Ph.D. Thesis

Model Transformation By-Example

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

Ao.Univ.-Prof. Mag. Dr. Christian Huemer
Institute of Software Technology and Interactive Systems
Vienna University of Technology

Submitted at the
Vienna University of Technology
Faculty of Informatics

by

Michael Strommer
0025232
Schliemanngasse 9/6
A-1210 Vienna

Vienna, May 2008

Danksagung

Diese Arbeit wäre ohne die Unterstützung und Hilfe von KollegInnen, BetreuerInnen, FreundInnen und Familie nicht möglich gewesen. Mein besonderer Dank gilt meinen beiden BetreuerInnen Gerti Kappel und Christian Huemer die mich bei der Wahl des Themas und der Durchführung der Dissertation unterstützt haben. Meinen Kollegen Manuel Wimmer und Horst Kargl möchte ich für die zahlreichen Diskussionen danken, die immer wieder spannende Ergebnisse geliefert haben. Manuel Wimmer möchte ich besonders für die Idee zu dem Thema dieser Dissertation danken.

Meinen beiden Diplomanden Abraham und Gerald Müller gilt mein Dank für ihre Unterstützung bei der Implementierung eines Prototyps.

Natürlich möchte ich mich auch ganz besonders bei meinen Eltern Eva und Norbert Strommer sowie meinen Freunden bedanken die immer ein offenes Ohr für meine Anliegen und Probleme hatten. Und zuletzt möchte ich noch meinem Freund Jürgen Falb für seine Geduld und Hilfsbereitschaft in den letzten Jahren danken.

Abstract

Model-Driven Engineering (MDE) is getting more and more attention as a viable alternative to the traditional code-centric software development paradigm. With its progress, several model transformation approaches and languages have been developed in the past years. Most of these approaches are metamodel-based and, therefore, require knowledge of the abstract syntax of the modeling languages, which in contrast is not necessary for defining domain models using the concrete syntax of the respective languages.

Based on the by-example paradigm, we propose Model Transformation By-Example (MTBE), to cope with shortcomings of current model transformation approaches. Our approach allows the user to define semantic correspondences between concrete syntax elements with the help of special mapping operators. This is more user-friendly than directly specifying model transformation rules and mappings on the metamodel level. In general, the user's knowledge about the notation of the modeling language and the meaning of mapping operators is sufficient for the definition of model transformations. The definition of mapping operators is subject to extension, which has been applied for the definition of mapping operators for the structural and the behavioral modeling domain. However, to keep things transparent and user-friendly, only a minimal set of mapping operators has been implemented. To compensate for the additional expressiveness inherent in common model transformation languages we apply reasoning algorithms on the models represented in concrete as well as in abstract syntax and on the metamodels generating adequate transformation code.

In order to fulfill the requirements for a user-friendly application of MTBE, proper tool support and methods to guide the mapping and model transformation generation tasks are a must. Hence, a framework for MTBE was designed that builds on state-of-the-art MDE tools on the Eclipse platform, such as the Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF), the Atlas Transformation Language (ATL), and the Atlas Model Weaver (AMW). The decision to base our implementation on top of Eclipse and further Eclipse projects was driven by the fact, that there is a huge community we can address with our MTBE plug-in.

Finally, we evaluate our approach by means of two case studies covering the structural as well as behavioral modeling language domain.

Kurzfassung

Die modellgetriebene Softwareentwicklung kann immer mehr als ernst zu nehmende Alternative zur klassischen Softwareentwicklung angesehen werden. Im Zuge des Entwicklungsprozesses der modellgetriebenen Softwareentwicklung sind in den letzten Jahren auch zahlreiche Methoden zur Modelltransformation entwickelt worden. Viele dieser Ansätze basieren auf den Metamodellen der jeweiligen Modellierungssprachen und setzen daher ein Wissen über die abstrakte Syntax voraus, das für die Modellierung von Modellen mit eben diesen Sprachen nicht notwendig ist.

Aufgrund dieser Einschränkungen heutiger Modelltransformationsansätze entstand die Idee zu *Model Transformation By-Example* (MTBE), welches auf beispielgetriebenen Methoden basiert. Dieser Ansatz ermöglicht BenutzerInnen mit geeigneten Mappingoperatoren semantische Beziehungen zwischen Elementen, definiert in einer konkreten Syntax, zu spezifizieren. Auf diese Weise lassen sich Modelltransformationen und semantische Beziehungen benutzerfreundlicher spezifizieren als dies auf der Metamodellebene der Fall wäre. Das Wissen der BenutzerInnen über die konkrete Syntax einer Modellierungssprache und die Bedeutung der Mappingoperatoren genügen in den meisten Fällen, um Modelltransformationen zu erzeugen. Der MTBE-Ansatz unterstützt die Spezifikation von weiteren Mapping-Operatoren. Im Rahmen der Dissertation wurden Mapping-Operatoren für Sprachen zur Strukturmodellierung und zur Verhaltensmodellierung entwickelt. Die Anzahl der Mappingoperatoren wurde bewusst niedrig gehalten, um die Überschaubarkeit und Benutzerfreundlichkeit nicht zu gefährden. Der dadurch entstandene Verlust an Ausdrucksstärke wurde durch Reasoningalgorithmen, sowohl auf der Metamodell- als auch auf der Modellebene größtenteils kompensiert, um ausführbaren Transformationscode generieren zu können.

Neben der konzeptuellen Definition von MTBE ist für den Nachweis der Praxistauglichkeit auch eine Werkzeugunterstützung gefordert, welche die Möglichkeit zum Modellmapping und zur Generierung von Transformationscode bietet. Daher wurde ein Framework konzipiert und umgesetzt, das auf bewährten Modellierungsanwendungen der Eclipse-Umgebung aufbaut. Zu diesen Anwendungen zählen etwa das Eclipse Modeling Framework (EMF), das Graphical Modeling Framework (GMF), die Atlas Transformation Language (ATL) und der Atlas Model Weaver (AMW).

Schließlich wurde MTBE anhand von zwei Fallstudien getestet und evaluiert. Diese beiden Fallstudien decken sowohl Struktur- als auch Verhaltensmodellierung ab.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Software development is a complex task. Developers have tried to overcome complexities by different kinds of methodologies (e.g., object oriented programming, domain specific languages, patterns, etc.) and technologies (e.g., CASE-Tools, XML, etc.). One of the latest paradigms is Model-Driven Engineering (MDE) [8], which aims at defining a framework for modeling, metamodeling and model transformation. Model transformations are used to transform between any kinds of models, no matter what domain they cover, as long as these models fulfill the requirements of a specific model transformation approach. In general, corresponding modeling language definitions in form of metamodels are required in a model transformation process. How model transformation approaches work is best explained fol-

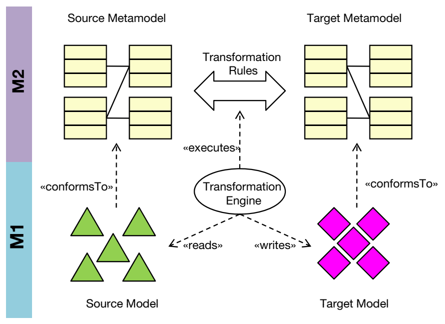


Figure 1.1: Model transformation pattern according to [21].

