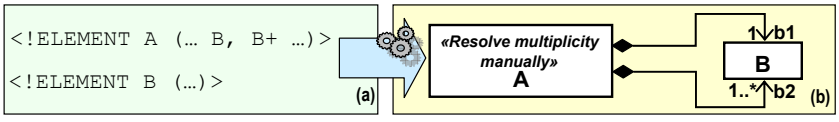

Figure 2.11: Heuristic 4 - Cardinalities Identification

Example. In Figure 2.11(a), an example for the awkward cardinality deficiency of DTDs can be found. The problem is that the first and the second `XMLContentParticle` *B* of `XMLElemType` *A* together may represent one set of elements with a cardinality restriction of '2..*' instead of representing two separate sets of elements. Consequently, in the corresponding metamodel (cf. Figure 2.11(b)), this ambiguity is marked by the annotation ≪*Resolve cardinality manually*≫. This annotation indicates that in the validation and refactoring step the user has to decide, if *b1* and *b2* are two separate sets or if *b1* and *b2* should be merged into one set with a cardinality of '2..*'. This heuristic has been applied for example in the context of WebML's hypertext model (cf. Listing 3.2 in Subsection 3.1.3).

Heuristic 5 - XOR Constraints Identification

If two `XMLAttributes` within an attribute list declaration of an element type are both of type `IDREF(S)` and have been declared as `#IMPLIED` they might represent two excluding `EReferences` in the metamodel and hence require an XOR constraint. Heuristic 5 annotates these `EReferences` (or `EAttributes` if the reference could not be resolved automatically by Heuristic 1) with ≪*Resolve XOR constraint manually*≫ to indicate the possible need for an XOR constraint.

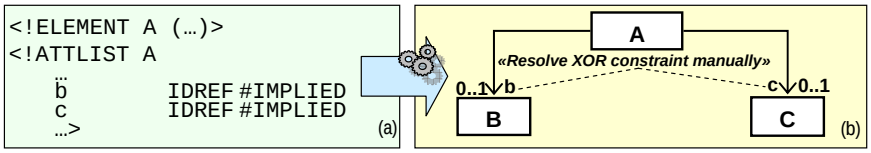

Figure 2.12: Heuristic 5 - XOR Constraints Identification

Example. In Figure 2.12(a), an excerpt of a DTD is shown which defines an `XMLElemType` *A* containing two attributes, namely *b* and *c*. Both attributes are `IDREF`-typed and defined as `#IMPLIED` which means both attributes are optional. In this example, Heuristic 1 is used to resolve the `IDREF` attributes as `EReferences`. In some cases, however, two optional `IDREF`-typed attributes are mutually exclusive. Therefore, `EReferences` *b* and *c* are annotated with the annotation ≪*Resolve XOR constraint manually*≫ indicating that the user must decide if actually an XOR constraint exists between these two relationships or not. For an example application of Heuristic 5 in the context of WebML see Listing 3.4 in Subsection 3.1.3.

Heuristic 6 - Inheritance Identification

This heuristic is based on the assumption that the element type `XMLAnyET` sometimes is used as a container element for other concepts in the described language, i.e., concepts that

have similar properties and in this way may represent sub-types of the `XMLAnyET` element. Hence, Heuristic 6 annotates `EClasses` resulting from an `XMLAnyET` with ≪*Resolve XMLAnyET manually*≫ in order to propose a possible candidate for inheritance.

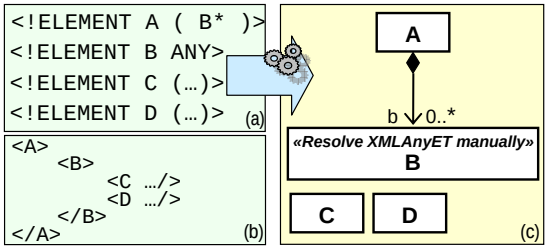

Figure 2.13: Heuristic 6 - Inheritance Identification

Example. In Figure 2.13(a), the abstract example shows a DTD, in which `XMLElemType` *A* contains an `XMLElemParticle` *B*. `XMLElemType` *B*, in turn, is defined as `XMLAnyET` stating that element *B* can be any `XMLElemType` or any text. In Figure 2.13(b), an example XML document is shown, where element *A* contains element *B* which in turn contains elements of type *C* and *D*. In Figure 2.13(c) the corresponding metamodel from the DTD definition contains amongst others an `EClass` *B* which is annotated with ≪*Resolve XMLAnyET manually*≫. This annotation indicates that the user has to decide, how to express the possibility that an instance of `EClass` *B* can contain instances of `EClass` *C* and *D*. In general, this possibility can be expressed as an inheritance relationship, defining `EClass` *B* as the super-class of `EClasses` *C* and *D*. In the context of WebML's hypertext model, an example application of Heuristic 6 is provided in Listing 3.5 in Subsection 3.1.4.

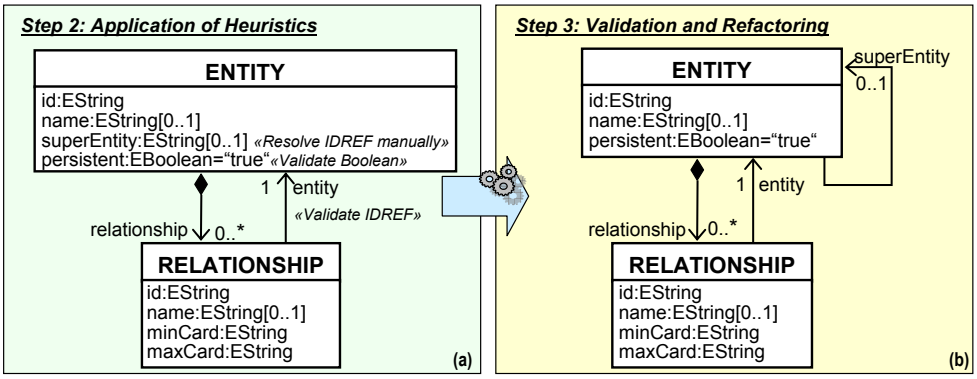

Figure 2.14: Example of Applying Manual Refactoring (Step 3)

2.3.3 Manual Validation and Refactoring of the Generated Metamodel

The second, manual phase of the transformation framework requires user interaction for validating and refactoring the automatically produced metamodel on the basis of domain-knowledge and specifically on the basis of the suggestions annotated by the applied heuristics. In the example shown in Figure 2.14(a), two annotations with respect to Heuristic 1 were introduced indicating that the user should validate, on the one hand, the directed reference introduced between *ENTITY* and *RELATIONSHIP* (≪*Validate IDREF*≫) and, on the other hand, the introduced attribute *superEntity* which was marked with ≪*Resolve IDREF(S) manually*≫. While the directed reference appears to be a correct transformation, the introduced attribute *superEntity* is, in fact, a reference to the *ENTITY* in its role as super-entity used to model inheritance in WebML and therefore shall be manually refactored accordingly by replacing the `EAttribute` with an `EReference` from *ENTITY* to itself with the role name *superEntity* (cf. Figure 2.14(b)). Furthermore, according to Heuristic 2, in the given example the `EAttribute` *persistent* has been annotated with ≪*Resolve possible EBoolean type manually*≫ to indicate the need of manual validation. In this case no manual refactoring is necessary and the annotation can be deleted.

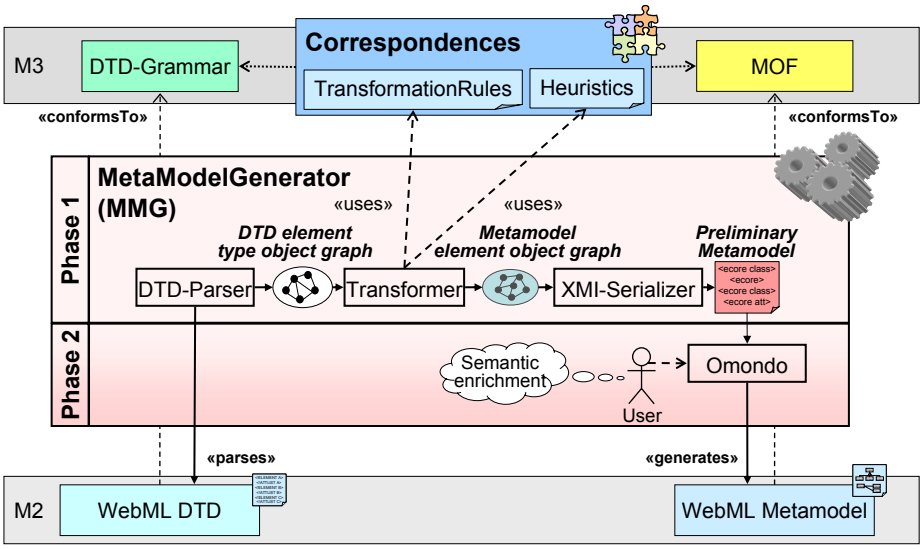

Figure 2.15: Architecture and Mode of Operation of the MMG

2.3.4 Implementation Architecture of the MetaModelGenerator

As already mentioned, the core component of the transformation framework is represented by the MetaModelGenerator. Figure 2.15 shows the details of its implementation architec-

ture. The MMG is based on the EMF and on an open source DTD parser[2]. In a first step a specific DTD, in this case, the WebML DTD, serves as input to the DTD parser, which builds a Java object graph of DTD markup declarations in memory. Then, each element type in the object graph is visited and transformed according to the transformation rules and heuristics described in Subsection 2.3.1 and Subsection 2.3.2 respectively. Each transformation rule is implemented as a separate Java method which takes DTD element type objects as input and generates the objects for the corresponding Ecore elements. If a transformation rule uses a heuristic, then the corresponding method calls a helper method which implements that heuristic. As soon as the complete element object graph of the Ecore-based metamodel has been generated, the default XMI serializer of EMF is activated in order to serialize the metamodel as an XMI file. This XMI file can be loaded into OMONDO[3], a graphical editor for Ecore-based metamodels available as an Eclipse plug-in. In a last step, the annotations created to indicate that a heuristic has been applied, should be validated by the user and the metamodel should be refactored accordingly.

[2] www.wutka.com/dtdparser.html
[3] www.omondo.com

Chapter 3

Evaluation of the DTD to Ecore Framework

Contents

This chapter is dedicated to the evaluation of the DTD2Ecore framework. In Section 3.1, the transformation framework is applied to the WebML DTD and the resulting WebML metamodel is presented. A discussion of the metamodel's completeness with respect to the WebML DTD and the metamodel's quality is given in Section 3.2. Note that the generated WebML metamodel has been used in [Sch07] to define an aspect-oriented extension of the WebML language.

3.1 Case Study on WebML

The Ecore-based metamodel for WebML resulting from the application of the DTD2Ecore transformation framework to the WebML DTD is subject of this section. In particular, the rationale behind some of the manual refactoring decisions shall be explained. Additionally, the WebML metamodel shall be illustrated by relating it to a concrete modeling example, i.e., a demo WebML model that is shipped with WebRatio. This way, the intention is to

briefly explain the language and notation to those unfamiliar with WebML and at the same time indicate the relationship between the model and the metamodel specification as well as informally show the equivalence of the metamodel with the original WebML DTD. A more profound evaluation of the WebML metamodel is provided in Section 3.2.

Please note that the following figures depicting some of the metamodel's packages have been simplified for readability purposes. For the same reason, XOR constraints are illustrated in UML syntax, i.e., with XOR annotated dependencies. For an in-depth description of each modeling concept the reader is referred to [CFB+03]. Before describing some of the packages in more detail and explaining some of the refactoring actions (cf. Subsection 3.1.2 - 3.1.7), an overview on the overall package structure is given.

3.1.1 Root Package "WebML"

The WebML designers have used parameter entities as a mechanism to structure the WebML's language specification. Thus, the WebML language definition consists of several DTDs with *WebML.dtd* being the root DTD that imports the others, which is expressed in Listing 3.1.

Listing 3.1: WebML's Concepts Grouped With External DTDs

```
<!-- WebML.dtd -->
<!ENTITY % StructureDTD SYSTEM "Structure.dtd">
%StructureDTD;
<!ENTITY % NavigationDTD SYSTEM "Navigation.dtd">
%NavigationDTD;
<!ENTITY % PresentationDTD SYSTEM "Presentation.dtd">
%PresentationDTD;
...
```

While *Structure.dtd* and *Navigation.dtd* define the main language concepts that have been introduced in [CFB+03], other rather tool-related DTDs have been introduced in the WebRatio tool. In contrast to previous work [SWK06], where the main focus has been WebML's main language concepts, in this thesis all of WebML's DTDs are considered. This allows for migrating existing WebML models that have been generated using WebRatio into models conforming to the metamodel specification without loosing any information (cf. Subsection 3.2.1) and thus profiting from further MDE techniques such as model transformation.

Figure 3.1 presents a bird's eye view of the resulting WebML metamodel, i.e., its packages and their interrelationships. This structural organization of the WebML concepts has been automatically generated on the basis of Heuristic 3. While *Structure.dtd* corresponds to the *Structure* package in Figure 3.1 and contains concepts for modeling the content level of a web application, the *Navigation* package contains modeling concepts for the hypertext level and has been automatically generated from the *Navigation.dtd*. The rather large *Navigation* package has been manually reorganized into four sub-packages, namely *HypertextOrganization*, *Hypertext*, *ContentManagement*, and *AccessControl*. In addition, the package *Basic* has been introduced, which includes typical abstract concepts, e.g., ModelElement and Named-

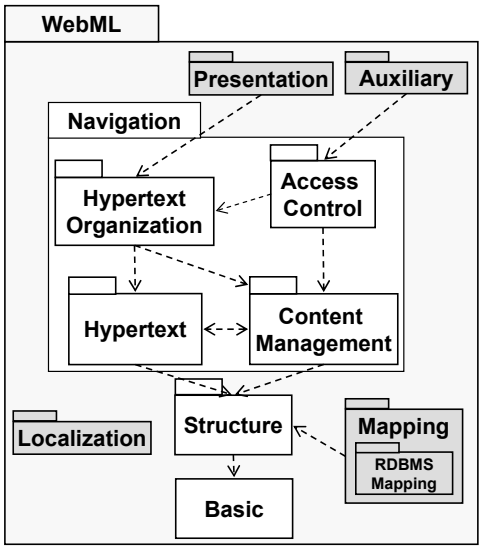

Figure 3.1: Overview of WebML Metamodel Packages

Element, from which all other WebML concepts are derived. The additional gray-shaded packages have been generated from the tool-related DTDs: First, the *Mapping* package imports the *RDBMSMapping* package which provides concepts for specifying the mapping of WebML's content model to a relational database, second, the *Localization* package offers modeling concepts for multilingual web applications, third the *Presentation* package defines concepts for modeling the *Look & Feel* of web applications, and fourth, the graphical illustration and positioning of WebML's notational elements within the WebRatio modeling editor is determined by concepts defined in the *Auxiliary* package.

In the following, a detailed description of the tool-related packages is omitted in favor of presenting the actual WebML language, i.e., the Structure, Navigation, and Basic packages, as well as some of the applied refactoring actions. In order to better illustrate the semantics of the metamodel, the corresponding part of a concrete WebML modeling example will be provided for each package. For this reason, the *ACME* (A Company Manufacturing Everything) example model, which is a demo WebML model shipped with WebRatio, shall be used. It represents a company's website where users can browse and search products as well as special combinations of products. These products and combinations can be edited, extended, and deleted by administrators of the web application.

3.1.2 Package "Structure"

The Structure package (cf. Figure 3.2(a)) contains modeling concepts that allow modeling the content layer of a web application, which regards the specification of the data used by the application. Since, as already mentioned, WebML's content model is based on the ER-model, it basically supports ER modeling concepts: An Entity represents a description of common features, i.e., Attributes, of a set of objects.

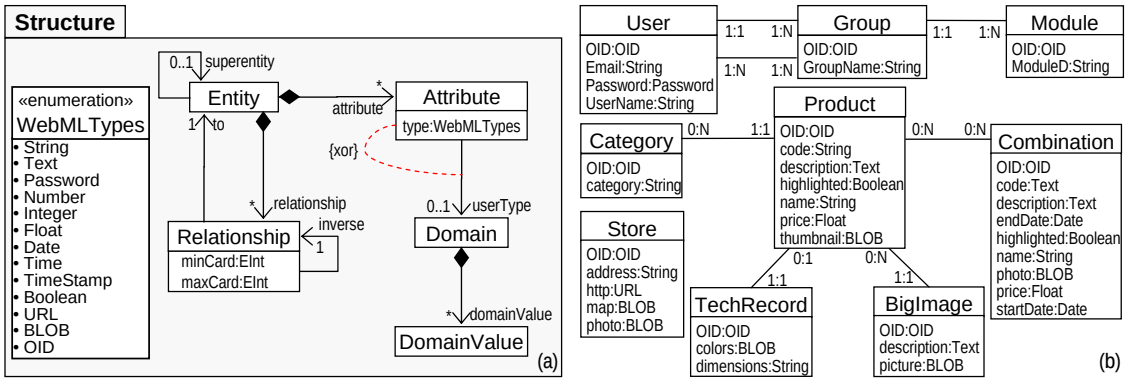

Figure 3.2: Structure

With respect to manual refactoring actions, an XOR constraint has been added to the metamodel in order to specify that Attributes can have either a predefined *type*, e.g., String, Integer, Float, Date, Time, and Boolean, or a *userType*, i.e., an enumeration type represented by Domain and DomainValue, respectively. Entities that are associated with each other are connected by Relationships whereby the type of the meta-attributes *minCard* and *maxCard* of Relationship have been changed from `EString` to `EInt`.

In Figure 3.2(b)[1], the WebML content model of the ACME web application is depicted. *Products* belong to one *Category* and can be described by a *TechnicalRecord* and several *BigImages*. Furthermore, *Products* can be offered within several *Combinations* with other *Products*. In addition, the web application provides information of available *Stores*. The *User*, *Group*, and *Module* entities are used for user management (e.g., normal users and administrators) and access control purposes.

3.1.3 Package "HypertextOrganization"

The HypertextOrganization package includes concepts for structuring the hypertext, i.e., it offers concepts for organizing modeling concept from the Hypertext package (cf. Subsec-

[1]For readability reasons, we do not incorporate "instance-of" relationships from the modeling example part to the metamodel part of the figure.

tion 3.1.4). More specifically, the Page concept is used to organize and structure information from the content level, e.g., ContentUnits from the Hypertext package. Siteviews and Areas in turn group Pages as well as operations on data from the content level, e.g., OperationUnits from the ContentManagement package (cf. Subsection 3.1.5). More specifically, Siteviews represent groups of areas and/or pages devoted to fulfilling the requirements of one or more user groups, while Areas are containers of Pages or nested sub-areas related to a homogeneous subject and are used to hierarchically organize the web application. These concepts are encapsulated within the HypertextOrganization package (cf. Figure 3.3(a)).

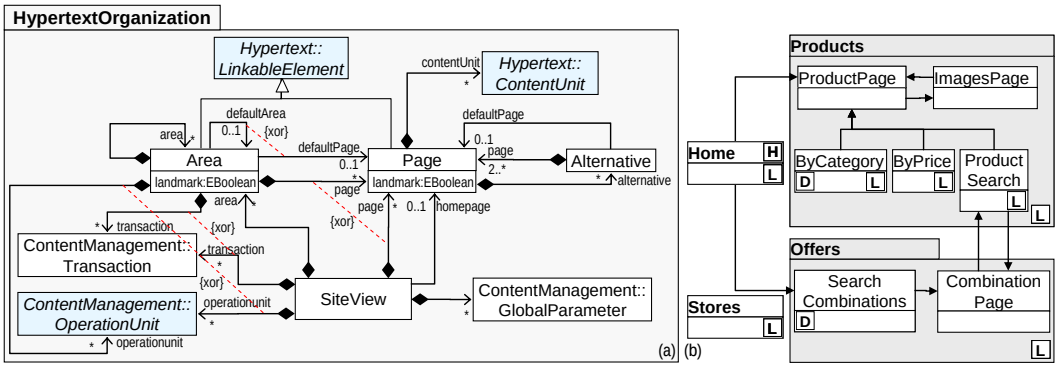

Figure 3.3: HypertextOrganization

With respect to refactoring actions, it was possible to identify an example of the awkward cardinalities problem (cf. Subsection 2.2.3) based on `EAnnotations` created by Heuristic 4. The definition of the Alternative concept requires the Alternative to have at least two sub-pages, which is expressed in the WebML DTD as is depicted in Listing 3.2.

Listing 3.2: Alternative has Two or More Sub-Pages

```
<!ELEMENT Alternative (Page, Page+)>
```

Yet, this definition found in the DTD might be interpreted differently in a metamodel. One possible interpretation is that the first `XMLContentParticle` represents a special page, e.g., a default page. The correct interpretation in the context of WebML [CFB+03] is, however, that the first and the second `XMLContentParticle` together represent one set of alternative Pages, i.e., one containment `EReference`, but with special restrictions on their cardinalities, i.e., 2..*. In metamodels, this constraint can be expressed unambiguously, which is shown by the *Alternative.page* reference in Figure 3.3(a).

While Rule 3 already detected, that Pages and Transactions can be contained by either a Siteview or an Area (cf. Listing 3.3), Heuristic 5 identified further possible candidates for XOR constraints in the HypertextOrganization package.

Listing 3.3: Page is Placed Either Within a Siteview or Within an Area

```
<!ELEMENT Siteview (... Page* ...)>
<!ELEMENT Area (... Page* ...)>
```

In Listing 3.4, an Area can have either a *defaultArea* or a *defaultPage*, but not both at the same time.

Listing 3.4: Area has Either a defaultPage or a defaultArea

```
<!ELEMENT Area (...)>
<!ATTLIST Area
    defaultPage IDREF #IMPLIED
    defaultArea IDREF #IMPLIED
    ...>
```

In the DTD, the attribute list declaration of Area is not able to ensure this constraint at the instance level. Therefore, an XOR constraint has been introduced to specify that either the *defaultPage* `EReference` or the *defaultArea* `EReferences` occurs at the instance layer:

```
context Area inv:
    defaultArea.oclIsUndefined()<>defaultPage.oclIsUndefined()
```

In the ACME WebML model, separate Siteviews for users and administrators have been designed. The first one, the *Web* Siteview, is depicted in Figure 3.3(b). The *Products* Area groups all Pages presenting some information about products and the *Home* Page (H) acts as the entry point of the Siteview. The default page of an Area (D) such as the *ByCategory* Page of the *Products* Area is the one displayed when the Area is entered. Furthermore, Pages and Areas declared as *landmark* (L) are reachable from all other Pages or Areas within their enclosing Siteview or enclosing Area. In this respect, the *landmark* represents a compact way of specifying a set of links to a Page or Area, respectively.

3.1.4 Package "Hypertext"

The hypertext layer represents a view on the content layer of a web application, only, and thus, the Hypertext package reuses concepts from the Structure package, namely, Entity, Relationship, and Attribute. The Hypertext package (cf. Figure 3.4(a)) summarizes Content-Units used, for example, to display information from the content layer, which may be connected by Links in a certain way. Based on Heuristic 6, a candidate `EClass` for introducing inheritance has been identified. In WebML, Pages contain different kinds of ContentUnits (cf. Listing 3.5).

Listing 3.5: Page Contains Different Kinds of ContentUnits

```
<!ELEMENT Page (ContentUnits, ...)>
<!ELEMENT ContentUnits ANY>
```

The XMLAnyET, however, does not restrict which element types are allowed at the instance layer. Again, these constraints have to be ensured by the WebRatio tool. In the metamodel, this problem could be resolved by manually introducing a generalization hierarchy, which includes the additional abstract classes ContentUnit, DisplayUnit, and SortableUnit. This way, it can be ensured that Pages contain sub-classes of ContentUnit, only, and handle the large amount of different kinds of ContentUnits more easily by reducing redundant structural feature definitions.

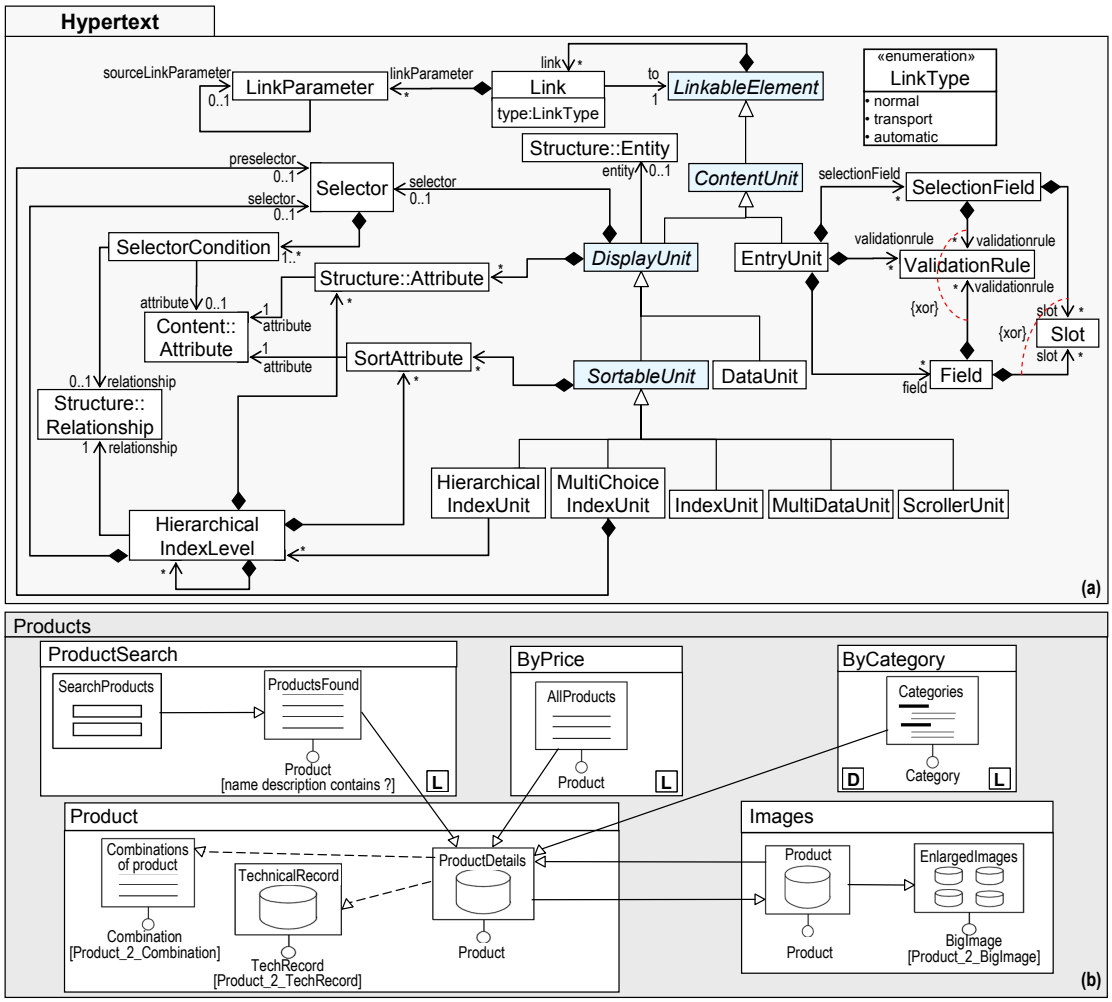

Figure 3.4: Hypertext

The abstract class LinkableElement has been manually introduced in order to cope with other language concepts that can also be connected by Links. This was necessary, since the

`IDREF`-typed *XMLAttribute to* of the Link `XMLElemType` declaration does not restrict the referenced elements to those that the designer originally intended to reference (cf. Listing 3.6).

Listing 3.6: Link Targets are not Specified

```
<!ELEMENT Link (...)>
<!ATTLIST Link
    to IDREF #REQUIRED
    type (normal|automatic|transport) 'normal'
    ...>
```

Furthermore, besides ContentUnits, there are other LinkableElements in the Hypertext-Organization package (cf. Subsection 3.1.3), namely Page and Area, as well as in the Content-Management package (cf. Subsection 3.1.5), namely OperationUnits. More specifically, three disjoint LinkTypes are available in WebML, i.e., *normal*, *automatic*, and *transport* (cf. Figure 3.4(a)). Besides this Link concept, there are also the OKLink and KOLink modeling concepts from the ContentManagement package, which are specifically used to define Links from OperationUnits to other LinkableElements. Consequently, there are multiple sourceElement-link-targetElement tuples of which some are allowed in WebML, only (cf. Table 3.1).

Table 3.1: Linking Possibilities in WebML

From\To	Content Unit	Operation Unit	Page	Area
Content Unit	*normal* *automatic* *transport*	*normal* *transport*	✗	✗
Operation Unit	*transport* *OK* *KO*	*transport* *OK* *KO*	*transport* *OK* *KO*	*transport* *OK* *KO*
Page	✗	*normal* *transport*	*normal* *transport*	*normal*

These sourceElement-link-targetElement tuples, however, are not restricted by the WebML DTD but are implicitly ensured within the WebRatio tool. Aiming at a precise definition of sourceElement-link-targetElement tuples, in the WebML metamodel, the introduction of the LinkableElement concept, which acts as a super-class for all possible sources and targets, is not enough. Consequently, a set of appropriate OCL constraints restricting the tuples to those that are allowed in WebRatio have been introduced (cf. Table 3.1). For example, a Page cannot link ContentUnits, which can be specified with the following OCL constraint:

```
context Page inv:
    self.link->forAll(l | not l.to.oclIsTypeOf(ContentUnit))
```

Figure 3.4(b) shows a refined view of the *Web* Siteview presented in Figure 3.3(b) depicting in detail the *Products* Area: While the *ByPrice* Page uses the IndexUnit *AllProducts* for listing links to all products in ascending order according to their price, the *ByCategory* Page displays a linked list of all products organized according to their categories using a HierarchicalIndexUnit. The *ProductSearch* Page provides an EntryUnit *SearchProducts* with one Field where the user can enter a keyword and displays the found products as an IndexUnit. A single SelectorCondition of the IndexUnit's Selector defines that only those products are to be retrieved, where the keyword is part of the name or the description of the product. A specific product is shown by the *Product* Page, which is linked by all other Pages via Links of type *normal*. The *ProductDetails* DataUnit represents one product from the content model and displays the specified Attributes, only. Furthermore, an additional DataUnit retrieves the technical record of the product and an additional IndexUnit displays a linked list of combinations where the specific product is part of. The information about what technical records and what combinations to retrieve is transported via LinkParameters of Links of type *transport* (dashed arrows), which are neither navigable by nor visible to users. Finally, the *Images* Page again shows some details of a product using a DataUnit and a set of images of the product using a MultiDataUnit.

3.1.5 Package "ContentManagement"

The ContentManagement package contains modeling concepts that allow the modification of data from the content layer. Similar to the generalization hierarchy in the Hypertext package, additional abstract classes have been introduced to the ContentManagement package on the basis of `EAnnotations` created by Heuristic 6 (cf. Figure 3.5(a)), i.e., OperationUnit, ContentManagementUnit, EntityManagementUnit, and RelationshipManagementUnit. In particular, the introduction of the OperationUnit `EClass` ensures that Areas and Siteviews from the HypertextOrganization package contain sub-classes of OperationUnit, only. Since the specific ContentManagementUnits are able to create, modify, and delete Entities as well as establish or delete Relationships between Entities from the content layer, the ContentManagement package reuses concepts from the Structure package, namely Entity and Relationship.

Furthermore, redundant definitions of the same concept have been identified, namely Selector. As an example, a RelationshipManagementUnit may have two Selectors, with one being used in the role of a sourceSelector and the other one being used as a targetSelector. In the WebML DTD, this is expressed as follows (cf. Listing 3.7).

Listing 3.7: Roles of the Selector Concept

```
<!ELEMENT DisconnectUnit (SourceSelector?, TargetSelector?,...)>
<!ELEMENT Selector(SelectorCondition+)>
<!ELEMENT SourceSelector (SelectorCondition+)>
<!ELEMENT TargetSelector (SelectorCondition+)>
```

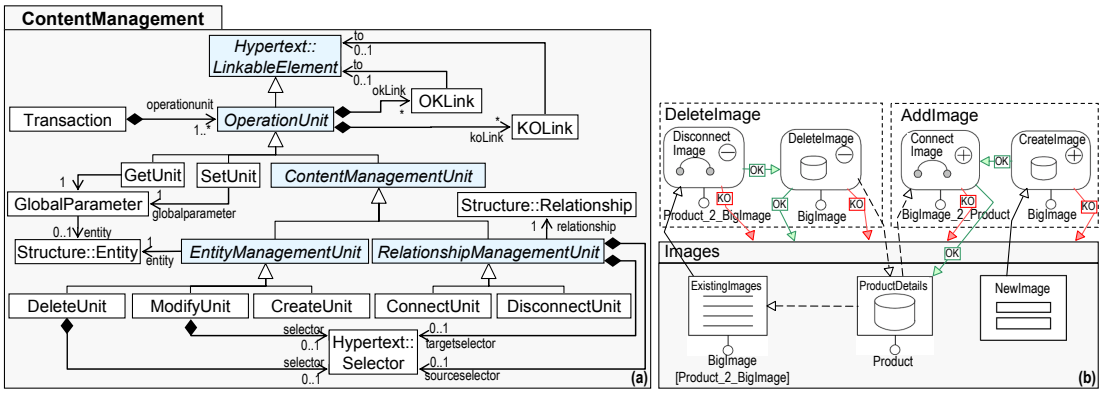

Figure 3.5: ContentManagement

Since the *Targetselector* `XMLElemType` declaration and the *SourceSelector* `XMLElemType` declaration are identical to the *Selector* `XMLElemType` declaration, one can conclude that they represent the same concept as the *Selector* but are used in a special context. In contrast, in metamodels, this context information can be defined as roles, i.e., incorporated by the `EReferences`' names. Therefore, the WebML metamodel only contains the Selector `EClass`, which is referenced by the RelationshipManagementUnit as a *sourceSelector* and *targetSelector*, respectively (cf. the `EReferences` role names in Figure 3.5(a)). A similar example can be found in the Hypertext package, where a Selector can act as *preselector* for MultiChoiceIndexUnits (cf. Figure 3.4(a)).

In the *Administrator* Siteview of the ACME web application, administrators can add, edit, and delete products, combinations, and stores. The *Images* Page in Figure 3.5(b) is part of the *ProductEditing* Area and allows adding and deleting images of a specific product. The Page displays product details in a DataUnit, an IndexUnit of existing images for the product, and an EntryUnit allowing the upload of further images. Selecting an image from the IndexUnit activates the Transaction *DeleteImage*, which has similar semantics as typical database transactions. First, a DisconnectUnit disconnects the image and the products by deleting the specific instance of the relationship, then the OKLink is followed, the image is deleted using a DeleteUnit, and via a second OKLink the *Images* Page is reached again. In case of an error, the KOLinks are followed to the *Images* Page. The *AddImage* Transaction is activated when the user uploads a new image. It first creates a new image with a CreateUnit and then connects it to the specific product with a ConnectUnit.

3.1.6 Package "AccessControl"

In Figure 3.6(a), the AccessControl package groups concepts for controlling the access to Siteviews, namely LoginUnit, LogoutUnit, and ChangeGroupUnit.

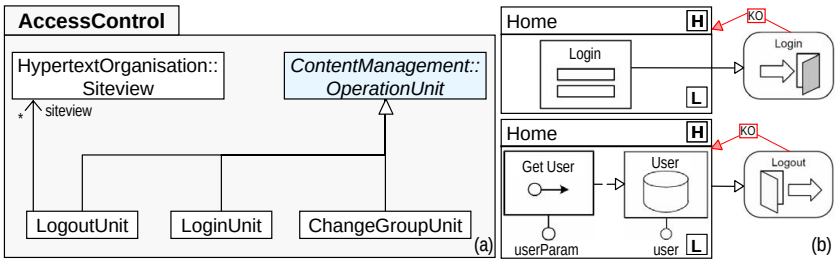

Figure 3.6: AccessControl

The example shows the *Web* Siteview, i.e., the *Home* Page of normal users (cf. Figure 3.6(b)). Administrators have to log in via the EntryUnit *Login*. The LoginUnit verifies username and password and switches to the user's default Siteview, i.e., the *Administrator* Siteview. In the *Home* Page of the *Administrator* Siteview, user information is displayed with the *User* DataUnit. The respective user is obtained from the session with a GetUnit (cf. Subsection 3.1.4). A user logs out via LogoutUnit, which forwards the user back to the *Web* Siteview for normal users.

3.1.7 Package "Basic"

The Basic package consists of three abstract concepts, which encompass some features needed by the majority of WebML's modeling constructs. The ModelElement metaclass represents the root element of the WebML language from which all others inherit, either directly or indirectly. The IdentifiedElement concept encompasses an `EAttribute` *id* as well as containment `EReferences` to the Comment and Property concepts of WebML. Finally, NamedElement represents modeling concepts having an `EAttribute` *name*. Since almost all elements of this package are abstract concepts, no excerpt of the ACME modeling example is provided in Figure 3.7.

3.2 Discussion of the Generated WebML Metamodel

In the following, a discussion on the evaluation of the generated WebML metamodel is provided and shall give an indication on the applicability of the semi-automatic transformation approach. This evaluation is conducted, first, with respect to the metamodel's completeness

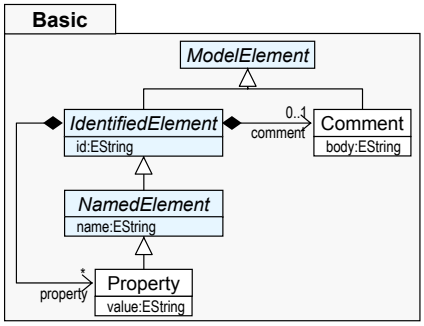

Figure 3.7: Basic Elements

compared to the language concepts defined in the original WebML DTD (cf. Subsection 3.2.1) and, second, on the basis of certain quality metrics (cf. Subsection 3.2.2).

3.2.1 Completeness Criteria

The completeness criteria is fulfilled at the meta-level M2 if the generated WebML metamodel contains all concepts defined within WebML's DTD and the WebRatio tool. At the model level M1 this means that the WebML models can be exchanged in a lossless way, i.e., instances of the WebML metamodel can be transformed to XML documents conforming to the WebML's DTD and vice versa.

Although WebML does provide a formal definition of the semantics of its concepts [CF01], [BCF02], a formal verification of the completeness criteria is not an option. This is due to the fact that currently within EMF the definition of semantics is not provided without executing the model itself. Nevertheless, a first prerequisite for completeness at the M2 level is provided by the fact that each WebML concept present in the DTD is dealt with by at least one transformation rule of the framework, which in turn assures for each WebML concept that there exists at least one counterpart in the metamodel.

In addition, completeness at the M2 level can be further underpinned by considering the M1 level. Taking a first step towards evaluating completeness at the M1 level, an "example-based" strategy has been followed, i.e., an existing WebML reference example has been remodeled on the basis of the generated metamodel. To do so, a tree-based modeling editor for the metamodel has been generated using EMF in order to completely remodel WebRatio's demo example, the ACME E-Store. In addition, the example has been extended by those WebML language concepts not covered in the original example[2].

In a second step, the tree-based modeling editor was enhanced with an import/export facility to demonstrate whether models could be exchanged with the WebRatio tool in a

[2]The modeling editor and the ACME example are available at http://big.tuwien.ac.at/projects/webml/

lossless way[3]. With that facility it is possible to import the extended ACME example from WebRatio into the modeling editor and subsequently to export the model back into a WebRatio XML document. A comparison of the original XML document defined by WebRatio and the exported XML document from the modeling editor with the XML Differencing facility of StylusStudio[4] demonstrated that both were equivalent.

Admittedly, it has to be noted that this is only a first step towards justifying the semantic equivalence of the WebML metamodel and the original language specification not least since the evaluation shall comprise a larger set of more complex examples.

3.2.2 Quality Metrics

The WebML metamodel and its quality characteristics in terms of expressiveness, accuracy, and understandability have evolved considerably during the three-step transformation process. In order to illustrate this evolution, a set of metrics inspired by [MSZJ04] have been applied to the metamodel versions resulting from each step of the transformation process, i.e., the application of transformation rules, the employment of heuristics, and the manual validation and refactoring. The results of applying these metrics are summarized in Table 3.2.

Interpreting these metrics, one can observe that the introduction of a package structure, inheritance, and roles as well as the resolution of the awkward cardinalities deficiency has had a great impact on the understandability and readability of the metamodel. For the manual refactoring phase, the specific impact of introducing the `Basic` package shall be pointed out. The left-hand side of the manual refactoring phase column in Table 3.2 depicts the metrics of the refactoring actions without considering the introduction of the `Basic` package. The metrics on the right-hand side then depict the numbers for the final metamodel and illustrate the positive effect of the introduction of the `Basic` package. In particular, the introduction of inheritance through 17 abstract `EClasses` has helped to decrease complexity by reducing redundant `EAttributes` and `EReferences`. In this respect, the introduction of the three abstract `EClasses` from the `Basic` package played an important role. The identification of 3 roles has diminished the number of `EClasses`, while the resolution of awkward cardinalities has diminished the number of `EReferences`. All in all, the number of 707 modeling elements in the WebML DTD has been reduced to 487 modeling concepts in Ecore (i.e., counting `EClasses`, `EAttributes`, `EReferences`, and `EEnums`). The application of grouping mechanisms according to Heuristic 3 and further manual refactorings also had a positive effect on the language's readability in terms of an introduced package structure and a reduced ratio of `EClasses` per `EPackage`. After manual refactoring, the maximum number of `EClasses` per `EPackage` decreased from 53 to 26.