lowing the basic model transformation pattern shown in Figure 1.1. First, we distinguish between layers M1 and M2 introduced by the OMG in [68]. M1 contains the models, which are instances of metamodels that reside on M2. Second, transformation rules between so called source and target metamodel are specified. Third, a transformation engine reads one source model conforming to the source metamodel and writes a target model conforming

to the target metamodel. A model can be transformed into a semantically corresponding model (horizontal transformation) or into a model on another level of abstraction (vertical transformation). This thesis focuses on horizontal model transformations that are used in various metamodel-based tool integration scenarios, e.g., exchanging models in different languages via model transformations. Horizontal model transformations are also considered by a special kind of MDE, namely Model-Driven Architecture (MDA) [64], an Object Management Group (OMG) initiative. MDA delivers concepts for metamodels representing both an abstract syntax of the corresponding modeling language and also a data structure for storing models in repositories. However, this implementation specific focus does not result in a user-friendly approach to model transformations due to two major reasons:
Concept Hiding. First, metamodels are not only based on first class concepts, but usually include constraints that are hidden in textual descriptions or, in best case, as formal constraint in special section. However, non-first class, i.e. hidden, concepts are available for notation purposes. For example, in the core of the UML metamodel (defined in the UML Infrastructure [65]) the concept *attribute* is hidden in the class *Property*. Properties can only be attributes, if the property has a relationship *owningClass* to a class. When the user has to define model transformations these hidden concepts must be re-engineered and expressed in complex rules by a manual process.
Metamodel vs. Model. Second, defining model transfromations using state-of-the-art technologies like ATL (ATLAS Transformation Language) [40] require a software engineer or designer to be familiar with metamodeling concepts. Furthermore, she must be aware of language specialties in the abstract syntax (AS) and how these specialties are mapped to the model using the concrete syntax (CS). In most cases a software engineer is neither an expert in metamodeling nor in specifying the CS of a modeling language.

1.2 Existing Work

Model Transformation Approaches. Model transformation is considered as one of the key technologies in MDE. Therefore, a lot of research efforts address this subject to expedite the dissemination of the MDE paradigm. A good overview of current model transformation technologies is given in [21]. All of these approaches are based on metamodels. As a consequence, the user needs to be experienced in metamodeling. In addition, she must understand the metamodels involved in a transformation scenario in order to specify transformation rules with common model transformation approaches such as ATL and graph transformation approaches. This complexity is shown on top of Figure 1.2. The three big question marks indicate the places where a lack of knowledge is commonly prevalent. This lack has then to be compensated by rarely available specialists knowing the corresponding modeling languages by hard. To summarize, the definition of model transformations is a complex and time consuming task since the user needs detailed know-how in the following three areas:

1. Metamodeling in general, i.e., to know the concepts used in the metamodeling language.
2. A sound understanding of the modeling languages and of their concepts.
3. An excellent command of the transformation language being used for the purpose of model transformation.

By-Example Approaches. By-Example approaches have a long tradition and originated in the early 1970ies [20]. These approaches promote the use of examples in one way or the other to overcome complexity of selected problems in the field of computer science. The most prominent by-example approach is definitely *Query By Example* (QBE) [93], which is presented in detail in Section 2.4. The basic idea of QBE is using a simple notation for defining queries on database tables. The notation is based on a table-like visualization of a database schemes, where sample instances are defined by the user. The user's knowledge of notation elements and their meaning is reused by using such a language design approach. It is rather is to learn the additional semantics for the QBE language needed for specifying proper queries compared to learning the fairly complex declarative query language SQL. Besides QBE we present several other example favoring approaches in Section 2.4.

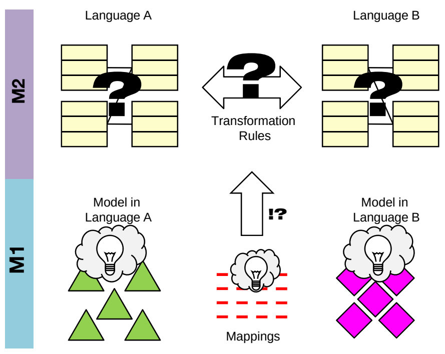


Figure 1.2: Typical distribution of knowledge considering software architects.

1.3 Contribution of the Thesis

In this thesis we propose *Model Transformation By-Example* (MTBE) following the idea of the by-example approaches. MTBE emerged during the realization of the ModelCVS project [42] that fosters model and tool integration by means of metamodel mapping and model transformation. In contrast to MTBE, the ModelCVS approach focuses on the metamodels and does not take the models and their concrete syntax into consideration. By our MTBE approach we allow the definition of inter-model mappings representing semantic correspondences between concrete domain models (M1 layer). This is more user-friendly than directly specifying model transformation rules based on metamodels (M2 layer). This advantage of MTBE is illustrated at the bottom of Figure 1.2 where we show two models as instances of two metamodels. The lights in the bubbles indicate that the user knows how to correctly interpret the concrete syntax elements. It follows that the user is able to define adequate mappings between these CS elements. The inter-model mappings are used to generate the transformation rules in by-example manner, taking into account the existing mapping (notation) between abstract and concrete syntax elements. The notation includes the constraints how elements from the abstract syntax (metamodel) are related to the concrete syntax. By applying MTBE to the Eclipse Modeling Framework (EMF) [12] and to the Graphical Modeling Framework (GMF) [23] it is possible to reuse the already available constraints to derive transformation rules expressed in ATL. The user's knowledge about the CS of the modeling language is sufficient for the definition of semantically corresponding model transformations. Hence, neither a detailed understanding about the abstract syntax (metamodel) nor about the notation are required. However, it is essential to align two models of the the same problem domain, to automatically derive the transformation rules.

This leads to the following **hypothesis**:
It is feasible to develop model transformations in a by-example manner by:

1. *Defining a conceptual foundation for MTBE.*
2. *Building a solid prototype on these conceptual foundations.*
3. *Evaluating MTBE by means of case studies.*

We identified two major **contributions** of this thesis:

Leveraging Model Transformations. Our MTBE approach helps to ease the production of transformation code by (semi-)automatically generating ATL rules from mappings on the model layer. Although we cannot provide a solution for fully-automatically code generation, we deliver a MTBE process that helps the software engineer safe a considerable amount of time. Especially, the ATL code generation promotes the generation of declarative code and does not mix it with imperative code fragments. This increases readability.

Implementing MTBE. In the literature we find two conceptual approaches to MTBE, which have been developed around the same time. The first one has been developed by ourselves [89], the other one has been proposed by Varro [84]. At the time of starting the thesis no tool support and framework design was available. In this thesis we present a framework design and a functioning proof-of-concept prototype that follows our conceptual approach. It allows the definition of model mappings, the reasoning on these mappings as well as the generation of transformation code.

1.4 Big Picture and Structure of the Thesis

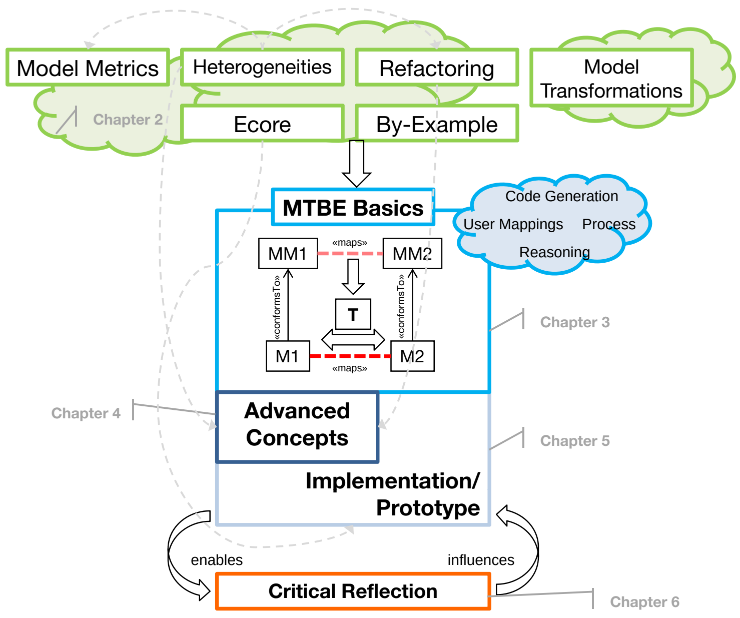


Figure 1.3: Big picture of this thesis.