[3]Note that currently, the import/export facility supports the WebML content model only.
[4]www.stylusstudio.com

Concerning accuracy, the resolution of `IDREF(S)`-typed `XMLAttributes` into `EReferences`, the introduction of `EBoolean`-typed `EAttributes` instead of enumerations and the definition of constraints have considerably contributed to a more precise language. E.g., Heuristic 1 already correctly resolved 39, that is 41% `IDREF`-typed `XMLAttributes` into `EReferences` making the relationship between `EClasses` explicit. Further 56 `EStrings` had to be resolved manually into `EReferences`. From 50 enumeration-typed `XMLAttributes`, 36 were resolved correctly by Heuristic 2 as `EBooleans`. Moreover, further 4 `EEnums` were eliminated reducing their number to 8, the respective `EAttributes` were refactored to `EBooleans`. Nevertheless, 3 more `EEnums` were introduced due to domain knowledge obtained from the WebRatio tool. This results in the final number of 11 `EEnums`, a reduction of `EStrings` in favor of an increase of `EEnum` attributes. And finally, 32 additional constraints have been defined, thus, achieving a more precise WebML metamodel.

Table 3.2: Metamodel Metrics

Metrics		Phase 1 Automatic Transformation		Phase 2 Manual Refactoring	
		Step 1	Step 2	Step 3	
All Modeling Concepts (EClass, EEnum, EAttribute, EReference)		707	707	580	487
EPackage		1	7	11	12
nested EPackage depth (Heuristic 3)		1	3	3	
EClass		96	96	104	107
abstract		0	0	14	17
inheriting from multiple EClasses		0	0	6	20
maximum inheritance depth		0	0	5	7
average inheritance depth		0	0	1.22115	1.66355
annotated with «Resolve XMLAnyET manually» (Heuristic 6)		-	3	-	
annotated with «Resolve Multiplicity manually» (Heuristic 4)		-	1	-	
EClasses/EPackage	MIN	96	1	1	
	MAX	96	53	26	
	AVG	96	13	9	8
EAttribute		338	338	241	191
EString		278	278	180	150
annotated with «Resolve IDREF manually» (Heuristic 1)		-	51	-	
annotated with «Resolve IDREFS manually» (Heuristic 1)		-	5	-	
annotated with «Resolve XOR manually» (Heuristic 5)		-	17	-	
EBoolean (Heuristic 2)		0	46	41	
EEnum		50	14	16	
EInteger		0	0	4	
EReference	annotated with «Validate IDREF» (Heuristic 1)	-	39	65	
	annotated with «Validate IDREFS» (Heuristic 1)	-	0		
	annotated with «Resolve XOR manually» (Heuristic 5)	-	5	-	
Containment EReference		234	234	159	113
EEnum		12	12	11	
annotated with «Resolve possible Boolean type manually» (Heuristic 2)		-	6	-	
OCL constraints	XOR constraint	5	5	10	
	other constraints	-	-	27	
Identified Roles		-	-	3	

Chapter 4

Summary and Related Work

Contents

4.1 Summary

In this part of the thesis, the prominent web modeling language WebML has been bridged to MDE for exploiting MDE benefits such as standardized storage, exchange, and transformation of models. To do so, the WebML language specifications partly available in form of a DTD and partly hard-coded in WebML's modeling tool has been reused to generate a MOF-based WebML metamodel in terms of EMF's Ecore through a semi-automatic transformation process. As a consequence, this part's contributions are as follows:

First, when comparing a language specified in MOF to one specified on the basis of DTDs, it is obvious that DTDs considerably lack extensibility, readability, and understandability for humans, and above all expressiveness. In this respect, a set of eight deficiencies of DTDs when used as a language specification mechanism has been identified.

Second, having elaborated on the concepts of DTDs and MOF as well as their correspondences, a set of rules and heuristics for transforming arbitrary DTDs into MOF-based metamodels has been provided.

Third, a tool, i.e., the MetamodelGenerator for supporting a semi-automatic transformation process from DTD to MOF has been developed. As a consequence, the transformation approach enables the graphical representation of any DTD-based language in terms of MOF-based metamodels and thus, enhances the understandability of those languages.

Fourth, the resulting metamodel for WebML represents an important prerequisite and thus, an initial step towards a transition to employ model-driven engineering techniques (e.g., model transformations or language extensions through profiles) within the WebML approach. It also enables interoperability with other MDE tools and furthermore represents another step towards a common reference metamodel for Web modeling languages.

The resulting WebML metamodel has been evaluated concerning its completeness and quality giving in particular indication on the applicability of the semi-automatic transformation approach. For evaluating completeness, an "example-based" strategy has been followed in that a WebML reference example has been remodeled on the basis of a tree-based modeling editor supporting the generated metamodel. A prototype of an import/export facility of that editor was used to demonstrate that models could be exchanged with the WebML's modeling tool in a lossless way. For evaluating the quality of the metamodel, a set of quality metrics was applied to show the improvement of the metamodel during the semi-automatic transformation process.

4.2 Related Work

With respect to the presented approach of a semi-automatic generation of a MOF-based metamodel for WebML, four areas of related work can be distinguished: First, approaches aiming at the design of metamodels for web modeling languages, second, approaches dealing with the transformation of DTDs to MOF-based metamodels, third, own related work concerning approaches for bridging technical spaces, and fourth, model management, in particular the ModelGen operator.

4.2.1 Defining Metamodels for Web Modeling Languages

To the best of or knowledge, there is currently just one closely related approach focusing on the definition of a UML 2.0 Profile for WebML [MFV06]. The motivation of this approach is to facilitate the interoperability of the WebRatio tool with existing UML modeling tools. More specifically, WebML has been manually remodeled using MOF and in a second step a UML profile has been inferred from it. The approach followed in this thesis differs from the approach of Moreno et al. [MFV06] in three ways. First, we strive for a domain-specific language, for which today, tool support can easily be provided, e.g., based on the EMF. Second, the presented WebML metamodel has been semi-automatically generated from WebML's DTD-based language specification. Third, since also tool related concepts have been considered in the transformation, this approach provides the prerequisite for migrating existing WebML models to MOF, while the WebML profile requires developers to re-model existing WebML models from scratch.

Besides this closely related work, in the context of WebML, three other web modeling approaches which are currently defined on top of a metamodel need to be mentioned, namely W2000 [BCMM06], UWE [KK03], and Muller *et al.* [MSFB05]. W2000 [BCMM06], originally has been defined as an extension to UML. In [BGM02], the provision of a metamodel based on MOF 1.4 [OMG02] has been motivated and adopted as a necessity for providing tool support for an evolving language definition. The metamodel of UWE [KK03] has been designed as a conservative extension to the UML 1.4 metamodel [OMG01]. It is intended as a step towards a future common metamodel for the web application domain, which envisions supporting the concepts of all existing web modeling methods. Muller *et al.* [MSFB05] present a model-driven web application design and development approach through the Netsilon tool. The tool is based on a metamodel specified with MOF 1.4 and the Xion action language. The decision for a metamodel-based approach has been motivated by the fact that in the web application domain the semantic distance between existing modeling elements (e.g., UML) and newly defined modeling elements is becoming too large.

This work is complementary to W2000 and UWE in that a metamodel is proposed for another prominent web modeling language, i.e., WebML. Furthermore, in contrast to these approaches, the WebML metamodel presented in this thesis has been generated semi-automatically, instead of manually deriving it from an existing language definition. Finally, the resulting WebML metamodel is based on Ecore and thus, basically corresponds to MOF 2.0, while the metamodels of the other approaches are based on MOF 1.4.

4.2.2 Transforming between DTDs and Metamodels

There have already been several approaches focusing on the transformation between XML and metamodels [WSKK06]. Summarizing these results, the approaches can be classified according to the direction of the transformation and the concrete formalisms used as source/-target of the transformation. Considering the direction, one can distinguish between forward and backward approaches, regarding the used formalisms the approaches focus on the XML side either on DTDs or XML Schema and on the model side either on MOF, UML or ER, respectively. In the context of this approach, especially those approaches conducting a forward transformation from DTD to MOF are closely relevant. To the best of our knowledge, currently there is no such approach but there are two approaches [BCFK99] and [Sof00] transforming DTDs into UML models which are also closely related, not least since UML is based on MOF. There are, however, two differences with respect to the presented approach. First, a straightforward transformation on basis of the correspondences between the two formalisms is extended by employing a set of heuristics dealing with potential ambiguous correspondences, thus facilitating a manual refactoring of the resulting metamodel. Second, the approach is based on a higher level of abstraction, meaning that the WebML DTDs are considered at the meta-level M2 whereas the other approaches relate

domain DTDs to the model-level M1. Because of this higher level of abstraction, it is possible to transform WebML models in terms of XML documents conforming to the WebML DTD into instances of the corresponding WebML Ecore metamodel, representing in fact a so called "linguistic instantiation" according to [AK03], and to validate if these models indeed fully conform to the WebML Ecore metamodel which is not facilitated by the other approaches. Note that, with respect to UML models, an XML document could in principle be mapped onto an object model, which represents an "ontological instantiation" [AK03] at M1. However, the problem is that the object model must not fulfil the constraints given by the UML model and thus, the "conforms to"-relationship between the XML document and the DTD is lost.

4.2.3 Bridging Technical Spaces

The presented approach for bridging DTDs with Ecore, can be reused for building other technical space bridges. For example, in [WK05], we proposed a generic mechanism for the semi-automatic generation of bridges between the grammar technical space and the model technical space based on the EBNF of a given language. We decided to use EBNF [Wir77, ISO96] as meta-language for the grammar technical space, because it is the most commonly used meta-language for defining programming languages. In the model technical space, our approach is based on MOF as used for the DTD 2 Ecore framework. A bridge between the grammar technical space and model technical space is useful in many software development tasks. For example, platform-specific metamodels are needed in order to transform platform-independent UML models into platform-specific models, such as Java models, from which Java code can be generated. But not only forward engineering can benefit, also reverse engineering of existing software systems is a suitable field of application. Regarding the latter, the OMG is working on model-based reverse engineering and software modernization. For that purpose a special work group for Architecture-Driven Modernization (ADM) has been initiated. The main target of ADM is to rebuild existing applications, e.g, legacy systems, as models. A bridge between the grammar technical space and the model technical space can act as a basic infrastructure tool to support various ADM tasks.

The framework supporting our proposed approach exploits the fact that both, the grammar technical space and the model technical space make a distinction between meta-languages *(M3)* and languages *(M2)*. Figure 4.1 shows the main idea by a correspondence relation at the M3 layer between EBNF and MOF. The main idea is to find correspondences between the concepts of the meta-languages EBNF and MOF. These correspondences are used for defining bridges for the *M2* as well as for the *M1* layer.

EBNF is a reflexive language, i.e., EBNF can be again described in EBNF. We utilize this property for constructing an attributed grammar, which defines, on the one hand, a gram-

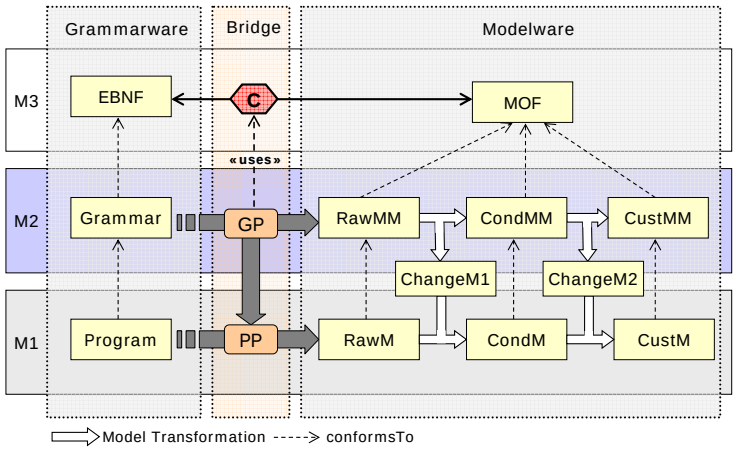

Figure 4.1: Overview on the EBNF 2 MOF Framework

mar for EBNF, and on the other, implements the correspondences between EBNF and MOF as transformation rules. A compiler-compiler takes the attributed grammar as input and generates a parser called *Grammar Parser (GP)*. First, the GP converts grammars defined in EBNF into *Raw Metamodels*, and second, it generates a parser for programs conforming to the processed grammar. The latter is called *Program Parser (PP)*. Via the PP, programs can be transformed into *Raw Models*. The Raw Metamodel and the Raw Model are expressed in *XML Metadata Interchange* (*XMI*) [OMG05c], thus they can be processed by tools form the model technical space. This means, the grammar parser and the program parser act as the main bridging technologies between grammar technical space and model technical space. It is important to note that both parsers are automatically generated from grammars in combination with correspondences between EBNF and MOF.

Once the Raw Metamodel and the Raw Model are created, we have reached the model technical space. However, the Raw Metamodel and the Raw Model can grow very big in terms of number of classes and references. To eliminate this drawback, some transformation rules for optimization are introduced, which can be automatically executed by model transformation engines. The optimization rules are applied to the Raw Metamodel and the outcome of this transformation is called *Condensation Metamodel*. Not only the metamodel has to be optimized, but also the model has to be adjusted in such a way that it conforms to the Condensation Metamodel. This adjustment is defined by a *Change Model* (cf. *ChangeM1* in Figure 4.1) including all required information to rebuild the Raw Model as a *Condensation Model*.

Furthermore our approach provides a mechanism to add additional semantics to the metamodel that cannot be expressed in EBNF. These additional semantics are attached to

the Condensation Metamodel by user-annotations. In particular, these annotations cover aspects like identification/reference semantics, data types and improved readability. The annotated Condensation Metamodel is automatically transformed into a *Customized Metamodel*. Again, the changes in the metamodel layer must be propagated to the model layer. For this task, a second *Change Model* (cf. *ChangeM2* in Figure 4.1) is introduced. The Change Model covers all user-defined modifications and propagates them to the Condensation Model, which is finally transformed into a *Customized Model*.

The main reason why the optimizations are done in the modelware and not in the grammarware is that the framework is designed to be used by model engineers. Apart from that two further reasons have influenced our design decision. First, the optimization rules require potentially complete parse trees, which are available from the Raw (Meta)models. Second, MOF, in contrast to EBNF, has an inherent annotation mechanism, therefore we decided not to directly annotate the EBNF grammars and generate the optimized metamodel in one step.

The concrete transformation and optimization rules as well as possible user annotations may be found in [WK05]. Furthermore, a case study is presented where the MiniJava grammar [Sta05] has been used as input for the EBNF 2 MOF framework.

Compared to our proposed DTD 2 Ecore framework in Section 2.3, the same mining pattern for metamodels and models has been implemented in the EBNF 2 MOF framework. Furthermore, a similar semi-automatic approach is applied, the first step is the automatic generation of an initial metamodel which is refined and semantically enriched in the second step based on user annotations in order to enhance the quality of the metamodels. Summarizing, it can be said that if language definitions are available in a declarative way, the proposed mining pattern and mining process can be applied to define a bridge between two technical spaces.

4.2.4 Model Management: ModelGen Operator

The presented mining pattern is comparable to the *ModelGen* operator proposed by Bernstein [Ber03] in the field of model management. The basic idea of model management is to provide a high-level language which is based on a set of generic operators such as *ModelGen*, *Match*, *Merge*, or *Diff*. These operators can be composed in so-called *scripts* in order to solve more complex integration scenarios which requires the execution of a sequence of mapping operators. In particular, the *ModelGen* operator is used when a source schema *S1* defined in terms of the data model *M1* is given, as well as a target data model *M2*. The *ModelGen* operator is capable of producing a target schema *S2* defined in terms of data model *M2* which semantically corresponds to the source schema *S1*. Furthermore, also instances of source schema *S1*, i.e., the data, is transformed to instances of source schema *S2*. Atzeni et al. [ACB05, ACG07] provided an implementation of the *ModelGen* operator regarding com-

monly used data models such as ER, UML class diagrams, and the relational data model. For the implementation of *ModelGen*, the authors decided to use a so-called *supermodel*, a model that integrates the modeling constructs of commonly used data models and acts as a "pivot" model. In addition, adapters are used to transform ER, UML class diagrams, and relational models into the supermodel formalism. If one wants to transform an ER model into a relational model, then the ER model is transformed with simple one-to-one translations into a supermodel model, then a set of generic operators is applied on the model as long as it only uses concepts of the relational data model, e.g., many-to-many relationships are transformed into additional relations, and finally, the model can be translated with a simple one-to-one transformation into a relational model.

Although our mining pattern also exploits the fact that correspondences on one meta-layer can be used for transforming schemas on the next lower meta-layer as well as for transforming instances of those schemas, we are aiming at the semantic enrichment of the schemas, in our case the automatically produced metamodels. In particular, in the manual tasks, the user has to incorporate constraints in the metamodels which have not been captured in the original language definitions. Therefore, in addition to transformations, we are using on the one hand heuristics and on the other hand user annotations or refactorings to improve the design and precision of the language description. Furthermore, we are not using a "pivot" model, as it is done in [ACB05, ACG07] by employing a supermodel, instead we are defining bi-literal bridges between two technical spaces. However, the applicability of a "pivot" model to ease the definition of model transformations has been demonstrated by Murzek and Kramler [MK07] in the area of business process model integration. For future work, we want to clarify, first, if it is possible to find a "pivot" meta-language for currently used meta-languages of different technical spaces, and second, if it is possible to abstract from bi-literal transformation rules into generic rules based on the "pivot" meta-language.

Part II

Mapping

Chapter 5

A Framework for Building Mapping Operators

Contents

The second part of this thesis elaborates on the following integration scenario. Two metamodels, both describing different modeling languages but defined with the same metalanguage, in our case the Meta Object Facility, have to be bridged in order to ensure model exchange between modeling tools. Figure 5.1 highlights this integration scenario in the overall context of the thesis, i.e., the focus lies on how correspondences can be defined between metamodels and how these correspondences can be used to automatically transform models.

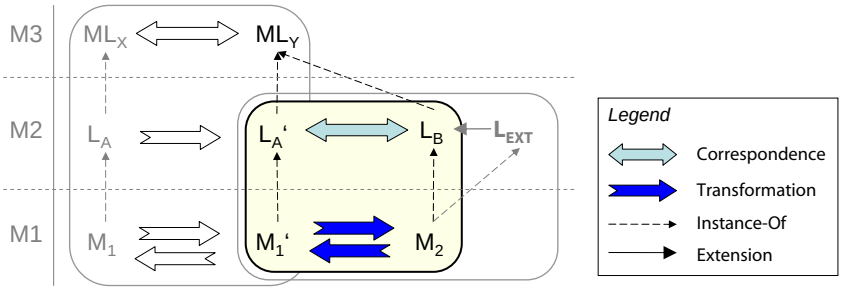

Figure 5.1: Integration Scenarios Revisited – Focus of Part II

Current best practices for realizing seamless exchange of models among different modeling tools employ model transformation languages to realize necessary mappings between concepts of the metamodels defining the modeling languages supported by different tools. Existing model transformation languages, however, lack appropriate abstraction mechanisms for resolving recurring kinds of structural heterogeneities one has to primarily cope with when creating such mappings. Therefore, this chapter proposes a framework for building reusable mapping operators which allow the automatic transformation of models. For each mapping operator, the operational semantics is specified on basis of Colored Petri Nets, providing a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations, thus facilitating understanding and debugging.

5.1 Motivation

Interoperability between modeling tools. As already mentioned in Chapter 1, with the rise of Model-Driven Engineering (MDE) models become the main artifacts of the software development process. Hence, a multitude of modeling tools is available supporting, however, due to lack of interoperability, it is often difficult to use tools in combination, thus the potential of MDE cannot be fully utilized. For achieving interoperability in terms of transparent model exchange, current best practices comprise creating model transformations based on mappings between concepts of different tool metamodels.

Problem Statement. We have followed the aforementioned approach in various projects such as the ModelCVS project [KKK+06b] focusing on the interoperability between legacy case tools (in particular CA's AllFusion Gen) with UML tools and the MDWEnet project [VKC+07] trying to achieve interoperability between different tools and languages for web application modeling. The prevalent form of heterogeneity one has to cope with when creating such mappings between different metamodels is *structural heterogeneity*, a form of heterogeneity well-known in the area of database systems [BLN86, KS96]. In the realm of metamodeling structural heterogeneity means that semantically similar modeling concepts are defined with different metamodeling concepts leading to differently structured metamodels. Current model transformation languages, provide no appropriate abstraction mechanisms or libraries for resolving recurring kinds of structural heterogeneities. Thus, resolving structural heterogeneities requires to manually specify partly tricky model transformations again and again which simply will not scale up having also negative influence on understanding the transformation's execution and on debugging.

Contribution. The contribution of this part of the thesis is twofold. First, in this chapter, a framework is proposed for building reusable mapping operators which are used to define so-called metamodel bridges. Such a metamodel bridge allows the automatic transformation of models since for each mapping operator the operational semantics is specified on

basis of Colored Petri Nets. Colored Petri Nets provide a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations, thus facilitating understanding and debugging. Second, to demonstrate the applicability of our approach we apply the proposed framework for defining a set of mapping operators subsumed in our mapping language called CAR which is presented in Chapter 6. This mapping language is intended to resolve typical structural heterogeneities occurring between the core concepts usually used to define metamodels, i.e., class, attribute, and reference, as provided by the OMG standard MOF [OMG04].

5.2 Metamodel Bridging Framework

In this section, we describe the conceptual architecture of the proposed *Metamodel Bridging Framework* in a by-example manner. The proposed framework provides two views on the metamodel bridge, namely a *mapping view* and a *transformation view* as illustrated in Figure 5.2.

5.2.1 Overview of the Metamodel Bridging Framework

At the mapping view level, the user defines mappings between elements of two metamodels (M2). Thereby a mapping expresses also a relationship between model elements, i.e., the instances of the metamodels [BM07]. In our approach, we define these mappings between metamodel elements with mapping operators standing for a processing entity encapsulating a certain kind of transformation logic. A mapping operator takes as input elements of the source model and produces as output semantically equivalent elements of the target model. Thus, it declaratively describes the semantic correspondences on a high-level of abstraction. A set of applied mapping operators defines the mapping from a left hand side (LHS) metamodel to a right hand side (RHS) metamodel further on subsumed as *mapping model*.

For actually exchanging models between different tools, the mapping models have to be executed. Therefore, we propose, in addition to the mapping view, a transformation view which is capable of transforming models (M1) from the LHS to the RHS on basis of Colored Petri Nets [Jen92].

5.2.2 Mapping View

For defining mapping operators and consequently also for building mapping models, we are using a subset of the UML 2 component diagram concepts. With this formalism, each mapping operator can be defined as a dedicated component, representing a modular part of

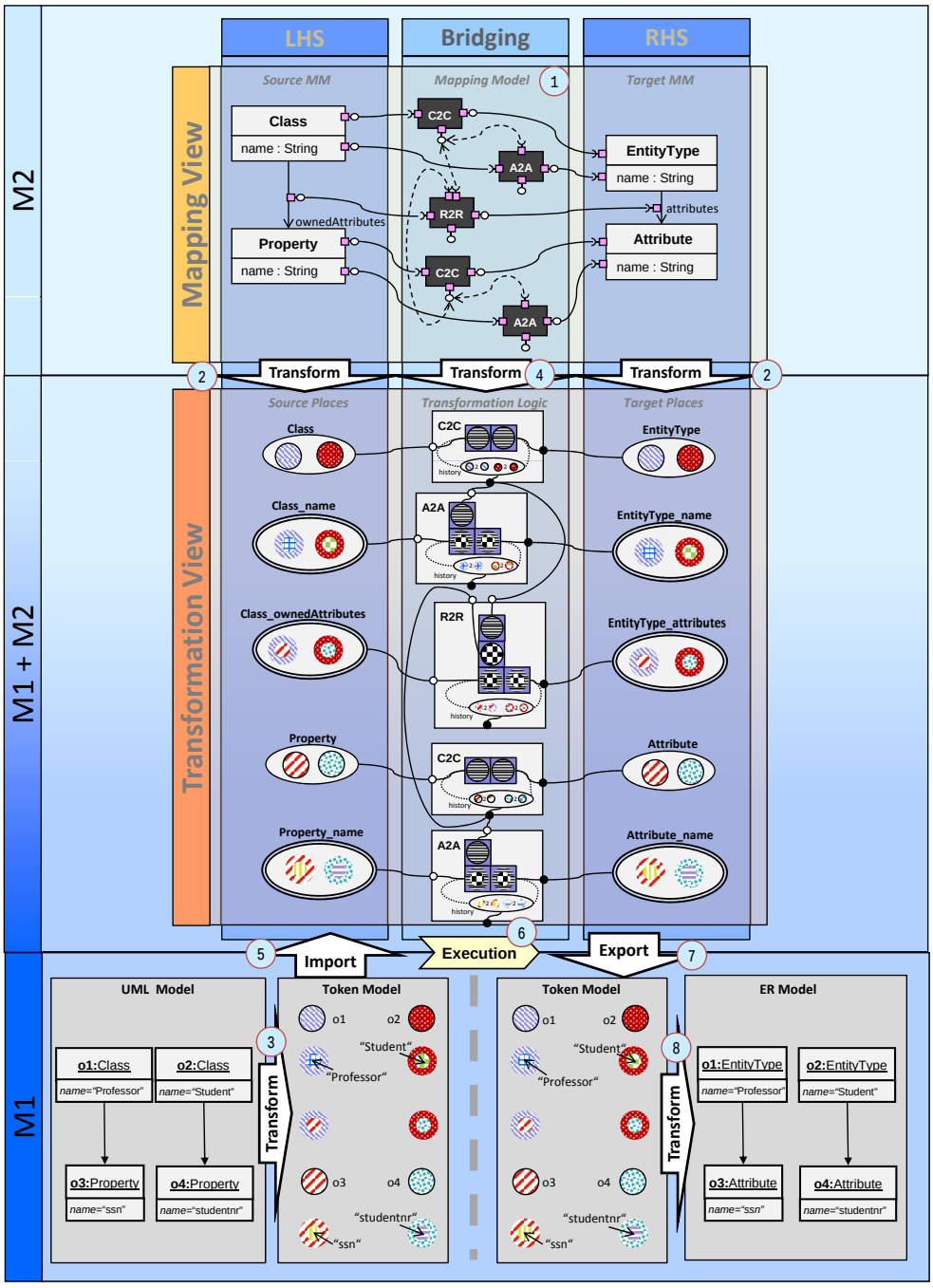

Figure 5.2: Metamodel Bridging Framework by Example

the mapping model which encapsulates an arbitrary complex structure and behavior, providing well-defined interfaces to the environment. The resulting components are collected in a mapping operator library which can be seen as a domain-specific language for bridging metamodels. The user can apply the mapping operators expressed as components in a plug&play manner, i.e., only the connections to the provided and required interfaces have to be established manually.

Our motivation for using UML 2 component diagrams for the mapping view is the following. First, many software engineers are likely to be familiar with the UML component diagram notation. Second, the provided and required interfaces which can be typed, enable the composition of mapping operators to resolve more complex structural heterogeneities. Third, the clear separation between *black-box* view and *white-box* view of components allows switching between a high-level mapping view and a detailed transformation view, covering the operational semantics, i.e., the transformation logic, of an operator.

Anatomy of a mapping operator. Each mapping operator (as for example shown in the mapping model of Figure 5.2) has *input ports* with required interfaces (left side of the component) as well as *output ports* with provided interfaces (right side of the component). Because each mapping operator has its own *trace model*, i.e., providing a log about which output elements have been produced from which input elements, an additional *providedContext port* with a corresponding interface is available on the bottom of each mapping operator. This port can be used by other operators to access the trace information for a specific element via *requiredContext ports* with corresponding interfaces on top of the operator.

In the mapping view of Figure 5.2 (cf. step 1), an example is illustrated where a small part of the metamodel of the UML class diagram (cf. source metamodel) is mapped to a part of the metamodel of the Entity Relationship diagram (cf. target metamodel). In the mapping view, source metamodel elements have provided interfaces and target metamodel elements have required interfaces. This is due to the fact that in our scenario, models of the LHS are already available whereas models of the RHS must be created by the transformation, i.e., the elements of the LHS must be streamed to the RHS according to the mapping operators. Consequently, *Class* and *Property* of the source metamodel are mapped to *EntityType* and *Attribute* of the target metamodel with *Class2Class* (*C2C*) operators, respectively. In addition, the C2C operator owns a providedContext port on the bottom of the component which shall be used by the requiredContext ports of the appropriate *Attribute2Attribute* (*A2A*) and *Reference2Reference* (*R2R*) operators to preserve validity of target models. In particular, with this mechanism it can be ensured that values of attributes are not transformed before their owning objects has been transformed and links as instances of references are not transformed before the corresponding source and target objects have been transformed.

5.2.3 Transformation View

The transformation view is capable of executing the defined mapping models. For this, so called *transformation nets* [RWK07, Rei08] are used which are a special kind of Colored Petri Nets consisting of source places at the LHS and target places at the RHS. Transitions between the source and target places describe the transformation logic located in the bridging part of the *transformation net* as shown in Figure 5.2.

Transformation nets provide a suitable formalism to represent the operational semantics of the mapping operators, i.e., the transformation logic defined in the white-box view of the component due to several reasons. First, they enable the execution of the transformation thereby generating the target model out of the source model, which favors also debugging of a mapping model. Second, they allow a homogeneous representation of all artefacts involved in a model transformation (i.e., models, metamodels, and transformation logic) by means of a simple formalism, thus being especially suited for gaining an understanding of the intricacies of a specific metamodel bridge.

In the next paragraphs, we discuss rules for assembling metamodels, models, and mapping models into a single transformation net and how the transformation can actually be executed.

Places represent Metamodels. First of all, places of a transformation net are used to represent the elements of the source and target metamodels (cf. step 2 in Figure 5.2). In this respect, we currently focus on the three major building blocks of metamodels (provided, e.g. by meta-metamodels such as MOF), namely *class*, *attribute*, and *reference*. In particular, classes are mapped onto one-colored places whereby the name of the class becomes the name of the place. The notation used to visually represent one-colored places is a circle or oval as traditionally used in Petri Nets. Attributes and references are represented by two-colored places, whereby the name of the containing class plus the name of the reference or of the attribute separated by an underline becomes the name of the place (cf. e.g. *Class_name* and *Class_ownedAttributes* in Figure 5.2). To indicate that these places contain two-colored tokens, the border of two-colored places is double-lined.

Tokens represent Models. The tokens of the transformation net are used to represent the source model which should be transformed according to the mapping model. Each element of the source model is expressed by a certain token, using its color as a means to represent the model element's identity in terms of a String (cf. step 3 in Figure 5.2). In particular, for every object, a one-colored token is produced, whereby for every link as an instance of a reference, as well as for every value of an attribute, a two-colored token is produced. The *fromColor* for both tokens refers to the color of the token that corresponds with the containing object. The *toColor* is given by the color of the token that corresponds with the referenced target object or the primitive value, respectively. Notationally, a two-colored token consist of a ring (carrying the fromColor) surrounding an inner circle (depicting the

toColor).

Considering our example shown in Figure 5.2, the objects *o1* to *o4* of the UML model shown in the M1-layer are transformed into one-colored tokens. Each one-colored token represents an object identity, pointed out by the object name beneath the token. E.g., the tokens with the inner-color *"Student"* and *"Professor"* have the same outer-color as their containing objects and the token which represents the link between object *o1* and *o3* has the same outer-color as the token representing object *o1* and the inner-color corresponds to the one-colored token representing object *o3*.

Transitions represent Mapping Models. The mapping model is expressed by the transformation logic of the transformation net connecting the source and the target places (cf. Step 4 in Figure 5.2). In particular, the operational semantics of the mapping operators are described with transitions, whereby the behavior of a transition is described with the help of preconditions called *query-tokens* (LHS of a transition) and postconditions called *generator-tokens* (RHS of a transition). Query-tokens and generator-tokens can be seen as templates, simply visualized as color patterns, describing a certain configuration of tokens. The precondition is fulfilled and the transitions fires, if the specified color pattern described by the query-tokens matches a configuration of available input tokens. In this case, the postcondition in terms of the generator-tokens produces the required output tokens representing in fact the necessary target model concepts.

In the following, the most simple mapping operators used in our example are described, namely C2C, A2A, and R2R.

C2C. The white-box view of the C2C operators as shown in the transformation view of Figure 5.2 ensures that each object instantiated from the class connected to the input port is streamed into the mapping operator, the transition matches a single token from the input port, and streams the exact token to the output port. This is expressed in the transition by using the most basic query-token and generator-token combination, both having the same color pattern. In addition, every input and output token combination is saved in a history place representing the trace model which is connected to the *providedContext* port and can be used as trace information by other operators.

A2A. The white-box view of the A2A operator is also illustrated in the bridging part of the transformation view in Figure 5.2. Two-colored tokens representing attribute values are streamed via the input port into the mapping operator. However, a two-colored token is only streamed to the output port if the owning object of the value has been already transformed by a C2C operator. This is ensured in that the transition uses the same color pattern for the one-colored query-token representing the owning object streamed from the *requiredContext* port and for the outer color of the two-valued query-token representing the containing object of the attribute value. Only, if a token configuration matches this precondition, the two-colored token is streamed via the generator-token to the output port. Again, the input tokens and the corresponding output tokens are stored in a history place

which is connected to the *providedContext* port.

R2R. The white-box view of the R2R operator shown in the transformation view of Figure 5.2 consists of three query-tokens, one two-colored query-token representing the link and two one-colored query-tokens for accessing trace information from C2C operators. The two-colored query-token must have the same inner and outer colors as provided by the C2C trace information, i.e., the source and target objects must be already transformed. When this precondition is satisfied by a token configuration, the two-colored token representing the link is streamed via the generator-token to the output port.

Execution of the transformation logic. As soon as the metamodels are represented as places, which are furthermore marked with the respective colored tokens depicting the concepts of the source model (cf. step 5 in Figure 5.2), the transformation net can be started. Now, tokens are streamed from the source places over the transitions into the target places (cf. step 6 in Figure 5.2).

Considering our running example, in a first step only the transitions of the C2C operators are able to fire due to the dependencies of the A2A and R2R operators. Hence, tokens from the places *Class* and *Property* are streamed to the appropriate places of the RHS and all combinations of the queried input and generated output tokens are stored in the trace model of the C2C operator. As soon as all necessary tokens are available in the trace model, depending operators, i.e., the A2A and R2R operators, are also able to fire.

Generation of the target model. After finishing the transformation, the tokens from the target places can be exported (cf. step 7 in Figure 5.2) and transformed back into instances of the RHS metamodel (cf. step 8 in Figure 5.2).

In our example, the one-colored tokens *o1* to *o4* contained in the target places are transformed back into objects of type *EntityType* and *Attribute*. The two-colored tokens which represent attribute values, e.g., *"Professor"* and *"Student"*, are assigned to their containing objects, e.g., *o1* and *o2* whereas *"ssn"* and *"studentnr"* are assigned to *o3* and *o4*. Finally, the two-colored tokens which represent links between objects are transformed back into links between *o1* and *o3*, as well as between *o2* and *o4*.

5.2.4 Implementation Architecture of the Metamodel Bridging Framework

In this subsection, the implementation of the metamodel bridging framework is described. More specifically, this framework has been implemented based on the infrastructure provided by the Eclipse Modeling Framework (EMF)[1] and the Graphical Modeling Framework (GMF)[2] as two separate Eclipse plug-ins. First, we have developed a plug-in called *CAR Mapping ENvironment* (CARMEN) used for graphically defining mapping models between

[1] www.eclipse.org/emf
[2] www.eclipse.org/gmf

two metamodels shown in the upper part of Figure 5.3. Second, for executing the mappings and for defining Transformation Nets manually from scratch, we have developed a plug-in called *Transformations On Petri nets In Color* (TROPIC) shown in the lower part of Figure 5.3. In the following, it is shortly explained how the plug-ins fit together and which functionality is supported as is illustrated in Figure 5.3.

CARMEN. In order to provide a graphical mapping environment, we developed a prototypical implementation of a mapping editor based on the GMF, which is able to load two different metamodels in UML class diagram notation. As soon as the metamodels are loaded, the user can build a graphical mapping model between them using a set of available mapping operators. In particular, the initial set of mapping operators comprises the CAR mapping operators which are introduced in Chapter 6. For new mapping operators, the user can define new subclasses of the abstract class *Mapping* thereby defining their abstract syntax as well as is able to specify a customized graphical syntax in cases where the default component-based syntax is not desired. In addition, the mapping editor provides the validation of the mapping model that facilitates finding errors in the mappings. This validation support is especially needed for situations where mapping models are subsequently automatically processed in order to transform models as is the case in our approach. For executing the mappings, the user can activate the generation of a corresponding transformation net which can be loaded into TROPIC. Furthermore, this requires, if new mapping operators are introduced by the user, also new generation rules have to be provided by the user to represent these operators within transformation nets.

TROPIC. TROPIC provides a graphical editor based on GMF to build transformation nets by hand, however, in our approach, it is intended that transformation nets are automatically generated from mapping models. Therefore, TROPIC is able to load the generated transformation nets into the graphical editor and visualize the transformation net by a default layout. Besides the graphical editor, TROPIC consists of three further subcomponents: first, a component for transforming models into tokens, second, a component for executing the transformation nets, i.e., streaming tokens from the source places to the target places by firing enabled transitions, and third, a component for transforming tokens back into models.

Open issues concerning tool support are first, a more user-friendly definition of new mapping operators by using additional editors for modeling the abstract syntax and the concrete syntax as well as the operational semantics of the operators without extending the metamodel and implementing transformation rules by hand, and second, the separate plug-ins should be integrated into one single environment which has two different views on the same integration problem. This means, the user should build the mapping model and can switch immediately to the transformation view for testing the defined mappings without an additional generation step.

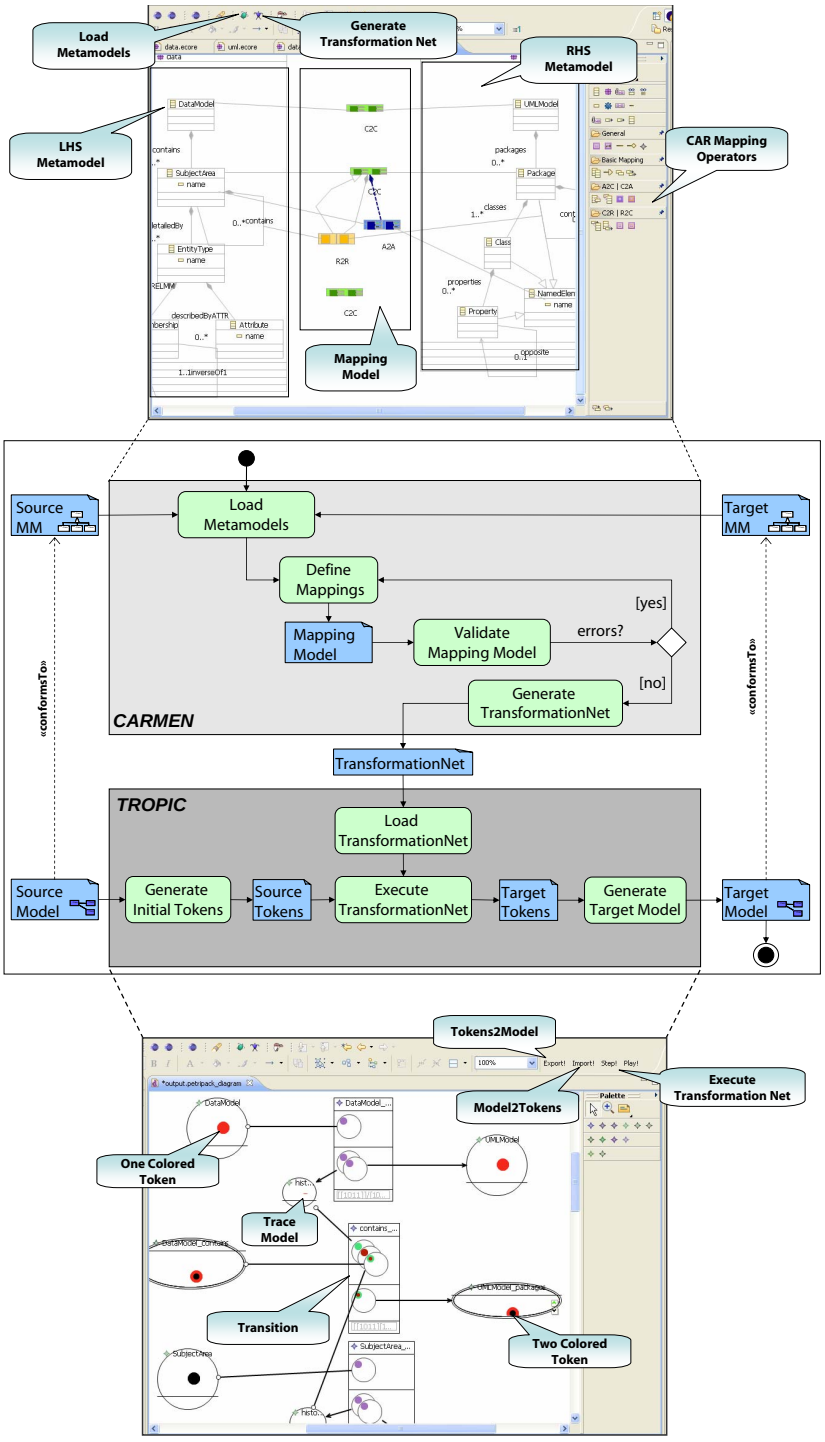

Figure 5.3: CARMEN and TROPIC – Activities, Artifacts, and Tooling

Chapter 6

CAR – A Mapping Language for Resolving Structural Heterogeneities

Contents

To demonstrate the applicability of the Metamodel Bridging Framework, we apply the proposed framework for defining a set of mapping operators which are intended to resolve typical structural heterogeneities occurring between the core concepts usually used to define metamodels. In Section 6.1, a motivating example is presented which introduces structural heterogeneities between metamodels. To resolve these heterogeneities, in Section 6.2 the syntax and operational semantics of the operators forming the CAR mapping language are presented and applied to solve the integration example. Finally, in Section 6.3, we present how a reuse mechanism based on inheritance between mapping operators reduces the size of mapping models, thus improving their readability.