In Figure 1.3 we provide an overview of the research topics addressed in this thesis. By this figure it becomes evident that the thesis is structured according to the arguments presented in the hypothesis. Each of the five colored blocks is discussed in its own chapter. The figure has to be read from top to bottom because lower concpets in an upper area are further elaborated in a lower area. In addition, dashed gray lines with arrows show strong dependencies from one part of the thesis to another one.

On top of this figure we show technologies and approaches that significantly influenced our MTBE approach. An overview of these technologies and approaches is given in Chapter 3. We present work on metamodeling by means of Ecore, metamodel heterogeneities, refactoring patterns for metamodels, and our model metric for measuring the explicitness of metamodels. There is a strong dependency between heterogeneities and both, model metrics and refactoring because the latter two specifically face the problem of the former one.

Chapter 4 MBTE basics represents the core of our MTBE approach. We address the conceptual MTBE process consisting of five different steps, each of which is discussed in subsequent sections.

Based on the basic concepts of MTBE, we discuss advanced extensions in Chapter 5. We give a detailed description of advanced concepts and useful extensions in order to maximize the amount of code, which is automatically generated. These concepts comprise, for example, special reasoning algorithms to cope with troublesome metamodel heterogeneities. Hence, our findings about heterogeneities play a major role for the contribution of this chapter.

The second major contribution of this thesis , i.e., the implementation of MTBE, is described in Chapter 6. We present employed technologies and frameworks that provide the basis for our MTBE framework. Furthermore, we explain the implementation aspects in detail. In Chapter 7 we elaborate on two case studies having proved the advantages of our approach. This chapter also covers a critical reflection of our work on MTBE. This critical reflection also leads to directions for future work and enhancements for our implementation being discussed in Chapter 8.

Chapter 2

State of the Art

Contents

In this chapter we give an overview of the research and work that is related most in the context of this thesis. The two major fields of related work are already visible in the title of the thesis - Model Transformation By-Example. We first focus on model transformation approaches in general. Next we present tried and tested work of common by-example approaches.

2.1 Introduction to Model Transformation

A key technique for automatic management of modeling artifacts are model transformations [54]. Several model transformation approaches and languages have been proposed in the past six years [21]. Czarnecki and Helsen provide a very comprehensive and detailed categorization of various model transformation approaches. Following [21] we define the basic concepts of model transformation as follows:

- A source model that conforms to a given source metamodel.
- A target model that conforms to a given target metamodel.

- Some sort of transformation definition that is based on the source and target metamodels.
- A transformation engine that executes the transformation and produces a target model from some source model.

Note that these basic concepts may be further extended. For example, one could allow for the use of more than one source and target (meta)models. Areas for model transformations are manifold. There are horizontal as well as vertical model transformation scenarios possible. We face a vertical transformation when moving from platform independent models to platform dependent models or even to code. When switching or migrating to other modeling languages at the same level we are designing horizontal model transformations.

2.2 Some Model Transformation Approaches in Detail

After this brief and very basic introduction to model transformations we continue by describing three of the latest model transformation technologies and approaches. To foster a better understanding of these approaches we underpin our explanations by small examples.

2.2.1 QVT

QVT stands for Query/Views/Transformations and represents a specification for model transformations still under development by the OMG [70]. The goal of this specification is to standardize model transformations. Besides the outstanding final version there is no tool implementation available supporting the complete QVT standard so far. QVT defines four different ways of specifying model transformations as depicted in Figure 2.1, i.e., four different transformation languages. Relations and Core are of a declarative style whereas Operational Mappings and the Black Box are languages favoring the imperative style. The complete Operational Mappings language is supported by the SmartQVT [1] tool, which is available as Eclipse plug-in. Another tool implementation of QVT supporting imperative as well as the declarative Relations language is ATL [2] described in the next subsection. In Example 1 we briefly demonstrate the use of the Relations approach, which is then mapped automatically to the Core, acting as operational semantic basis for Relations.

Example 1. *Assume we have two metamodels SimpleUML and SimpleRDBMS and want to transform instances of these metamodels. The first mapping that needs to be defined is PackageToSchema as is shown in Listing 2.1. The direction of model transformation is specified during runtime, which makes relation PackageToSchema directionless. The domain in a relation is just a typed variable, that originates from a given metamodel. This may be seen as a pattern that is going to be matched within*

[1]http://smartqvt.elibel.tm.fr/
[2]http://www.eclipse.org/m2m/atl/

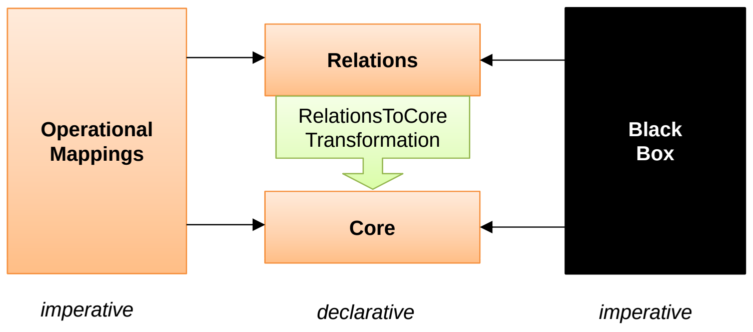


Figure 2.1: Relationships between QVT metamodels [70].

input and output models. Both name attributes are bound to the same variable pn, which acts as a condition forcing them to be the same in both models in order to let the pattern match.

Listing 2.1: QVT Relation for mapping packages to schemas.

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS){
  top relation PackageToSchema{
    pn: String;

    domain uml p:Package {
      name=pn
    }
    domain rdbms s:Schema {
      name=pn
    }
  }
}
```

2.2.2 ATL

The Atlas Transformation Language (ATL) was introduced by the Atlas Group and the TNI-Valiosys Company in 2003 [14]. ATL aims at providing a practical implementation for the MOV/QVT [70] standard. As such it provides a transformation engine able to transform any given source model to a specified target model. However, before the engine can however perform the transformation the user has to specify a proper ATL program based on some valid and executable metamodels. ATL supports queries, views and transformations. The transformation language is based on rules that are either matched in a declarative way or called in an imperative way. Besides rules, ATL provides so called helpers that are similar to e.g. Java methods in declarative style. For a complete description of the abstract syntax of ATL and its execution semantics we refer to the ATL user manual [5] as well as the project site [4]. To illustrate ATL we will give a small example.

Example 2. *Assume we have two metamodels $MM1$ and $MM2$ as well as two model instances $M1$ and $M2$ such that the instance of functions $M1 \lhd_t^l MM1$ and $M2 \lhd_t^l MM2$ hold. $MM1$ defines just one meta class, say $Operation$ with one attribute $String : name$. On the other hand $MM2$ also just defines one meta class, i.e., $Method$, which contains the attribute $String : signature$. We now define an ATL transformation program that transforms any given $M1$ into $M2$. The code that transforms these simple models is depicted in Listing 2.2. The first two lines define the header section of an ATL program. This primarily determines input metamodels and conforming models as well as output metamodels and conforming models. What follows is an ATL matched rule that matches elements from the source model in its source pattern ("from part") and transforms them into elements form the target model with its target pattern ("to part"). Matched elements are bound to variables that can be used throughout the current rule, e.g. in the "to part" in order to assign attributes. Variables are also bound to target elements that are created by this rule. This makes sense if a rule contains multiple target pattern elements.*

Listing 2.2: ATL snippet MM1 2 MM2.

```
module MM1_2_MM2
create OUT : MM2 from IN : MM1;

rule Operation2Method {
    from
            o : MM1!Operation
    to
            m : MM2!Method (
                signature <- o.name
            )
}
```

2.2.3 Graph Grammars

Besides the above presented approaches to define model transformations, we want to elaborate on graph transformations. Several approaches have been developed that use graph grammars to transform from source to target models [86, 83, 53, 76]. The description of basic concepts of graph grammars are based on the very intuitive presentation in [34]. As the name already suggests, graph transformations are highly based on graphs, which consist of vertices V and edges E. A vertex $v1$ in V is connected to say $v2$ through an edge e such that $s(e)$ yields the source $v1$ and $t(e)$ yields the target $v2$. Also, we can distinguish between two different kinds of graphs, i.e., type graphs and instance graphs. Type graphs capture concepts from the real world, which have been derived by generalization of real world entities. Based on these types specified in a type graph, we can build instance graphs that represent snapshots of possible realizations. We could also think of type and instance graphs as metamodels and models, respectively. Having specified the modeling space for very basic graph transformations, we now turn to the transformation rules that operate on the instance graphs. The specifications of these rules is also done by means of graphs. The concrete syntax for specifying these rules is the same as the one used to specify instance

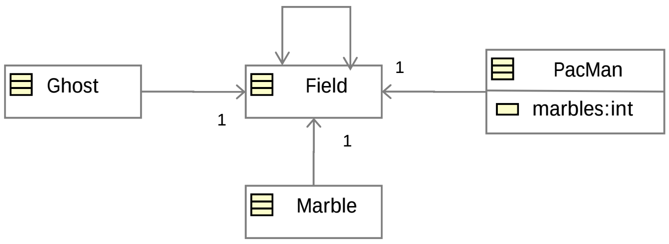


Figure 2.2: PacMan type graph.

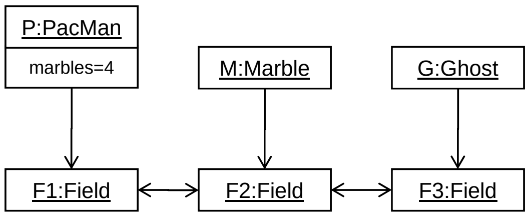


Figure 2.3: Instance graph for the PacMan type graph.

graphs, which in turn is based on UML object diagram concrete syntax. In order to explain basic semantics of these graph transformation rules we give a small example. The example is again taken from [34] due to simplicity and didactic reasons.

Example 3. *Everyone knows the PacMan game. It should therefore be no problem to construct a metamodel for this simple game, which contains the concepts of $Field$, $Ghost$, $Marble$ and $PacMan$ as shown in Figure 2.2. The type graph also contains the relationships through which these four generalizations of game concepts are related to each other. Figure 2.3 depicts a corresponding instance graph. We now aim at moving our PacMan instance P to the Field F2 in order to acquire the Marble M. Therefore this instance graph can be considered as source graph of our transformation. A sample transformation rule that moves our PacMan instance to Marble M is specified in Figure 2.4. Formally, a graph transformation rule is specified as $p : L \to R$, where p is the name of the rule and L and R are possible subgraphs of source and target instance graphs G and H. Our source instance graph is depicted in Figure 2.3. The subgraph L can be considered as precondition and R as post-condition. So, whenever L matches in a source instance graph the postcondition R has to create the corresponding target graph H. Our move(p) rule matches a subgraph of our source instance graph G and therefore transforms the affected regions into an target instance graph, where P:PacMan moves from F1:Field to F2:Field, the M:Marble is removed and instead the attribute marbles of P:PacMan increased to 5.*

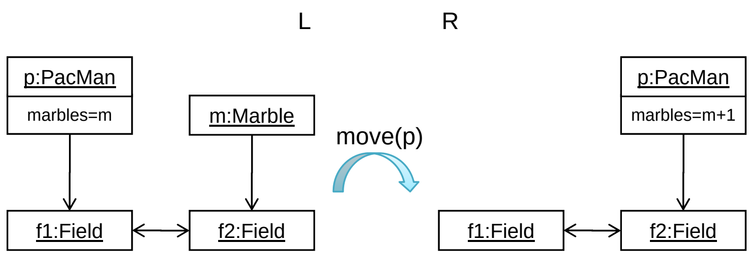


Figure 2.4: Graph transformation rule that moves PacMan.

2.2.4 Advanced Model Transformation Features

Generic Model Transformations. Typically model transformation languages, like ATL or graph grammars, 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. Varró et al. [85] define a notion of generic transformations, which are closely related to templates of C++ or generics in Java. Both templates and generic transformation rules are not bound to specific types at design time, but dynamically bound at runtime. Generic transformation rules are best suited for recurring transformation problems such as transitive closures. These generic transformations are supported by the VIATRA2 framework, implemented as an Eclipse plug-in. In contrary to our approach VIATRA2 does not foster an easy to debug execution model. Graph transformation rules in VIATRA2 may be further extended by Abstract State Machine rules to introduce control mechanisms for graph transformation rules and, hence, formulate complex transformation programs. This introduces an additional formalism within the framework. In addition there exists no explicit mapping model between source and target model.

Transformation Patterns. Very similar to the idea of generic transformation is the definition of reusable idioms and design patterns for transformation rules described in [1]. Instead of claiming to have generic rules, the authors propose the documentation and description of recurring problems in a general way, i.e. the definition of patterns. These patterns, which are independent of any type system in terms of a metamodel, may be reused by the transformation engineer in a concrete scenario. Summarizing, this approach solely targets documentation, whereas implementation issues on how these patterns could be implemented in a generic way - remain open. [1] does support idioms, which describe recurring design problems specific to a certain application domain.

Mappings for bridging metamodels. Another way of reuse can be achieved by the abstraction from transformation rules to model mappings as is done in our framework or by the ATLAS Model Weaver (AMW) [26]. AMW lets the user extend a very generic weaving metamodel, which allows the definition of correspondences between any two metamodels. Through the extension of the base weaving metamodel one can easily define new weaving operators or reuse existing ones of other weaving languages. The semantics, of these weaving operators is determined by the transformation model that takes the concrete weavings as input and generates a transformation model operating on the concrete models of two modeling languages. This transformation model acting upon the weaving model and producing a transformation model by itself is called Higher Order Transformation (HOT). For an application of the combination of AMW and HOT using Atlas Transformation Language [2] see [27]. The semantic of weaving operators, is specified by means of the HOT. This makes it very difficult to debug a possibly unintended result of the transformation process.

Weaving models as introduced by [26] are considered as specialization of the more general concept of a mapping model. In [33] the authors present an approach, which allows the definition of inter-model mappings using UML. They build on the semantics of mathematical relations and apply those definitions on model mappings in the software engineering field. With these formal foundations they define the visual syntax and the semantics of model mappings by relying on and extending the UML.

2.3 Why We Benefit from Examples?

Examples play a key role in the human learning process. There exist numerous theories on learning styles in some of which examples are to be seen as major artifact. For a description of today's popular learning style theories see for example [60, 28, 44]. All these theories, especially the referenced ones, are quite similar concerning their notion of a learning dimension, e.g., visual, sensing, active, and reflective. The sensing dimension promotes the use of examples.

Examples have proven to be helpful in the deduction of general rules, as exemplified in [63]. It is therefore by no accident that instructive and intuitive examples are common tools in teaching of computer science. Especially in those fields that are strongly driven by algorithms and mathematics. Examples help to introduce abstract terms and definition such as a state space. Nievergelt and Behr [63] use the cannibal and missionary puzzle to introduce state spaces and in succession generalization from a concrete problem. This way of presenting complex and abstract theories and models makes them understandable to beginners and therefore targets the needs of students.

In order to better understand this example-based strategy to the understanding of complex scientific results we will motivate the field of modal logic by Example 4.

Example 4. *When trying to introduce modal logic to students, often the same intuitive and simple example is chosen. In the literature this example is referred to as "'wise-men puzzle"' [36]. Basically the problem situation looks as follows. There are three wise men and five hats, which three of them are red and two of them are white colored. Each wise men now gets one hat put on, but does not see*

	M1	M2	M3
1.	R	R	R
2.	R	R	W
3.	R	W	W
4.	W	R	R
5.	W	W	R
6.	R	W	R
7.	W	R	W

Table 2.1: Wise-men puzzle: state space.

of which color it is made of. Instead he can see what color the hats of the other two wise men have. Suppose now that the first two wise men say they don't know which color their hat has. The question remains: Does the third man know, which color his head is made of?

A solution for this problem can be easily found by first writing down all possible states as is shown in Table 2.1. M1...M3 denotes each of the wise men and W and R stand for white and red, respectively. In order to solve the problem we can eliminate solutions that can't be true, assuming that the first two wise men have said the truth and are indeed wise concerning their reasoning capabilities. Solution number 3 and 7 can be deleted right away as one of the two wise men would know the color of his hat. We can delete one more possible solution, if we let the second wise man take the first man's statement into account. His not knowing the solution too, we can also exclude number 2. Because if 2 was the solution the second man could reason about the first man's statement and conclude that his own hat was red. But he does not know his hat, so we can erase 2 as possible solution. This last step is of course most difficult and needs usually some time to think about. Now that we have only solutions number 1,4,5 and 6 left we can say for sure that the third man wears a red hat.

This relatively simple example of reasoning about knowledge has approved itself in lectures about modal logic as a starting point. The intuitive way of finding a solution, e.g., for the wise men puzzle may serve as basis for a more general, formal way of solving such problems in computer logics. One modal logic capable of formalizing and solving the wise men puzzle would be $KT45^n$ [36].

2.4 Common By-Example Approaches

MTBE was inspired by popular *by example approaches*, some of which we present briefly in the following subsections. But what does by example really mean? What do all these approaches have in common? The main idea, as the name already suggests, is to give the software kind of examples, how things are done or what the user expects, and let it do the rest automatically. In fact this idea is closely related to fields such as machine learning or speech recognition. Common to all *by example approaches* is the strong emphasis on user friendliness and a "short" learning curve. According to [20] the *by example paradigm* dates back to 1970 - see “Learning Structure Descriptions from Examples” in [90].

2.4.1 Query By Example

One of the most well known *by example approaches* was developed by Zloof in 1975. It is called Query by Example (QBE) [93] and acts as a visual language for relational data base management and manipulation like SQL. Data base operations originally supported by this approach are:

- queries on tables,
- insertions on tables,
- deletions on tables,
- updates on tables,
- table creations,
- table updates,
- query data base meta information.

The main advantage of QBE lies in its simplicity, which makes QBE easy to learn and use. This is achieved by imitating the user's thought process, when it comes to formulating a request to the data base. Also, this concrete syntax for the formulation of requests has the same style and uses the same operations throughout QBE, i.e., for each of the above mentioned data base operations. The functionality is however limited compared to data base query languages, such as SQL. With the rise of abstractness and reduction of complexity one looses flexibility in the design of problem solutions. QBE is therefore not used for any data base management scenario.

Figure 2.5 shows a simple QBE query demonstrating how the visual syntax looks like.

The central syntactical artifact in QBE is the skeleton table, which is empty at the beginning of every QBE application. An arbitrary number of such empty tables will be instantiated upon one working canvas, if operations on more than one table is required by the user. These tables serve as open space for examples given by the user. Tables in QBE allow for two different inputs:

- constant elements (e.g. 1980-01-27 in Figure 2.5) and
- example elements also called variables (e.g. ANY in Figure 2.5).

PERSON	SSN	NAME	DAY OF BIRTH
	P.$\underline{\text{ANY}}$	P.$\underline{\text{ANY}}$	1980-01-27

(a)

PERSON	SSN	NAME	DAY OF BIRTH
	P.$\underline{\text{ID}}$	P.$\underline{\text{ANY}}$	1980-01-27

PROJECT	SSN	PROJECTNAME
	$\underline{\text{ID}}$	P.$\underline{\text{ANY}}$

(b)

Figure 2.5: Simple retrieval of data within table PERSON.

Note that example elements are always underlined. *P.* has the meaning of print and therefore selects all the columns that should be presented in the result set of the query. Example elements or variables are - as the name suggests - placeholders for any single tuples stored in a specific column in that table. It follows that for every column a variable preceded by a print operation *P.* is defined the result table for those columns is returned. This is of course equivalent to projection $\pi_{column,...}(Table)$ in relational algebra. Constant elements on the contrary represent a concrete value of an attribute that has to be matched in order to return a result set. Moreover QBE allows the specification of conditions upon attribute values that must be satisfied.

Example 5. *Having these syntactic and semantic definitions in mind one can easily understand the example given in Figure 2.5(a). This QBE query selects the tuples consisting of SSN and NAME from the table PERSON, whose DAY OF BIRTH equals 1980-01-27. In SQL this QBE query would be expressed as follows:*