6.1 Motivating Example

Based on experiences gained in various interoperability projects [KKK+06a, KKK+07, WSSK07, WSS+07] it has been shown that although most meta-metamodels such as MOF offer only a core set of language concepts for defining metamodels, numerous structural heterogeneities occur when defining modeling languages.

As an example for structural metamodel heterogeneity consider the example shown in Figure 6.1. Two MOF-based metamodels represent semantically equivalent core concepts of the UML class diagram in different ways. Whereas the LHS metamodel uses only a small set of classes, the RHS metamodel employs a much larger set of classes thereby representing most of the UML concepts which are in the LHS metamodel implicitly defined as attributes or references explicitly as first class citizens. More specifically, five structural metamodel heterogeneities can be found which require mapping operators going beyond the simple *one-to-one* mappings provided by the mapping operators in Section 5.2.

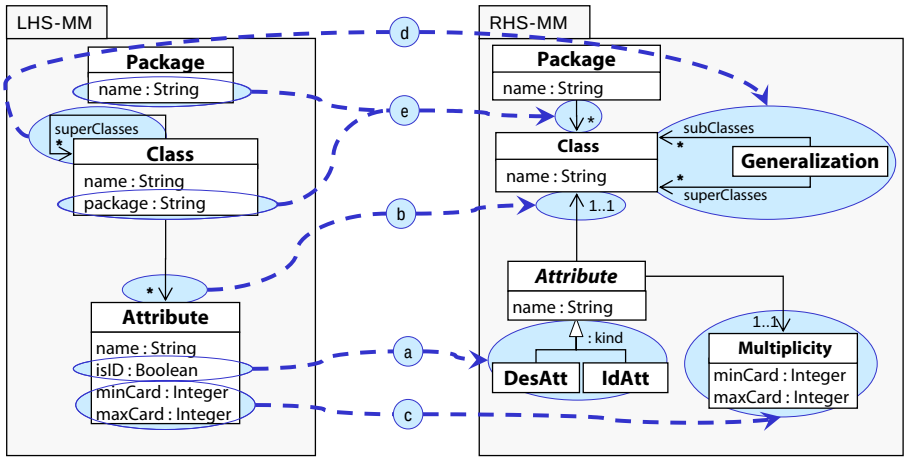

Figure 6.1: Structural Heterogeneities Between Metamodels - Example

6.2 Mapping Operators

6.2.1 Overview of the CAR Mapping Language

At this time, we provide nine different core mapping operators for resolving structural metamodel heterogeneities as depicted in Figure 6.1. These nine mapping operators result from the possible combinations between the core concepts of meta-metamodels, namely *class*, *attribute*, and *reference*, which also led to the name of the CAR mapping language.

These mapping operators are designed to be declarative and it is possible to derive executable transformations based on transformation nets. One important requirement for the CAR mapping language is that it should be possible to reconstruct the source models from the generated target models, i.e., any loss of information during transformation should be prevented. In Figure 6.2, the mapping operators are divided according to their functionality into the categories *Copier*, *Peeler*, and *Linker* which are explained in the following.

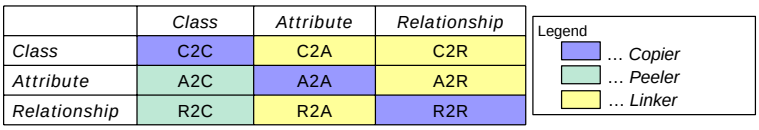

Figure 6.2: CAR Mapping Operators

Copier. The diagonal of the matrix in Figure 6.2 depicts the symmetric mapping operators of the CAR mapping language which have been already discussed in Section 5.2. The term symmetric means that the input and outport ports of the left side and the right side of the mapping operators are of the same type. This category is called *Copier*, because these mapping operators copy one element of the LHS model into the RHS model without any further manipulations.

Peeler. This category consists of mapping operators which create new objects by "peeling"[1] them out of values or links. The A2C operator bridges heterogeneities which are resulting from the fact that a concept is expressed as an attribute in one metamodel and as a class in another metamodel. Analogously, a concept can be expressed on the LHS as a reference and on the RHS as a class which can be bridged by a R2C operator.

Linker. The last category consists of mapping operators which either link two objects to each other out of value-based relationships (cf. A2R and R2A operator) or assign values or links to objects for providing the inverse variants of the A2C and R2C operators (cf. C2A and C2R operator).

To resolve the structural heterogeneities depicted in Figure 6.1, in the following subsections the necessary mapping operators are discussed in detail, comprising besides a variation of the C2C operator mainly mapping operators falling into the above mentioned peeler and linker categories.

6.2.2 Conditional C2C Mapping Operator

Problem. In MOF-based metamodels, a property of a modeling concept can be expressed via a discriminator of an inheritance branch or with an additional attribute. An example

[1]Note that the term "peeling" is used since when looking at the white-box view, the transformation of an attribute value into an object requires in fact to generate a one-colored token out of a two-colored token.

for this kind of heterogeneity can be found in Figure 6.1(a), namely between *Attribute.isID* on the LHS and the subclasses of the class *Attribute* on the RHS. This heterogeneity is not resolvable with a primitive C2C operator per se, because one class on the LHS corresponds to several classes on the RHS whereby each mapping is only valid under a certain condition. On the model level, this means that a set of objects has to be splitted into several subsets based on the object's attribute values.

Solution. To cope with this kind of heterogeneity, the C2C operator has to be extended with the capability of splitting a set of objects into several subsets. For this we are annotating the C2C operator with OCL-based preconditions assigned to ports as depicted in Figure 6.3. These preconditions supplement the query-tokens of the transitions by additionally allowing to specify constraints on the source model elements. The reason for introducing this additional mechanism at the black-box view of a mapping operator is that the user should be able to configure the C2C operator without having to look into the white-box view of the operator, realizing its basic functionality.

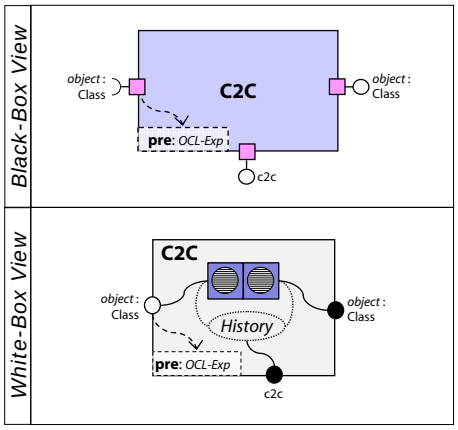

Figure 6.3: Conditional C2C Mapping Operator

Example Application. In the example shown in Figure 6.7[2], we can apply two C2C mapping operators with OCL conditions, one for mapping *Attribute* to *DesAtt* with the precondition *Attribute.isID = false*, and one for mapping *Attribute* to *IdAtt* with the precondition *Attribute.isID = true*. In addition, this example shows a way how mappings can be reused within a mapping model by allowing inheritance between mappings. This mechanism allows to define certain mappings directly between *superClasses* and not for each *subClass* combination again and again (cf., e.g., the A2A mapping between the *Attribute.name* attributes). More details about this inheritance mechanism are given in Section 6.3.

[2]The corresponding transformation net is shown in Figure 6.8.

6.2.3 R2R Mapping Operator with Annotations

Problem: In MOF-based metamodels, a relationship between two classes can be expressed via a reference from Class A to Class B or, the opposite way round, from Class B to Class A. When these two metamodels have to be bridged, it is not possible to use the basic R2R operator, because it is not enough to simply copy the links from left to right. Instead, the inverse reference has to be computed in order to build the target model appropriately. An example for this kind of heterogeneity can be found in Figure 6.1(b), namely between the reference from *Class* to *Attribute* on the LHS and the reference from Attribute to Class on the RHS.

Solution: The computation of inverse references can be seen as a variability point of the R2R mapping operator. Therefore, we propose to use an additional annotation mechanism, namely *tagged/value* pairs, for configuring an operator to support different variants of a bridging problem. We decided to use tagged/value pairs in addition to OCL annotations, because some variants of operators can be easily configured via tags, which are tedious to express in OCL for each problem again and again. More specifically, the annotation mechanism enables to support more than one white-box view of a mapping operator. A way to avoid this kind of annotations would be to provide a different type of integration operator for every situation. The other extreme case would be to have a single generic integration operator that is parameterized for every situation. Of course, the first solution would result in an overly large set of operators, with probably only minor differences in behavior. Similarly, the other extreme would simply result in an equally large set of parameters mimicking the different types of operators. Hence, the question arises when to use or not to use parameterizations or dedicated operators, respectively. We decided to use a manageable number of mapping operators and user-definable parameters that pin down semantic variations concerning specific runtime behaviors. Consequently, mapping operators which have the same black-box view but different white-box views, are defined as one single operator which is configurable with tagged/value pairs for deciding which transformation logic, i.e., which white-box view, is chosen.

Figure 6.4 illustrates how the R2R mapping operator can be annotated in order to be also capable of computing inverse references. In the black-box view, the user can define via the *inverse* tag which is of kind boolean, if the inverse reference should be computed or not . If the inverse tag is not set, the standard R2R white-box view is used (cf. left white-box view in Figure 6.4), which only copies links from left to right. However, if the inverse tag is set, then the right white-box view shown in Figure 6.4 is used, which matches a two colored token from its input place, and produces an inverted token, i.e., fromColor and toColor are interchanged, for the output place.

Example Application. In Figure 6.7, a *R2R* mapping operator is applied with *inverse* tag set to true for mapping the reference from *Class* to *Attribute* of the LHS metamodel to the

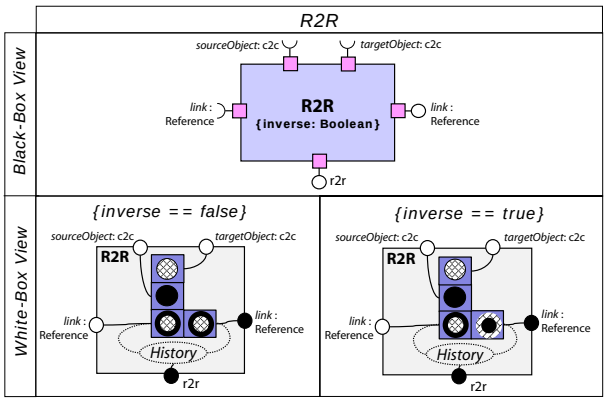

Figure 6.4: R2R Mapping Operator with Inverse Annotation

reference from *Attribute* to *Class* of the RHS metamodel.

6.2.4 A2C Mapping Operator

Problem. In Figure 6.1(c), the attributes *minCard* and *maxCard*, which are part of the class *Attribute* at the LHS, are at the RHS part of a dedicated class *Multiplicity*. Therefore, on the instance level, a mechanism is needed to "peel" objects out of attribute values and to additionally take into account the structure of the LHS model in terms of the attribute's owning class when building the RHS model, i.e., instances of the class *Multiplicity* must be connected the corresponding to instances of class *Attribute*.

Solution. The black-box view of the A2C mapping operator as illustrated in Figure 6.5 consists of one or more required interfaces for attributes on the LHS depending on how many attributes are contained by the additional class, and has in minimum three provided interfaces on the RHS. The first of these interfaces is used to mark the reference which is responsible to link the two target classes, the second is used to mark the class that should be instantiated, and the rest is used to link the attributes of the LHS to the RHS. Additionally, an A2C operator has a required interface to a C2C, because the source object is splitted into two target objects, thereby only one object is created by the A2C, the other has to be generated by a C2C operator which maps the LHS class to its corresponding target RHS class.

The white-box view of the A2C operator shown in Figure 6.5 comprises a transition consisting of at least two query-tokens. The first query-token guarantees that the *owningObject* has been already transformed by a C2C operator. The other query-tokens are two-colored tokens representing the attribute values which have as fromColor the same color as the first query-token. The post-condition of the transition consists of at least three generator-tokens.

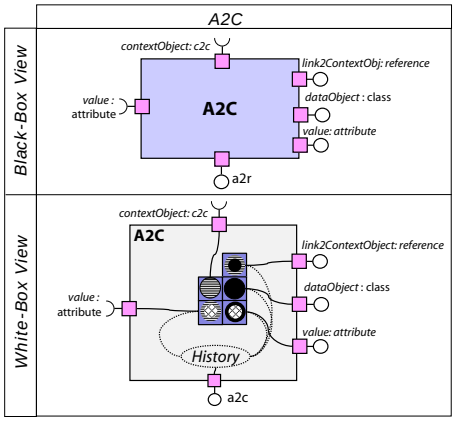

Figure 6.5: A2C Mapping Operator

The second generator-token introduces a new color, i.e., this color is not used in the pre-condition part of the transition, and therefore, the generator-token produces a new object with an unique identity. The first generator-token is used for linking the newly created object appropriately into the target model and the other two-colored generator tokens are used to stream the values into the newly generated object by changing the fromColor of the input values.

Example Application. In Figure 6.7, the attributes *minCard* and *maxCard* are mapped to attributes of the class *Multiplicity*. Furthermore, the reference between the classes *Attribute* and *Multiplicity* is linked by the A2C mapping as well as the class *Multiplicity*. To assure that the generated *Multiplicity* objects can be properly linked to *Attribute* objects, the A2C mapping is in the context of the C2C mapping between the *Attribute* classes.

6.2.5 R2C Mapping Operator

Problem. In Figure 6.1(d), the reference *superClasses* of the LHS metamodel corresponds to the class *Generalization* of the RHS metamodel. This kind of heterogeneity requires an operator which is capable of "peeling" an object out of a link and to additionally preserve the structure of the LHS in terms of the classes connected by the relationships at the RHS.

Solution. The black-box view of the R2C mapping operator, as depicted in Figure 6.6, has one required interface on the left side for pointing to a reference. On the right side it has three provided interfaces, one for the class which stands for the concept expressed as reference on the LHS and two for selecting the references which are responsible to connect the object which has been peeled out of the link of the LHS into the RHS model. To determine the objects to which the peeled object should be linked, two additional required interfaces

on the top of the R2C operator are needed for determining the corresponding objects of the source and target objects of the LHS.

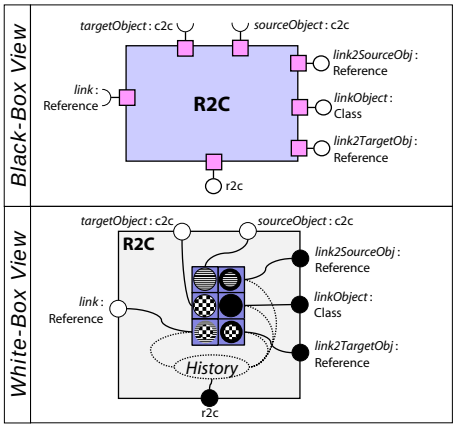

Figure 6.6: R2C Mapping Operator

The white-box view of the R2C mapping operator, as illustrated in Figure 6.6, consists of a pre-condition comprising three query-tokens. The input link is connected to a two-colored query-token, the fromColor corresponds to the query-token standing for the source object and the toColor corresponds to a query-token standing for the target object. The post-condition of the transition introduces a new color and is therefore responsible to generate a new object. Furthermore, two links are produced by the other generator-tokens for linking the newly generated object with the corresponding source and target objects of the LHS.

Example Application. In Figure 6.7, the reference *superClasses* in the LHS metamodel is mapped to the class *Generalization* by an R2C operator. In addition, the references *subClasses* and *superClasses* are selected for establishing an equivalent structure on the RHS as existing on the LHS. For actually determining the *Class* objects which should be connected via *Generalization* objects, the R2C operator has two dependencies to C2C mappings. This example can be seen as a special case, because the reference *superClasses* is a reflexive reference, therefore both requiredContext ports of the R2C operator point to the same C2C operator.

6.2.6 A2R Mapping Operator

Problem. The value-based vs. reference-based relationship heterogeneity shown in Figure 6.1(e) resembles the well-known difference between value-based and reference-based relationships, i.e, corresponding attribute values in two objects can be used to "simulate" links between two objects. Hence, if the attribute values in two objects are equal, a link ought to be established between them.

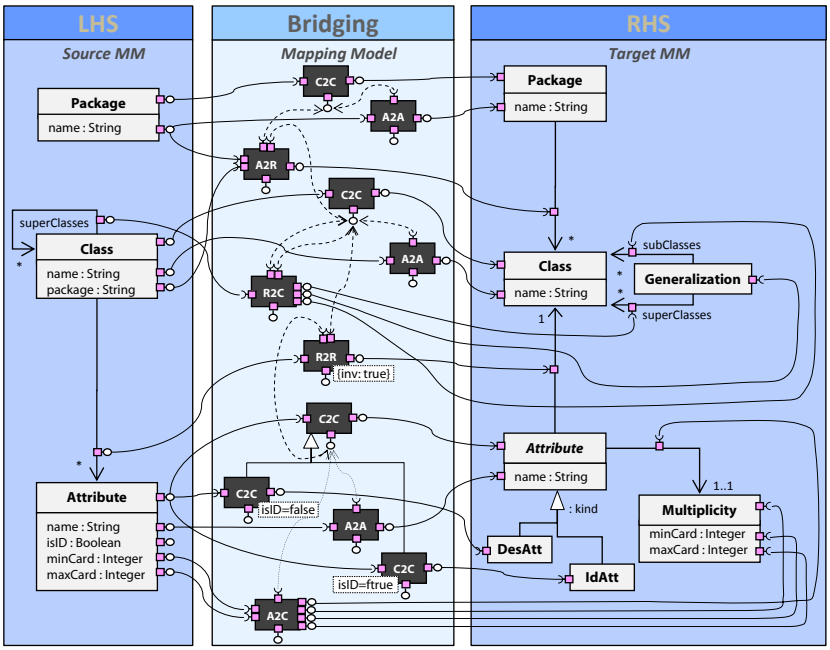

Figure 6.7: Example Resolved with CAR (Mapping View)

Solution. For bridging the value-based vs. reference-based relationship heterogeneity, the A2R mapping operator as shown in Figure 6.9 provides on the LHS two interfaces, one for marking the *keyValue* attribute and another one for marking the *keyRefValue* attribute. On the RHS, the operator provides only one interface for marking the reference which corresponds to the *keyValue/keyRefValue* attribute combination.

The white-box view of the operator comprises a transition which has four query-tokens. The first two ensure that the objects which are referencing each other on the LHS have been already transformed. The last two are the *keyValue* and *keyRefValue* query-tokens whereby the inner-color (representing the attribute values) is the same for both tokens. The generator-token of the transition produces one two-colored token by using the outer-color of the *keyRefValue* query-token as the outer-color and the outer-color of the *keyValue* query-token as the inner-color.

Example Application. In Figure 6.7, the A2R operator is used to map the *Package.name* attribute as the key attribute and the *Class.package* attribute as the keyRef attribute of the LHS metamodel to the reference between *Package* and *Class* on the RHS metamodel.

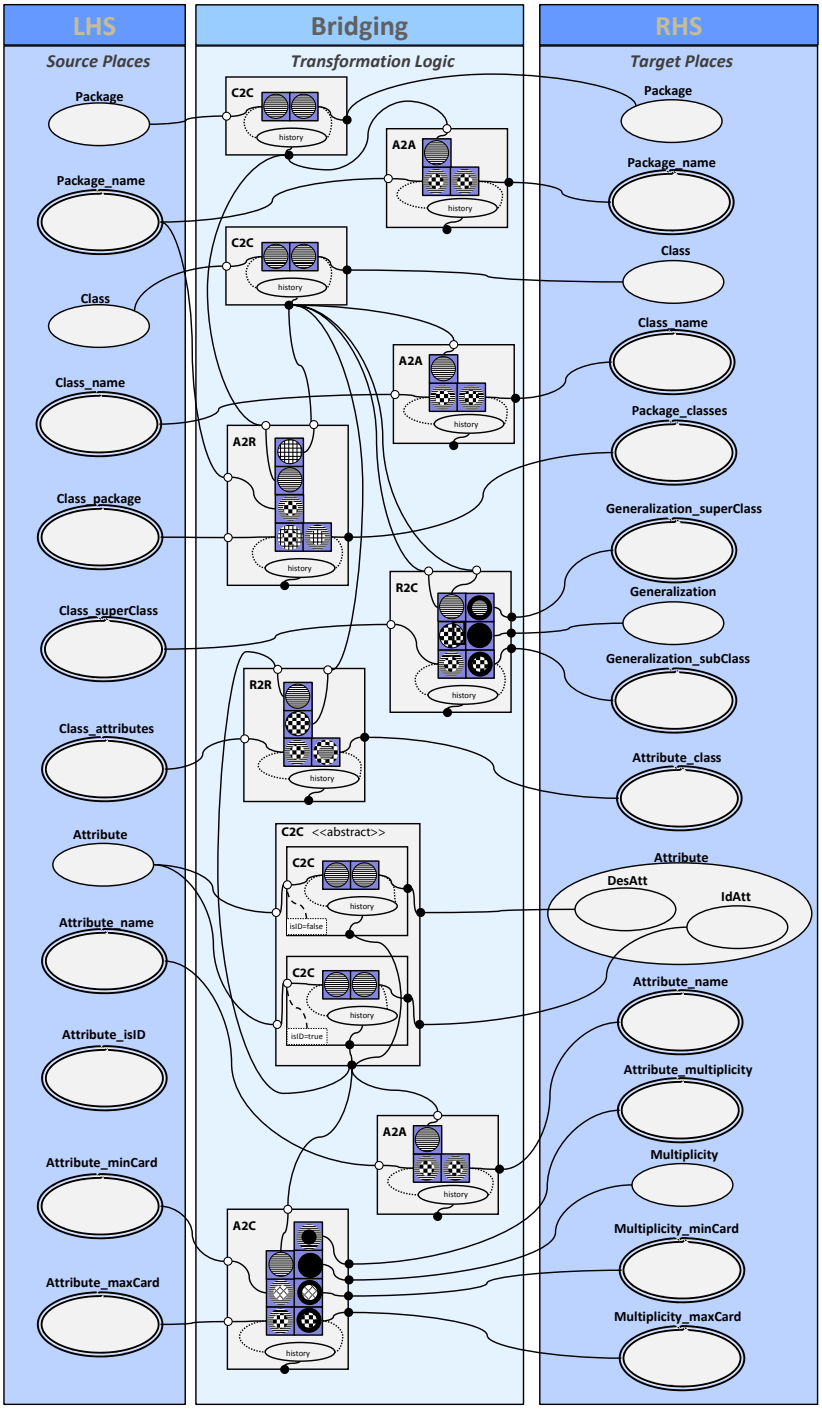

Figure 6.8: Example Resolved with CAR (Transformation View)

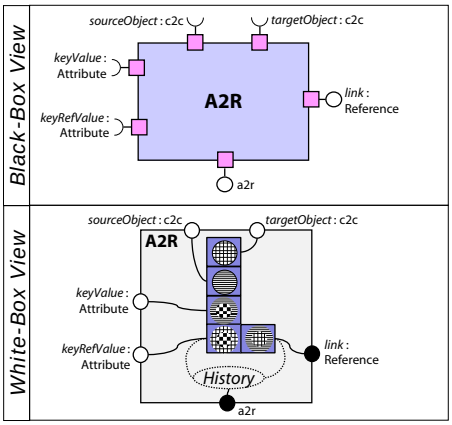

Figure 6.9: A2R Mapping Operator

6.3 An Inheritance Mechanism for Mapping Operators

Why is an inheritance mechanism needed? MOF allows to define class hierarchies in metamodels through generalization. A generalization defines an is-a relationship between a specialized class (*subclass*) and a more general class (*superclass*) where the subclass inherits all features, i.e., attributes and references, of the superclass. Consequently, when one defines mappings between metamodels which heavily use generalizations leading to huge taxonomies, it should be possible to reuse previously defined mappings between general classes for mappings between subclasses.

Required reuse mechanisms. To facilitate the definition of mapping models between MOF-based metamodels, reuse can be achieved in three ways:

- Mappings between features of superclasses can be defined once for the superclasses and can be inherited by mappings between subclasses which can define additional feature mappings between the subclasses. This reuse mechanism can be compared to code reuse through implementation inheritance supported by common object-oriented programming languages.
- In addition to sharing of feature mappings, the following mapping situation frequently occurs when two metamodels have to be integrated. Assume, we have on the LHS a superclass which has a multitude of subclasses. In contrast, on the RHS we only have a single class, i.e., the class hierarchy on the LHS is collapsed into a single class inhibiting all features of the LHS class hierarchy at the RHS, which is equivalent to the LHS superclass and also to its subclasses. To avoid that for each subclass a mapping to the

RHS class must be defined, it should be possible to apply the mapping between the LHS superclass and the RHS class also for indirect instances of the LHS superclass.

- A refinement mechanism is needed to define for certain subclasses specific mappings, which refine the mapping between the superclasses. Of course, the feature mappings between the superclasses should be also applied for such submappings.

6.3.1 Inheritance for C2C Mappings

For reusing existing mappings, we introduce the possibility to define generalization relationships between C2C mappings. This means, the user can define general mappings between superclasses called *supermappings* and more specific mappings between *subclasses* called *submappings* which can be used to refine (i.e., the source and target types of the supermappings) and extend (i.e., new feature mappings can be introduced) the supermappings. As concrete syntax for generalization relationships between C2C mappings, we reuse the notation of UML generalization relationships between classes, i.e., a line with a hollow triangle as an arrowhead.

Concerning the second reuse mechanism, it has to be noted that there are also cases in that the mentioned behavior of applying a supermapping for indirect instances is not desired. Sometimes it is required that only direct instances should be transformed and not indirect instances. Therefore, certain configuration parameters for mappings are required in order to express such integration details in the mapping model.

We allow generalization relationships only for C2C mappings for inheriting feature-mappings which are dependent on C2C mappings such as symmetric mappings (A2A, R2R), or asymmetric mappings (A2C, R2C, A2R, and their inverse operators). This is due to the fact that C2C operators are responsible for providing the context information for all other CAR mapping operators. The introduction of generalization relationships between C2C mappings results in an extension of the C2C operator as shown in Figure 6.10. In this figure, the class *C2C*, representing the C2C mapping operator, is extended for defining generalization relationships by setting the references *supermappings* and *submappings* accordingly. Furthermore, for allowing different kinds of supermappings, i.e., if a mapping is itself executable or if it is applicable for unmapped subclasses, two additional boolean attributes, namely *C2C.isAbstract* and *C2C.isApplicable4SubClasses*, are defined for the *C2C* metaclass. These two attributes enable the user to configure the behavior of the mappings in more detail.

One important constraint for generalization relationships between C2C operators is that if a generalization between two C2C operators is defined, the participating LHS classes of the supermappings and the submappings must be either in a generalization relationship or it must be actually the same class. Of course, the same constraint must hold on the RHS. These two constraints must be ensured, because the submappings inherit the feature mappings of the supermappings and therefore, the features of the superclasses must be also available

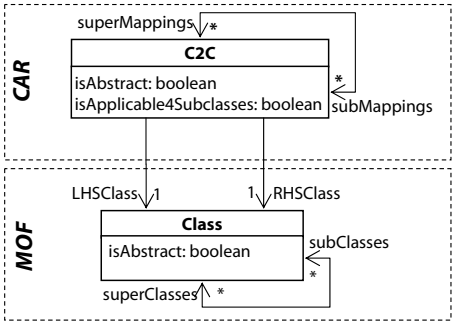

Figure 6.10: Abstract Syntax of C2C Operator Extended with Generalization Relationships

on instances which are transformed according to the submappings. The OCL constraints shown in Listing 6.1 can be used to validate a mapping model with respect to the correct usage of generalization relationships between C2C mappings.

Listing 6.1: Well-formedness Rules for C2C Generalizations

```
context C2C
  inv1: self.superMappings -> forAll(s|s.LHSClass.subClasses ->
          union(s.LHSClass) -> contains(self.LHSClass));
  inv2: self.superMappings -> forAll(s|s.RHSClass.subClasses ->
          union(s.RHSClass) -> contains(self.RHSClass));
```

6.3.2 Symmetric Mapping Situations

For further explanations how to use generalization between C2C operators, we assume at first that the mapping problem is symmetric, i.e., the same generalization structure is available on the LHS and on the RHS, and that only single inheritance is used for defining the metamodels. In particular, we assume that on the LHS and on the RHS a superclass with various subclasses exists. Asymmetric mapping problems, i.e., one side has a taxonomy and the other has not, and integration scenarios where metamodels use multiple inheritance are discussed later in this section.

In general, for symmetric mapping problems we can distinguish between three different cases of mapping models as exemplified in Figure 6.11. Thereby for each case it is assumed that there is at least a mapping between the superclasses of the LHS and the RHS. For explicitly defining these three mapping models, we assume that one can navigate from a superclass to its subclasses via the *subClasses* reference (cf. lower half of Figure 6.10) as well as there exists a helper method *hasMapping()* that checks if a class has been mapped with a C2C operator or not.

Case A. $\forall\ x$ IN *LHSsuperClass.subClasses* | $\neg$ *x.hasMapping(), meaning that only the superclasses are mapped and no single mapping between subclasses exists as shown in Figure 6.11 (a).*

Case B. $\forall\ x$ IN *LHSsuperClass.subClasses* | *x.hasMapping()*, *meaning that all subclasses refine the mapping between the superclasses as shown in Figure 6.11 (b).*

Case C. $\exists\ x, y$ IN *LHSsuperClass.subClasses* | $\neg$ *x.hasMapping()* $\wedge$ *y.hasMapping()*, *meaning that some subclasses exist that do not refine the mapping between the superclasses and some subclasses exist that do not, as shown in Figure 6.11 (c).*

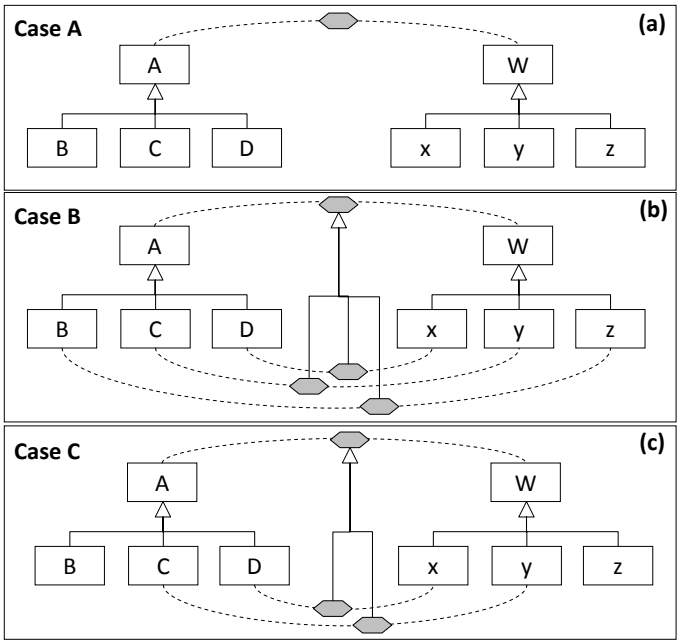

Figure 6.11: Mapping Model Cases for Symmetric Inheritance

For interpreting the C2C mappings as transformations, in addition to the consideration of the various mapping model cases, the meta-properties of classes participating in the supermappings have to be considered. In particular, attention has to be given if a class is an abstract class or a concrete class (cf. Figure 6.10 *Class.isAbstract*) when transformation rules have to be generated in order to ensure that only concrete classes are instantiated in the target model. This means, orthogonal to the different mapping models, we have to distinguish four combinations of superclass on the LHS and on the RHS, namely (1) both are abstract, (2) both are concrete, (3) the LHS superclass is abstract and the RHS superclass is concrete, and finally (4) the LHS superclass is concrete and the the RHS superclass is abstract. When these four combinations are examined for each of the three mapping model cases, we receive twelve mapping situations which may lead to different interpretations of the mapping models.

In the following subsections, we elaborate on *complete*, *incomplete*, and *non-applicable* mappings with respect to configurations of supermappings as well as mapping situations. With configuration we mean the property setting of the supermappings, namely if a supermapping itself is executable (attribute *C2C.isAbstract*) and if a supermapping can be applied on indirect instances (attribute *C2C.isApplicable4SubClasses*). With the term complete mapping, we mean that each queried object on the LHS can be transformed according to the mappings in a RHS object. With incomplete mappings, we mean that at least one object is queried on the LHS by the mappings which cannot be instantiated in the RHS model, and finally, the term none-applicable mapping means that generally no object can be queried on the LHS according to the mappings. For incomplete mapping situations an error message should be provided to the user in order to give feedback how to adjust the mapping model. For none-applicable mappings only a warning should be produced that the mappings must be further refined in order to be effective on the instance layer.

The default configuration of the C2C operator for each mapping situation specifies that the supermapping itself is executable and applicable for indirect instances. In order to give the user more possibilities to explicitly define other interpretations of supermappings, we furthermore allow three non-default supermapping configurations, thereby the first configuration allows to define abstract supermappings with the capability to be applied for indirect instances, and the other two configurations allow reuse of depending mappings of supermappings without applying the supermappings on indirect instances. We discuss the default mapping configuration together with the explicit options for changing this default configuration in detail and present a summary of this discussion in Table 6.1.

Default Mapping Configuration. The first variant of supermapping configurations is that the supermapping itself is executable (*C2C.isAbstract=FALSE*) and applicable to transform instances of subclasses having no refined mappings (*C2C.isApplicable4SubClasses=TRUE*). We decided to use this variant as the default configuration of C2C mappings, because it is the most natural interpretation to transform direct and indirect instances – for which no refined mappings are available – if a mapping between two superclasses is defined. However, when we take a closer look on this configuration, we can find mapping situations which have no correct interpretation or the mappings are not even applicable on one single instance. The combination of the mapping model cases and the superclass combinations lead to twelve mapping situations which are discussed in the following with respect to consequences on the generated transformation logic, i.e., which instances can be actually transformed from the LHS to the RHS, in order to identify if the mapping situation is complete, incomplete or none-applicable. The *Config 1* block of Table 6.1 summarizes the discussed mapping situations and illustrates which of them produce complete, incomplete, and none-applicable transformations. One can clearly see that mapping situations where the RHS class is not a concrete class lead to problems when not all subclasses have refined

mappings.

Case A: Only superclasses are mapped.

- A1: Indirect instances of the LHS superclass cannot be transformed, because the RHS superclass cannot be instantiated and it is not possible to determine which subclass should be used instead. Therefore, an error message has to be given to the user that she should refine the supermapping with appropriate submappings for each subclass.
- A2: Each LHS instance (direct and indirect) can be transformed into an instance of type RHS superclass.
- A3: Same as A2, however, only indirect instances have to be transformed.
- A4: Same problem as in A1. Feedback to the user has to be given for such mapping configurations.

Case B: All subclasses refine the mapping between the superclasses.

- B1: Each indirect instance can be transformed into an instance of a RHS subclass according to the refined submapping.
- B2: Same as B1, however, additionally direct instances of the LHS superclass are transformed into instances of the RHS superclass.
- B3: Same as B1, however, only indirect instances have to be transformed.
- B4: Indirect instances can be transformed according to the submappings. However, direct instances of the LHS superclass cannot. Again, an error message has to be given to the user.

Case C: Some, but not all subclasses refine the supermapping.

- C1: Instances of mapped subclasses can be transformed, however, the rest of the instances cannot. This means, the supermapping has to be refined for each subclass.
- C2: All instances can be transformed. Instances of mapped subclasses can be instantiated from specific subclasses, instances of unmapped subclasses are instantiated directly from the RHS superclass.
- C3: Same as C2, however, only indirect instances have to be transformed.
- C4: Same as C1. Furthermore, direct instances of the LHS superclass cannot be transformed, too.

Non-default Mapping Configurations. In the following, the differences of non-default mapping configurations with respect to the default mapping configuration are discussed which are visualized in Table 6.1 by shaded cells.

The second configuration variant, summarized in the *Config 2* block of Table 6.1, is that the supermapping is abstract, but should be applied on indirect instances for which no specific mappings are available. For this configuration only the situation 4 of case B differs from the default configuration. With configuration 2, this mapping situation is complete, because only indirect instances of the LHS superclass are transformed, the direct instances are not queried because of the abstract supermapping.

Table 6.1: Overview on Mapping Situations with Respect to Complete, Incomplete, and Non-Applicable Configurations

		SuperClass.isAbstract		Case A	Case B	Case C
		LHS	RHS	∀x **IN** subClasses | ¬ x.hasMapping()	∀x **IN** subClasses | x.hasMapping()	∃ x, y **IN** subClasses | ¬ x.hasMapping()∧ y.hasMapping()
Config 1 *isAbstract = F* *isAppl4SubCl = T*	(1)	*T*	*T*	✗	✓	✗
	(2)	*F*	*F*	✓	✓	✓
	(3)	*T*	*F*	✓	✓	✓
	(4)	*F*	*T*	✗	✗	✗
Config 2 *isAbstract = T* *isAppl4SubCl = T*	(1)	*T*	*T*	✗	✓	✗
	(2)	*F*	*F*	✓	✓	✓
	(3)	*T*	*F*	✓	✓	✓
	(4)	*F*	*T*	✗	✓	✗
Config 3 *isAbstract = F* *isAppl4SubCl = F*	(1)	*T*	*T*	~	✓	✓
	(2)	*F*	*F*	✓	✓	✓
	(3)	*T*	*F*	~	✓	✓
	(4)	*F*	*T*	✗	✗	✗
Config 4 *isAbstract = T* *isAppl4SubCl = F*	(1)	*T*	*T*	~	✓	✓
	(2)	*F*	*F*	~	✓	✓
	(3)	*T*	*F*	~	✓	✓
	(4)	*F*	*T*	~	✓	✓

Legend:

✓	complete
✗	incomplete
~	none-applicable

The third configuration variant, summarized in the *Config 3* block of Table 6.1, is about using the supermapping only for reusing feature-mappings and for transforming direct instances. For case A1 and case A3 we get a different interpretation as for the default configuration, because the supermapping is not used for indirect instances and no direct instances exist for the LHS superclass, thus the supermapping is not applicable at all and need to be refined with concrete submappings. Furthermore, for this kind of mapping configuration,

the case C1 is complete, because the supermapping is not applied on indirect instances.

The fourth configuration variant, summarized in the *Config 4* block of Table 6.1, is about using the supermapping only for reusing feature-mappings. This means, the supermapping is not used for transforming direct and indirect instances. Thus, Case A directly leads to none-applicable mappings, which have to be refined by concrete submappings. Furthermore, there are no problematic cases when submappings exist, because only the submappings are executed which have their own refined target classes and are therefore not dependent on properties of the RHS classes of their supermappings.

6.3.3 Representing Inheritance within Transformation Nets

In this subsection we discuss how C2C generalization relationships influence the generation of transformation nets and consequently the execution of the transformation logic. On overall design goal is naturally to express new language concepts at the black-box view – such as mapping generalizations in this case – as far as possible by means of existing transformation net mechanisms.

Basic Idea. When we take a closer look on supermappings with a standard configuration, we see that these mappings must provide the context, i.e., the trace model information, for all dependent mappings. This means, the supermappings must also provide context information about the transformation of indirect instances, e.g., for assigning attribute values of indirect instances when the attribute is contained by the superclass. Consequently, a supermapping is derived into a transformation component which contains the union of its own trace model for logging the transformation of direct instances of the superclass and the trace models of its submappings for logging the transformation of indirect instances. Therefore, the corresponding transformation components of the submappings are nested into the transformation component of the supermapping. For constructing the union of trace models of nested transformation components, each nested component gets an arc from its own trace model to the union trace model of the outer component. Mappings which depend on the supermapping are connected to the union trace model available on the outer component and mappings which are dependent on submappings are directly connected to the individual trace models of the nested components.

Figure 6.12 illustrates the derivation of generalization relationships into transformation net components. For describing the basic mapping rule how generalization relationships are represented in transformation nets, it is assumed that all mappings are concrete mappings and it is not considered if a mapping is applicable for subclasses or not. The mapping $C2C_1$ of the Mapping Model shown on the LHS of Figure 6.12 is transformed into the outer component $C2C_1$, which consists of a transition for transforming direct instances and of two subcomponents $C2C_{2.1}$ and $C2C_{2.2}$. In addition, the outer component provides a union trace model of the transformation components $C2C_1$, $C2C_{2.1}$, and $C2C_{2.2}$. Because the map-

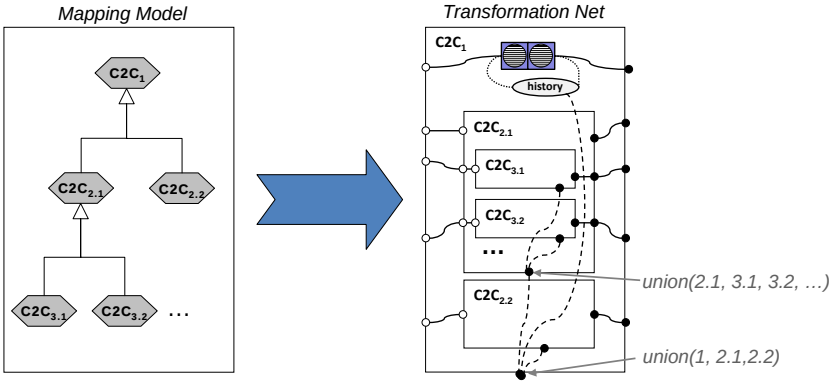

Figure 6.12: Representing Inheritance Structures with Nested Transformation Components

ping $C2C_{2.1}$ has two submappings, the corresponding transformation component has also two sub-components $C2C_{3.1}$ and $C2C_{3.2}$. In addition, the component $C2C_{2.1}$ provides a union trace model of itself and the subcomponents $C2C_{3.1}$ and $C2C_{3.2}$

Mapping Rules for Supermapping Configurations. In addition to the derivation of inheritance structures to nested transformation components, specific derivation rules for the configuration variants of the supermappings are needed to represent *abstract* and *concrete* mappings in transformation nets as well as the *applicability* of supermappings for subclasses. In particular, the following four rules, which are summarized in Figure 6.13, are sufficient to generate transformation nets for all possible supermapping configurations. The mapping model shown in Figure 6.13 is used as an example input mapping model for describing the mapping rules and comprises a mapping between the superclasses *C1* and *C1'* of the LHS and RHS metamodels and between the subclasses *C2* and *C2'*, whereby the subclass *C3* of the LHS remains unmapped.