```
SELECT SSN, NAME
FROM PERSON
WHERE DAYOFBIRTH='1980-01-27';
```

Example 6. *Joins upon several tables can be achieved by using the same variable names across different tables as depicted in Figure 2.5(b). Here the variable name* $\underline{\text{ID}}$ *is referenced by table PROJECTS to ensure, only tuples, that are related via a distinct SSN to each other, form a solution. Again, in SQL this QBE query looks like:*

```
SELECT SSN, NAME, PROJECTNAME
FROM PERSON p, PROJECT pr
WHERE p.SSN = pr.SSN AND DAYOFBIRTH='1980-01-27';
```

From these two examples one recognizes that the query formulation in QBE is more intuitive and easier to learn in comparison to the query language SQL. Especially when more complex tasks, such as grouping, are used, SQL queries soon get harder to grasp and maintain. The prime target audience of QBE are of course people who are not familiar with programming languages.

2.4.2 Programming By-Example

Programming by Example (PBE) is also known in literature as *Programming by Demonstration*. In any case the paradigm of programming by means of visual tool support aims to simplify the task of creating a computer program. Instead of hand coding every single instruction of an executable program, a tool supports the user by visual abstractions from common programming languages. There exist various approaches how these abstractions take place. In the following we give a few examples that illustrate some of these approaches.

Example 7. *The Stagecast Creator[3] [79, 77] is kind of an integrated development environment (IDE) that lets one create simulations and games without a programming language just by knowing semantics of visual abstractions and rules. This tool originated from KidSim developed by Cypher et al. in 1994 [78]. The major contribution of the Stagecast Craeator is the combination of the programming by demonstration (PBD) and the visual before-after rules paradigms. The former is concerned about the way the user can record a certain behavior of the future program whereas the latter is concerned about the visualization of the instructions recorded in this PBD manner.*

Example 8. *Consider Excel macros, which record every step the user takes to later automatically do exactly the same again without user input. No command of Visual Basic is needed in order to perform complex tasks. This approach is best described by the metaphor: "Watch what I do".*

Example 9. *Another example for PBE could be the flash timeline and its support for enabling automatic animation creation. Flash also implements visual before-after rules as the user specifies some sort of pre and post conditions for Flash instances on different keyframes, which represent discrete points in time. Figure 2.6 shows various screenshots demonstrating this way of specifying animations. At first the user inserts a keyframe at the beginning and draws a circle. Next she creates another keyframe some moments ahead and moves the circle, which has automatically been pasted at the same position as the first one, to some place on right side of the canvas. Finally a motion tween is inserted between these two points in time and the frames in between, capturing the movement of the circle, are interpolated accordingly. This is in fact a major relief in the development of animations. The Flash Actionscript 3.0 code, which would do this animation, is depicted in Listing 2.3.*

Listing 2.3: ActionScript 3.0 code for the transition presented in Figure 2.6.

```
import fl.motion.Animator;
var test_xml:XML = <Motion duration="18" xmlns="fl.motion.*"
  xmlns:geom="flash.geom.*" xmlns:filters="flash.filters.*">
  <source>
    <Source frameRate="12" x="150" y="151" scaleX="1" scaleY="1"
      rotation="0" elementType="movie␣clip" instanceName="test" symbolName="wheel">
```

[3]www.stagecast.com

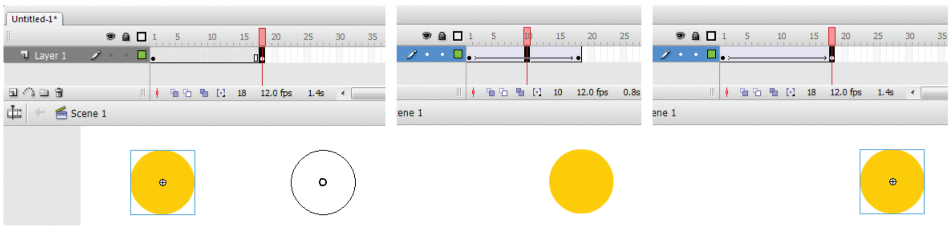


Figure 2.6: Simple transition in Flash using motion tweens.

```
          <dimensions>
            <geom:Rectangle left="-40" top="-40" width="80" height="80"/>
          </dimensions>
          <transformationPoint>
            <geom:Point x="0.5" y="0.5"/>
          </transformationPoint>
        </Source>
      </source>

      <Keyframe index="0" tweenSnap="true" tweenSync="true">
        <tweens>
          <SimpleEase ease="0"/>
        </tweens>
      </Keyframe>

      <Keyframe index="17" tweenSnap="true" tweenSync="true" x="151.95">
        <tweens>
          <SimpleEase ease="0"/>
        </tweens>
      </Keyframe>
    </Motion>;