- Rule 1 - Concrete/Applicable Supermapping: When a supermapping is concrete, a transition is available in the outer transformation component for transforming direct instances of the superclass and indirect instances for which no specific mappings are available. Because only direct and indirect instances of subclasses without specific mappings should be transformed by the transition of the outer component, an OCL condition is attached on the inputPort which leads to the transition in order to reject tokens for which more specific mappings are available. Such constraints can be defined with the OCL function *oclIsTypeOf* which gets as parameters the superclass and all subclasses for which no specific mappings have been defined in the mapping model (cf. OCL condition *oclIsTypeOf(C1|C3)*). If there is a more specific mapping between subclasses, a nested component is produced and the tokens are not streamed via the superclass mapping, instead the subPlace generated from the LHS subclass

gets an additional arc which leads to a more specific transformation component.

- Rule 2 - Abstract/Applicable Supermapping: Although the supermapping is abstract, a transition resides directly in the outer component, which is not applicable for direct instances but for transforming all indirect instances for which no specific mapping has been applied (cf. OCL condition *oclIsTypeOf(C3)*).
- Rule 3 - Concrete/Non-Applicable Supermapping: If a supermapping is defined as concrete and non-applicable for unmapped subclasses then an outer component is produced which consists of a transition for transforming direct instances of the superclass (cf. OCL condition *oclIsTypeOf(C1)*).
- Rule 4 - Abstract/Non-Applicable Supermapping: When a supermapping is abstract and non-applicable for unmapped subclasses only the outer component is generated for providing a union trace model for its submappings.

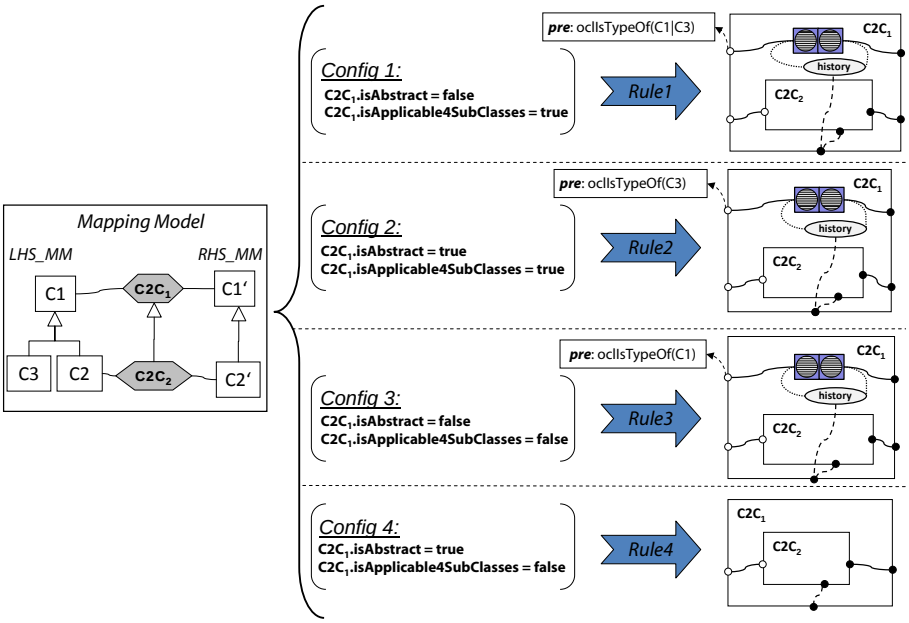

Figure 6.13: Representing C2C Configurations in Transformation Components

The following three design alternatives exist for transformation nets to model the applicability of the supermapping transition on subPlaces. First, we could extend the place modeling constructs with tags such as "superTransition is applicable". However, the introduction of such a transformation net feature would violate our design goal that the transformation net formalism should not be changed. The second possibility is to generate for

each unmapped class an additional arc from the corresponding source place to the outer component generated for the supermapping. This variant would lead to much more complicated transformation nets and to many duplicated arcs, which simply does not pay off the information gain for the user. Therefore, we decided for a third variant, the usage of OCL constraints as explained for Rule 1 to 3.

Example. To summarize this subsection, a concrete integration example, as shown in Figure 6.14, is discussed on the mapping view and on the transformation view. In the LHS metamodel, a class *Person* is specialized into *Supplier*, *Employee*, and *Costumer* classes. The RHS metamodel consists also of a superclass *Person*, and of *Client*, *Staff*, and *ShareHolder* subclasses. Each LHS class can be mapped to a RHS class, except the class *Supplier*. Hence, the LHS class *Person* is mapped with a C2C mapping operator to the RHS class *Person*. The properties of this C2C are set to *isAbstract=FALSE* and *Applicable4subclasses=TRUE*. Consequently, each instance of the LHS class *Person* is transformed into an instance of the RHS class *Person*, as well as each instance of a subclass which has no further refinement mapping is also transformed into an instance of the RHS *Person* class. For example, each instance of the class *Supplier* becomes an instance of the class *Person*. Additionally, the name attribute of the LHS class *Person* is mapped by an A2A mapping operator to the name attribute of the RHS class *Person*.

The subclasses *Employee* and *Customer* of the class *Person* on the LHS are mapped by C2C mappings to *Staff* and *Client* of the RHS, respectively. Additionally, the attributes of these classes, namely *Customer.cuNr*, *Employee.emNr*, and *Client.clNr*, *Staff.stNr*, are mapped by A2A mappings, respectively. Due to the fact that each of the subclasses inherit the attributes of the superclass – the attribute *Person.name* – the A2A mapping between the superclasses is also inherited by the C2C mappings by setting the *superMappings* reference to the C2C mapping which resides between the *Person* classes.

The corresponding transformation net for the presented mapping model is depicted in the *Transformation View* of Figure 6.14. The *Person* classes become places which comprise for each subclass an inner place. As subclass places are nested in superclass places, the inheriting submappings are nested in the transformation component which correspond to the supermappings. The outer transformation component, corresponding to the supermapping, contains a transition, because the *isAbstract* property of the C2C mapping is set to *FALSE*. Furthermore, due to the *isApplicable4subclasses* property of the C2C mapping, which is set to TRUE, the outer transformation component of the transformation net receives an additional OCL constraint, namely *oclTypeOf(Person | Supplier)*. Due to readability purposes, we refrain from displaying these features in Figure 6.14. Consequently, each direct instance of type *Person* from the LHS is transformed into an instance of class *Person* on the RHS. Furthermore, this OCL constraint ensures that each instance of subclasses of the class *Person*, which have no refined mapping, is also transformed by the supermapping into an instance of type *Person* on the RHS.

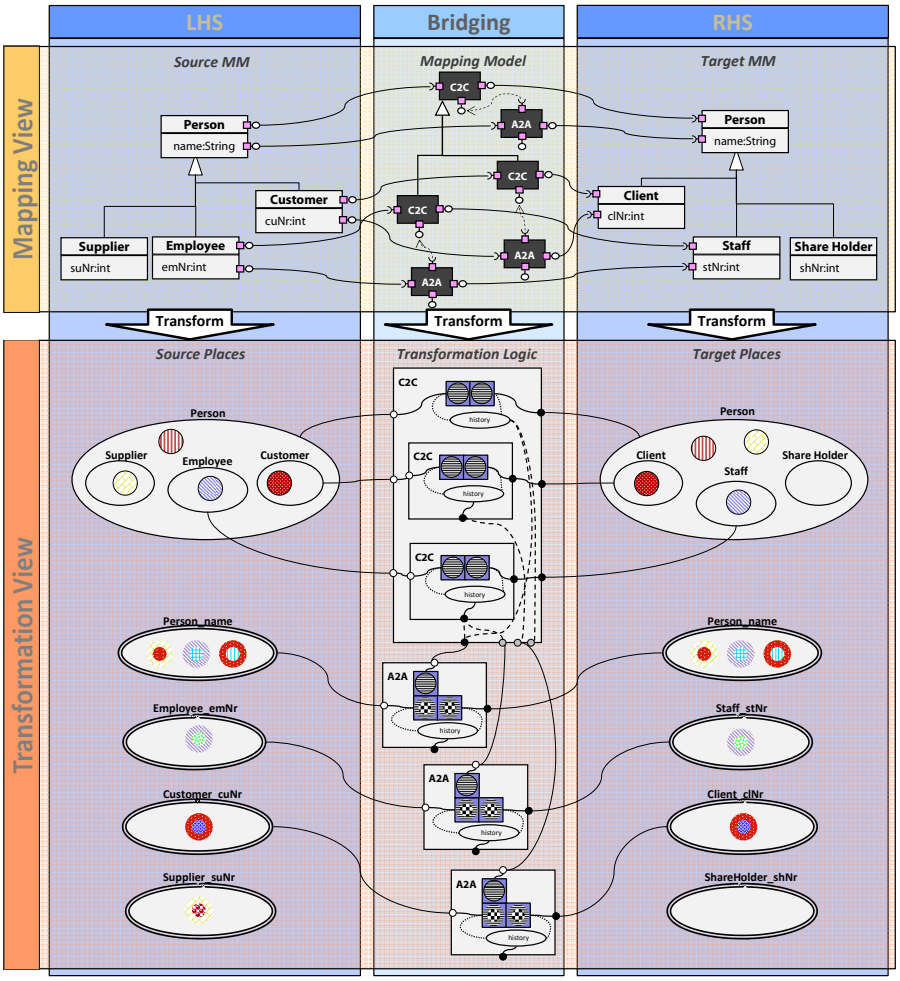

Figure 6.14: Inheritance between C2C Mappings – Symmetric Example

The attribute *Person.name* can be transformed only if the containing instance which can be of type *Person*, *Employee*, or *Customer* has been already transformed. Consequently, the A2A transformation component for name values must be in the context of three C2C transformation components. This is achieved by the trace model provided by the black port in the middle of the bottom of the outer C2C transformation component. This trace model unifies the individual trace models of the C2C transformation components. The other A2A operators are connected to the gray ports which link directly to individual trace models of the nested components.

6.3.4 Asymmetric Mapping Situation – Hierarchy vs. Collapsed Hierarchy

In the previous subsections we only elaborated on the symmetric integration scenario, where the hierarchical structures of the metamodels are fully corresponding to each other. In practice, however, the inheritance structures between metamodels can vary considerably. The worst case of asymmetric mapping situations is that on one side an inheritance hierarchy is applied to structure the metamodel and on the other side no inheritance relationships are used at all. For example, Figure 6.15 illustrates a mapping example which comprises a typical refactoring pattern from object-oriented programming [Fow99], namely the way how an inheritance hierarchy can be flattened. Thus, when two metamodels have to be bridged, it is possible that one metamodel uses an inheritance structure and the other metamodel does not. In particular, this means, in one metamodel an additional attribute is used to split a set of objects into various subsets, whereas on the other side these subsets are explicitly expressed through subclasses.

When one wants to bridge two metamodels where one metamodel has a generalization hierarchy and the other one has a collapsed hierarchy, generalization between C2C mappings should be used. More specifically, as introduced in Subsection 6.3.1, an OCL constraint allows to use generalization between C2C operators also for cases where the participating classes of the supermapping and the submapping are actually the same. Exactly this special case is exploited for defining supermappings for asymmetric cases as is explained in the following.

The following C2C mappings are needed for resolving the asymmetric bridging case as shown in Figure 6.15. First, a mapping from the single class on the LHS to the superclass on the RHS is defined which comprises all possible feature mappings. Second, for each subclass of the RHS a mapping is defined to the class on the LHS. The mappings inherit from the previously defined mapping. In addition, each submapping gets an OCL condition which selects from all instances of the LHS class a certain subset of instances which matches to the subclass, i.e., which fulfill the OCL condition. All submappings must have mutually exclusive conditions in order not to instantiate duplicates on the RHS side.

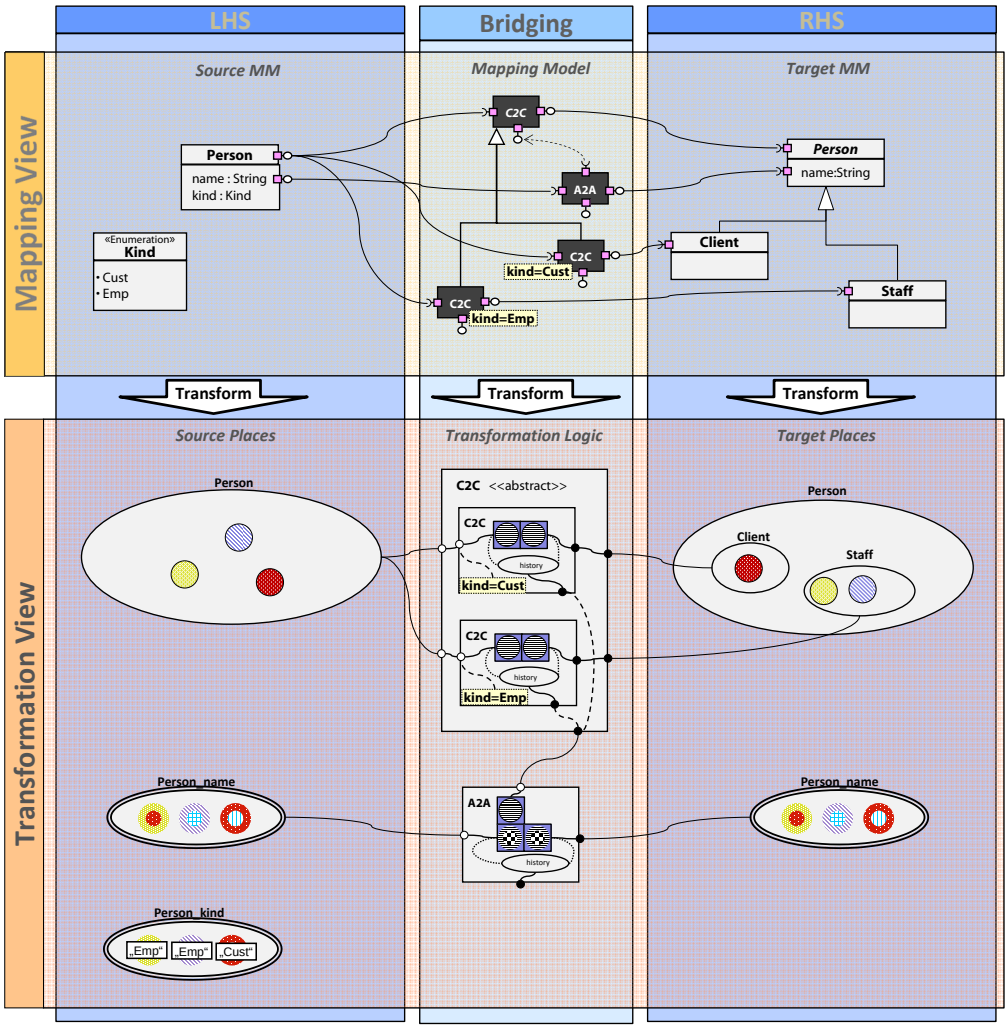

Figure 6.15: Inheritance between C2C Mappings – Asymmetric Example

6.3.5 Multiple Inheritance for C2C Mappings

MOF allows not only single inheritance between classes, but also multiple inheritance. This means in metamodels, classes can collect features from more than one superclass. In this subsection, we discuss the consequences of multiple inheritance provided by MOF for our proposed C2C generalization mechanism based on two examples. The first example covers the case that both metamodels use multiple inheritance (*symmetric case*) and the second example covers the case that one metamodel uses multiple inheritance and the other metamodel makes use of single inheritance and duplication of attributes for avoiding multiple inheritance (*asymmetric case*).

Before we continue with discussing the consequences of multiple inheritance in metamodels for our proposed mapping language, we first describe how multiple inheritance is expressed with the *Place* concept of the transformation net formalism. We decided for our approach to represent multiple inheritance in transformation nets as intersecting places, thereby the intersection of places itself represents a place, which can be used for connecting arcs to transitions.

In Figure 6.16, an example metamodel is presented which makes use of multiple inheritance. In particular, the class *CustomerEmployee* is the subclass of *Customer* and *Employee* and consequently inherits the features of both superclasses. When we take a look on the corresponding transformation net representation, we see that the places for *Customer* and *Employee* have an intersecting part, which contains the place for *CustomerEmployee* tokens. We decided to use this representation, because when looking at the inheritance structures from an *extensional viewpoint* (the extension of a class is the set of all its instances), we can define the concepts of Figure 6.16 as the following instance sets (the numbers of the instance sets can be found as annotations in Figure 6.16).

- Extension(Person) = {1,2,3,4}
- Extension(Customer) = {2, 4}
- Extension(Employee) = {3, 4}
- Extension(CustomerEmployee) = {4}

The extensions of the classes *Customer* and *Employee* have an intersecting part (cf. instance set 4) which can be appropriately expressed with intersecting places in the transformation net formalism.

6.3.5.1 Symmetric Mapping Situation

For reusing feature mappings from more than one supermapping, we offer in our mapping models the possibility to define multiple inheritance for C2C operators as is shown in Figure

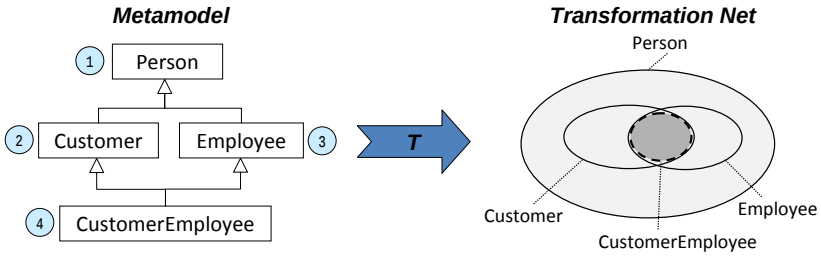

Figure 6.16: Representing Multiple Inheritance with Intersecting Places

6.10 where the upper cardinality of the reference *superMapping* is not restricted. The design rationale for this extension is that if a class has more than one superclass it makes sense to reuse the feature mappings from all superclasses and not only from one. Furthermore, this means that multiple inheritance between mapping operators should only be used for such cases where multiple inheritance structures exist also in the metamodel.

In our approach, multiple inheritance for C2C mappings can be seen as N single inheritance relationships, at least on the mapping layer. This means, we can use the individual inheritance relationships for reusing feature mappings, for applying the supermappings also for indirect instances, and the target classes of supermappinges can be refined within a submapping.

Representing Multiple Inheritance between C2C operators in Transformation Nets. Considering the representation of multiple inheritance by means of transformation nets, as for single inheritance, the supermappings must provide the union trace models of the submappings. However, the fact that in case of multiple inheritance, a submapping has more than one supermapping, aggravates the mapping of the C2C mappings into transformation net components. The main driver for the complication is that, on the one hand, we have more than one transformation component in the transformation net because we can have more than one supermapping and, on the other hand, only one token exists for an instance which is actually an indirect instance of more than one superclass. This means, it is not possible to simply nest the submapping into the supermappings as it was done before to get a fully working transformation net.

In order to adapt the mapping of generalizations of C2C mappings for supporting also multiple inheritance, there are several possibilities how the mapping model can be represented in transformation nets in order to realize the intended transformation behavior. In the following, we discuss three alternatives together with their advantages and disadvantages.

- *Updating trace models from the outside*: The first alternative would be that the trace models of the supermapping components are updated from the outside, i.e., from other

components. This would allow that the submapping component resides outside of the supermapping components, but requires that for each streamed token, the trace model of the supermapping is updated by the submapping. However, this solution should be avoided, because an update of the trace models should only be possible within a component and not from outside in order to have a clear separation between the black-box and white-box view.

- *Nesting supermappings as well as submappings into one component*: The second alternative is very similar to the nesting of submappings into supermappings, however, now the supermappings and submappings are nested into one component on the same level. This allows that the individual trace models of the nested components can be unified into one common trace model which is available on the outside of the outer component. However, this solution is in contradiction with the previous mapping rules for single inheritance where submappings are nested into supermappings.
- *Duplicating tokens for the transformation*: For using the same mapping rules for multiple inheritance as for single inheritance, the tokens of subclasses which inherit from more than one superclass have to be duplicated. Furthermore, this allows to duplicate also the submapping component and nest them into the supermapping components. In particular, for each supermapping component, a duplicated token must be available in order to be streamed through the component of the supermapping. This is required, because the trace model of the supermappings must also provide information about the indirect instances. After streaming each duplicated token through the supermapping components, the tokens have to be merged again into one single token which can then be stored in the target place.

We have decided to use the third alternative, because it allows us to use the same derivation rules for producing the transformation nets as for single inheritance by nesting the submapping components into the supermapping components, plus using the additional duplicator and merge components in the source and target places.

Example. To summarize the discussed mapping rules, the integration example illustrated in Figure 6.17 shows a symmetric integration scenario for multiple inheritance in metamodels. Each class of the LHS can be mapped to a class of the RHS. The mapping between the *CustomerEmployee* classes inherits from the mappings between the superclasses which furthermore provide the context for two A2A mappings. In order to apply these two A2A mappings for the submapping, the transformation net is built as follows. First, two transformation components are generated for the supermappings. Second, each component comprises a nested component for transforming instances of the class *CustomerEmployee*. Third, in order that both nested components can transform all *CustomerEmployee* instances, a duplicator component is interposed between the *CustomerEmployee* place and the nested components. By duplicating the tokens, it can be ensured that both components can provide the

union trace models, i.e., for their direct instances and for their indirect instances. Before the tokens representing *CustomerEmployee* instances can be put into the target place, a merge component must ensure that the duplicated tokens are again merged into one single token. With the help of the duplicate and merge workaround, it can be ensured that both components can provide complete trace models, which are needed for the depending components, in this case the A2A operator in order to transform *cuNr* values and *emNr* values also for *CustomerEmployee* instances.

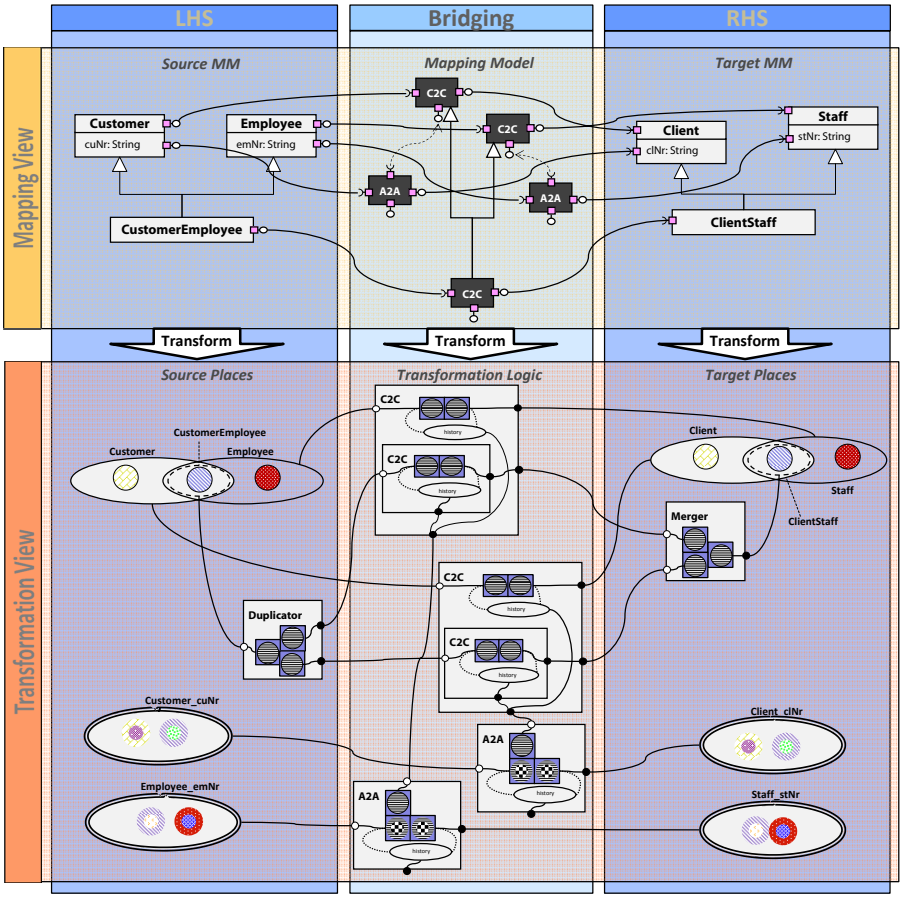

Figure 6.17: Multiple Inheritance between C2C Mappings – Symmetric Example

6.3.5.2 Unmapped Subclasses

Until now, it has been discussed how multiple inheritance influences the transformation logic for submappings, only. Now, we want to discuss what happens if no submappings are

defined for subclasses which inherit from more than one superclass. In particular, in cases where a class inherits from multiple classes, which have been mapped with C2C mappings applicable for unmapped subclasses, it is not clear how the instances of the subclass should be actually transformed.

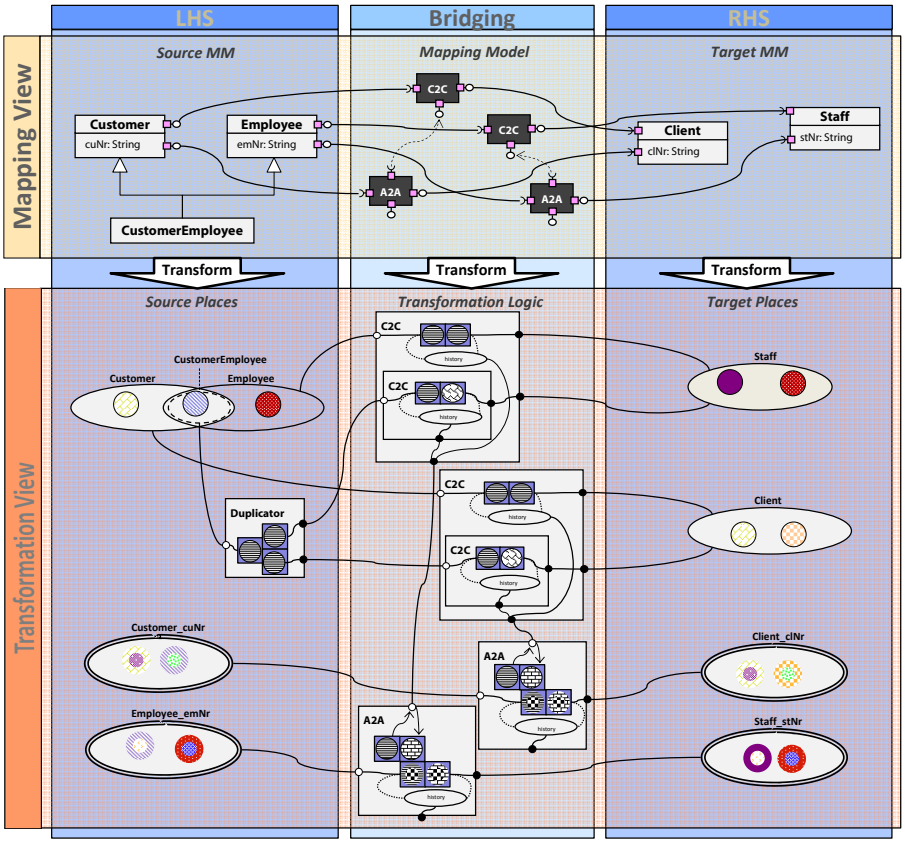

Figure 6.18: Unmapped Subclasses and Multiple Inheritance – Example

In the example shown in Figure 6.18, the classes *Customer* and *Employee* of the LHS are mapped to their corresponding classes on the RHS. However, for the class *CustomerEmployee* of the LHS, no mapping can be defined, because this concept is not available on the RHS. For deciding if and how instances of *CustomerEmployee* are transformed, one has to look at the properties of the C2C mappings between the superclasses. When both are not applicable for subclasses, instances of *CustomerEmployee* are simply ignored in the transformation. In cases where only one C2C mapping is applicable for subclasses, then the same transformation component is generated as presented before for single inheritance and no conflicts arise in the transformation. However, in cases where both C2C mappings are ap-

plicable for subclasses, the problem arises that for an instance of the unmapped subclass only one corresponding token exists, but two transformation components are available for such instances. As mentioned before, for solving this problem, a duplicator component is introduced as discussed before, but additionally, a second problem exists. On the RHS, the duplicated tokens cannot be merged into one single token, because two instances should be available in the RHS, one residing in the *Customer* place and one residing in the *Employee* place. This requires that two different tokens are generated out of one single token. Unfortunately, with the streamer components, this is not possible. Therefore, we need a modified C2C component for transforming instances of unmapped subclasses which are capable of generating a new token color. The extension of the streamer C2C component into a so-called *ColorChanger* is shown in the upper part of Figure 6.19. The generation of a new color can be easily defined, only a new color pattern has to be used in the generation part of the transition.

The modification of the C2C mapping operator also influences the trace model, i.e., now the input and output colors are no longer the same as the normal streamer mapping operator guarantees. Consequently, also depending feature mappings, such as the A2A mappings shown in Figure 6.18, have to be modified, because it is no longer sufficient to stream only the two-colored token from left to right. Instead, the outer color of the two-colored tokens contained by instances of unmapped subclasses have to be changed to the newly generated color for the containing instance. Therefore, the A2A components also need some modification concerning the querying of the trace model provided by the C2C operator, in particular for the ColorChanger C2C components. More specifically, the trace model of the ColorChanger C2C has not only to be asked: *Has this instance been already transformed?* Instead, the question should be: *Has the instance been transformed, and if yes, into which element?* This means, the color can change in the C2C operator, and therefore, an extended query-token concept is needed in the A2A mapping operator. In particular, the incoming outer-value must be send to the C2C trace model, and the C2C trace model must send back the color of the generated element. Until now, the input tokens had the same colors as the output tokens, i.e., only streamer C2C mappings has been used as context for feature mappings.

For using other mapping operators which produce also new colors for output tokens, we extend the query part of the A2A mapping operator as shown in the lower part of Figure 6.19. The query tokens for querying the trace models now consist of two color patterns. First, a token (shown on the upper left side of the transition) is sent to the trace model indicated by the arrow pointing to the required trace port of the A2A operator. In addition, a second token (shown in the upper middle of the transition) displays a color pattern which represents the answer (it is assumed that the input element has been already transformed) of the trace model indicated by an arrow pointing from the required trace port to the token. This additional token is required, because an additional color for the generation part is needed, in order to be able to access the probably newly generated token color for the con-

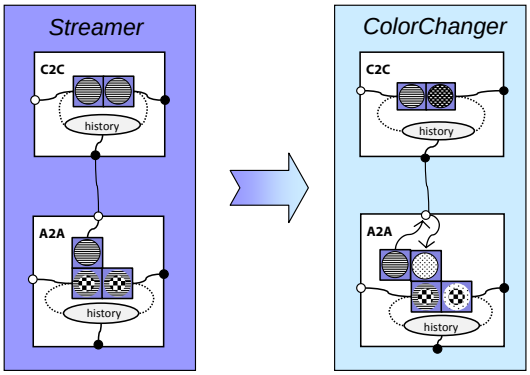

Figure 6.19: From Streamer to ColorChanger

taining object. In the generation part, the token gets as outer color the color pattern queried from the trace model of the C2C operator assigned and the inner color is the same value as for the input token.

6.3.5.3 Asymmetric Mapping Situation – Multiple Inheritance vs. Single Inheritance

So far we have assumed that both metamodels make use of multiple inheritance in the same way. However, the use of multiple inheritance can be easily avoided by the duplication of features which allows one to use single inheritance solely. Therefore, in this subsection we discuss how to deal with integration scenarios where on one side single inheritance is used and on the other side multiple inheritance.

Example. Figure 6.20 shows such an integration example where the metamodel on the LHS uses multiple inheritance and the metamodel on the RHS uses single inheritance, only. Nevertheless, both metamodels express the same information, namely that a set of persons is divided into customers and employees, thereby some employees are also customers. The LHS metamodel expresses the fact that employees can be also customers by defining a class *CustomerEmployee* which is a subclass of the classes *Customer* and *Employee*. In contrast, the RHS metamodel expresses this aspect by separating first customers and employees which are not customers from each other (cf. classes *Customer* and *EmployeeNoCustomer* in Figure 6.20). Subsequently, customers which are also employees are modeled by the class *CustomerEmployee* which is a subclass of the class *Customer*, only. The avoidance of extensionally overlapping classes lead directly to redundancies on the intensional level, namely, as one can immediately see, the attribute *emNr* has to be defined in several classes, namely for *EmployeeNoCustomer* and also for *CustomerEmployee*.

This two different modeling styles regarding the usage of inheritance lead to one main heterogeneity problem in this example, namely dividing the set of *CustomerEmployee.emNr*

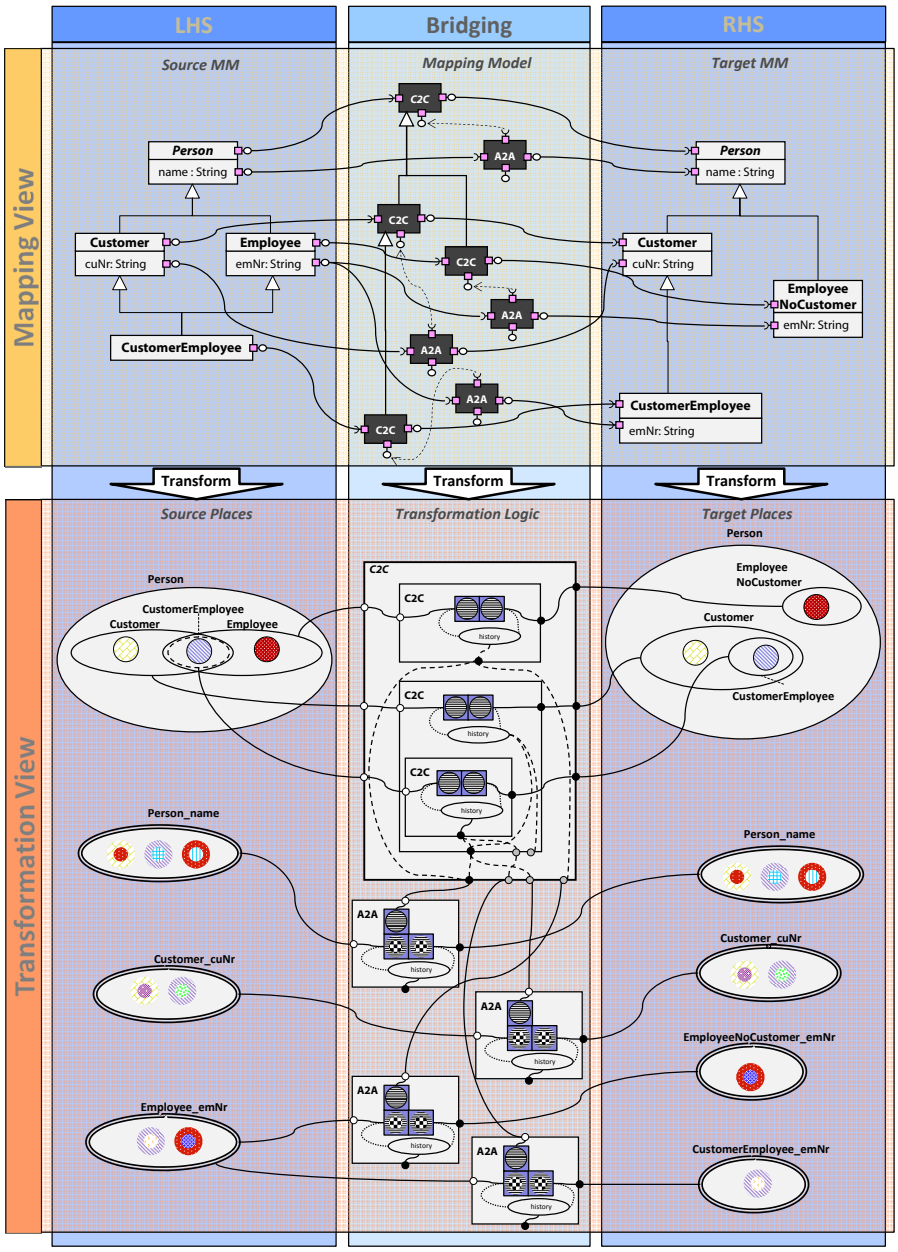

Figure 6.20: Multiple Inheritance between C2C Mappings – Asymmetric Example

values on the LHS into the subsets *EmployeeNoCustomer.emNr* and *CustomerEmployee.emNr* on the RHS. For overcoming this heterogeneity problem, the mapping model is built as illustrated in Figure 6.20. The tricky part of this integration example is how to map the classes *Employee* and *CustomerEmployee* of the LHS to the classes *EmployeeNoCustomer* and *CustomerEmployee* of the RHS as well as how to define the inheritance relationships between the mappings.

Mapping View. First, a mapping is needed for transforming direct instances of the class *Employee*, i.e., employees which are not simultaneously customers. This can be achieved by mapping the *Employee* class of the LHS to the *EmployeeNoCustomer* class on the RHS. However, to ensure that only direct instances are transformed via this mapping, the subclass *CustomerEmployee* needs a specific mapping, because instances of *CustomerEmployee* on the LHS should be transformed into *CustomerEmployee* instances on the RHS and of course not into *EmployeeNoCustomer* instances. Therefore, the *CustomerEmployee* class on the LHS holds its own C2C mapping to the *CustomerEmployee* class on the RHS. Now the question arises from which C2C mappings should the last mentioned mapping inherit. To answer this question, one has to take a closer look at the inheritance structures in the LHS metamodel and in the RHS metamodel. It can be seen that for the mapping, it is only allowed to inherit from the mapping between the *Customer* classes, because an inheritance relationship between *Customer* and *CustomerEmployee* is available in the LHS metamodel and in the RHS metamodel. A specialization of the mapping between *Employee* and *EmployeeNoCustomer* is not allowed, because an inheritance relationship is missing and simply not meaningful between *EmployeeNoCustomer* and *EmployeeCustomer* in the RHS metamodel. With this set of C2C mappings and the usage of one generalization relationship it is guaranteed that instances of *Customer* at the LHS are transformed into *Customer* instances at the RHS, direct instances of the class *Employee* into instances of class *EmployeeNoCustomer*, and instances of *CustomerEmployees* into *CustomerEmployees* instances.

Through the generalization relationship between the mapping *Customer2Customer* and the mapping *CustomerEmployee2CustomerEmployee* it is achieved that also for customerEmployees the values of the *custNr* attribute are transformed, because this featureMapping is inherited from the supermapping. What is missing is the transformation of *empNr* attribute values. Here comes the heterogeneity of different inheritance structures into play. The single set of *Employee.empNr* values on the LHS has to be splitted into two sets on the RHS, namely *EmployeeNoCustomer.empNr* and *CustomerEmployee.empNr*. This can be expressed in the mapping model by using two A2A mappings for the *Employee.empNr* attribute on the LHS. One A2A mapping is applied from *Employee.empNr* to *EmployeeNoCustomer.empNr* and is therefore in the context of the *Employee2EmployeeNoCustomer* mapping. The second A2A mapping is also applied to map *Employee.empNr* to *CustomerEmployee.empNr*, but this time the mapping is in the context of the *CustomerEmployee2CustomerEmployee* mapping. It has to be noted, that these two A2A mappings which are using trace models of different C2C

operators are sufficient to split the attribute values into two sets.

Transformation View. The corresponding transformation net is shown in the transformation view of Figure 6.20. The derivation of the C2C mappings into transformation components and the nesting of transformation components for expressing generalization relationships is done as already explained. What we want to discuss now in more detail is the linking of the corresponding components of the feature mappings to the provided trace models of the C2C components in order to realize the intended transformation behavior. The A2A mapping for the *Person.name* attributes is connected to the union trace model of the *Person2Person* transformation component, because for each person (each instance in our example is an indirect instance of *Person*) the value of the *name* attribute has to be transformed. Therefore, we need the whole trace information of all nested transformation components, which is provided by the most outer transformation component. The A2A mapping for *Customer.custNr* values depends on the *Customer2Customer* component, because the union trace model for direct instances of *Customer* and for *CustomerEmployee* instances is needed. For transforming the *Employee.empNr* values, two transformation components are required. First, an A2A transformation component is used to transform *Employee.empNr* values of direct instances of *Employee* to the *EmployeeNoCustomer.empNr* place. Therefore, this A2A transformation component depends on the *Employee2EmployeeNoCustomer* component, which ensures that *Employee.empNr* values contained in direct instances of the class *CustomerEmployee* remain in the source place, because no history information about direct instances of the *CustomerEmployee* class can be provided from the required trace model of the *Employee2EmployeeNoCustomer* component. In order to transform the remaining tokens as well, a second A2A transformation component is used, which depends on the *CustomerEmployee2CustomerEmployee* transformation component and streams the tokens to another target place, namely to the *CustomerEmployee.empNr* place. The *CustomerEmployee-2CustomerEmployee* trace model only provides information about the transformation of direct instances of the *CustomerEmployee* class and therefore ensures that no values of direct instances of the class *Employee* are streamed through the dependent A2A transformation component.