    var test_animator:Animator = new Animator(test_xml, test);
    test_animator.play();
```

Example 10. *Besides the paradigm of PBE there exists the one of visual programming languages, which seems to be closely related to PBE as it also fosters the development of complex programs by means of visual abstractions. An example of these visual programming languages would be Quartz Composer of Apple Inc. coming along since Mac OS X v10.4 Tiger. Quartz Composer is used for efficient creation and prototyping of animations and simulations. A domain specific modeling language (DSL) is used to represent the data that is then rendered with OpenGl. These animations can later be easily integrated in other development environments such as Apples Cocoa or Carbon.*

2.4.3 Web-Scheme Transformers By-Example

Lechner et al. [47] follow the original approach of QBE, but with extensions for defining scheme transformers, which is demonstrated in the area of web application modeling with

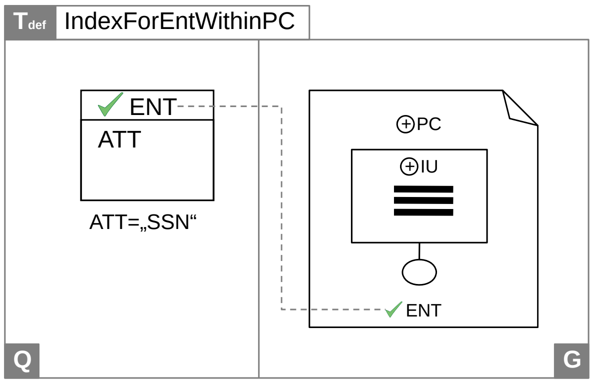


Figure 2.7: Simple TBE rule.

WebML [15]. Therefore, the original QBE approach is extended by introducing a generation part (WebML model after transformation) in the template definitions in addition to the query part (WebML model before transformation). Finally, XSLT code is generated to transform the WebML models, which are represented as XML files within the accompanying tool *WebRatio*. This approach is called transformers by-example (TBE).

Example 11. *Figure 2.7 shows the transformer graphical concrete syntax of the TBE approach. On top of the rectangle appears the name of the transformer. On the left marked with a Q resides the query template of the transformer. And on the right marked with a G we find the generative template. Within those parts we find elements from the WebML language specification. Our query part now selects all entity types ENT that are composed of an attribute ATT called SSN and transforms them into a page class containing an index unit for a specific entity type. This entity type is however connected to those entity types ENT from the query part, which have at least one attribute with name SSN. We have highlighted this relationship with a dashed line.*

Chapter 3

Prerequisites for MTBE

Contents

Before we introduce our MTBE approach we provide a detailed discussion on the terms and technologies used to realize MTBE, both conceptually and practically. Metamodeling may be performed in various ways leading to heterogeneities or hidden concepts on the metamodel and model layer. Therefore, any MTBE approach needs to be aware of these heterogeneities because of their strong impact on the model transformation code. One way to cope with these heterogeneities and to solve them is based on refactoring patterns. This is shown in this chapter. Additionally, we present a metamodel metric that is used to detect some heterogeneities or give an overview to what extent they are prevailing. Furthermore, we explain the difference between concrete syntax and notation, as these two concepts are central to the MTBE approach.

3.1 The Metamodeling Stack

Model Transformation By-Example defines not a new transformation language but instead relies on existing implementations of such languages. Those implementations in turn rely on the metamodeling structures of the development environment, such as Eclipse. In the case of MTBE we build upon the Eclipse Modeling Framework (EMF) [12]. The EMF defines Ecore, which acts as a basic modeling language for the creation of metamodels. Ecore is basically an implementation of the OMG's Meta Object Facility (MOF) [68]. However, instead of implementing all details of MOF, Ecore comprises just the EMOF specification and, thus, leaves out many of the core constructs of UML 2.0. The Ecore model is depicted in Figure 3.2 and is described in more detail in Subsection 3.1.1.

The UML Infrastructure specification [65] presents a four-level metamodel hierarchy that relates MOF to UML, Models of UML and runtime instances based on these UML models. The levels are labeled as M3, M2, M1 and M0. Between these layers there exist so called «$instanceOf$» relationships. The metamodeling architecture is also shown in Figure 3.1. In the context of MTBE we refer to the metamodeling structure as modelware technical space. We replaced the MOF model on M3 by Ecore that acts as our metametamodel for defining metamodels, e.g. $MM1$, to represent modeling languages. The relation among models on different layers is stereotyped by the label «$conformsTo$» as suggested in Bézivin [13]. The MTBE approach is concerned with the modeling languages and their models. Since, we do not include runtime instances in our work, we shaded this layer in Figure 3.1. The major artifacts in MTBE are therefore metamodels and models.

3.1.1 The Ecore Model - Core Concepts Reviewed

As mentioned earlier the Ecore model is an implementation of the EMOF specification with a focus on tool integration and interoperability issues, in contrast to the OMG's focus on meta data repositories. Figure 3.2 shows nearly all class definitions of Ecore. Meta classes in gray represent abstract classes. References in blue depict so called derived *EReferences* and are computed automatically from other existing and required *EReferences*. The core concepts of Ecore and for the building of metamodels are the metaclasses *EClassifier* and *EStructuralFeature* with their concrete specializations *EClass*, *EAttribute* and *EReference*, respectively. As described in the figure *EAttribute* and *EReference* inherit both from the super class *EStructuralFeature* and are contained via the *eStructuralFeature* reference in an *EClass* instance. Ecore supports all simple Java types like boolean, int, float as well as various

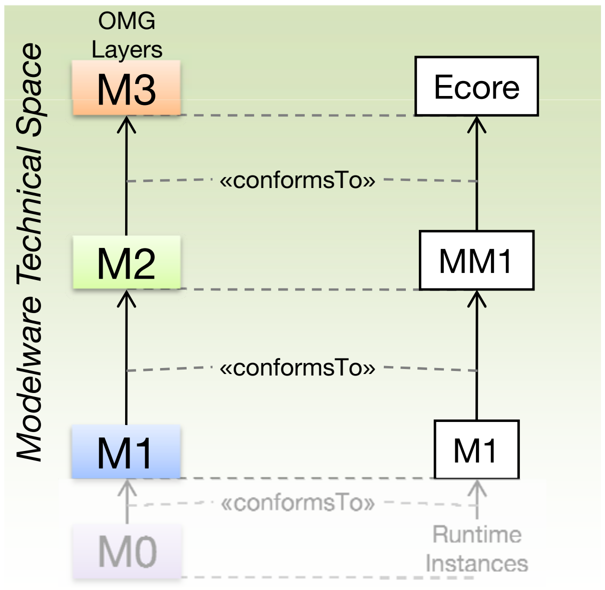


Figure 3.1: The OMG's metamodeling stack in the light of MTBE [68]

object types such as java.util.Date. For the definition of modeling languages also meta-classes for Operations or Packages for structuring are provided. The *EFactory* reflects an implementation specific design pattern as the name already suggests. *EModelElement* and *ENamedElement* may be seen as convenience classes encapsulating common features.

3.1.2 Differences between Ecore and MOF

Although the above presented model strongly relates to MOF, there exist some major differences in the design of these two meta modeling languages, which are briefly discussed in this subsection.

- **References vs. Associations** EMOF does not support the modeling of relationships between elements. CMOF on the other side imports UML constructs to attain the concept of an *Association*. This *Association* class is responsible for navigability expressed by *ownedEnd* and *navigableOwnedEnd* features pointing to *Properties* acting as roles. For a visualization in terms of a UML class diagram see [68]. Because of the use of UML association this also includes the concept of a role. Formal Semantics for both are found in [56]. In general, UML associations are bidirectional allowing to obtain the classes on each end. In Ecore we have the concept of an *EReference* in terms of a meta class to establish links or edges between modeling entities or nodes. An *EReference* is

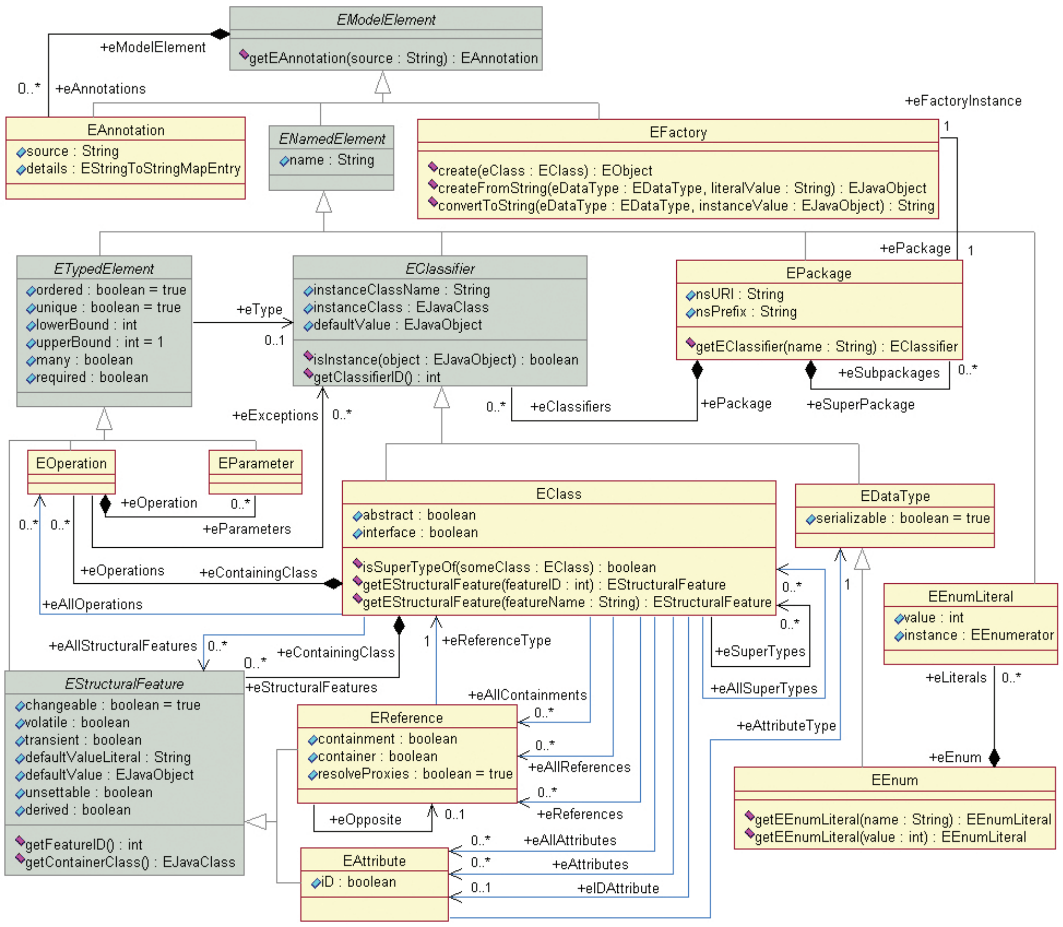


Figure 3.2: The Ecore metamodel for creating models [22].

always part of an *EClass* an has an *eType*, which points to one specific *EClass* including the one it is contained within to allow for self references. In contrast to MOF ,an *EReference* only allows navigating in one direction, i.e., an instance of *EClass* containg the *EReference*, navigates to the the other *EClass* but not vice versa. In order to allow navigating both directions one has to define a second *EReference* on the other *EClass* and to declare the two separate *EReferences* opposite to each other with the *eOpposite* feature. Another difference is that Ecore does not explicitly support the concept of a role. *EReferences* do have attributes for name and multiplicity, but this is not a compensation for UML roles.