This example shows the value of using an explicit notion of trace models together with a union of trace models for automatically synchronizing transformation net components. With the automatic synchronization provided by the Petri Net semantic, the splitting of sets into subsets comes for free. Thus, no additional control structures and OCL constraints are required, and the derivation of transformation nets from mapping models is straightforward.

Chapter 7

Summary and Related Work

Contents

7.1 Summary

In this part of the thesis, we have introduced a framework allowing the definition of mapping operators and their application for building metamodel bridges. Metamodel bridges are defined by the user on a high-level mapping view which represents the semantic correspondences between metamodel elements and are tested and executed on a more detailed transformation view which also comprises the transformation logic of the mapping operators. The close integration of these two views and the usage of models during the whole bridging process further enhances understandability and the possibility of debugging the defined mappings in terms of the mapping operators. The applicability of the framework has been demonstrated by implementing mapping operators subsumed under the mapping language CAR for resolving common structural metamodel heterogeneities. Furthermore, we have shown how the mappings can be reused within a mapping model by using inheritance. The inheritance mechanism introduced for the CAR mapping language has been also derived into appropriate transformation net concepts by building union trace models out of individual trace models of single mapping operators.

7.2 Related Work

With respect to our approach we can distinguish between two categories of related work: first, related work in the field of model-driven engineering concerning the design of reusable

model transformations, and second, related work in the field of ontology engineering concerning the usage of dedicated mapping languages for bridging structural heterogeneities.

7.2.1 Reusable Model Transformations

Generic Model Transformations. Typically, model transformation languages, e.g., ATL [JK06] and QVT [OMG05b], allow to define transformation rules based on types defined as classes in the corresponding metamodels. Consequently, model transformations are not reusable and must be defined from scratch again and again with each transformation specification. One exception thereof is the approach of Varró et al. [VP04], who propose the notion of generic transformations within their VIATRA2 framework, which in fact resembles the concept of templates in C++ or generics in Java. VIATRA2 also provides a way to implement reusable model transformations, although it does not foster an easy to debug execution model as is the case with our proposed transformation nets. In addition, there exists no explicit mapping model between source and target metamodel which makes it cumbersome to reconstruct the correspondences between the metamodel elements based on the graph transformation rules, only.

Transformation Patterns. Very similar to the idea of generic transformations is the definition of reusable idioms and design patterns for transformation rules described by Karsai et al. [AVK+04]. Instead of claiming to have generic model transformations, the authors propose the documentation and description of recurring problems in a general way. Thus, this approach solely targets the documentation of transformation patterns. Indications how these patterns could be implemented in a generic way are not given.

Mappings for bridging metamodels. Another way of reuse can be achieved by the abstraction from model transformations to mappings as is done in our approach or by the ATLAS Model Weaver (AMW) [FBJ+05]. AMW lets the user extend a generic so-called weaving metamodel, which allows the definition of simple correspondences between two metamodels. Through the extension of the weaving metamodel, one can define the abstract syntax of new weaving operators which roughly correspond to our mapping operators. The semantics of weaving operators are determined by a higher-order transformation that take a weaving model as input and generates model transformation code. Compared to our approach, the weaving models are compiled into low-level transformation code in terms of ATL which is in fact a mixture of declarative and imperative language constructs. Thus, it is difficult to debug a weaving model in terms of weaving operators, because they do not explicitly remain in the model transformation code. Furthermore, the abstraction of mapping operators from model transformations expressed in ATL seems more challenging compared to the abstraction from our proposed transformation net components.

7.2.2 Ontology Mapping for Bridging Structural Heterogeneities

In the field of ontology engineering, several approaches exist which make use of high-level languages for defining mappings between ontologies (cf. [KS05] for an overview). For example, in Maedche et al. [MMSV02], a framework called *MAFRA* for mapping two heterogeneous ontologies is proposed. Within this framework, the mapping ontology called *Semantic Bridge Ontology* usually provides different ways of linking concepts from the source ontology to the target ontology. In addition to the Semantic Bridge Ontology, MAFRA provides an execution platform for the defined mappings based on services whereby for each semantic bridge type a specific service is available for executing the applied bridges. In [SdB05], Scharffe et al. describe a library of so called *Ontology Mediation Patterns* which can be seen as a library of mapping patterns for integrating ontologies. Furthermore, the authors provide a mapping language which incorporates the established mapping patterns and they discuss useful tool support around the pattern library, e.g., for transforming ontology instances between different ontology schemas.

The main difference to our approach is that ontology mapping approaches are based on Semantic Web standards, such as *OWL* and *RDFS*, and therefore contain mapping operators for typical description logic related mapping problems, e.g., *union* or *intersection* of classes. We are bridging metamodels expressed in MOF, a language which has only a partial overlap with OWL or RDFS, leading to different mapping problems. Furthermore, in contrast to the ontology mapping frameworks, we provide a framework allowing to build new mapping operators by using well-known modeling techniques not only for defining the syntax but also for the operational semantics of the operators.

Part III

Roundtrip Transformations

Chapter 8

Why Roundtrip Transformations?

Contents

The third part of this thesis elaborates on roundtrip transformations and how this kind of transformations can be systematically engineered. Figure 8.1 highlights the focus of this part. In particular, for supporting this integration scenario, a mechanism is needed to extend the target language (cf. L_B) with concepts of the source language (cf. $L_A{}'$), which are not available in the target language. This means, the language extension (cf. L_{EXT}) represents the concepts which are in the source language but not in the target language.

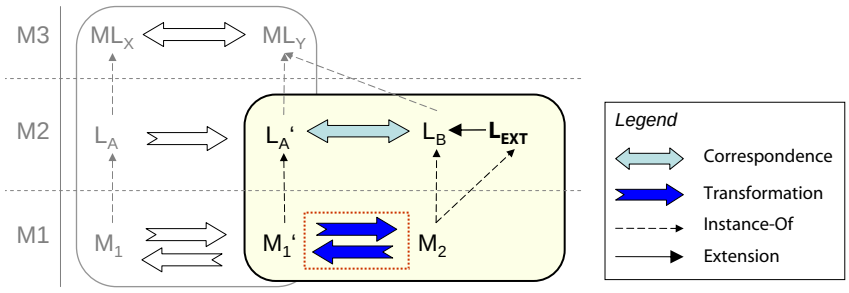

Figure 8.1: Integration Scenarios Revisited – Focus of Part III

This chapter introduces the term roundtrip transformations and explains why this kind of transformation is actually needed in the context of tool integration. Furthermore, two integration scenarios requiring roundtrip capability are presented and used as running examples or case studies in the subsequent chapters.

8.1 Motivation

The basic case of tool integration occurs when two different tools' modeling languages conceptually overlap to a large extent. This means, that both modeling languages cover the same or very similar domains, in a way that semantically equivalent modeling concepts can be identified in either metamodel and models can be transformed correspondingly. Variations address *directionality* and *completeness* of a transformation.

A transformation may be *bidirectional*, allowing two-way transformations between metamodels. In case a tool, for instance a code generator, is purely consuming and not producing models, unidirectional transformation suffice, e.g., in the typical MDA scenario where code is generated from a platform-specific model. However, in case of bridging two tools, bidirectional transformations are mostly necessary in order to move from tool *A* to tool *B* and then back again from *B* to *A*. This process is further on called *roundtrip*. The CAR language presented in Chapter 5 of this thesis is a bidirectional mapping language in the sense that two unidirectional transformation nets (both going in opposite directions) can be derived from one mapping model.

In case modeling languages of two different tools are not entirely overlapping, meaning that either some modeling concepts or features of modeling concepts are available in one modeling language which cannot be expressed in the other modeling language, a transformation may be lossy. Thus, although bidirectional transformations are available, the initial model M_a of tool *A* may be different from the roundtrip result M'_a which is computed by translating M_a into M_b via the transformation T_{a2b} and the application of T_{b2a} on M_b to produce M'_a or short $M'_a := T_{b2a}(T_{a2b}(M_a))$. The main reason for not roundtripping transformations is the fact that *bijective* mappings are not always possible to establish between metamodels as for example reported in [Ste07].

In this part of the thesis, we provide two approaches for supporting the engineering of roundtrip transformations. In particular, we focus on two bridging cases, namely when two modeling languages are completely overlapping in terms of their modeling concepts or the modeling concepts of a language are only a subset of another, but the modeling concepts themselves have not completely overlapping features. Thus, there are *missing correspondences* [LN07], as already mentioned in 1, between the features of the source and target metamodels, i.e., no mappings are possible for those kind of metamodel elements. If elements of the source metamodel are not part of a mapping, there is definitely an information loss, meaning that the source model cannot be completely reconstructed from the generated target model.

Both approaches enrich transformations in order to achieve that the models before and after roundtrip are equal $M_a == T_{b2a}(T_{a2b}(M_a))$ by explicitly storing information, which would normally get lost due to missing correspondences when performing the round-trip, within the target models in terms of annotations. This means, we are proposing lightweight

approaches for which the use of current model transformation techniques is sufficient, in contrast to using heavyweight approaches such as complex model management systems [Mel04] or logging the information in separate files.

We decided to use annotation mechanisms for storing information which has no corresponding parts in the target model, because annotation mechanisms are quite common in currently used modeling languages, such as Ecore or UML. However, the annotation mechanism can range from simple comment-like strings via tagged/value pairs to sophisticated language extension mechanisms of general but extensible modeling languages such as the extension of UML through profiles.

Saving all information of the source models in the target models, i.e., corresponding but also non-corresponding information, offers several advantages in tool bridging scenarios compared to saving non-corresponding information in separate files or models. First, in tool modernization scenarios [OMG05a, Ulr05] it is often helpful to explore information of models of a source metamodel which is not directly representable within the target modeling language, e.g., for understanding design decisions or for reproducing constraints which are not directly representable in the target modeling language. Second, the information can be further processed in subsequent tasks, being, for example, the driver for model transformation or code generation for a specific platform. Third, no additional tooling is necessary such as a special repository which can provide merge functionalities for incorporating the initial information back into the resulting model after roundtrip.

Various model exchange scenarios realized in the ModelCVS project have shown that information loss during a transformation is in most cases a *cross-cutting concern*, i.e., it is not bound to a specific transformation rule, but rather spread over the whole transformation leading to more complex transformation code compared to code which only covers the transformation of corresponding elements. Therefore, the definition of transformation logic for corresponding elements should be separated from transformation logic responsible for saving information in annotations taking care of missing correspondences.

In the next sections, two approaches are presented for tackling the problem of information loss in roundtrip scenarios. Both approaches allow for clearly separating the specification of mappings for corresponding parts from the specification of non-corresponding information in annotations. Both approaches are based on the CAR mapping language which has been introduced in part two of this theses. The first approach can be seen as an aspect-orientated extension of the CAR mapping language, whereas the second approach focusses on the integration of DSLs with UML by using the mapping model not only for generating the transformations, but also for generating language extensions in terms of UML profiles.

AspectCAR. This approach represents an aspect-oriented extension of the CAR mapping language, which can be used to enrich existing CAR mapping models to prevent information loss during roundtrip. The separation in base mapping models and aspects allows to separate mappings between semantically corresponding elements from mappings for sav-

ing additional information in annotations. A weaver component can then combine the base mapping model with the aspects in order to produce the intended mapping model which is capable of preserving important information during roundtrip. This approach is individually configurable with respect to the annotation mechanism and which information should be saved to ensure roundtrip.

ProfileGen. One often recurring bridging scenario is to integrate DSLs with UML. However, due to the fact that UML is a generic object-oriented modeling language which is not tailored to a specific domain or platform, the bad thing is that a transformation inherently looses information when moving from a DSL to UML. The good message is that UML provides a sophisticated extension mechanism in terms of profiles which can be used to extend UML for specific domains and platforms. Therefore, we present an approach dedicated to bridging DSLs with UML based on profiles. The profile definitions are used for storing information from the DSL models which is not directly representable in UML. The general idea of this approach is to use a mapping model also for automatically generate UML profiles which in turn can be used by the model transformations.

8.2 Integration Scenarios

In this section, we introduce two integration scenarios where in the first scenario loss of meta-information exists whereas in the second scenario a loss of information occurs. These two scenarios are subsequently used to elaborate on the proposed approaches for enabling roundtrip transformations.

8.2.1 Scenario 1: Loss of Meta-Information

In this subsection, we present a scenario in which important *meta-information* is lost, even though for the modeler all visible model elements can be exchanged between the tools. With meta-information, we mean in this case *administrative* and *technical* information [(NI04] which is not part of the modeling domain but is needed to achieve and preserve the model by the tooling environment such as unique identifiers for model elements.

Consider the following bridging scenario as depicted in Figure 8.2. Two modeling tools (cf. *ToolA* and *ToolB*) have to be bridged based on their metamodels to ensure model exchange between the tools (cf. step 1). One requirement for the bridging solution is that models can be simultaneously updated by different users leading to concurrent changes of models (cf. step 2). Consequently, this means, when updated models from different modelers are integrated in one consolidated version (cf. step 3), the models have to be merged via a *merge* operator which in turn requires a possibility to compute the differences between the models via a *diff* operator [Ber03] (cf. step 4 and 5). In this scenario it is assumed that the diff operator is essentially based on the unique identities of the model elements and is

only available for one tool A. In contrast, tool B does not make use of explicit identifiers (because no merge or diff functionalities are intended) and therefore, the problem arises that the original identifiers from models of tool A get lost during roundtrip. Without any additional solution, only new identity values can be automatically generated when transforming models from tool B to tool A. This meta-information loss prevents the usage of the diff techniques and simultaneous updating of the models cannot be supported. This means, the transformations T_{a2b} and T_{b2a} have to be enriched with transformation logic for saving the identifiers in additional annotations (T_{a2b}) and to retrieve the annotated values in order to set the identifier values accordingly (T_{b2a}).

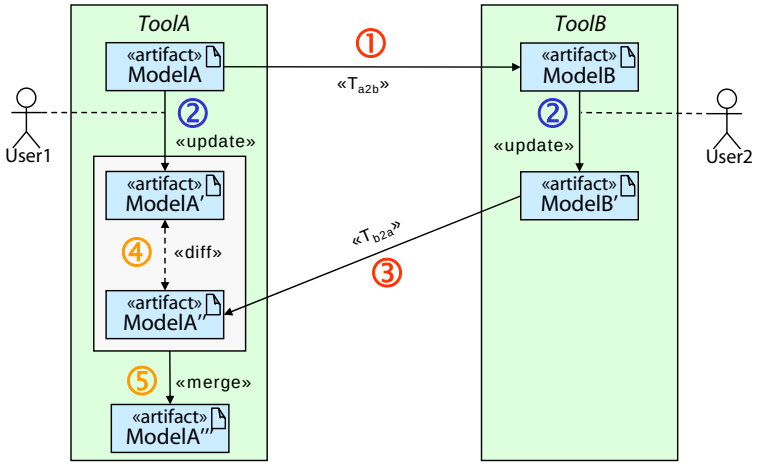

Figure 8.2: Supporting Simultaneous Updates by Different Users

8.2.2 Scenario 2: Loss of Information

The common case in model exchange is that it is not possible to bridge each modeling concept with all its features from one modeling language to another. Thus, the target models are only a projection of the source models. For example, consider that in a software development department one development team uses an UML tool for modeling data-intensive applications and a second team uses a dedicated database design tool, whereas the second tool can be seen as a domain-specific modeling tool for databases which allows the user to define much more technical details about the persistence layer of the application. In contrast, the UML tool provides a platform independent modeling style also for the persistence layer. Although, there is a high overlap of the used languages, they are not exactly on the same abstraction level, a circumstance which further aggravates the bridging task. In such cases, it makes sense that also the difference between the models can be accessed

in both tools, e.g., also the platform specific information should be available in the UML tool. However, when only plain UML is used, there is no way to represent platform specific information, e.g., database configuration details such as the average occurrence of instances of a class, which is necessary for optimizing the performance of databases.

Table 8.1: Model Metrics for Data Model/Class Diagram Roundtrip

Metrics	Initial AFG Model	Generated UML Model	AFG Model after Roundtrip	Diff in %
#Objects	156	165	156	0
#Values	1099	156	156	85,8
#Links	44	54	36	18,2
#Containment Links	155	164	155	0
File Size	32,8 KB	16 KB	14,6 KB	55,5

In the ModelCVS project, we have bridged the AllFusion Gen[1] (AFG) modeling tool and its DSL for defining database schemas with the UML Rational Software Modeler[2] tool. Thereby, the first case study was to bridge structural modeling, i.e., the AFG Data Model with the UML Class Diagram. The first attempt was to bridge AFG with plain UML. The resulting bridge was not appropriate for using the tools in combination. Because, although we have defined for each AFG modeling concept a specific transformation rule, a lot of information was lost during roundtrip or even though after the first step when moving from the DSL to UML. Table 8.1 summarizes the roundtrip scenario by depicting some model metrics for each step in the roundtrip.

The main reason for the massive loss of information was the fact that on the meta-attribute level only a minimal overlap between the languages exists. In most cases, only the *name* attribute of the modeling concepts may be bridged, but all platform specific attributes of the AFG modeling language may not. When we take a look at the model metrics in Table 8.1, the initial AFG model and the generated UML model have nearly the same amount of objects and containment links, only some derived objects are additionally instantiated in the UML model. This means, the same model structure can be reproduced on the UML side. However, when we compare the amount of values, we see that a huge amount of information gets lost in the first step. In particular, when comparing the number of values of the initial AFG model and the resulting AFG model after roundtrip, 85,8 % of values are lost during the roundtrip. For links which are not containment links we see that more links exist in the generated UML model compared to the initial AFG model. This is due to the fact, that also derived links are generated for the aforementioned additionally derived objects. Therefore, even though we have more links and objects on the UML side, less information is expressed and some links cannot be reconstructed when transforming the UML models back to AFG models. Finally, the information loss has of course an effect on the file size, namely the resulting file after roundtrip has only half the size of the initial file.

[1] http://ca.com/us/products/product.aspx?ID=256
[2] http://www-306.ibm.com/software/awdtools/modeler/swmodeler

Chapter 9

AspectCAR – An Aspect-oriented Extension for the CAR Mapping Language

Contents

In this chapter, an aspect-oriented extension for the CAR mapping language called AspectCAR is presented. To define this language extension, a reference architecture for aspect-oriented modeling is employed in Section 9.1. While in Section 9.2, AspectCAR is explained by example, Section 9.3 provides a critical discussion concerning some limitations of the presented approach.

9.1 Motivation

In this section, we present an approach for saving information in roundtrip scenarios which would be lost focusing only on the semantically corresponding parts of the metamodels. More specifically, we introduce an approach which is based on aspect-oriented concepts, for separating the definition of mappings for semantically corresponding elements between metamodels (*one-to-one*, *one-to-many*, and *many-to-many* mappings) from the definition how to save information which has no semantically corresponding parts (*one-to-zero* and *zero-to-one* mappings) in annotations.

Introducing aspect-orientation for mapping models. In this subsection, we describe the basic ingredients of aspect-orientation and how aspect-orientation can be applied for mapping models. The concept of separation of concerns can be traced back to Dijkstra [Dij76]

and Parnas [Par72]. Its key idea is the identification of different concerns in software development and their separation by encapsulating them in appropriate modules or parts of the software. Aspect-Oriented Software Development (AOSD), adopts this idea and further aims at providing new ways of modularization in order to separate crosscutting concerns from traditional units of decomposition during software development.

From a software development point of view, aspect-orientation has originally emerged at the programming level with AspectJ as one of the most prominent protagonists. Meanwhile, the application of the aspect-oriented paradigm is no longer restricted to the programming level but more and more stretches over phases prior to the implementation phase of the software development life cycle such as requirements engineering, analysis, and design. This development is also driven by the simultaneous rise of MDE employing models as the primary artifact in software development. In the context of this, Aspect-Oriented Modeling (AOM) languages attract more and more attention. Consequently, AOM can also be applied for mapping and transformation models.

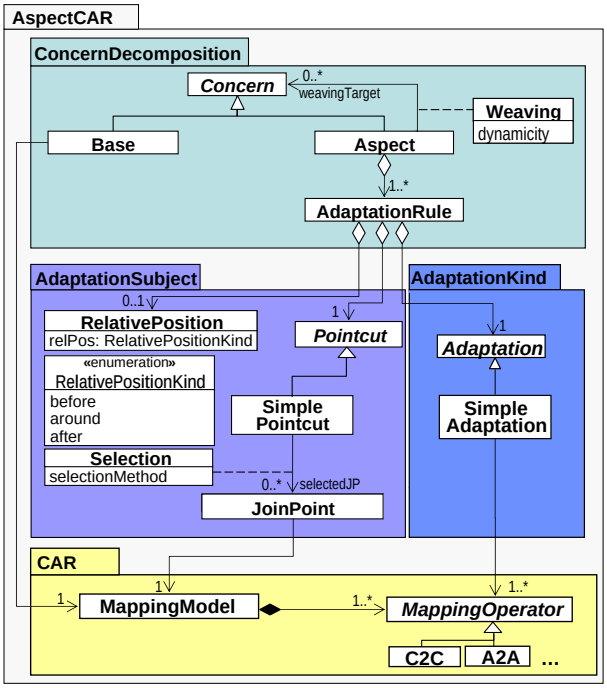

Figure 9.1: AspectCAR

In Figure 9.1, the architecture of our aspect-oriented approach for the CAR mapping language is shown which is based on a reference architecture for aspect-oriented modeling proposed by Schauerhuber et al. [SSK+06, Sch07]. Because we introduce only an asymmet-

ric aspect-oriented approach[1] for the CAR mapping language, we reuse only those parts of this architecture concerning asymmetric concepts. In the following, we elaborate on common aspect-oriented concepts and how the aspect-oriented extension is integrated into the CAR mapping language.

Concern Decomposition. Concern decomposition deals with the general decomposition of the system under development, in our case the mapping model, into concerns and their interrelationships. A *concern* is an inclusive term for *aspect* and *base*, which is depicted using generalization in Figure 9.1. A distinction between aspect and base concerns means supporting the asymmetric approach to decomposition. A base is a unit of modularization formalizing a non-crosscutting concern. An aspect is a unit of modularization formalizing a crosscutting concern. In AOSD, the composition of aspects with other concerns, is called *weaving*. In general, one can distinguish between two ways of weaving aspects into the base model, namely static (i.e., at design time) and dynamic (i.e., at runtime). For our approach, design time weaving is sufficient, because we are weaving aspects into the base mapping model which is in turn transformed into a transformation net, so the weaving happens before the actual execution of the transformation. An aspect's adaptation rules introduce *adaptations* at certain points of concerns. Consequently, an adaptation rule consists of an adaptation describing how to adapt the base model, and a *pointcut* describing where to adapt the concern.

Language. The language package describes the language underlying the specification of base and aspect, in our case the CAR mapping language. Concerns are formalized using elements of the CAR mapping language, in particular the mapping operators. With respect to aspect-orientation, mapping operators serve two purposes. First, they may represent join points and thus, in the role of join points specify where to introduce adaptations. Second, mapping operators are used for formulating an adaptation.

Adaptation Subject. The adaptation subject package describes the concepts required for identifying where to introduce an aspect's adaptations. A *join point* specifies where an aspect might insert adaptations. Thus, a join point is a representation of an identifiable element of the underlying language used to capture a concern. The *join point model* comprises all elements of a certain language where aspects are allowed to introduce adaptations. A *pointcut* represents a subset of the join point model, i.e., the join points used for specifying certain adaptations. In our approach, the selection of join points as pointcuts is done by OCL-like queries on the mapping models. A *relative position* may provide further information as to where adaptations have to be introduced. This is necessary since in some cases, selecting join points by pointcuts, only, is not enough to specify where adaptations have to be inserted, since an adaptation can be introduced for example before or after a certain

[1] AOM approaches follow different schools of thoughts, some adhering to an asymmetric view supporting the distinction between aspect and base, while others have a symmetric understanding where all concerns are treated on an equal basis and no distinguished base model is selected [HOT02].

join point. Still, in many other cases, a relative position specification is not necessary, e.g., when a new attribute is introduced into a class, the order of the attributes is insignificant. For our purposes, we do not use relative positions, because the mapping operators can be connected to all existing ports of the mapping components, thus, the user can decide the exact position where the aspect is woven into the mapping model.

Adaptation Kind. The adaptation kind package comprises the concepts necessary to describe an aspect's adaptation. An adaptation specifies in what way the concern's structure or behavior is adapted, i.e., enhanced, replaced or deleted. For our purposes, the use of constructive adaptation effects (cf. enhancements) is sufficient, because we only have to introduce mappings for storing information of the source model with no corresponding elements in the target model within annotations. This means, the existing mappings for semantically corresponding elements should not change, i.e., there is no deletion or replacements of already existing mappings.

9.2 AspectCAR by Example

9.2.1 Running Example

For describing our approach, the following running example is used which basically adheres to the previously discussed scenario 1 about a loss of meta-information.

The running example deals with the specification of a bridge between two metamodels depicting two heterogenous data structures (an array and a linked list) in terms of a mapping model, which is compiled into a transformation net that finally executes the transformation process. Figure 9.2 shows the source and target metamodels, as well as the input model and the desired output model. As shown, a transformation between these two metamodels has to transform array input models into linked-list output models. The transformation specification in-between the metamodels is given in an extended version of the CAR mapping language, which comprises several operators whose exact transformation net semantics will be given in the following section when describing the runtime level. The C2C (Class2Class) operator takes objects from the *Element* class as input, and outputs them into the *Node* class. Analogously, the R2R (Reference2Reference) operator streams links from the reference *contains* to the reference *head*. The C2C operator offers another output port *history*, which offers the possibility to access all yet transformed tokens. The *2-Buf* operator connected to C2C's *history* port sequentially fills an internal buffer of size two, which is again provided as output port. A *Linker* operator takes the two objects in the 2-Buf operator, and produces a link between them which is streamed into the *next* reference. A *back-link* is produced by the *Inverter* operator from the *next* reference and streams them into the *prev* reference. The 2-Buf, Linker, and Inverter operators can be seen as an extension of the core CAR mapping language for converting graph structures. The operational semantics of the

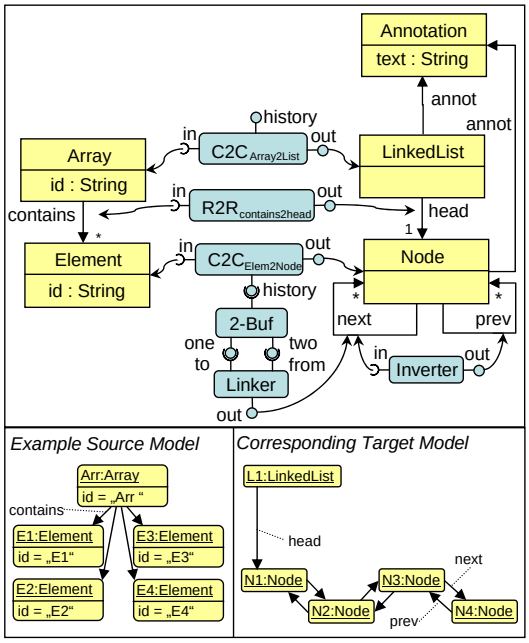

Figure 9.2: Running Example – Base Mapping Model

operators are presented in a by-example manner in the next section.

9.2.2 Problem of the Bridging Solution

As one can see in Figure 9.2, the *id* values of the LHS model are lost when moving to the RHS. Fortunately, the RHS metamodel provides an annotation mechanism (cf. class *Annotation* in the RHS metamodel) for each metamodel element. This means, each model element can have arbitrary annotations, each annotation containing a string value. This annotation mechanism can be exploited for storing the *id* values of the LHS models also in the RHS models. Thus, the *id* values can be used for the inverse transformation to set the same values in the newly generated LHS model as in the original one before roundtrip. Consequently, the diff technique based on *id* values can then be used for merging the models.

For saving the *id* values, each C2C mapping would require an additional component for transforming the *id* values into *Annotation* objects, having in turn a text value which corresponds to the *id* value. For this purpose, the previously introduced A2C operator can be used. However, the adaptation of the transformation with the capability of saving the *id* values can be seen as a cross-cutting concern which is manifested in the whole transformation. In particular each C2C mapping has to be extended with an A2C operator. In order to avoid defining this transformation logic for each *id* attribute again and again, aspect-orientation

can be applied to encapsulate this transformation logic and to weave this functionality into the base mapping models which is discussed for this example in the next subsection.

Additionally to these "manually" assembled components, certain operators can cross-cut a transformation specification: Because the target metamodel classes do not have *id* attributes, these should be stored within an annotation for eventual round-tripping. This can be accomplished by the A2C component, which henceforth crosscuts the transformation of every object and is therefore woven with every C2C component. The transformation specification is itself a model, and due to the component-like assembly, existing model weavers can be used to merge the aspect model into the base transformation specification.

9.2.3 Defining the Aspect

The top of Figure 9.3 shows the aspect's definition in a notation inspired by XWeave [GV07]. The query *%allC2C* in the aspect *ID2Annotation* selects all C2Cs, with three additional subqueries *in.id*, *history* and *out*, relative to the current C2C operator. The results of *in.id* and *history* are bound to the *value* and *object* ports of the A2C operator. This operator instantiates for every transformed object a new Annotation object (*class* port) which is linked up (*ref* port) with the according *Node* object and sets its *text* attribute (*att* port) to the value of the source objects' *id* attribute. The result of the *out* query determines the classes to which an *annot* reference will be added. In the middle of Figure 9.3, the resulting mapping model (cf. Composed Mapping Model) of the weaving process is illustrated,

9.2.4 Generated Transformation Net

After the weaving process is carried out by the weaver component on the mapping level, generation takes place to produce a transformation net out of the composed mapping model. The top of Figure 9.4 shows a transformation net resulting from the above integration specification. Note that places marked as *ordered* provide contained tokens in a sorted fashion. For instance, the R2R component's transition matches *ArrE1* - the "first" input token. Furthermore, according to the multiplicity of a reference, a place (e.g. *head*) can have a capacity, which constrains the amount of tokens a place can hold. For simplicity reasons, the example assumes only a single array object, and since there is only a single ordered reference, the *Element* place is compiled into an ordered place as well, as not to unnecessarily complicate the example.

The middle and the bottom of Figure 9.4 show the transformation net during execution and in its finished configuration. For instance, one can see how the tokens streamed through the C2C component are stored in its history place. Note that the history place is duplicated in the lower C2C, as both the Att2Annot and the 2-Buf components are bound to it. The 2-Buf component takes in these tokens and fills its two-place buffer. Once the buffer is full (both places have a capacity of just one token), the Linker component's transition can fire

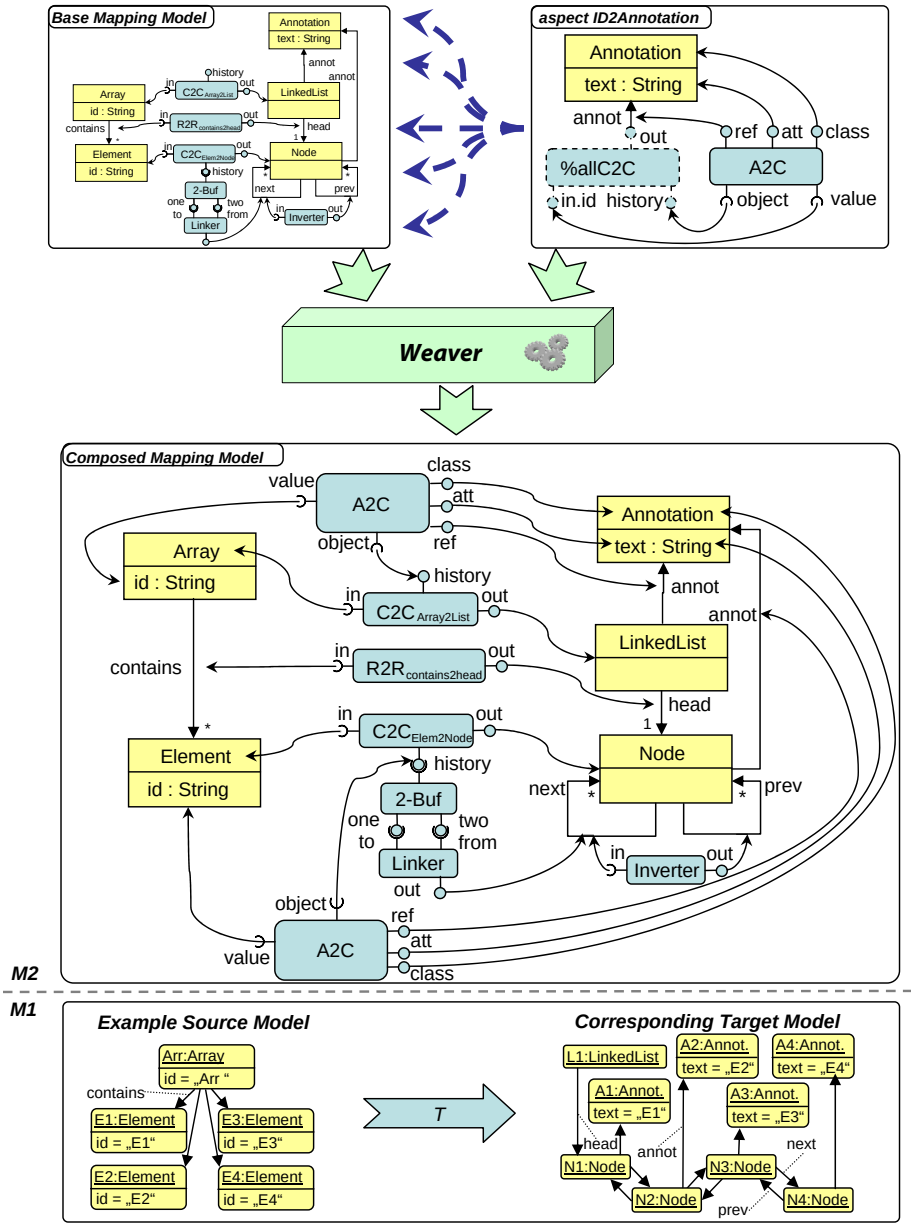

Figure 9.3: Running Example – Weaving Aspects into Mapping Models

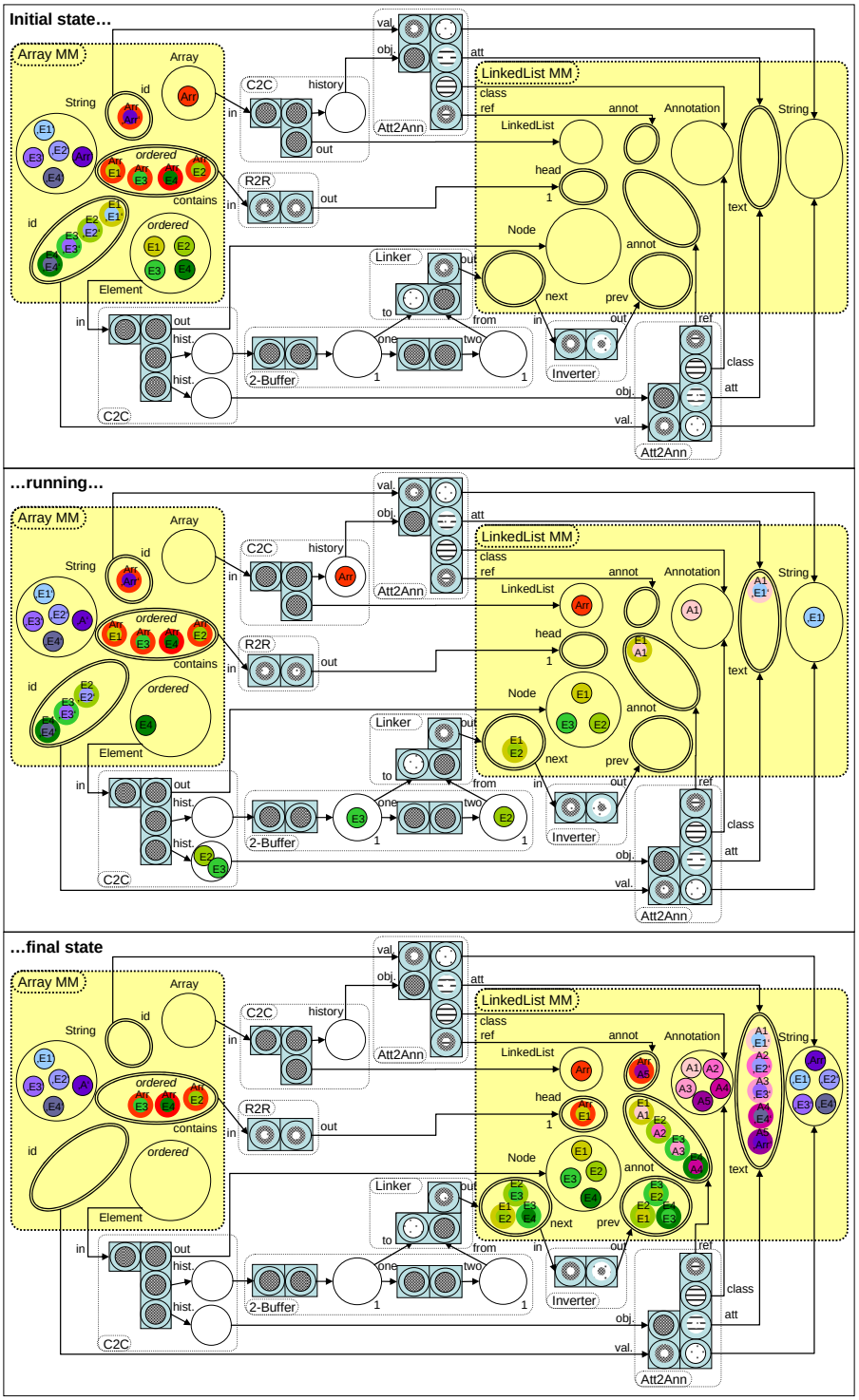

Figure 9.4: Running Example – Roundtrip-aware Transformation Nets

and empty the buffer, producing a two-colored token which is streamed into the *next* place. Thereby it is to note, that the creation of two-colored tokens for the *next* link is based on a certain state of the execution, rather than on the input model alone. Furthermore, one can see how the previously weaved operators form Petri Net patterns that become active after an *Array* or *Element* token was streamed. As an example, in the "running" net, the lower A2C pattern has already created an *Annotation* object with the corresponding value for the *E1* object, and is currently enabled to do the same for *E2* and *E3*, as both have already been handled by a C2C component. Analogously, the rest of the patterns stream tokens from source to target places, possibly depending on other patterns in turn. The actual firing order, however, is handled by the underlying Petri Net engine. Once the transformation process has finished, the final net configuration is used to instantiate a model that conforms to the target metamodel, as shown in the bottom-right corner of Figure 9.3. This running example has been implemented within the open Architecture Ware (oAW) framework[2]. In particular, we used the XWeave [GV07] component for defining the aspect definitions as well as for weaving the base models in combination with the aspects into composed models.

9.3 Critical Discussion

The presented aspect-oriented approach for remedy of information loss can be seen as a very first step, only. First of all, we use a very small set of aspect-oriented concepts, e.g., we have not introduced composite aspects, pointcuts, or adaptations for allowing a more fine-grained reuse of already existing aspect definitions. Second, we are only using adaptations which are enhancements, although using adaptations which are modifications of existing mapping operators of the mapping model would offer some interesting possibilities, for example, to exchange basic mapping operators with mapping operators providing a modified operational semantic. However, this would require to look into the implementation of the mapping components, i.e., the white-box view must be adapted, which leads to the third limitation of our presented approach. For more fine-grained adaptations, a weaving capability for the transformation nets would be necessary for adapting the white-box view of the components. In particular, changing pre-conditions and post-conditions of the transitions would offer a much more fine-grained adaptation approach.

Despite these limitations, the presented approach of weaving whole mapping operators into the existing mapping models shows the benefit of adapting an already existing mapping model with information preserving capabilities for attribute values. However, if a complete information preserving approach is needed, the usage of an annotation mechanisms based on string values falls too short. In particular, when in addition to values, also objects and links, as is the case in scenario 2, should be preserved in the transformation, sim-

[2] www.eclipse.org/gmt/oaw

ple string based annotation values are not appropriate and do not scale for large bridging tasks. One way to cope with such extended requirements is to use a more sophisticated annotation mechanism for models such as is provided in UML with the profile mechanism. In the following section, we present an approach for realizing information preserving transformations with the help of UML profiles for bridging scenarios which falls under the category of bridging DSLs to UML.

Chapter 10

ProfileGen – A Generator-based Approach for Bridging DSLs with UML

Contents

After discussing the motivation for developing a systematic approach for bridging domain specific languages (DSLs) with UML in Section 10.1, the subsequent sections present an approach for bridging DSLs to UML in a semi-automatic way. Finally, the applicability of the proposed approach is demonstrated in a case study.

10.1 Motivation

In software engineering in general and in MDE in particular, there is a movement from general-purpose languages (GPLs) to domain-specific languages (DSLs). For defining DSLs

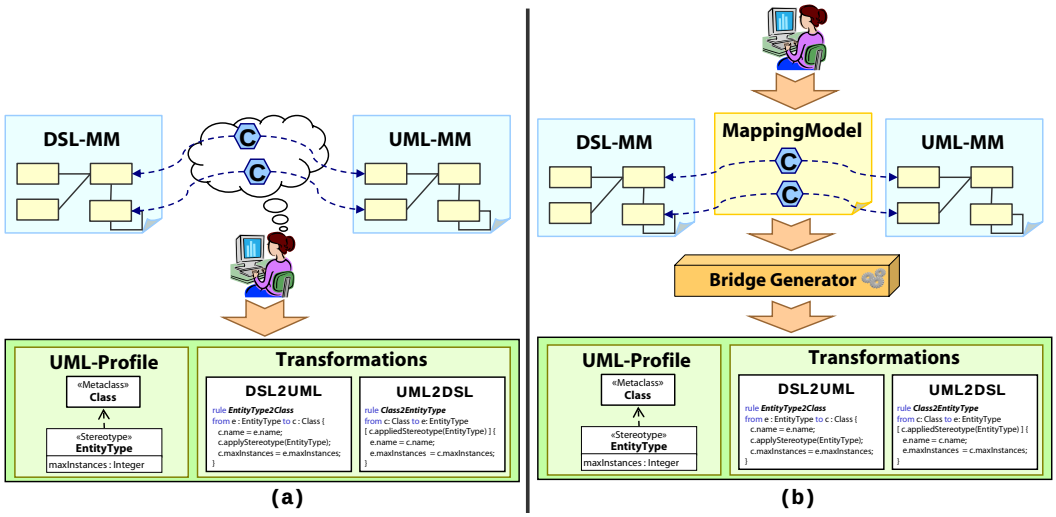

Figure 10.1: DSL/UML Integration. (a) Ad-hoc Approach, (b) Systematic Approach

in the field of MDE, *metamodels* and *UML profiles* are the proposed options. While metamodels, mostly based on the Meta Object Facility (MOF) [OMG04], allow the definition of DSLs from scratch, UML profiles are used to extend UML with domain-specific concepts.

In this part of the thesis, we focus on the interoperability between these two approaches, because the need to bridge modeling languages originally defined as DSLs to UML often arises in practice. For example, as already mentioned in Section 1, in the ModelCVS project [KKK+06b], our industry partner is using the CASE tool *AllFusion Gen*[1] from ComputerAssociate which supports a DSL for designing data-intensive applications and provides sophisticated code generation facilities. Due to modernization of the IT department and the search for an exit strategy (if tool support is no longer guaranteed), the need arises to extract models from the legacy tool and import them into UML modeling tools while at the same time the code generation of AllFusion Gen should in the future be usable for UML models as well. Besides these typical tool integration and interoperability issues, when buidling UML profiles from scratch, in a first step domain-specific modeling concepts are often collected in a metamodel and in a second step the corresponding UML profile is created, as it was done for example in [MFV07] for WebML.

Ingredients for a DSL/UML bridge. First of all, it has to be mentioned that the current practice in bridge development is characterized by ad-hoc solutions mainly focusing on implementation tasks as illustrated in Figure 10.1(a). First, one has to reason about correspondences between DSL and UML metamodel elements. Then, a profile must be defined

[1] http://ca.com/us/products/product.aspx?ID=256

for the DSL metamodel in which stereotypes for each metaclass of the DSL are added as well as tagged values for DSL features which are not directly representable in UML. Afterwards, one has to define the transformations from DSL to UML and back again, whereas the transformations are based on the UML profile definition, e.g., for assigning stereotypes and tagged values to UML model elements.

Drawbacks of ad-hoc implementation. The described approach for bridging DSLs to UML has several drawbacks, making the bridging task tedious and error-prone which results in low productivity and maintainability.

1. **Transformations and profiles are highly coupled**. A change in the profile definition requires changes in the model transformation code.

2. **Bridging covers many repetitive tasks**. For each DSL modeling concept nearly the same tasks have to be carried out in the integration process.

3. **No guidelines available**. Users are not familiar with the integration task since it is typically a one-time job.

4. **No explicit correspondences between DSL and UML elements**. No explicit mapping model is available for defining the correspondences between the languages on a high-level of abstraction. Instead one has to start with an implementation of the UML profile and the transformation rules directly.

10.2 Overview of the DSL2UML Bridging Approach

We propose a semi-automatic approach for bridging DSLs with UML and introduce two additional facilities for the integration process, as can be seen in Figure 10.1(b). First, we propose the use of an explicit *mapping model* which is built manually. Second, we provide a *Bridge Genenerator* component which is capable of generating the required profiles and transformations from the mapping models. The resulting integration process is as follows: First, the user defines the correspondences between the DSL metamodel and the UML metamodel in terms of a mapping model on basis of an interactive mapping environment. The mapping model is expressed in terms of a dedicated metamodel bridging language representing the pre-requisite for automatic processing. Out of the mapping model, the Bridge Generator automatically generates the UML profile and in case that an uni-directional model transformation language is used, it generates also the necessary transformations from the DSL to UML and back again.

Benefits of this systematic integration approach. Our approach tackles the four mentioned drawbacks of the ad-hoc implementation approach in the following way:

1. **Mapping model is the single source of information**. In our approach, it is sufficient that one modifies the mapping model, only. The changes made in the mapping model are automatically propagated to the UML profile and to the transformation definitions. Hence, the high coupling between the UML profiles and the transformations is transparent to the user.

2. **Repetitive tasks are automated by model transformations**. The only manual task is the definition of the mapping model. All subsequent implementation artifacts, i.e., the UML profile and the transformations are automatically produced by the Bridge Generator component. Furthermore, the Bridge Generator component ensures that the profiles and transformations are always developed in the same manner for the same kind of integration problem, leading directly to the next benefit.

3. **Guideline support for systematic integration**. First, the mapping model is built with a specific mapping language, and second, there is an explicit integration method for corresponding elements and non-corresponding elements of DSLs and UML implemented in the Bridge Generator component.

4. **Explicit representation of correspondences**. Our approach supports an explicit mapping model which represents the whole integration specification in a single conceptual model. Furthermore, currently used documentations such as mapping tables can be automatically generated out of the mapping model.

Qualitative Criteria for generated UML profiles. The generated UML profiles should preserve the following qualitative criteria which are based on criteria for integrated schemas produced out of a set of independent schemas in the field of database integration [BLN86]:

- **Correctness**: The generated profiles have to ensure the same modeling possibilities and constraints as the DSL metamodel. Furthermore, the profile definitions must not be in contradiction with plain UML, i.e., UML must be extended in a consistent way.

- **Completeness**: All information of the DSL metamodel must be represented in the UML metamodel which is extended with the profile. This means, it should be possible to transform the DSL models into UML models from which the original DSL models can be completely reconstructed.

- **Minimalism**: Minimalism requires that there are no redundant information definitions in the UML models. This is possible, when a modeling feature can be defined in plain UML and as tagged/values, leading to redundant definition of information. In such cases it is possible for the user to set for one modeling feature different values which leads directly to inconsistent models.

- **Understandability**: One major requirement is that the profile should be understandable by human users, e.g., inheritance between stereotypes for sharing common properties and conditions should be used where appropriate. Understandability of profiles ensures that subsequent tasks, such as extending the profile by hand or using the profiles in model transformations or code generation scripts, can be achieved, so to say from understandability follows usability.

Tooling. For supporting our approach with proper tooling, the following components are needed as a prerequisite for implementing a DSL2UML bridging framework. First of all, for step 1, a mapping editor is needed, which allows to define proper mappings between a DSL metamodel and an UML metamodel. This leads directly to the next requirement, a metamodel for UML. We reuse the UML metamodel of the UML 2 project[2] from Eclipse. For using UML models and UML profiles within transformations, we need a transformation language and a corresponding engine fulfilling two requirements. First, the use of UML profiles for extending the base UML metamodel, i.e., in addition to the UML metamodel, profiles must be loaded. Second, there must be a mechanism for calling external APIs, in our case the API of the UML 2 metamodel, in order to apply profiles, stereotypes, and tagged values on UML model elements. Currently, no transformation language and engine supports the use of UML profiles as it is intended by the UML standard. The main problem is that UML profiles are instances of the UML metamodel and at the same time extend the UML metamodel with new concepts. However, when using transformation languages supporting an arbitrary number of input models, the UML profiles can be at least read in and processed as normal input models. Due to these requirements, we decided to use the ATLAS Transformation Language (ATL) [JK06] instead of the transformation net formalism presented in Chapter 5, because ATL supports access to the UML 2 API as well as provides the possibility to use arbitrary input models.

10.3 DSL2UML Bridging Language

In this section we introduce the abstract syntax of the language used to describe the mappings between DSL and UML metamodels named DSL2UML bridging language. In particular, we are using the CAR mapping language concepts from Part 2 of this thesis as the basis for the DSL2UML bridging language and extend it with concepts for mapping types, such as data types and enumerations. Furthermore, we are reusing a generic mapping tool, called ATLAS Model Weaver (AMW) [FBJ+05], which provides a graphical user interface for building mapping models and is available for the Eclipse platform. The mapping language of the AMW can be easily extended with new mapping concepts. Consequently, for the definition of the DSL2UML bridging language, as one can be seen in Figure 10.2, we reuse

[2] www.eclipse.org/uml2

the core weaving language of the ATLAS Model Weaver, displayed in the *mwcore* package in Figure 10.2, which defines abstract concepts for namespaces such as *ModelRef* and *ElementRef* and linking semantics, such as a *WModel*, *WLink*, and *WLinkEnd*. Note that each concept in the core weaving language is defined abstract. The extension of these abstract concepts with concrete concepts is sufficient to use the AMW tool for defining mapping models between two metamodels.

Our dedicated bridging language, defined in the package *DSL2UMLbridgingLanguage*, consequently refines the abstract core weaving concepts with concrete concepts. In particular, we defined a metamodel bridging language which allows symmetric mappings, meaning one-to-one mappings between elements which are instances of the same meta concept. Furthermore, also asymmetric mappings can be used for describing m-to-n mappings between elements which are instances of arbitrary meta concepts. In the following, we briefly describe each concrete subclass of the class *WLink*, which represents the mapping operators of the bridging language, whereas first, the symmetric operators and then the asymmetric operators are presented. The semantics of the mapping operators are the same as presented for the CAR mappping language in Section 6. Therefore, in the following only a short summary of the operators are given.

Symmetric Mapping Operators

- **C2C-Mapping**. This kind of mapping operator allows the user to map classes of the DSL metamodel to classes of the UML metamodel. One pecularity of *C2C* mappings is, that they can have *superMappings*, i.e., the feature mappings from super mappings are inherited to sub mappings. This allows for reuse of already provided mapping information. Furthermore, the user can mark *C2C* mappings as abstract, meaning that they have to be refined with concrete sub mappings. Finally, *C2C* mappings can have invariants expressed in the Object Constraint Language (OCL) [OMG05d] for defining conditions when a mapping should be executed.

- **A2A-Mapping**. This kind of mapping allows to map corresponding attributes between DSLs and UML metamodels. In order to derive executable transformations for each *A2A* mapping, the *C2C* mapping that contains the *A2A* mapping has to be specified.

- **R2R-Mapping**. This kind of mapping is similar to *A2A* mappings with the difference that corresponding references are mapped instead of attributes.

- **E2E-Mapping**. In order to transform attribute values correctly between DSL and UML models, corresponding enumerations must be mapped. Therefore, we introduce *E2E* mappings to allow the definition of equivalent enumerations.

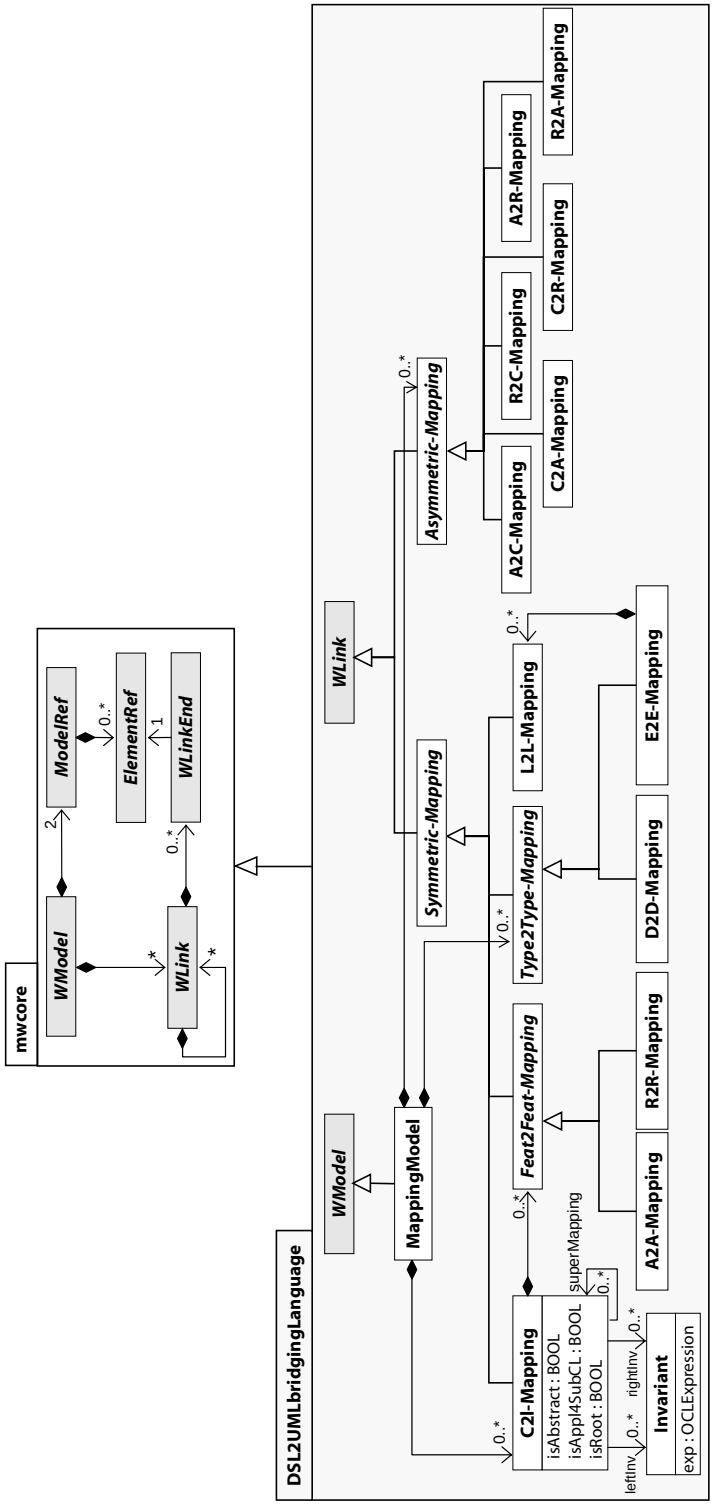

Figure 10.2: DSL2UML Bridging Language

- **L2L-Mapping**. This kind of mapping is used to define equivalences between literals of enumerations and consequently, each *L2L* mapping is owned by an *E2E* mapping. Note that only enumerations which have completely corresponding literals can be mapped, i.e., each literal must be mapped to exactly one other literal.

- **D2D-Mapping**. This kind of mapping is used to define equivalences between data types of DSLs and UML. It is required because (1) data types which are semantically identical are sometimes named differently and (2) DSLs often support specific data types which are not available in plain UML and therefore have to be additionally defined in profiles.

Asymmetric Mapping Operators. This kind of mapping operators are used to bridge structural heterogeneities as presented in part two of this thesis. Summarizing, we have the A2C-/C2A-Mapping for resolving the attribute vs. class heterogeneity, the R2C-/C2R-Mapping for resolving the reference vs. class heterogeneity, and finally, the A2R-/R2A-Mapping for bridging the attribute vs. reference heterogeneity.

10.4 Automatic Generation Process

The distinction between mapped and unmapped elements is the main driver for the two-step bridge generation process which is discussed in this subsection:

- *Step1 – UML Profile Generation*: The first step of the generation process is the profile generation.

- *Step2 – Model Transformation Generation*: As soon as the profile is available, the model transformation generation can be carried out.

10.4.1 UML Profile Generation

For the UML profile generation, the mapping model is parsed and for each mapping as well as for each unmapped element appropriate generation rules are called which together produce the content of the profile. In the following, the profile generation rules are presented, namely rules for symmetric mapping operators, subsequently rules for asymmetric mapping operators, and finally, additional rules which are necessary to derive meaningful and understandable profile definitions. These generation rules are implemented by a Java component based on the UML 2 project[3].

[3] www.eclipse.org/uml2

10.4.1.1 Generation Rules for Symmetric Operators

In the following, generation rules for symmetric operators are discussed. First, the generation rule for C2C mappings is presented which is the driver for creating stereotypes. Second, the generation rule for Feat2Feat mappings is presented. This rule is responsible for generating tagged values. Third, the generation rule for Type2Type is discussed which is responsible for generating types in the profile, such as data types or enumerations.

Profile Generation Rule 1 - C2C Operator. To prevent any loss of information, it is required that each DSL metaclass is mapped in the final mapping model. Unmapped DSL metaclasses are only allowed in incomplete mapping models, since they are not included in the generation process and consequently no complete bridge can be generated. Thus, instances of unmapped classes cannot be transformed. For each *C2C* mapping, a stereotype is generated which extends the UML metaclass being referenced by the right end of the mapping. If the *C2C* mapping has super mappings, the generated stereotype is a substereotype of the stereotypes generated for the super mappings. Listing 10.1 summarizes the stereotype generation from C2C-Mappings in pseudo code.

Listing 10.1: Stereotype Generation

```
FOR EACH c2c IN mappingModel.getC2C–Mappings(){
  CREATE Stereotype{
    self.setName(c2c.getRightClass().getName());
    self.setExtension(c2c.getLeftClass());
    FOR EACH superMapping IN c2c.getSuperMappings() {
      self.getSuperStereotype().add(
        resolveCreatedStereotypeFor(superMapping));
    }
  }
}
```

Profile Generation Rule 2 - Feat2Feat Operators. The next step concerns the generation of the tagged values from *Feat2Feat* mappings, i.e., *A2A* and *R2R* mappings. Assume the following *C2C* mapping: *DSL::Class* $\leftrightsquigarrow$ *UML::Class*. If we want to derive the tagged values for the stereotype which is generated for the DSL class, we must analyze the features of the DSL class and whether they are mapped to corresponding UML features or not. We can distinguish the following three distinct cases:

- **DSL::Class.features** $\cap$ **UML::Class.features**. Features which are available in the DSL and in the UML metamodel should be linked in the mapping model via feature mappings, i.e., the values of the DSL features are directly representable with UML features.
- **DSL::Class.features** $\setminus$ **UML::Class.features**. Features which are only available in the DSL must not be mapped via feature mappings to UML features. Furthermore, this

means that the values of these features are not representable in plain UML. Therefore, for each DSL feature which is not mapped to an UML feature a tagged value is generated. In particular, if the feature is an attribute then a tagged value having as type a data type is generated. If the feature is a reference, a tagged value is generated which can link to the stereotype actually representing the referenced type in the DSL metamodel.

- **UML::Class.features \ DSL::Class.features**. Features which are only available in UML must be specially treated. Though not relevant for profile generation, this case is relevant for generating model transformations. The problem is that the values of this kind of features cannot be set with values of the DSL models. In order to produce valid UML models, we must distinguish between optional and mandatory features. Optional features are easy to handle since a *null* value can be assigned. Mandatory features are more problematic. If there is no default value defined for the feature, the user must specify a value in the mapping model which is automatically assigned for this particular feature.

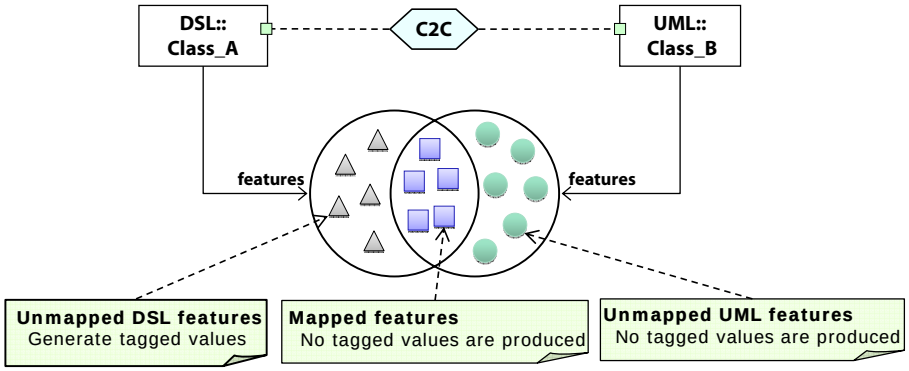

Figure 10.3: Profile Generation Rule 2 - Feat2Feat Operators

Profile Generation Rule 3 - Type2Type Operators. The last step in the generation process concerning symmetric mappings is about type mappings, i.e., (un)mapped *Enumerations*, *Literals*, and *DataTypes*. If an enumeration of the DSL metamodel fully corresponds to an enumeration of the UML metamodel then they are mapped with an *E2E* mapping. This *E2E* mapping further requires that each literal is mapped to exactly one literal with a *L2L* mapping. No further definitions are required in the UML profile. Otherwise, an additional enumeration with corresponding literals must be generated in the profile. The treatment of *DataTypes* is similar to that of enumerations.

10.4.1.2 Generation Rules for Asymmetric Operators

After discussing the generation rules for symmetric operators, we present the generation rules for asymmetric operators. These rules can influence the generation rules of symmetric operators, because asymmetric operators depend on at least one symmetric C2C operator. More specifically, new elements have to be generated in the profile or already existing elements have to be deleted. For discussing how the generation rules of asymmetric operators influence the generation rules of symmetric ones, we are using mapping model examples and illustrate how the profile generation take place for these examples.

Profile Generation Rule 4 - A2C Operator. For all attributes of a DSL class which are linked to the LHS of the A2C operator, no tagged values are generated for the stereotype generated out of the C2C mapping on which the A2C depends. Furthermore, no additional profile definitions such as stereotypes or tagged values have to be generated for the A2C mappings.

Example. For an example application of the generation rule for the A2C operator consider Figure 10.4. The input for the generation rule is the mapping model shown on the LHS, whereby the DSL class *X* semantically corresponds to the UML class *A*. The attribute *att1* of DSL class *X* corresponds to the attribute *att1* of UML class *B* which can be mapped via the A2C operator.

The result of the generation rule is shown on the RHS of Figure 10.4. As explained above, the generation process produces for the C2C mapping a stereotype *X* which extends the UML class *A*. For attribute *att1* of DSL class *X*, no tagged value for the stereotype *X* is generated, because this attribute is mapped to the attribute *att1* of the UML class *B* and therefore the value of the DSL attribute can be directly represented in plain UML.

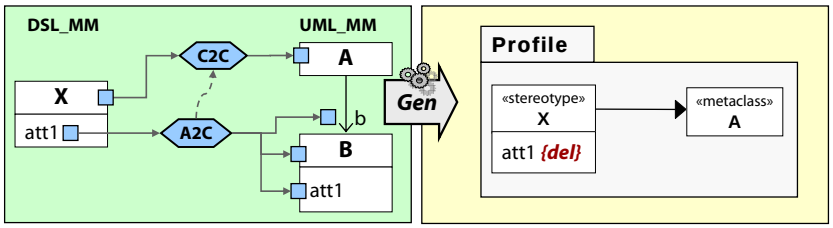

Figure 10.4: Profile Generation Rule 4 - A2C Operator

Profile Generation Rule 5 - C2A Operator. For a DSL class which is mapped with a C2A operator, no stereotype is generated in the UML profile, because there is no directly corresponding class in UML, thus no UML metaclass can be extended. Also for the reference and

attributes of the DSL side participating in the C2A mapping, no tagged values have to be generated. However, for unmapped features of the DSL class which is mapped with a C2A operator, tagged values have to be generated for the stereotype generated out of the C2C mapping on which the C2A operator depends.

Example. For the application of the generation rule for the C2A operator consider the example in Figure 10.5. In this figure, two DSL classes are mapped to the same UML class, thereby the DSL class *X* directly corresponds to the class *A* of the UML metamodel. The DSL class *Y* does not have a directly corresponding class on the UML side, but its attribute *att1* corresponds to the attribute *att1* of the UML class *A*. Furthermore, for the attribute *att2* no corresponding attribute is available in UML.

The result of the generation rule is shown on the RHS of Figure 10.5. First, for all mapped attributes of the DSL class *Y* (e.g., *att1*) no tagged values have to be generated for the stereotype *X* generated out of the C2C mapping on which the C2A mapping depends. However, for each unmapped feature of the DSL class *Y* (e.g., *att2*), a tagged value has to be produced for the stereotype *X*. In order to avoid name clashes of tagged values, the name of the containing DSL class is used as prefix (cf. *Y_att2* in Figure 10.5). For the reference between the two DSL classes, no tagged value has to be generated, because the grouping of the values is expressed at the UML side by the fact that the UML class directly contains all necessary attributes. Furthermore, for the DSL class *Y*, no stereotype has to be defined in the profile, because there is no UML class on which the stereotype would be applicable.

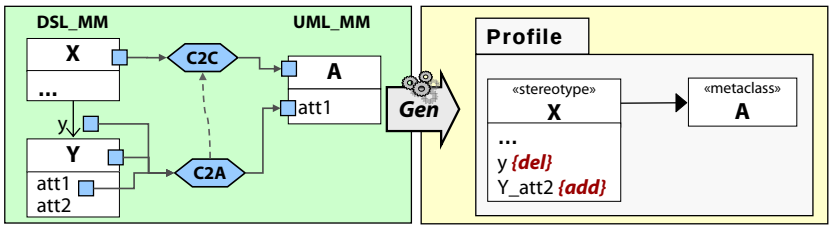

Figure 10.5: Profile Generation Rule 5 - C2A Operator

Profile Generation Rule 6 - R2C Operator. For references which are mapped with an R2C mapping, no tagged values have to be generated, because this reference can be directly represented in UML in terms of an association class. Besides this, no additional profile elements have to be generated for the R2C operator.

Example. For the application of the generation rule for the R2C operator, consider the example shown in Figure 10.6. The DSL classes *X* and *Y* have corresponding classes on the UML side, but the reference between *X* and *Y* corresponds to the UML class *AB* connecting the UML classes *A* and *B*.

The result of the generation rule is shown on the RHS of Figure 10.6. This kind of mapping influences the profile generation for the C2C operator, because for the reference *y*, no tagged value has to be generated. This is due to the fact that this reference is directly representable in UML via the path from class *A* to class *B* via the association class *AB*.

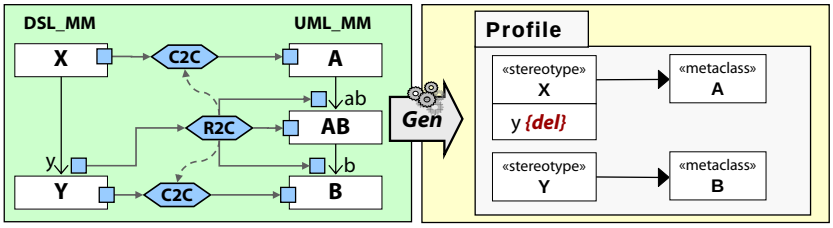

Figure 10.6: Profile Generation Rule 6 - R2C Operator

Profile Generation Rule 7 - C2R Operator. For each DSL class which is mapped with a C2R mapping, no stereotype has to be generated. Furthermore, also for the participating references of the C2R operator no tagged values are required in the profile definition. However, for unmapped features of DSL classes mapped with a C2R mapping, additional tagged values are needed for the stereotype generated out of the C2C mapping.

Example. For the application of the generation rule for the C2R operator consider the example in Figure 10.7. The DSL classes *X* and *Y* correspond directly to classes on the UML side, but the DSL class *XY* which connects the DSL classes *X* and *Y* corresponds to the reference *b* on the UML side. The attribute *att1* of the DSL class *XY* has no counterpart on the UML side.

The resulting profile is shown at the RHS of Figure 10.7. The C2R operator ensures that for the references *xy* and *y* no tagged values are generated. However, for each unmapped feature of the class *XY* (cf. *XY.att1*), an additional tagged value (cf. *X.XY_att1*) has to be generated for the stereotype generated out of the C2C mapping on which the C2R operator depends. For the class *XY* itself, no stereotype has to be generated, because no corresponding class on the UML side is available. Thus, the stereotype would not be applicable on UML elements.

Profile Generation Rule 8 - A2R Operator. The A2R operator influences the profile generation in the following way. For the *id* attribute, a tagged value is generated which is owned by the stereotype generated out of the C2C mapping, except the *id* attribute is mapped with an additional A2A operator to an UML attribute. For the *idref* attribute, no tagged value has to be generated, because the value-based relationship can be expressed in plain UML via a simple link.

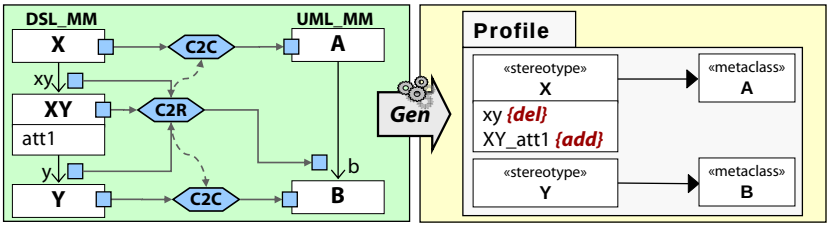

Figure 10.7: Profile Generation Rule 7 - C2R Operator

Example. For the application of the generation rule for the A2R operator consider the example illustrated in Figure 10.8. The DSL classes *X* and *Y* have corresponding classes on the UML side, however, the reference *b* between the UML classes *A* and *B* is on the DSL side expressed through value-based relationships (cf. attributes *id* and *idref*). Therefore, the attributes of the DSL side are mapped via an A2R mapping to the reference *b* on the UML side.

The resulting profile is shown at the RHS of Figure 10.8. The *id* tagged value of the stereotype *Y* is necessary, because the attribute *id* of the DSL class *Y* has no additional A2A mapping. However, the *idref* tagged value of the stereotype *X* has to be eliminated, because in UML the relationship between instances of class *A* and *B* should be set via the reference *b* and not via *id/idref* tagged values.

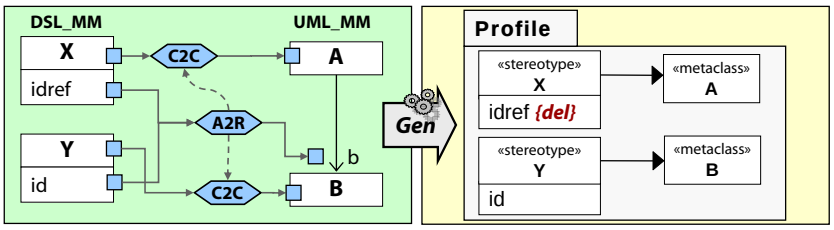

Figure 10.8: Profile Generation Rule 8 - A2R Operator

Profile Generation Rule 9 - R2A Operator. First of all, it has to be noted that this case does not occur when DSLs are integrated with UML, because the UML metamodel is carefully engineered and therefore all relationships are explicitly modelled as references. However, for completeness of the description how CAR mapping operators can be used for profile generation, we shortly discuss also this case[4]. In particular, only tagged values generated

[4]At least, generation rule 9 can be applied for metamodels which are aware of profile extensions and use value-based relationships.

for references which are participating in R2A mappings have to be deleted from the profile definition.

Example. For the application of the generation rule for the R2A operator, consider the example in Figure 10.9. The DSL classes *X* and *Y* have corresponding classes on the UML side, however the reference between them corresponds to the *id/idref* attribute combination on the UML side.

The result for the generation rule is shown at the RHS of Figure 10.9. For the reference *y* between the DSL classes, no tagged value should be generated, because relationships between instances of class A and B can be directly expressed in UML via *id/idref* attributes.

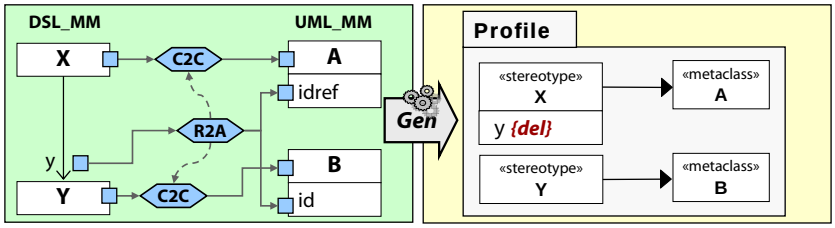

Figure 10.9: Profile Generation Rule 9 - R2A Operator

10.4.1.3 Additional Profile Generation Rules

In this subsection, we discuss additional profile generation rules which are necessary to fulfill the aforementioned qualitative criteria for generated profiles. These additional generation rules concern unmapped subclasses, constraints, bi-directional references, derived properties, and refined feature mappings.

Unmapped Subclasses. As for the generation of transformations out of CAR mapping models, it has to be clarified how the inheritance relationships between C2C operators and their configuration settings influence the profile generation. In order to get working transformations, the same constraints as for the core CAR mapping language must be fulfilled by the mapping models as discussed in Section 6.3. In addition to these constraints, it also has to be discussed how unmapped subclasses of mapped superclasses influence the profile generation.

Furthermore, it has to be distinguished, if a stereotype is defined as a concrete stereotype or as an abstract stereotype. A concrete stereotype is applicable on the instances of the corresponding UML metaclass. In contrast, an abstract stereotype is not applicable, but only its concrete substereotypes are. This means, an abstract stereotype can only be used for grouping common properties of its substereotypes. We decided to generate concrete

stereotypes for C2C mappings where the DSL class is concrete, because on the UML side, corresponding objects have to be annotated with an appropriate stereotype. Thus, the generated stereotype must be applicable on instances. In cases where the DSL class is abstract, an abstract stereotype is generated. This design decision is also the basis for the next generation rule concerning unmapped subclasses of mapped superclasses.

We decided against concrete stereotypes for abstract superclasses which can be used for annotating indirect instances for which no specific mappings are available, because when moving back from UML models to DSL models, there is no way to determine how to split the big set of objects all annotated with the same stereotype of the superclass into subsets according to the subclasses of the DSL metamodel. Furthermore, for ensuring that no information is lost during roundtrip, tagged values are required also for features of unmapped subclasses. Therefore, we decided to generate for unmapped subclasses of mapped superclasses substereotypes which extend the stereotype generated for the mapping between the superclasses. However, these substereotypes only inherit the extension relationship to the metaclass of the superstereotype, i.e., no specific metaclass is given for such stereotypes, and for each feature of the subclasses a tagged values is produced.

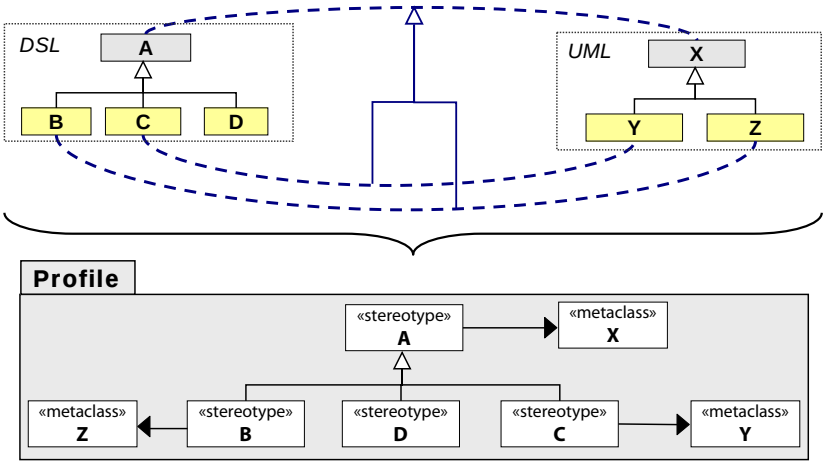

Figure 10.10: Stereotype Generation for Unmapped Subclasses

The upper half of Figure 10.10 shows an example for the generation of stereotypes for mapped superclasses, mapped subclasses, as well as for unmapped subclasses. The DSL class *A* is mapped to the UML class *X* with a C2C mapping which is also the supermapping for the C2C mappings between class *B* and *Z* as well as class *C* and *Y*. The DSL class *D* remains unmapped. For this mapping model, the profile is generated as depicted in the lower half of Figure 10.10. First, for the supermapping the stereotype *A* is generated. Second, for the submappings the stereotypes *B* and *C* are created which have refined extension

relationships to UML metaclasses. Third, for the unmapped class *D*, a stereotype is generated which has no refined extension relationship, but may have tagged values generated for properties of the DSL class *D*.

Constraints. For A2A and R2R mappings, additional reasoning capabilities have to be provided. This is needed, because features themselves have various properties, such as data types and multiplicity constraints. In cases where an upper multiplicity greater than one is allowed, unique and ordered constraints can be defined. Furthermore, this means that for mapped features the properties or constraints should be not contradicting. However, there are many possible mapping cases where this requirement cannot be ensured, because constraints are differently defined on the DSL and on the UML side. In particular, the following two cases have to be distinguished when generating profiles out of mapping models, namely, the DSL is more restrictive compared to UML or the opposite case, namely UML is more restrictive.

- **Stricter DSL constraints**: First, in the DSL metamodel, there can be more restrictive constraints than defined in the UML metamodel. For example, in the DSL metamodel a reference named *superEntity* with multiplicity *zero-to-one* is mapped with an R2R mapping to a reference *superClasses* of the UML metamodel having as multiplicity *zero-to-many*. In such cases, OCL constraints can be defined for profile elements to ensure that stricter constraints are checked on the UML models which have the profile applied. In particular, with OCL constraints it is possible to restrict the usage of UML. For our previous example, the following OCL constraint can be automatically generated which must be checked for each stereotype application:

  ```
  context Class:
  inv1: superClasses.upper = 1
  ```

- **Stricter UML constraints**: In the DSL metamodel, there can be less-restrictive constraints than in the UML metamodel. For this, we can just invert the previous example, i.e., assume that in the DSL metamodel a reference with multiplicity *zero-to-many* exists which is mapped with an R2R mapping to an UML reference having a multiplicity of *one-to-one*. For such cases, OCL constraints cannot be used, because it is only allowed to further restrict UML with profiles as also stated in the UML superstructure specification [OMG05e] (page 649):

  > *"A profile must provide mechanisms for specializing a reference metamodel (such as a set of UML packages) in such a way that the specialized semantics do not contradict the semantics of the reference metamodel. That is, profile constraints may typically define well-formedness rules that are more constraining (but consistent with) those specified by the reference metamodel."*

Contradicting this requirement would mean that we build UML models which are no longer conform to the UML metamodel. Fortunately, such mapping situations where the constraints on the UML side are more restrictive than on the DSL side are very rare, because most constraints in the UML metamodel are very general, e.g., most of the multiplicity constraints are zero-to-many.

Additionally to the constraints defined directly in the metamodel, OCL constraints can be applied to ensure further well-formedness rules which have also to be considered for the generation of OCL constraints for an UML profile out of a DSL metamodel. These additionally defined OCL constraints for the DSL must be also ensured from the generated UML profile. In particular, the OCL constraints of the DSL metamodel must be converted into constraints which use the corresponding terms of the UML profile. More specifically, this means that constraints defined for a class of the DSL metamodel must be converted into constraints for the corresponding stereotype, whereby actually constrained features of the DSL class are on the UML side either properties of the UML metaclass or generated tagged values.

Bi-directional References. In Ecore-based metamodels, it is possible to define bi-directional references. More specifically, such references are defined as two uni-directional references, whereby both references point to each other via a so-called *eOpposite* reference. Setting two uni-directional references can be regarded as an additional constraint on the model level, namely if an object *o1* has a link to object *o2*, then *o2* must also have a link to *o1*.

However, defining bi-directional references with two uni-directional references introduces various mapping problems, e.g., one can define a R2R mapping for one reference but not for the *eOpposite* reference. When we look at mapping models which consists of bi-directional references, the following cases can occur which are summarized in Figure 10.11.

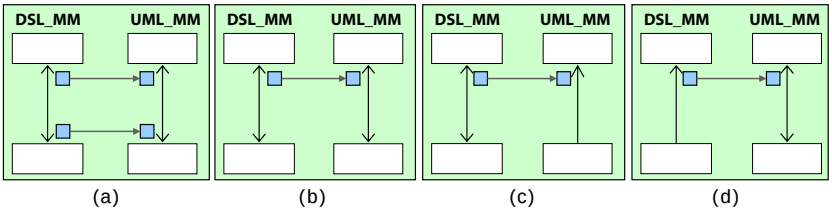

Figure 10.11: Mapping Cases Involving Bi-directional Associations

In the first case shown in Figure 10.11(a), a bi-directional reference is mapped to a bi-directional reference on the UML side. This case is not problematic, because no tagged values have to be derived and the transformation of instantiated links is straightforward.

Furthermore, the same constraints are ensured on the UML side, i.e., it is assumed that the references have the same multiplicities. If this is not the case, the constraint generation described in the previous paragraph comes into play.

In the second case shown in Figure 10.11(b), on both sides bi-directional references exists, but only one reference is mapped. For such cases, a warning must be given to the user, because it is not clear why only one reference is mapped. In most cases, the other reference should also be mapped. However, for all other cases, it is necessary to adapt the tagged value generation. In particular, for the unmapped reference on the DSL side, either no tagged value is generated or a tagged value is generated which is derived as the inverse reference of the mapped reference.

In the third case shown in Figure 10.11(c), on the DSL side, a bi-directional reference exists and on the UML side only an uni-directional reference corresponds to the bi-directional one, in particular only to one reference end. For such cases, either no tagged values are produced for the unmapped reference end or a derived tagged value is generated. Furthermore, the multiplicity of the unmapped reference end of the DSL has to be considered. Because of the missing reference end on the UML side, there is no restriction of the multiplicity of the missing end. This means, only for unrestricted references, no additional constraints are necessary. For all other cases, additional OCL constraints have to be generated for ensuring that the linking of objects on the UML side offers exactly the same possibilities and restrictions as on the DSL side.

The fourth case shown in Figure 10.11(d) is that on the DSL side a uni-directional reference exists, which corresponds to a reference being part of a bi-directional association on the UML side. In principle, no problems are encountered for the profile generation. However, problems can occur in the model transformation execution when on the UML side the unmapped reference has a cardinality different from *zero-to-many*. For example, if the multiplicty is *one-to-one*, then the problem occurs that on the DSL side models may exist that cannot be represented in UML, e.g., one object is not linked at all or a single object is linked by two other objects at the same time.

Derived Properties. In the UML metamodel, several properties are set as derived properties, i.e., meta-property *derived* is set to true meaning that these properties cannot be set directly, instead they are computed from other properties. An example in the UML metamodel is the reference *nestedPackage* of the class *Package* which actually represents a derived reference, namely the owned members of a *Package* that are of type *Package*. This reference is computed as a subset of the reference *packagedElement*:

```
nestedPackage:= packagedElement->select(e|e.oclIsTypeOf(Package));
```

Feature mappings to derived properties must be especially treated in the generation process, because when generating the transformations normally the feature mappings result in

property assignments which lead to runtime errors in ATL[5]. However, derived properties are appropriate for the query parts of transformations. They cannot be set, but they can be accessed. In our example, the *nestedPackage* reference can be used for selecting only nested packages out of all owned elements for defining specific feature mappings for this derived reference. This means, for derived properties which are subsets of other properties, when moving from DSL to UML, the superset property has to be filled in the generation part[6], but when moving from UML to DSL, the subset can be used in the query part directly. Therefore, the mapping model needs to be validated, if no mapping points to a derived property, which is different from subset relationships of the UML metamodel.

Superclass/Subclass Feature Mappings. In CAR mapping models it is allowed that features of superclasses can be mapped in submappings between subclasses as shown at the left hand side of Figure 10.12. However, when it is required to derive stereotypes from such mapping models as shown at the RHS of Figure 10.12, the problem arises that for the superstereotype (generated for the super C2C mapping) a tagged value is generated for the attribute *x2*, which is unmapped in the context of the superclass, but mapped in the context of the subclass. For the substereotype, however, no tagged value is necessary for the attribute *x2*, because an A2A mapping is available, meaning that a corresponding attribute exists in plain UML. For ensuring that for a single attribute of the DSL metamodel only one value can be specified within an UML model, we have to redefine the tagged value[7] as derived, where the value is exactly the same as the value of the UML attribute *b1*. Note that for the user it is not possible to set derived tagged values directly, because in UML, derived properties are automatically readonly.

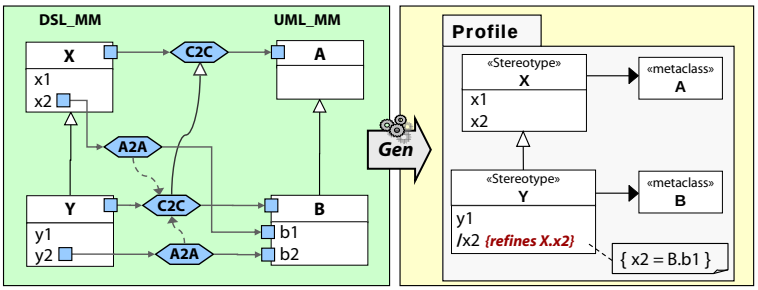

Figure 10.12: Generation Rule for Superclass/Subclass Feature Mappings

[5]Derived features cannot be set directly by ATL. In such cases, an exception is thrown and the transformation execution is aborted.

[6]ATL allows to use several assignments for one target property by automatically computing the union of all assigned elements.

[7]In UML 2 it is possible to redefine tagged values, because a tagged value is actually a property which can be redefined.

10.4.2 Model Transformation Generation

After derivation of the UML profile, the model transformations can be generated. Due to brevity reasons we will not discuss the model transformation generation in detail, but give an overview of the main characteristics of the generation process.

Generation Rules for Symmetric Operators. As already mentioned, we generate model transformation rules for mapped classes and for unmapped subclasses of mapped superclasses which transform instances of the source class to instances of the target class. If the target model is an UML model we must also apply the profile. Therefore, we have to check if the C2C mapping is the *rootMapping* (*C2C.isRoot = true*) in order to apply the profile to the generated UML element (a direct or indirect instance of the class *Package*). This is required in order to be able to assign the corresponding stereotypes to the generated objects and to set tagged values for unmapped DSL features. When transforming attribute values, we have to take care that enumerations are treated correctly, i.e., if the enumeration is not mapped then the generated literal has to be assigned, else the mapped literal has to be assigned.

Listing 10.2: Transformation Generation Template

```
rule c2c.getLeft.name() + _2_ + c2c.getRight().name
   extends getRulesfor(c2c.superMappings) {
        from s : DSL!c2c.getLeft() [c2c.getLeftInv()]
        to t : UML!c2c.getRight() (
          //set attributes/references for mapped features
     )
    do{
          if(c2c.isRoot()){ t.applyProfile(getLeftModel.name); }
          t.assignStereotype(c2c.getLeft().name);
          //set tagged values for unmapped features
    }
}
```

10.4.3 Validating Mapping Models

In order to ensure that only useful UML profiles and working model transformations are derived from a mapping model, certain constraints should be checked before the generation process is actually started. It has to be noted that it is not possible to represent the following constraints directly in the metamodel, therefore OCL constraints are used to define additional well-formedness rules. In the following, we present some OCL constraints to give the reader an idea which kinds of problems can be automatically detected in the mapping model.

For deriving working ATL transformations, OCL constraints are needed for the CAR mapping language as presented in Chapter 6. For example, basic OCL constraints can be defined for verifying if it is possible to derive working transformation rules from A2A mappings. Two attributes mapped with an A2A mapping should be compatible with respect to their meta-properties. For example, it can be verified if both attributes have the same,

compatible, or at least partly compatible data types. This can be ensured by the following OCL constraint that checks if the type of the attribute from the DSL metamodel (cf. `self.leftEnd`) is the same type or a subtype of the type of the attribute from the UML metamodel (cf. `self.rightEnd`).

```
context A2A:
inv: self.leftEnd.type.oclIsKindOf(self.rightEnd.type)
```

Finally, there are some constraints that need to be checked directly on the mapping model. For example, an OCL constraint is needed for verifying that each mapping model has at least one root mapping. This is required, because for root mappings, transformation rules are produced that apply the UML profile on the UML model. Without a root mapping, no UML profile application is achieved within the model transformation, consequently no stereotypes can be applied on UML model elements. In such cases, the execution of the transformation results in a runtime error. Therefore, the following OCL constraint should be evaluated before the generation process is started. In particular, the constraint checks if at least one C2C mapping exists in the mapping model that has set the *isRoot* attribute to true.

```
context MappingModel:
inv: self.C12C1-Mappings -> exists(m|m.isRoot)
```

10.5 Implementation Architecture of the ProfileGen Framework

We have developed a prototype called "ProfileGen" which supports our approach for bridging DSLs with UML. First of all, we decided to build on existing components available in the Eclipse[8] platform. As already mentioned before, as model repository we use the EMF. As mapping environment we use the ATLAS Model Weaver[9] which is built on EMF. The Eclipse UML2 project[10] is used as UML framework for two reasons, first, this project provides a complete implementation of the UML 2.0 metamodel, and second, it is compatible with several common UML tools such as IBM Rational Software Modeler or Magic Draw. Finally, we use the ATLAS Transformation Language[11] (ATL) for transforming the DSL models into UML models and vice versa.

As can be seen in Figure 10.13, the ProfileGen prototype allows to build mapping models between arbitrary DSL metamodels and the UML metamodel which can be validated with

[8] www.eclipse.org
[9] www.eclipse.org/gmt/amw
[10] www.eclipse.org/uml2
[11] www.eclipse.org/m2m/atl

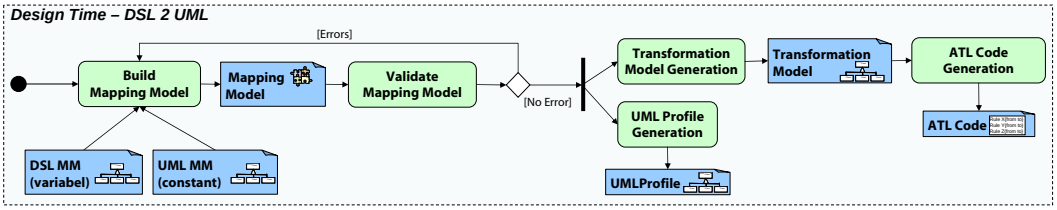

Figure 10.13: Tasks Supported by ProfileGen

respect to the abstract syntax defined with the metamodel together with OCL constraints. If errors exists in the mapping model, then the user has to correct the mapping model, else the automatic generation process can be started. More specifically, the profile is generated and in addition, a transformation model is created from which the ATL transformation code is derived. We decided to use an explicit transformation model instead of directly generating ATL code, because it allows to support further transformation languages in the future, such as the operational part of the QVT standard.

10.5.1 Mapping Editor

In Figure 10.14, a screenshot of the mapping editor for defining the mapping models is shown. As mentioned before, the mapping editor is based on AMW and uses a dedicated bridging language for mapping elements of metamodels which are defined in Ecore.

The user interface of the mapping editor is divided into the following four parts:

1. On the LHS, the elements of the DSL metamodel are shown in a tree viewer.

2. On the RHS, the elements of the UML metamodel are displayed.

3. The mapping model is placed in the middle and comprises user-defined mappings between the DSL metamodel and the UML metamodel. The user can create new mappings via the context menu. Via drag-and-drop it is possible to define the mapping ends.

4. In the left lower part of the mapping editor, a property view is used to define additional properties of mappings, such as inheritance relationships between mappings as well as root or abstract properties of mappings.

5. In the right lower part of the mapping editor, a problem view displays errors existing in the mapping models, found by the validation support. Note that these errors have to be corrected before the generation process can be started.

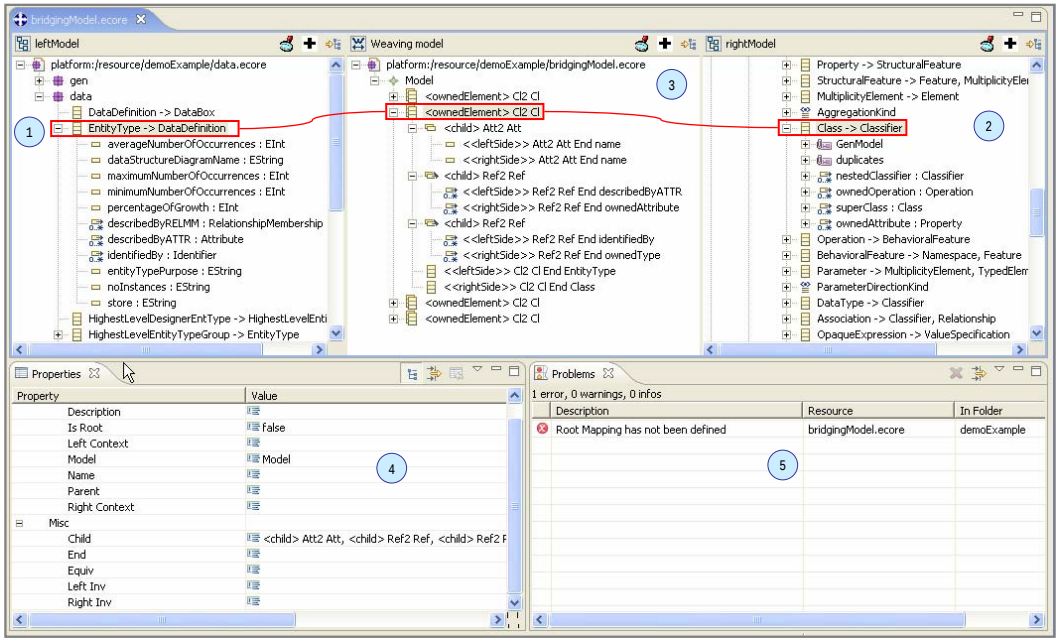

Figure 10.14: Mapping Editor Based on AMW

10.5.2 Executing DSL2UML Bridges

As already mentioned, we did not find model transformation languages and execution engines capable of using UML profiles as language definitions which extend the base UML metamodel. Hence, currently the only way to make use of UML profiles in model transformations is that the profiles are additional input models for the model transformation.

Figure 10.15 illustrates this problem for both transformation directions. The left hand side shows the runtime configuration for transforming DSL models into UML with ATL. The user has to define the DSL model as first input model, and the UML profile as second input model. The output model is the UML model, which both conforms to the UML metamodel and to the UML profile, which also conforms to the UML metamodel. This configuration reveals the problem, that UML profiles are located on the M1 layer (because they conform to the UML metamodel being located at the M2 layer), and also on the M2 layer (because UML models are located at the M1 layer and instantiate the profile, which is thereby located at the M2 layer). The work-around of reading UML profiles as additional input models hampers the definition of model transformations. In particular, stereotypes cannot be used as type definitions in query and generation patterns, and furthermore, profile and stereotype applications as well as setting tagged values can only be done in the imperative parts of ATL transformation rules. Since ATL only supports tracing for its declarative part, it is not pos-

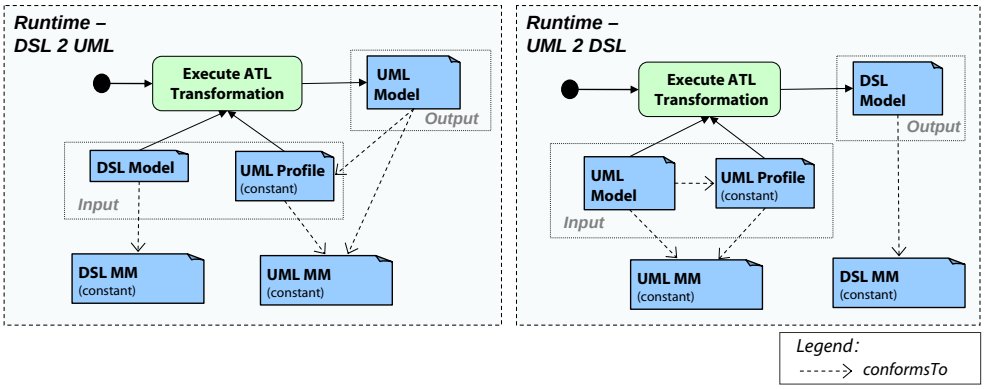

Figure 10.15: ATL Runtime Configuration

sible to automatically provide trace models for objects generated in imperative code blocks of ATL.

10.6 Case Study

In this section we present our approach within a case study for bridging parts of the ComputerAssociate's DSL of the *AllFusion Gen* CASE tool and IBM's *Rational Software Modeler* which implements the UML 2.0 standard. First, we briefly describe the involved metamodels, then, we give an overview of the mappings, and finally, we present the details for one particular C2C mapping. Further details of the case study can be found on our project's web site[12] including the details for all C2C mappings.

10.6.1 AllFusion Gen's Data Model

The metamodel for the data model of AllFusion Gen is illustrated in the package *DataModel* of Figure 10.16 and contains concepts that allow modeling the data used by the applications. Since AllFusion Gen's data model is based on the ER model, it supports ER modelling concepts like *EntityTypes*, *Attributes*, and *Relationships*. In addition to the ER modeling concepts, a grouping mechanism in terms of *SubjectAreas* can be used that is allowed to contain *EntityTypes* as well as further *SubjectAreas*. *EntityTypes* can have zero-or-one supertype. Furthermore, two concrete subtypes of the abstract *EntityType* concept can be distinguished, namely *AnalysisEntityType* and *DesignEntityType*. As already mentioned, AllFusion Gen is typically used for modeling data intensive applications which make excessive use of database technologies. Therefore, the data model allows the definition of platform specific information

[12] www.modelcvs.org/prototypes

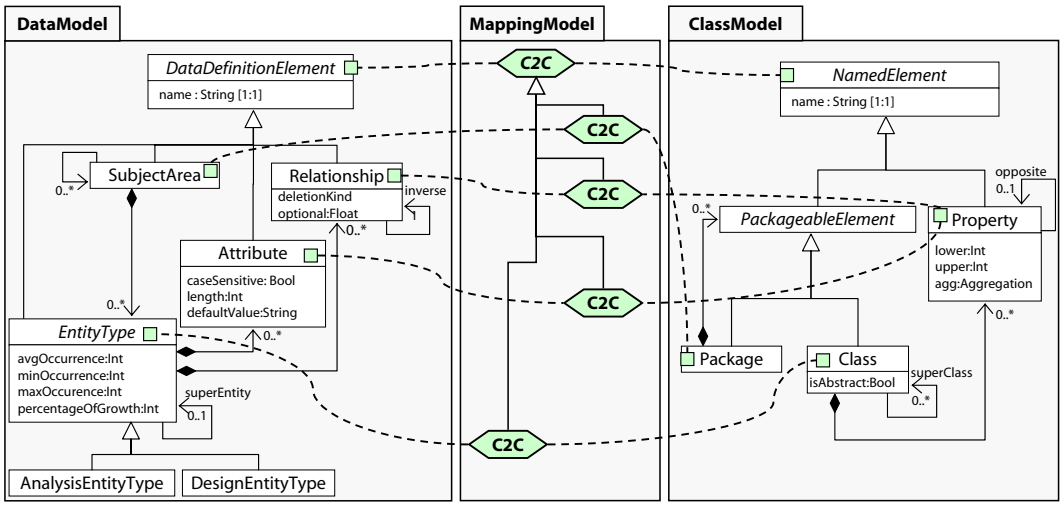

Figure 10.16: C2C Mappings at a Glance

typically usable for generating optimized database code, e.g., *EntityTypes* have special occurrence configurations.

10.6.2 UML Class Diagram

It is obvious that the corresponding UML model type for AllFusion Gen's data model is the class diagram. In this work we only present those part of the UML metamodel which is relevant for integration purposes. The metamodel excerpt is shown in Figure 10.16 in the package *ClassModel*. In UML, *Packages* can contain further *Packages* as well as *Classes*. *Classes* can be defined as either abstract or concrete and can have properties as well as an arbitrary number of superclasses. *Properties* represent attributes if the opposite property is not set, or role ends if the opposite property is set.

10.6.3 Overview of the Mapping Model

In this subsection, we present an overview of the mappings (cf. package *MappingModel* in Figure 10.16), covering only the C2C mappings. Both metamodels make use of inheritance which results in abstract superclasses containing the *name* attribute, only. In order to allow for reuse of mapping information in sub mappings and to minimize the number of feature mappings, the abstract classes are mapped with an abstract C2C mapping which is used as super mapping for all other C2C mappings. *SubjectArea*, *EntityType*, *Attribute*, and *Relationship* are mapped to *Package*, *Class*, and *Property*, respectively. While the first two mappings are obvious, the last two mappings have both the same target class. This is due the fact that

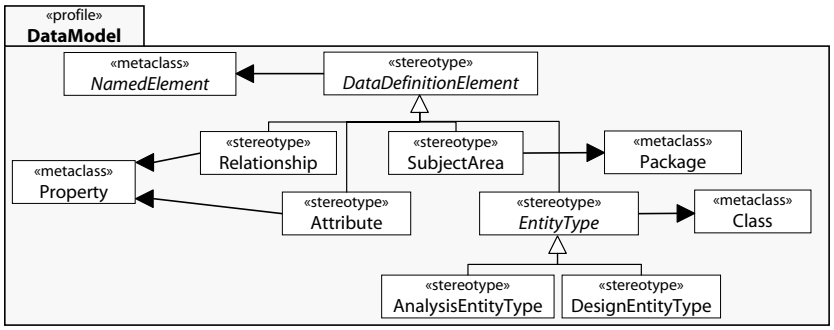

Figure 10.17: Stereotypes for C2C Mappings

UML does not distinguish explicitly between attributes and relationships in the metamodel. *AnalysisEntityType* and *DesignEntityType* are not mapped to UML metamodel elements, because both concepts would be mapped to *Class* in UML. Hence we decided to reuse the mapping of the abstract *EntityType* class in order to infer that both subclasses should be also mapped to *Class*.

In Figure 10.17, the resulting stereotypes for the C2C mappings are shown. Besides concrete stereotypes, two abstract stereotypes have been produced: ≪DataDefinitionElement≫ is the superstereotype for all others and ≪EntityType≫ has two concrete substereotypes corresponding to the subclasses of the metaclass *EntityType*.

10.6.4 EntityType_2_Class Mapping in Detail

The mapping details for the C2C mapping between *EntityType* and *Class* are illustrated in the upper half of Figure 10.18. Both concepts support inheritance, but with the difference that *EntityTypes* only support single inheritance and *Classes* allow multiple inheritance. Despite this difference, the two references are mapped with a R2R mapping because they support a similar feature. Furthermore, *EntityTypes* can have attributes and references, whereas UML classes can have properties which can be divided into two distinct subsets. As discriminator, the opposite reference is used, whereas the property is either an attribute if the opposite value is null or a reference if the opposite value is not null. Thus, we can map the *attribute* and *relationship* references to the *property* reference with two R2R mappings with specific OCL conditions.

10.6.5 Profile Generation

The profile details resulting from the *EntityType_2_Class* mapping is shown in the lower half of Figure 10.18. An abstract stereotype ≪EntityType≫ is generated for the abstract meta-

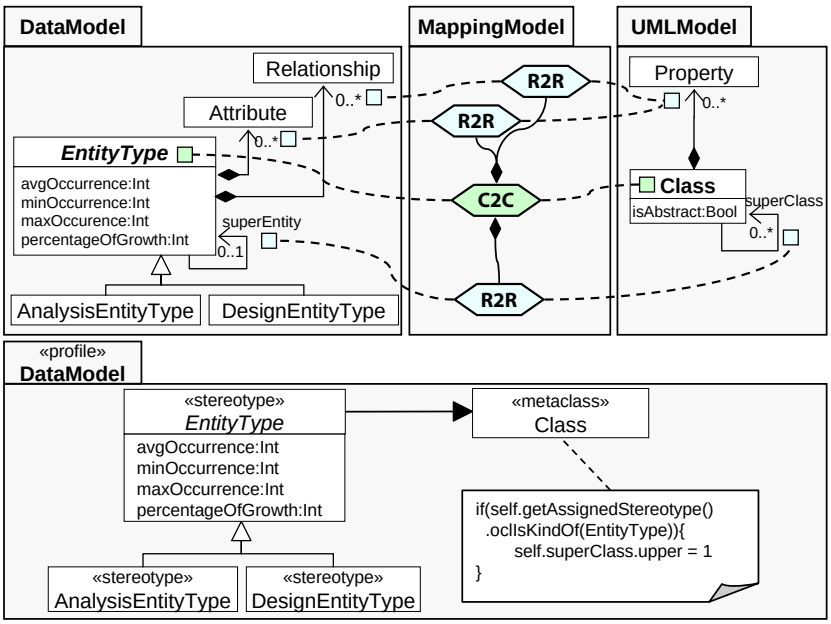

Figure 10.18: EntityType_2_Class Mapping

class *EntityType*. Furthermore, the metaclass EntityType has four platform specific attributes which are not available in UML. These attributes are represented in the ≪EntityType≫ stereotype as tagged values. Finally, the multiple inheritance mechanism of UML classes is restricted by assigning a special OCL constraint to the metaclass *Class*.

10.6.6 Transformation Generation

A simplified excerpt of the resulting ATL code is shown in Listing 10.3. An abstract transformation rule (cf. first rule in Listing 10.3) with three reference mappings is generated. However, the current ATL version does not allow to define *do* blocks for super rules, thus, *feature to tagged value* mappings must be defined in concrete sub rules. In fact, two concrete sub rules are generated, one for transforming *AnalysisEntityTypes* (cf. second rule in Listing 10.3) and one for *DesignEntityTypes*, implementing the *feature to tagged value* mappings - which are also defined for unmapped features of the superclasses, e.g., the *avgOccurrence* attribute.

10.6.7 Discussion

In this subsection, a discussion on the evaluation of the generated bridge between the AFG tool and the UML tool is provided and shall give an indication on the applicability of the

semi-automatic DSL2UML bridging approach. This evaluation is conducted, first, with respect to roundtrip capabilities of the generated bridge based on the integration scenario 2 presented in Section 8.2, and second, on the basis of certain metrics for the manually defined mapping model as well as for the generated artifacts, i.e., the profile and the model transformations.

Listing 10.3: Resulting ATL Code

```
module DSL2UML; create OUT:UML from IN:DSL;

abstract rule EntityType_2_Class {
   from s : DSL!EntityType
   to t : UML!Class (
        property <- s.attribute,
        property <- s.relationship,
        superClass <- s.superEntity
   )
}

rule AnalysisEntityType_2_Class extends
    EntityType_2_Class{
   from s : DSL!AnalysisEntityType
   to t : UML!Class
   do{
    t.assignStereotype(AnalysisEntityType);
    t.setTaggedValue("avgOccurrence",
        s.avgOccurrence);
    ...
    }
}
```

Integration Scenario 2 Revisited. In Section 8.2, the presented motivating scenario 2 summarized information loss for an input model which conforms to the afore presented metamodel of AllFusion Gen's Data Model. In this scenario, no UML profiles and adapted model transformations have been used. Now, the model metrics are again presented in Table 10.1, however, now the afore presented profile definitions and model transformations have been used for preserving information during roundtrip. As one can see in the most right column of the table, no difference regarding to the number of model elements between the initial AFG model and the AFG model after roundtrip exists. The information could be preserved by applying for each corresponding object an appropriate stereotype allowing to use tagged values for saving feature values which otherwise would be lost. Furthermore, a comparison of the initial AFG model and the resulting AFG model using the comparison facility of EMF Compare[13] demonstrated that both were equivalent, thus no information has been lost during roundtrip.

Metrics for Mapping Model, Profile, and Model Transformations. In addition to discussing the roundtrip capability of the generated bridge based on an example input model,

[13] www.eclipse.org/modeling/emft/?project=compare#compare

Table 10.1: Model Metrics for Data Model/Class Diagram Roundtrip Revisited

Metrics	Initial AFG Model	Generated UML Model	AFG Model after roundtrip	Diff in %
#Objects	156	165	156	0
#Values	1099	156	1099	0
#Links	44	54	44	0
#Containment Links	155	164	155	0
File Size	32,8 KB	58,5 KB	32,8 KB	0
#Applied Stereotypes	-	156	-	-
#Tagged Values	-	951	-	-

the number of model elements of the manually created mapping model compared to the number of elements of the generated bridge artifacts is presented. This should give an indication on how much effort requires the creation of the mapping model and using the generator in contrast to building the desired bridge artifacts manually from scratch.

Table 10.2 summarizes some metrics for the mapping model (cf. Table 10.2(a)), for the generated profile (cf. Table 10.2(b)), and finally for the generated model transformations (cf. Table 10.2(c)). We decided to use as metrics for the mapping model, first, the number of applied mapping operators, and second, how many non-default property values have to be set by the user, because this is exactly the work the user has to do for creating the mapping model. For the profile, we use as metrics the number of profile elements the user has to create, i.e., how many elements have to be imported from the UML metamodel (used as types and metaclasses within the profile), how many stereotypes and tagged values have to be created, and finally, how many constraints and additional data types have to be defined within the profile. Finally, for the transformation code we simply use lines of code (LOC) as metric, just to give an indication how much effort the manual implementation of the transformations for both directions (DSL2UML, UML2DSL) would cause.

Interpreting these metrics, one can observe that the number of created elements of the mapping model is less than the number of elements of the profile definition. More specifically, one can see that even for the stereotype definitions alone, i.e., stereotypes, imported metaclasses, extension relationships, and generalization between the stereotypes, more elements have to be created (in sum 29 elements) than are needed for the whole mapping model (in sum 14 applied mapping operators). In addition, tagged values (which need additional property settings such as multiplicity or data types), enumerations with literals, data types, and constraints have to be defined.

Furthermore, for realizing model exchange between AFG and UML, 220 lines of ATL code are necessary, where most parts are declarative rules. However, for using profiles within the transformation, imperative code is needed, e.g., for applying profiles and stereotypes, for setting tagged values when moving from AFG to UML as well as for assigning tagged values as attribute values when we are going back from UML to AFG. Summarizing, it can be said that the generator based approach offers the possibility to derive a bridge,

i.e., profiles and transformations, from declarative mappings defined in an interactive environment. Compared to the tedious authoring of profiles by hand, e.g., supported by the Rational Software Modeler or by the Eclipse UML2 project, the ProfileGen tool offers faster development of profiles in cases where a metamodel is already existing.

Table 10.2: Bridge Metrics Overview. (a) Mapping Model Metrics, (b) Profile Metrics, and (c) Model Transformation Metrics

Manually Created

(a) Mapping Model Metrics

	Mapping Operator	***#***
Mapping Model	C2C	6
	A2A	1
	R2R	7
	Properties	7

Automatically Generated

(b) Profile Metrics

	Element Type	***#***
Profile	Element Import	8
	Stereotype	
	Abstract Stereotype	2
	Concrete Stereotype	6
	Relationship	
	Extension	6
	Generalization	7
	Data Type	
	Enumeration	1
	Literal	3
	Primitive Type	1
	Tagged Value	9
	Constraint	2
	Properties	27

(c) Model Transformation Metrics

	ATL File	***LOC***
Model Transformation	AFG2UML	120
	declarative	75%
	imperative	25%
	UML2AFG	100
	declarative	87%
	imperative	13%

Chapter 11

Summary and Related Work

Contents

11.1 Summary

In this part of the thesis, we have discussed the problem of information loss during roundtrip, which is a frequently occurring problem in tool integration practice. We further motivated that information loss is a cross-cutting concern, which is spread over the whole transformation and needs special treatment in the integration process. Therefore, we presented two approaches for establishing bridges between tools which are capable of saving information which is otherwise lost during roundtrip.

As first approach, we presented AspectCAR, a generic aspect-oriented extension of the CAR mapping language. With the help of AspectCAR the development of mappings representing semantic correspondences between metamodel elements can be separated from the development of mappings which are necessary to save information in annotations which would get otherwise lost during roundtrip. The approach has been presented with the help of a running example and finally some limitations have been discussed. It has to be noted that the user has full control over the mapping model with AspectCAR. This means, other concerns can be implemented as aspects as well, e.g., for logging information if there is a mismatch in abstraction, aggregation, or precision between source and target elements.

As second approach, we have introduced a semi-automatic approach for bridging DSL tools with UML tools. First, we presented a dedicated metamodel mapping language based on the CAR mapping language which can be used to build a mapping model between the DSL metamodel and the UML metamodel. Second, a set of rules for generating profiles out

of mapping models has been presented. With the help of these rules, it is possible to generate complete profile definitions, i.e., stereotypes, tagged values, enumerations, data types, and also constraints. Third, a discussion on the generation of ATL transformation rules for exchanging the models between the DSL tools and UML tools has been presented. Fourth, tool support has been developed for supporting our bridging approach which is built on existing components available on the Eclipse platform. We extended the ATLAS Model Weaver (AMW) in order to create mapping models in a more user friendly way, validate the mapping model for ensuring that only working bridge definitions are produced, and finally, for starting the automatic generation process. Further information about tool support can be found on our project site[1]. Finally, we presented a case study of the ModelCVS project and discussed the benefits of using a mapping model combined with a generator instead of manually implementing the bridge.

Summarizing, the presented approach allows for faster development and a higher maintainability of bridges between DSL and UML tools. The approach can also be applied for generating a profile out of a metamodel, only. Another application scenario would be that the generated profiles can be used to annotate already existing models, because UML profiles allow the dynamic extension of existing UML models. As soon as a UML model is marked with stereotypes and tagged values, it can be transformed into a DSL model which then can be the input for the DSL code generator component. This way, code generation facilities can also be supported for already existing UML models.

11.2 Related Work

With respect to the presented approaches for engineering information preserving roundtrip transformations, three areas of related work can be distinguished. First, approaches related to AspectCAR aiming at adapting existing model transformations, second, widely related work to ProfileGen, namely approaches dealing with the definition of DSLs on the one hand in terms of MOF-based metamodels and on the other hand using UML profiling, and third, closely related work to ProfileGen, namely establishing model exchange between DSL and UML tools.

11.2.1 Adapting Model Transformations

Adapting Model Transformations with Aspects. Closely related work to AspectCAR is presented in [VG07a, VG07b], where variability in model transformations and code generators is handled with aspect-oriented techniques. The authors propose to combine DSLs, aspect-orientation, and MDE in order to improve software product line engineering. In

[1] www.modelcvs.org

particular, the process starts with the definition of variability models, which drive the subsequent generation tasks. In order to realize variants of model transformations and code generators, aspect-oriented language concepts are offered by the model transformation language *Xtend* and also by the code generation language *XPand* of the openArchitectureWare framework[2]. Compared to AspectCAR, the presented approach in [VG07b] and [VG07a] is used to define variants of model transformations which are specific to a certain product line. On the contrary, we propose to use aspect-orientation to solve a more general problem, namely loss of information which occurs in each tool integration scenario. However, it has to be mentioned that in general, also the model transformation language *Xtend* could be used to define bridges between tools by using aspects for preserving information loss, whereas the resulting bridge is on a lower-level of abstraction which furthermore hampers the definition of aspects for ensuring roundtrip capability and reasoning on possible information loss.

Adapting Model Transformations with Model Transformations. Further related work to AspectCAR are approaches adapting a model transformation with the help of an additional model transformation[3], as presented for example in [Jou05]. The author proposes to use model transformations to introduce traceability issues in already existing model transformations where tracing is actually considered as a cross-cutting concern. The benefit of the proposed approach is that the traceability code is clearly separated from the rest of the model transformation code. Using a model transformation for adapting existing model transformations is similar to the AspectCAR approach, however, the pointcut expressions and adaptations are encoded and intermingled in the model transformation rules and are therefore not reusable in isolation.

11.2.2 Integrating DSLs with UML

Although many debates[4] on pros and cons of MOF-based metamodels and UML profiles are going on, only a few scientific work discussing DSL development with metamodels compared to UML profiles can be found and even less papers can be found on the integration of MOF-based metamodels with UML profiles. In this subsection, we discuss first widely related work concerning a comparison of different approaches for defining domain-specific modeling languages with OMG standards, and finally closely related work, namely two approaches concerning the visualization of DSL models with UML based on model trans-

[2] www.eclipse.org/gmt/oaw

[3] This mechanism is called *Higher-Order Transformation* (HOT), i.e., a transformation itself can be seen as a model and therefore can be input for another transformation called HOT, which again produces as output a transformation.

[4] For example, a very interesting debate about DSLs and UML profiles was a panel named *A DSL or UML Profile. Which would you use?* at the Models 2005 conference (http://www.cs.colostate.edu/models05/panels.html)

formations, as well as an approach for bridging DSLs defined as MOF-based metamodels with already existing UML profiles.

Comparing Modeling Approaches for defining DSLs. In several papers, the benefits and drawbacks of using MOF for defining metamodels from scratch or using UML profiles for tailoring UML have been discussed. For example, Weisemöller and Schürr [WS07] discuss the definition of a DSL for architectural description languages with the usage of UML profiles, the definition of a completely new language based on a MOF-based metamodel, and finally, by a heavyweight extension of the UML metamodel.

In contrast to our work, Weisemöller and Schürr only discuss as future work the combination of UML profiles with metamodels by using triple graph grammars for defining a bridge between DSL tools and UML tools. Besides bridging UML tools with DSL tools, Weisemöller and Schürr discuss some design alternatives how an example profile can be expressed as metamodels. This discussion could be a good starting point to develop a semi-automatic approach to transform UML profiles into MOF-based metamodels as pointed out as future work in Subsection 12.2.3 of this thesis.

Defining model transformations between DSLs and UML. In [GvD07], the authors propose to use UML as visualization technique for documenting domain-specific models. The motivation for using UML as visualization technique is that the development of graphical editors is still in its infancy, although there are some promising frameworks emerging such as the Graphical Modeling Framework (GMF). The authors also argument that it is not always necessary to have a graphical editor, e.g., the models can be also defined in concrete textual syntax (e.g., by employing the Textual Concrete Syntax framework [JBK06] or the Xtext framework [EV06]). However, other use cases, such as documentation of models, require some form of graphical representation. Therefore, Graaf and van Deursen propose to use model transformations to specify mappings between the DSL metamodel and the UML metamodel. With the help of the model transformations, the DSL models can be transformed into UML models which can be in turn visualized in UML tools. They exemplified their approach in the domain of software architecture, more specifically by using architecture description languages.

There are several differences to our approach. First, the authors propose to directly use model transformation languages for describing the mapping between the DSL and UML, i.e., they only focus on an uni-directional transformation. Second, they are not concerned of automatically deriving a profile for the DSL. If there is a semantic gap between the DSL and UML, stereotypes are directly generated within the transformation, thus the profile is more or less developed manually by the transformation engineer. Third, only a projection of the DSL model is achieved by the transformation, because for documentation purposes it is not necessary to represent each tiny detail of the DSL model.

Annotating DSL metamodels with UML notations. Brucker and Doser [BD07] propose an approach for annotating a DSL metamodel with concrete syntax information which may be seen as most closely related work to our approach. More specifically, the authors propose to use the UML notation as concrete syntax for graphical DSLs. Therefore, they use an OCL dialect to annotate the metaclasses, attributes, and associations in the metamodel and from these annotations model transformations can be derived. These model transformations can be used to transform DSL models into UML models annotated with stereotypes and vice versa. Moreover, the authors present how the user-interface of the UML tool ArgoUML[5] can be extended to provide a more concise presentation of the UML models.

Although the approach presented in [BD07] and ours target similar problems, the realization of both is different. First, we favor for an interactive environment based on a graphical environment, especially for searching for appropriate metaclasses used for a stereotype and for finding corresponding attributes and references. We believe that an interactive environment is required, because the UML metamodel is a very large metamodel which should be only explored on demand. Second, the approach of [BD07] has a strong coupling between the UML notation and the DSL metamodel. On the contrary, our approach has only a loose coupling between UML and the DSL, because the correspondences between them are stored in an external mapping model and not within the DSL metamodel. Third, we also allow to define mappings between data types, enumerations, and literals which are not considered in [BD07]. Fourth, in [BD07] it is not clearly stated, if an explicit profile definition is generated, as well as, if tagged values are generated or if all attributes and references of a DSL class have to be mapped to UML. Fifth, no constraints are generated for the UML models. The authors claim that this is not necessary because the constraints are defined through the metamodel. However, if a user builds an UML model by hand, constraints are necessary in order to check some well-formedness rules to ensure that, on the one hand, the model transformation to the DSL is working and on the other hand it seems to be tedious to transform an UML model into a DSL model, then checking the constraints, and finally, propagating identified errors back to the UML model. Therefore, we decided to incorporate all constraints of the metamodel also in the UML profile. Sixth, we are using a dedicated mapping language which consists of reusable mapping operators. In contrast, the authors in [BD07] propose to use a subset of OCL to describe the mapping between the DSL and UML with OCL queries which are not reusable except on basis of copy/paste. Finally, we employ the UML 2 framework for our ProfileGen framework instead of using a specific UML case tool as is done in [BD07] by using ArgoUML. We decided to use the UML 2 framework, because it provides a standard conform implementation of the UML 2 metamodel and several commercial and open source UML tools provide import/export functionalities for the UML 2 framework. Consequently, the UML 2 framework can be used as a common format which is not bound to a specific UML tool. Moreover, ArgoUML currently only supports UML 1.4

[5] http://argouml.tigris.org

providing only a restricted profile mechanism compared to UML 2.

Bridging DSL metamodels with UML profiles. In Abouzahra et al. [ABFJ05], the integration process starts with an already existing, probably manually defined UML profile and one has to define the mappings between the DSL metamodel and the UML profile elements. Subsequently, the model transformations between DSL models and UML models are automatically derived from these mappings. As stated in [ABFJ05], the proposed tool aims at automatically generating transformations between models and not on transforming a metamodel to a profile or the opposite. This means, our work is different in that we do not assume that an UML profile is available yet. Instead, we assume that the whole bridge, i.e., the transformations and the UML profile, has to be developed. Therefore, we generate not only the transformations between the DSL and UML, but also the UML profile is generated automatically.

Furthermore, the presented approach in [ABFJ05] does not use the standardized UML 2 metamodel with built in profile support, instead a proprietary profile metamodel has been used. Finally, also the proposed mapping language is quite different to our metamodel bridging language, in that sense that our language provides mapping operators with more specifically typed mapping ends. This allows more accurate validation of the mapping models which in turn ensures the generation of meaningful profile definitions and correctly working model transformations.

Chapter 12

Conclusion and Outlook

Contents

12.1 Major Contributions of the Thesis

In this thesis, various tool integration scenarios have been discussed and concepts, mechanisms, and tools supporting the establishment of tool bridges have been presented. Summarizing, it can be said that reuse of existing integration knowledge is crucial for providing a faster and more systematic development of tool bridges compared to using model transformation languages, only. It has to be noted that on the one hand, reuse may be achieved on several meta-layers, i.e., on the M3 and the M2 layer, and that on the other hand, integration knowledge may be generic, i.e., reusable for each integration scenario, but it may be also specific, i.e., reusable for a certain kind of integration scenario such as integrating domain-specific modeling tools with UML tools.

Concerning the first part of the thesis, reuse can be achieved by defining correspondences between the meta-languages of different technical spaces, from which transformations for producing metamodels and also models can be derived. This allows to provide generic adapters between technical spaces based on the M3-layer which can be reused to build specific adapters for pre-MDE modeling tools. Our proposed mining pattern and semi-automatic process further opens the door to the model technical space by showing how already existing languages definitions, e.g., residing in the XML technical space or in the grammar technical space, can be rebuilt as metamodels. Furthermore, the semi-automatic process allows to incorporate constraints into the metamodels, which are not expressed in

the source language definitions. The semantic enrichment of the language definitions is possible by reusing existing integration knowledge specific for a particular technical space combination by employing heuristics and a set of predefined user-annotations or refactorings.

The second part of the thesis addresses reuse of generic mapping operators which are used to resolve frequently occurring integration problems, e.g., structural metamodel heterogeneities, between two MOF-based metamodels. This kind of integration knowledge can be reused in nearly each integration scenario where two MOF-based metamodels have to be bridged. In our approach, the integration knowledge is encapsulated into mapping operators, thereby not only the syntax of the operators is defined, but also the operational semantics can be expressed in terms of Colored Petri Nets. Furthermore, we refrain from recommending only one single set of mapping operators which is intended to be capable of resolving each integration problem, instead we are proposing an open platform which supports the user by defining new mapping operators. To demonstrate the applicability of this framework, we provide a set of mapping operators subsumed in the CAR mapping language for resolving the prevalent form of structural heterogeneities.

The third part of the thesis focusses on how roundtrip transformations can be built in a systematic way. The first approach for ensuring roundtrip transformations is based on aspect-orientation. In particular, it has been shown how integration knowledge about roundtrip capabilities of transformations can be encapsulated into aspect definitions, which can be employed to adapt existing non-roundtripping transformations. These aspect definitions can be reused in various integration scenarios. The only precondition is that the CAR mapping language is used to define the tool bridges. The second approach is dedicated to the integration of domain-specific modeling tools with UML tools by using a generator-based approach for enhancing reuse of existing integration knowledge specific to this scenario. Instead of manually developing UML profiles for metamodels which describe DSLs, the user should indicate semantic correspondences between the elements of the DSL metamodel and the elements of the UML metamodel in terms of a mapping model. A generator component, which implements generation rules reflecting best practices how an UML profile should be built for a DSL metamodel, produces the bridge between the domain-specific modeling tools and UML tools.

12.2 Outlook

The work presented in this thesis leaves several issues open for further research. First, concerning Part I of this thesis, the presented DTD 2 Ecore framework can be used as foundation for building other technical space bridges, in particular for bridging XML schema with Ecore. Second, concerning Part II of this thesis, there are many issues for future work, such as extending the CAR mapping language by firstly, allowing new combinations of existing

mapping operators, and secondly, by proposing additional operators which further leads to the development of a methodology for finding mapping operators. Another issue is the automatic creation of mapping models between two metamodels, which has not been discussed in this thesis at all. Third, although the case study presented in Section 10.6 showed various benefits of using a semi-automatic approach for bridging domain-specific modeling tools with UML tools, various challenges concerning roundtrip transformations and the integration of metamodels with UML profiles lies ahead. In the following subsections, we elaborate on these open issues and future challenges in more detail.

12.2.1 Bridging Technical Spaces

DTD 2 Ecore as foundation for XML schema 2 Ecore

When XML schema is used for language engineering, it is not possible to generate an appropriate Ecore-based metamodel out of the XML schema without additional information. For example, within the Eclipse Modeling Framework, there is the possibility to define a metamodel with XML schema syntax, however, there are many "extensions" of pure XML schema constructs by using specific annotations, in particular, for every Ecore language feature that is not directly representable in XML schema, a specific annotation is available. However, when switching from already existing XML schemas to Ecore-based metamodels, these annotations are not provided. One possibility would be to annotate the XML schema with the proposed annotations of the Eclipse Modeling Framework, or to employ our semi-automatic approach for generating a first version of a metamodel, and then, enhancing the quality of the automatically generated metamodel by appling heuristics as well as refactorings. The benefit of the latter approach would be to have in addition to the generated metamodel, a bridge for transforming XML documents into models. For example, WebRatio uses XML schema for defining language extensions of the WebML language. To incorporate these extensions in the WebML metamodel, we developed a prototypical implementation of an XML schema 2 Ecore bridge in order to generate first metamodel drafts which have to be manually improved. However, this prototype can only be seen as a first step towards developing a complete bridging approach for XML schema and Ecore.

12.2.2 Mapping Metamodels

Extending the CAR mapping language

The first extension possibility of the CAR mapping language is the composition of existing mapping operators into composite operators. Such composite operators enhance the readability of the mapping model and allow faster development by avoiding the definition of trace model dependencies again and again. Figure 12.1 illustrates, how the C2C and A2C

operator can be composed into a CA2CC operator by nesting both individual operators into one component.

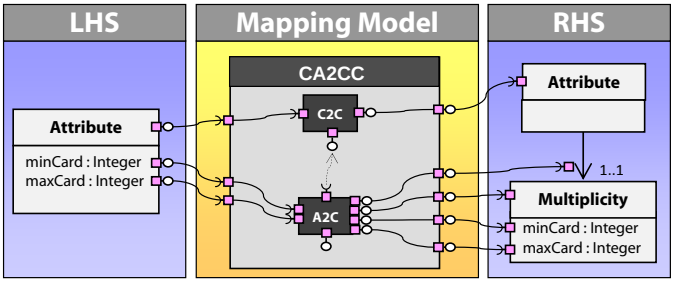

Figure 12.1: CA2CC Operator by Composing C2C and A2C Operators

The second extension is allowing new trace model dependencies between operators, or in other words, allowing other operators to provide their trace models as context for other mappings, in addition to the C2C operator. For example, the A2C and R2C mapping operators can provide the context for A2A and R2R mappings.

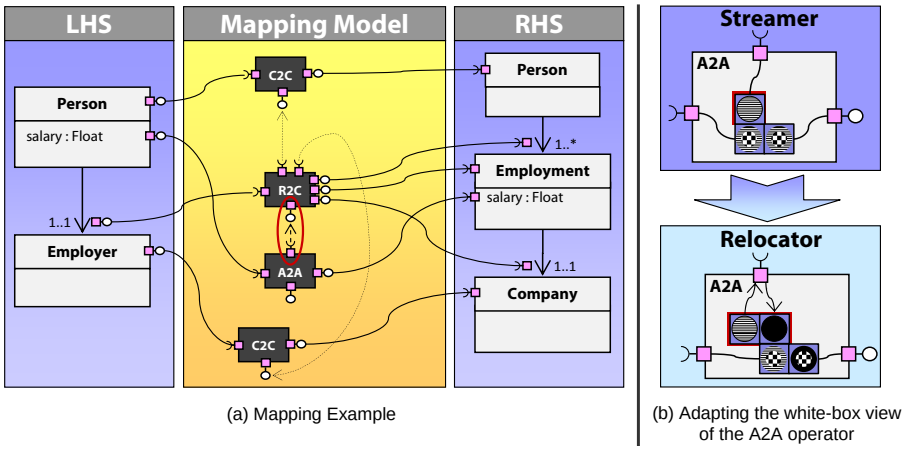

Figure 12.2: Using R2C Trace Models. (a) Mapping Example, (b) Adaptation of the A2A Operator

The example shown in Figure 12.2(a) requires the combination of an R2C with an A2A operator. Attention has to be given to the dependency between the R2C and A2A operators. In order to allow such kind of dependencies, the white-box view of the A2A operator as shown in the upper part of Figure 12.2(b) has to be adapted as shown in the lower part of Figure 12.2(b). The Relocator component has to provide an explicit query pattern token to

ask the trace model of the R2C operator which object color has been generated for the link which has the same outer color as the input value of the A2A component. For using the answer of the trace model within the A2A component, an explicit answer token pattern is introduced, which is used for the generator token as outer color.

The third extension possibility comprises finding additional mapping operators in a top-down approach as well as in a bottom-up approach. The CAR mapping language has been defined in a top-down approach by primarily studying existing database integration literature and investigating the MOF standard of the OMG. However, a bottom-up approach seems also promising to find new mapping operators. This can be done by developing transformation nets by hand and then "lifting" the transformation net components into mapping operators. For example, by building various transformation nets manually, we found out that mapping operators are quite often needed that generated elements in the target model which are not explicitly and even not implicitly represented in the source model, i.e., they cannot be produced out of elements from source models.

In Figure 12.3(a), an integration example is shown in that on the LHS only one class called *EntityType* exists, i.e., no root object representing the whole model is required on the instance level. On the RHS, a class *Model* exists which has a containment reference to the class *Class*. On the instance level, only one instance of *Model* is allowed to exist, which contains all instances of *Class*. This means, when moving from left to right, only one *Model* instance should be generated, but all instances of Class, which corresponds to EntityType instances, should be connected to the *Model* instance. For resolving this heterogeneity, we have introduced an *ObjectFactory* mapping operator (cf. Figure 12.3(b)), which is configurable, allowing to specify if for each transformed element of the context mapping, an object is generated or if only a single object is generated for all transformed elements. This configuration is possible via the *singleton* tag. In this example, it is necessary to set the *singleton* tag to true, because not for all classes an additional model object should be generated, but only for the first one. In case the singleton tag is set to true, the white-box view of the ObjectFactory has a place with only one initial token, standing for the singleton object which is streamed with the first input element from the contextObject port. When the following tokens are streamed from the contextObject port into the component, only the context2singleton link is generated.

Automatic establishment of mapping models

This thesis elaborated on the manual development of mapping models, only. However, when looking at the ontology or schema integration field, much effort is investigated into automatic matching approaches[1], which take two ontologies or schemas as input and automatically compute correspondences between them. It has to be noted that matching techniques is based on heuristics, thus the approaches do not claim for full completeness and

[1] For more information on matching techniques, the interested reader is kindly referred to the Ontology Alignment Evaluation Initiative (http://oaei.ontologymatching.org).

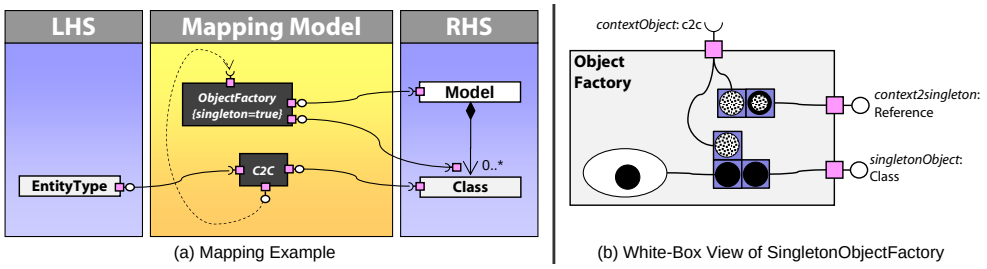

Figure 12.3: ObjectFactory Operator. (a) Mapping Example, (b) White-Box View

correctness of the mappings. We have picked up the idea of automatically computing a first draft of a mapping model, on the one hand, by using existing ontology matching tools in combination with a machine-learning approach, and on the other hand, by using a by-example based approach which also exploits the graphical notation of modeling languages. Both approaches, which are subject to future work, are presented in the following in more detail.

Smart Matcher. With the rise of the semantic web and the emerging abundance of ontologies, several automated matching approaches and tools have been proposed, for an overview see [RB01, SE05]. The typical output of such tools are simple one-to-one correspondences mostly based on schema information, e.g., similar names and structures of schema elements. However, these correspondences cannot cope with structural heterogeneities as is often the case between MOF-based metamodels, hence, these problems, on the one hand, lowers the quality of automatically computed correspondences[2], and on the other, must be resolved manually by defining additional transformation code. Furthermore, there is no automated evaluation of the quality of the alignments based on the instance level, because the matching approaches are not bound to a specific integration scenario, e.g., transformation or merge. In [KW08], we have proposed the SmartMatching approach, which can be seen as an orthogonal extension to existing matching approaches for increasing the quality of the automatically produced alignments for the transformation scenario. This is achieved by first, using the executable CAR mapping language for bridging structural heterogeneities, and second, by using concrete models to evaluate the quality of the alignments in an iterative and feedback-driven process inspired by machine learning approaches. Further steps are improving our current prototype, and the evaluation of the hypothesis that the preparation phase, i.e., building the test models, is less work than the reworking phase, i.e, validating the correspondences and programming additional transformation code which cannot be derived from simple mappings. Finally, we have to evaluate

[2]For an evaluation of existing ontology matching tools regarding their metamodel matching capabilities see [KKK+07].

our matching approach extension regarding completeness and correctness of the mappings with empirical experiments against already existing matching tools.

Model Transformation By Example. Current approaches for tool integration, as is also the case for the CAR mapping language and for the transformation nets, are only focussing on the abstract syntax in terms of MOF-based metamodels. However, this implementation specific focus makes it difficult for modelers to develop model transformations, because metamodels do not necessarily define all language concepts explicitly which are available for notation purposes. Therefore, in [WSSK07], we have proposed Model Transformation By Example, a by-example approach for defining semantic correspondences between graphical domain models which is more user-friendly then directly specifying correspondences between metamodels. The correspondences between domain models can be used to generate model transformations, by-example, taking into account the already defined mapping between abstract and concrete syntax elements. With this approach, the user's knowledge about the notation of the modeling language is sufficient for the definition of model transformations. Hence, no detailed knowledge about metamodels and model transformation languages or mapping languages is required. In particular, the proposed by-example approach for model transformations requires proper tool support and methods guiding the mapping of graphical domain models as well as the generation of the transformation code in order to fulfill the requirements for the user-friendly application. Therefore, future work is the implementation of a prototype in order to further evaluate our proposed approach in the large.

12.2.3 Integrating DSLs with UML

Lost information during roundtrip

The goal of supporting full round-trip transformations is currently not always achieved due to heterogeneity issues which have been first identified in the area of database integration [KS96]. These issues lead to value transformations that go beyond simple copying of values. Hence, some transformations can be inverted and some transformations can't. We are confronted with this challenge when there is a mismatch in *abstraction*, *aggregation*, or *precision* between the source and the target model elements. In the field of model-driven engineering, some issues are reported concerning the definition of bi-directional model transformations with QVT [Ste07]. In future work, these issues have to be clarified for our roundtrip transformation approaches, in particular, when feature mappings are not bijective.

Co-Evolution of UML Profiles

One "hot" topic in MDE is the co-evolution of models, transformations, code generation scripts, constraints, the concrete syntax definitions, and the modeling editor, to mention

just a few, when metamodels evolve over time. For example, this has been extensively discussed at the 7th OOPSLA Workshop on Domain-Specific Modeling[3]. Our approach based on a mapping model between the DSL metamodel and the UML metamodel may be exploited to co-evolve already existing UML profiles when the DSL metamodel evolves. For example, one simple case would be if an attribute of a DSL metaclass is deleted, also the corresponding tagged value in the profile may be automatically deleted (it is assumed that no feature mapping is available for the DSL attribute). This co-evolution capability would allow to reuse the mapping model also for evolved metamodels, and it is not necessary to create a new mapping model between the modified DSL metamodel and the UML metamodel.

Generate Metamodels from UML Profiles

What has not been discussed in this thesis is the opposite integration case, namely an UML profile already exists and a metamodel is needed. This direction has only barely been discussed in literature. In [WS07] a small example is presented how an UML profile can be modeled as metamodel, however, no integration approach, even not a completely manual methodology, has been published, yet. Information of the profiles can be more or less easily transformed into a metamodel, however, it is not clear which UML metaclasses and features should be available in the corresponding metamodel. One possibility would be to use the whole UML metamodel, however, this would explode the corresponding metamodel, consequently leading to a too complex metamodel even for a simple DSL, actually requiring a heavyweight extension of the UML metamodel.

A transformation language using profiles as language extensions

As mentioned in Subsection 10.5.2, currently no transformation language and corresponding transformation engine exist which support UML profiles as first class language extensions. Stereotypes should be usable as types in transformations rules, which means that rules can be directly applied on instances which have a certain stereotype assigned - so to say using the stereotype as a subclass. In particular, the Transformation Net formalism could be extended with the capability of using profiles as first class citizens. Stereotypes can be presented as one-colored places, tagged values as two-colored places. Only an additional adapter must be written in order to convert UML models into tokens as well as tokens into UML models, accordingly.

[3] www.dsmforum.org/events/DSM07

Bibliography

[ABFJ05] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*, San Diego, California, USA, 2005.

[ACB05] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. ModelGen: Model Independent Schema Translation. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05), Tokyo, Japan*, 2005.

[ACG07] Paolo Atzeni, Paolo Cappellari, and Giorgio Gianforme. MIDST: model independent schema and data translation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China*, 2007.

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[AT95] Paolo Atzeni and Riccardo Torlone. Schema translation between heterogeneous data models in a lattice framework. In *Proceedings of the 6th Working Conference on Data Semantics (DS-6), Atlanta, Georgia*, 1995.

[AVK+04] Aditya Agrawal, Attila Vizhanyo, Zsolt Kalmar, Feng Shi, Anantha Narayanan, and Gabor Karsai. Reusable Idioms and Patterns in Graph Transformation Languages. In *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs'04), Italy*, 2004.

[BCF02] Marco Brambilla, Sara Comai, and Piero Fraternali. Hypertext Semantics for Web Applications. In *Proceedings of the 10th Italian National Symposium on Advanced DataBase Systems (SEBD), Portoferraio, Italy*, 2002.

[BCFK99] Grady Booch, Magnus Christerson, Matthew Fuchs, and Jari Koistinen. UML for XML Schema Mapping Specification. Technical report, Rational Software and CommerceOne, August 1999.

[BCMM06] Luciano Baresi, Sebastiano Colazzo, Luca Mainetti, and Sandro Morasca. W2000: A Modeling Notation for Complex Web Applications. In Emilia Mendes

and Nile Mosley, editors, *Web Engineering: Theory and Practice of Metrics and Measurement for Web Development*, pages 335–364. Springer, 2006.

[BD07] Achim D. Brucker and Jürgen Doser. Metamodel-based UML Notations for Domain-specific Languages. In *Proceedings of the 4th International Workshop on Software Language Engineering (ATEM'07)*, 2007.

[Ber03] Philip A. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03), California*, 2003.

[BFW92] Alan W. Brown, Peter H. Feiler, and Kurt C. Wallnau. Past and future models of CASE integration. In *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering (CASE'92), Montreal, Canada*, 1992.

[BGM02] Luciano Baresi, Franca Garzotto, and Monica Maritati. W2000 as a MOF metamodel. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI'02), Orlando, Florida, USA*, July 2002.

[BGMT88] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of PCTE and PCTE+. *SIGSOFT Softw. Eng. Notes*, 13(5):248–257, 1988.

[BK98] Jan A. Bergstra and Paul Klint. The discrete time TOOLBUS — A software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, 1998.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Survey*, 18(4):323–364, 1986.

[BM07] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, China*, 2007.

[BSM+04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesely, 1st edition, 2004.

[CF01] Sara Comai and Piero Fraternali. A semantic model for specifying data-intensive Web applications using WebML. In *Proceedings of the 1st Semantic Web Working Symposium (SWWS'01), CA, USA*, July/August 2001.

[CFB+03] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 1st edition, 2003.

[Che76] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[DH05] AnHai Doan and Alon Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1):83–94, 2005.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Ear90] Anthony Earl. Principles of a reference model for computer aided software engineering environments. In *Proceedings of the International workshop on Environments on Software engineering environments, Chinon, France*, 1990.

[EV06] Sven Efftinge and Markus Voelter. oAW xText - A framework for textual DSLs. In *Proceedings of the Eclipse Modeling Symposium at the Eclipse Summit Europe Conference, Esslingen, Germany*, 2006.

[FBJ+05] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: a generic model weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM'05), France*, 2005.

[FFVM04] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An Introduction to UML Profiles. *European Journal for the Informatics Professional*, 5(2):5–13, April 2004.

[Fla02] Rony G. Flatscher. Metamodeling in EIA/CDIF—meta-metamodel and metamodels. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):322–342, 2002.

[Fow99] Martin Fowler. *Refactorings*. Addison-Wesley, 1st edition, 1999.

[Fow05] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? available at http://martinfowler.com/articles/languageWorkbench.html, June 2005.

[GCP01] Jaime Gómez, Cristina Cachero, and Oscar Pastor. Conceptual Modeling of Device-Independent Web Applications. *IEEE MultiMedia*, 8(2):26–39, 2001.

[GV07] Iris Groher and Markus Voelter. XWeave: models and aspects in concert. In *Proceedings of the 11th International Workshop on Aspect-oriented modeling (AOM'07), Nashville, TN*, 2007.

[GvD07] Bas Graaf and Arie van Deursen. Visualisation of Domain-Specific Modelling Languages Using UML. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, 2007.

[Haa07] Laura M. Haas. Beauty and the beast: The theory and practice of information integration. In *Proceedings of the 11th International Conference on Database Theory (ICDT'07), Barcelona, Spain*, 2007.

[HK87] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Survey*, 19(3):201–260, 1987.

[HK00] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 1st edition, 2000.

[HOT02] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical report, IBM Research Division, Thomas J. Watson Research Center, December 2002.

[Iiv96] Juhani Iivari. Why are case tools not used? *Communications of the ACM*, 39(10):94–103, 1996.

[ISO96] ISO/IEC. 14977:1996(E) Information technology – Syntactic metalanguage – Extended BNF, International standard, 1996.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering, (GPCE'06), Oregon, USA*, 2006.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer, 1992.

[JK06] Frederic Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference, Jamaica*, 2006.

[Jou05] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the 1st European Conference on Model Driven Architecture (ECMDA'05) – Workshop on Traceability, Nuremberg, Germany*, 2005.

[KAB02] Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. Technical Spaces: An Initial Appraisal. In *Proceedings of the 10th International Conference on Cooperative Information Systems (CoopIS'02), Carlifornia, Irvine*, 2002.

[KK03] Nora Koch and Andreas Kraus. Towards a Common Metamodell for the Development of Web Appliactions. In *Proceedings of the 3rd International Conference on Web Engineering (ICWE 2003), Oviedo, Spain*, July 2003.

[KKK+06a] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting Metamodels to Ontologies - A Step to the Semantic Integration of Modeling Languages. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06), Italy*, 2006.

[KKK+06b] Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Gerti Kappel, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration. In *Proceedings of Modellierung 2006, Austria*, 2006.

[KKK+07] Gerti Kappel, Horst Kargl, Gerhard Kramler, Andrea Schauerhuber, Martina Seidl, Michael Strommer, and Manuel Wimmer. Matching Metamodels with Semantic Systems - An Experience Report. In *Workshop Proceedings of Datenbanksysteme in Business, Technologie und Web (BTW'07), Germany*, 2007.

[KKR04] Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. Integrating XML and Relational Database Systems. *World Wide Web*, 7(4):343–384, 2004.

[KKR+06] Gerhard Kramler, Gerti Kappel, Thomas Reiter, Elisabeth Kapsammer, Werner Retschitzegger, and Wieland Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *Proceedings of the 1st International Workshop on Global integrated Model Management (GaMMa'06), Shanghai, China*, 2006.

[KS91] Won Kim and Jungyun Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, 24(12):12–18, 1991.

[KS96] Vipul Kashyap and Amit P. Sheth. Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. *VLDB Journal*, 5(4):276–304, 1996.

[KS05] Yannis Kalfoglou and W. Marco Schorlemmer. Ontology Mapping: The State of the Art. In *Dagstuhl Seminar Proceedings: Semantic Interoperability and Integration*, 2005.

[KW08] Horst Kargl and Manuel Wimmer. SmartMatcher - How Examples and a Dedicated Mapping Language can Improve the Quality of Automatic Matching Approaches. In *Proceedings of the 1st International Workshop on Ontology Alignment and Visualization (OnAV'08), Spain*, 2008.

[LC00] Dongwon Lee and Wesley W. Chu. Comparative Analysis of Six XML Schema Languages. *ACM SIGMOD Record*, 29(3):76–87, 2000.

[LLPM06] Björn Lundell, Brian Lings, Anna Persson, and Anders Mattsson. UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06), Italy*, 2006.

[LM05] Ralf Lämmel and Erik Meijer. Mappings make data processing go 'round. In *Pre-Proceedings of the International Summer School on Generative and Transformation Techniques in Software Engineering (GTTSE 2005), Braga, Portugal*, July 2005.

[LN07] Frank Legler and Felix Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In *Datenbanksysteme in Business, Technologie und Web (BTW'07), Aachen, Germany*, 2007.

[Mel04] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*. Springer, 2004.

[MFV06] Nathalie Moreno, Piero Fraternalli, and Antonio Vallecillo. A UML 2.0 profile for WebML modeling. In *Proceedings of the 2nd International Workshop on Model-Driven Web Engineering (MDWE 2006), Palo Alto, California, USA*, page 4. ACM Press, July 2006.

[MFV07] Nathalie Moreno, Piero Fraternali, and Antonio Vallecillo. WebML modeling in UML. *IET Software Journal*, 1(3):67–80, 2007.

[MK07] Marion Murzek and Gerhard Kramler. The Model Morphing Approach - Horizontal Transformations between Business Process Models. In *Proceedings of the 6th International Conference on Perspecitves in Business Information Research (BIR'07), Tampere, Finnland*, 2007.

[MMSV02] Alexander Maedche, Boris Motik, Nuno Silva, and Raphael Volz. MAFRA - A MApping FRAmework for Distributed Ontologies. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW'02), Spain*, 2002.

[Mot87] A. Motro. Superviews: virtual integration of multiple databases. *IEEE Trans. Softw. Eng.*, 13(7):785–798, 1987.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédéric Fondement, and Jean Bézivin. Platform independent Web application modeling and development with Netsilon. *Software and System Modeling*, 4(4):424–442, 2005.

[MSZJ04] Haohai Ma, Weizhong Shao, Lu Zhang, and Yanbing Jiang. Applying OO Metrics to Assess UML Meta-models. In *Proceedings of the 7th International Conference*

on the Unified Modelling Language: Modelling Languages and Applications (UML 2004), Lisbon, Portugal, October 2004.

[(NI04] National Information Standards Organization (NISO). Understanding Metadata. available at http://www.niso.org/standards/resources/UnderstandingMetadata.pdf, 2004.

[Obe88] Patricia A. Oberndorf. The Common Ada Programming Support Environment (APSE) Interface Set (CAIS). *IEEE Transactions on Software Engineering*, 14(6):742–748, 1988.

[OMG01] Object Management Group OMG. UML Specification Version 1.4. http://www.omg.org/docs/formal/01-09-67.pdf, September 2001.

[OMG02] Object Management Group OMG. Meta Object Facility (MOF) Specification 1.4. http://www.omg.org/docs/formal/02-04-03.pdf, April 2002.

[OMG03a] Object Management Group OMG. MDA Guide Version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf, June 2003.

[OMG03b] Object Management Group OMG. UML Specification: Infrastructure Version 2.0. http://www.omg.org/docs/ptc/03-09-15.pdf, November 2003.

[OMG04] Object Management Group OMG. Meta Object Facility (MOF) 2.0 Core Specification Version 2.0. http://www.omg.org/docs/formal/06-01-01.pdf, October 2004.

[OMG05a] Object Management Group OMG. Architecture Driven Modernization (ADM). http://www.omg.org/adm, June 2005.

[OMG05b] Object Management Group OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification., November 2005.

[OMG05c] Object Management Group OMG. MOF 2.0/XMI Mapping Specification 2.1. http://www.omg.org/docs/formal/05-09-01.pdf, September 2005.

[OMG05d] Object Management Group OMG. OCL Specification Version 2.0. http://www.omg.org/docs/ptc/05-06-06.pdf, June 2005.

[OMG05e] Object Management Group OMG. UML Specification: Superstructure Version 2.0. http://www.omg.org/docs/formal/05-07-04.pdf, August 2005.

[OMG08] OMG. *Common Object Request Broker Architecture (CORBA/IIOP)*. OMG, http://www.omg.org/spec/CORBA/3.1/, 2008.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[Rei08] Thomas Reiter. *T.R.O.P.I.C.: Transformation on Petri Nets In Color*. PhD thesis, Johannes Kepler University Linz, 2008.

[RWK07] Thomas Reiter, Manuel Wimmer, and Horst Kargl. Towards a runtime model based on colored Petri-nets for the execution of model transformations. In *Proceedings of the 3rd Workshop on Models and Aspects, Germany*, 2007.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Sch07] Andrea Schauerhuber. *aspectUWA - Applying Aspect-Orientation to the Model-Driven Development of Ubiquitous Web Applications*. PhD thesis, Vienna University of Technology, 2007.

[SD05] Andy Schürr and Heiko Dörr. Introduction to the special SoSym section on model-based tool integration. *Software and System Modeling (SoSym)*, 4(2):109–111, 2005.

[SdB05] François Scharffe and Jos de Bruijn. A language to specify mappings between ontologies. In *Proceedings of the 1st International Conference on Signal-Image Technology & Internet-Based Systems (SITIS'05), Yaounde, Cameroon*, 2005.

[SE05] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics IV*, pages 146–171, 2005.

[SHT+77] Nan C. Shu, Barron C. Housel, Robert W. Taylor, Sakti P. Ghosh, and Vincent Y. Lum. EXPRESS: A Data EXtraction, Processing, amd REStructuring System. *ACM Trans. Database Syst.*, 2(2):134–174, 1977.

[Sof00] Rational Software. Migrating from XML DTD to XMLSchema using UML. Rational Software White Paper, August 2000.

[SPD92] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1(1):81–126, 1992.

[SSK+06] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, and Manuel Wimmer. Towards a Common Reference Architecture for Aspect-Oriented Modeling. In *Proceedings of the 8th International Workshop on Aspect-Oriented Modeling (AOM'06), Bonn, Germany*, March 2006.

[Sta05] Ryan Stansifer. EBNF Grammar for Mini-Java. available at http://www.cs.fit.edu/ryan/cse4251/mini_java_grammar.html, August 2005.

[Ste07] Perdita Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), Nashville, USA*, 2007.

[SWK06] Andrea Schauerhuber, Manuel Wimmer, and Elisabeth Kapsammer. Bridging Existing Web Modeling Languages to Model-Driven Engineering: A Metamodel for WebML. In *Proceedings of the 2nd International Workshop on Model-Driven Web Engineering (MDWE'06), Palo Alto, California, USA*, July 2006.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Software and System Modeling*, 4(2):112–122, 2005.

[Ulr05] William Ulrich. A Status on OMG Architecture-Driven Modernization Task Force. available at http://www.omg.org/adm-status/, 2005.

[VG07a] Markus Voelter and Iris Groher. Handling Variabilitiy in Model Transformations and Generators. In *Workshop Proceedings of 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Montreal, Canada*, 2007.

[VG07b] Markus Voelter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, 2007.

[VIG05] Fabio Vitali, Angelo Di Iorio, and Daniele Gubellini. Design patterns for descriptive document substructures. In *Proceedings of the Extreme Markup Languages Conference, Montréal, Quebec, Canada*, August 2005.

[VKC+07] Antonio Vallecillo, Nora Koch, Cristina Cachero, Sara Comai, Piero Fraternali, Irene Garrigós, Jaime Gómez, Gerti Kappel, Alexander Knapp, Maristella Matera, Santiago Meliá, Nathalie Moreno, Birgit Pröll, Thomas Reiter, Werner Retschitzegger, Eduardo Rivera, Andrea Schauerhuber, Wieland Schwinger, Manuel Wimmer, and Gefei Zhang. MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods. In *Workshop Proceedings of 7th International Conference on Web Engineering (ICWE'07), Italy, July*, 2007.

[VP04] Dániel Varró and András Pataricza. Generic and Meta-transformations for Model Transformation Engineering. In *Proceedings of the 7th International Conference on the Unified Modeling Language (UML'04), Portugal*, 2004.

[W3C04] World Wide Web Consortium W3C. XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/XML Schema-0/, October 2004.

[W3C06] World Wide Web Consortium W3C. Extensible Markup Language (XML) 1.1 (Second Edition). http://www.w3c/TR/xml11/, September 2006.

[Was89] Anthony I. Wasserman. Tool integration in software engineering environments. In *Proceedings of the International Workshop on Software Engineering Environments, Chinon, France*, 1989.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[WK05] Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. In *Proceedings of Satellite Events at the MoDELS 2005 Conference, Jamaica*, 2005.

[WS07] Ingo Weisemöller and Andy Schürr. A comparison of standard compliant ways to define domain specific languages. In *Proceedings of the 4th International Workshop on Software Language Engineering (ATEM'07)*, 2007.

[WSKK06] Manuel Wimmer, Andrea Schauerhuber, Elisabeth Kapsammer, and Gerhard Kramler. From Document Type Definitions to Metamodels: The WebML Case Study. Technical report, Vienna University of Technology, March 2006.

[WSS+07] Manuel Wimmer, Andrea Schauerhuber, Michael Strommer, Wieland Schwinger, and Gerti Kappel. A Semi-automatic Approach for Bridging DSLs with UML. In *Workshop Proceedings of 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Canada*, 2007.

[WSSK07] Manuel Wimmer, Andrea Schauerhuber, Wieland Schwinger, and Horst Kargl. On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges. In *Workshop Proceedings of 7th International Conference on Web Engineering (ICWE'07), Italy*, 2007.

Curriculum Vitae

Manuel Wimmer, MSc.

Schanzstrasse 49/4
1140 Wien, Austria

Email: wimmer@big.tuwien.ac.at
Web: http://www.big.tuwien.ac.at/staff/wimmer

Date of Birth: 25-Mar-1980
Nationality: Austria

Education

PhD Studies in Business Informatics April 2005 - April 2008
Vienna University of Technology, Austria
Supervision:
o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel, Vienna University of Technology, Austria
a.Univ.-Prof. Mag. Dr. Werner Retschitzegger, Johannes Kepler University Linz, Austria

MSc Business Informatics October 2000 - March 2005
Vienna University of Technology, Austria
Supervision: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Thesis: Model Driven Architecture in der Praxis - Evaluierung von aktuellen Werkzeugen und Fallstudie.

Graduation from Secondary School June 1999

Work Experience

University Assistant April 2005 - April 2008
Vienna University of Technology, Institute of Software Technology and Interactive Systems
Courses:

- Object-oriented Analysis and Design (UML)
- Modeling Techniques and Methods (ER, UML)
- Web Engineering (Servlets, JSPs, XML, Web Services)

- Model Engineering (Metamodeling, Model Transformations, Code Generation)

Teaching Assistant March 2004 - March 2005
Vienna University of Technology, Institute of Software Technology and Interactive Systems
Courses:

- Object-oriented Analysis and Design (UML)
- Modeling Techniques and Methods (ER, UML)
- Web Engineering (Servlets, JSPs, XML, Web Services)

Tutor October 2002 - January 2005
Vienna University of Technology, Institute of Software Technology and Interactive Systems
Courses:

- Introduction to Programming with Java
- Object-oriented Analysis and Design (UML)
- Modeling Techniques and Methods (ER, UML)
- Web Engineering (Servlets, JSPs, XML, Web Services)

Awards & Achievments

Best Paper Award 2005
M. Wimmer, G. Kramler: Bridging Grammarware and Modelware. 4th Workshop in Software Model Engineering, in conjunction with MoDELS/UML'06, October 3rd, 2005.

Lower Austria Top-Scholarship 2003
(In German: Top-Stipendium des Landes Niederösterreich)

Publications

Book Chapters

1. "Modellgetriebene Softwareentwicklung in der Praxis". In Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger, "UML@Work - Objektorientierte Modellierung mit UML 2". 3rd Edition, dpunkt.verlag, 2005, pp. 343-368.

Journal Papers

1. Andrea Schauerhuber, Manuel Wimmer, Elisabeth Kapsammer, Wieland Schwinger, and Werner Retschitzegger. "Bridging WebML to Model-Driven Engineering: From

DTDs to MOF". IET Software Journal, Vol. 1, No. 3, Institution of Engineering and Technology, June 2007.

Conference Papers

1. Michael Strommer and Manuel Wimmer. "A Framework for Model Transformation By-Example: Concepts and Tool Support". Proc. of the 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'08), Zurich, Switzerland, July 2008. (to appear)
2. Gerti Kappel, Horst Kargl, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, Michael Strommer, and Manuel Wimmer. "A Framework for Building Mapping Operators Resolving Structural Heterogeneities". Proc. of the 7th International Conference on Information Systems Technology and its Applications (ISTA'08), Klagenfurt, Austria, April 2008. (to appear)
3. Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. "Model Transformation Generation By-Example". Proc. of the 40th Hawaii International Conference on Systems Science (HICSS'07), Hawaii, January 2007.
4. Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. "Lifting metamodels to ontologies - a step to the semantic integration of modeling languages". Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'06), Genova, Italy, October 2006.
5. Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. "On Models and Ontologies - A Layered Approach for Model-based Tool Integration". Proc. of Modellierung 2006, Innsbruck, March 2006.

Workshop Papers

1. Manuel Wimmer, Andrea Schauerhuber, Michael Strommer, Jürgen Flandorfer, and Gerti Kappel. "How Web 2.0 can leverage Model Engineering in Practice". Proc. of the Workshop on Domänenspezifische Modellierungssprachen (DSML'08), in conjunction with Modellierung'08, Berlin, Deutschland, March 2008.
2. Horst Kargl and Manuel Wimmer. "SmartMatcher - How Examples and a Dedicated Mapping Language can Improve the Quality of Automatic Matching Approaches". Proc. of the 1st International Workshop on Ontology Alignment and Visualization (OnAV'08), in conjunction with CISIS'08, Barcelona, Spain, March 2008.
3. Manuel Wimmer, Andrea Schauerhuber, Michael Strommer, Wieland Schwinger, and Gerti Kappel. "A Semi-automatic Approach for bridging DSLs with UML". Proc. of

the 7th OOPSLA Workshop on Domain-Specific Modeling, in conjunction with OOPSLA'07, Montreal, Canada, October 2007.

4. Michael Strommer, Marion Murzek, and Manuel Wimmer. "Applying Model Transformation By-Example on Business Process Modeling Languages". Proc. of the 3rd International Workshop on Foundations and Practices of UML, in conjunction with ER'07, Auckland, New Zealand, November 2007
5. Thomas Reiter, Manuel Wimmer, and Horst Kargl. "Towards a runtime model based on colored Petri-nets for the execution of model transformations". Proc. of the 3rd Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, in conjunction with ECCOP'07, Berlin, Germany, July 2007.
6. Manuel Wimmer, Horst Kargl, Martina Seidl, Michael Strommer, and Thomas Reiter. "Integrating Ontologies with CAR-Mappings". Proc. of the 1st International Workshop on Semantic Technology Adoption in Business (STAB'07), in conjunction with ESCT'07, Vienna, June 2007.
7. Manuel Wimmer, Andrea Schauerhuber, Wieland Schwinger, and Horst Kargl. "On the Integration of Web Modeling Languages: Preliminary Results and Future Challenges". Proc. of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007), in conjunction with ICWE'07, Como, Italy, July 2007.
8. A. Vallecillo, N. Koch, C. Cachero, S. Comai, P. Fraternali, I. Garrigós, J. Gómez, G. Kappel, A. Knapp, M. Matera, S. Meliá, N. Moreno, B. Pröll, T. Reiter, W. Retschitzegger, J. E. Rivera, A. Schauerhuber, W. Schwinger, M. Wimmer, G. Zhang: "MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods". Proc. of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007), in conjunction with ICWE'07, Como, Italy, July 2007.
9. Andrea Schauerhuber, Manuel Wimmer, Wieland Schwinger, Elisabeth Kapsammer, and Werner Retschitzegger. "Aspect-Oriented Modeling of Ubiquitous Web Applications: The aspectWebML Approach". Proc. of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), Tucson, Arizona, USA, March 2007.
10. Gerti Kappel, Horst Kargl , Gerhard Kramler , Andrea Schauerhuber , Martina Seidl, Michael Strommer, and Manuel Wimmer. "Matching Metamodels with Semantic Systems - An Experience Report". 12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web, Aachen, Germany, March 2007.
11. Horst Kargl, Michael Strommer, and Manuel Wimmer. "Measuring the Explicitness of Modeling Concepts in Metamodels". Proc. of the 1st Workshop on Model Size Metrics, in conjunction with MoDELS/UML'06, Genova, Italy, October 2006.
12. Andrea Schauerhuber, Manuel Wimmer, and Elisabeth Kapsammer. "Bridging existing Web Modeling Languages to Model-Driven Engineering: A Metamodel for WebML." Proc. of the 2nd International Workshop on Model-Driven Web Engineer-

ing (MDWE 2006), in conjunction with ICWE'06, Standford Linear Accelerator Center, Palo Alto, California, USA, July 2006.

13. Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer. "Towards a Common Reference Architecture for Aspect-Oriented Modeling". 8th International Workshop on Aspect-Oriented Modeling, in conjunction with AOSD 2006, Bonn, Germany, March 21, 2006.
14. Manuel Wimmer and Gerhard Kramler. "Bridging Grammarware and Modelware". Proc. of the 4th Workshop in Software Model Engineering, in conjunction with MoDELS/UML'05, Jamaica, October 2005.

Technical Reports

1. Andrea Schauerhuber, Wieland Schwinger, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. "A Survey on Web Modeling Approaches for Ubiquitous Web Applications". Technical Report, Vienna University of Technology, October 2007.
2. Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. "A Survey on Aspect-Oriented Modeling Approaches". Technical Report, Vienna University of Technology, October 2007.
3. Manuel Wimmer, Andrea Schauerhuber, Elisabeth Kapsammer, and Gerhard Kramler. "From Document Type Definitions to Metamodels: The WebML Case Study". Technical Report, Vienna University of Technology, March 2006.