- **Generalization** When modeling generalization among *EClasses* one has to be aware of the fact, that although Ecore allows multiple inheritance by defining *eSuperTypes* with multiplicity $0..n$, Java does not. However, Java offers an alternative for multiple inheritance by providing the construct of an interface that is implemented by sub classes acting as kind of a super class. But we can specify, whenever inheriting from at least two *EClasses*, which of the *EClasses* shall be the one getting extended in Java. In graphical concrete syntax this is annotated in a concrete metamodel with the stereotype «$extends$» at one generalization link.

- **Data Types** At the data type level, Ecore provides Java simple types and some object types as explained above. MOF on the other hand imports the UML PrimitiveTypes package from the UML infrastructure, which consists of four basic primitive types that shall be reused.

3.1.3 Models and Transformations

Now that we have introduced a language for meta modeling and have explained its core constructs, we turn to the field of model transformations and how they fit into our modelware technical space. Bézivin et. al. [9] argue for model transformations being just another kind of models, i.e., transformation models. They say:

"Model transformations can be abstracted to a transformation model."

Figure 3.3, which is based on their work, illustrates the general notion of a transformation model. Refinements of this presentation are found in [9]. In the left part of the figure we have marked the OMG's layers as well as corresponding example modeling artifacts $MM1$ and $M1$. Further, we have sketched in a second metamodel $MM2$ with corresponding model $M2$. As we already know from Chapter 2, a model transformation operates on a given source model to produce a specified target model. In addition, this model transformation has to be aware of the metamodels representing the types or

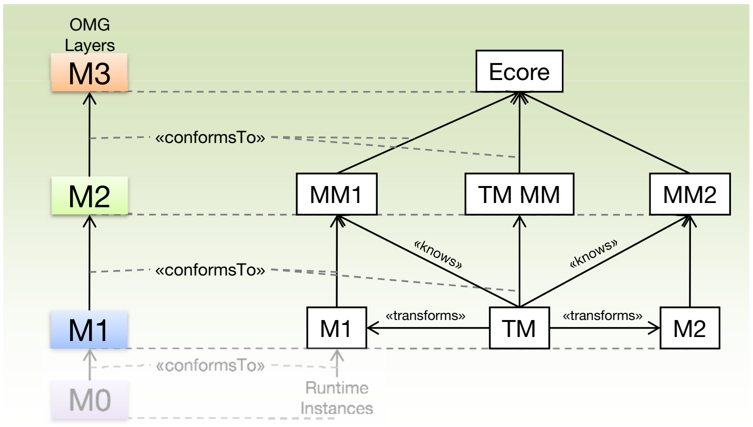


Figure 3.3: Model transformations in the modelware technical space.

concepts of these models, i.e., the instances of these metamodels. A model transformation is interpreted as a transformation model, e.g., TM in Figure 3.3, which is an instance of a transformation metamodel $TM\ MM$. This metamodel for defining model transformations is in turn an instance of the metametamodel Ecore. In Figure 3.3 following [9] we introduce two new association types. These are «$transforms$» and «$knows$». So a transformation model needs to know at least two different metamodels and transforms at least between two models. Note that we do not differentiate between source and target models. In general, we believe a transformation should not have a distinct direction. Instead, it should be possible to apply a transformation model both directions. However, this does not apply to lots of transformation language implementations. We also point out that a transformation model can be applied to various input models and can generate more than one output models at a time.

3.1.4 Horizontal vs. Vertical Model Transformations

MTBE primarily targets the use case of horizontal model transformations. But what about vertical model transformations from, e.g., platform independent to platform specific models (PIM, PSM)? A vertical model transformation results in a decrease in abstraction, whereas a horizontal model transformation leaves the level of abstraction unchanged. If UML is used to create an abstract model, i.e. a PIM, one can easily define stereotypes to extend the UML metamodel. The mapping of stereotypes has not yet been tested in the context of MTBE. Though this type of extension mechanism should not pose any problem.

Another way to model in a more concrete way building a separate metamodel. Then all concepts are represented in terms of a metamodel independent of the model. Taking

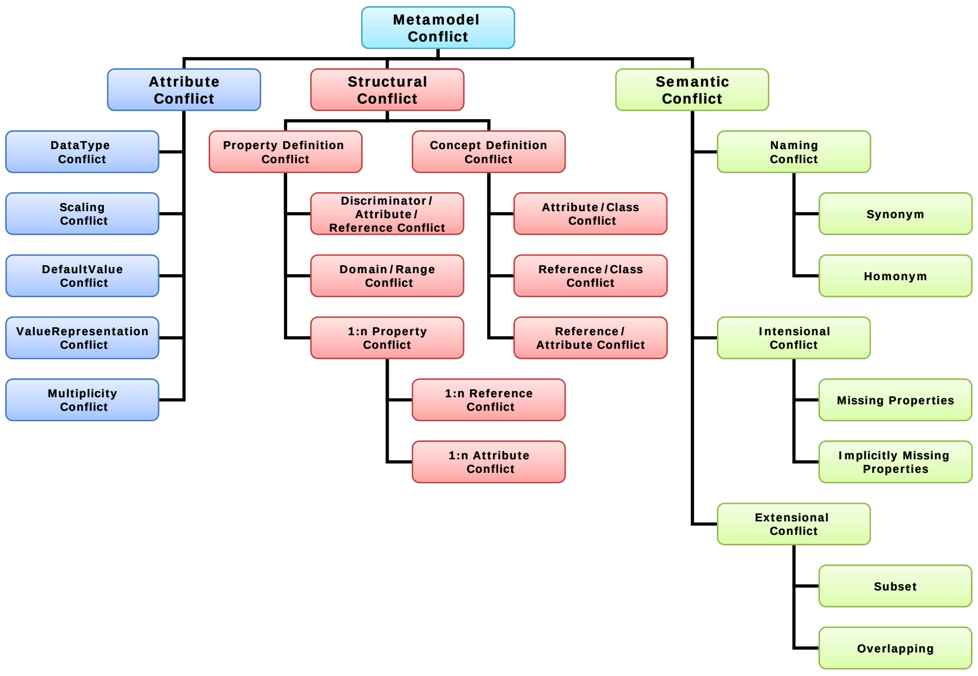


Figure 3.4: Taxonomy of schema integration conflicts.

two different metamodels as input MTBE can also be applied to vertical model transformations if desired. The reason why vertical model transformations are not the main objective of MTBE is, that there are typically no such big differences between the two metamodels neither in graphical notations nor in the concepts being used. We think that automated matching tools such as Coma++ [6] produce faster mapping results than manually defined correspondences.

3.2 Metamodel Heterogeneities

There has been a tremendous amount of work on analyzing and classifying different types of structural integration conflicts between schemas which represent the same "real world" domain. For this work, we reuse classifications of structural integration conflicts from [Kim], [KashyapSheth], [Härder], [Conrad], [NaumannLegler], and [Schmitt] as a basis for our classification of model integration conflicts.

When using a meta modeling language, such as MOF, it is quite likely that semantically equivalent modeling concepts are defined in different ways with different MOF concepts. Therefore, in the next paragraphs we look at integration conflicts between semantically equivalent metamodels which are not identically defined. Instead, they are defined using different MOF modeling concepts and modeling styles. Nevertheless, they present the same modeling concepts.

Figure 3.4 represents our taxonomy of schema integration conflicts which are classified into the following three main categories:

- Attribute Conflicts: This category covers conflicts which may occur between two corresponding attributes, i.e., the attributes are equivalent but not identically defined. Consequently, the values of the attributes are not the same, although the represent the same information.
- Structure Conflicts: Different structures representing equivalent modeling languages are the result either of defining properties of modeling concepts in different ways or of defining properties as concepts and vice versa.
- Semantic Conflicts: This category covers conflicts due to using different names for the same concept (synonyms) or same names for different concepts (homonyms). Furthermore, a concept may be a subset of another one or concepts may overlap. These four conflicts are with regards to extensional conflicts. In addition, we look at intensional conflicts such as (implicitly) missing properties.

In the following subsections, we elaborate on each category in more detail. In particular, we first describe the integration conflict in general and then discuss a typical integration example.

3.2.1 Attribute Conflicts

An attribute conflict occurs, when a semantic relationship between values of two attributes exists, but the attributes are representing the same information with different values or the attributes have different meta-properties. When using MOF-based metamodels, the following mismatches between attributes may occur:

- Data type: Semantically equivalent attributes can have different data types. Consequently, attribute values are syntactically different represented. In such cases it must be observed, if the values may be converted (restricted or unrestricted) or not. In MOF Strings, Integers, and Booleans may be used as data types.

  Example 12. *Figure 3.5 illustrates the case of a data type mismatch, where in language (a) the upper multiplicity attribute maxMulti is ot type Integer whereas in (b) the maxCard attribute is of type String.*

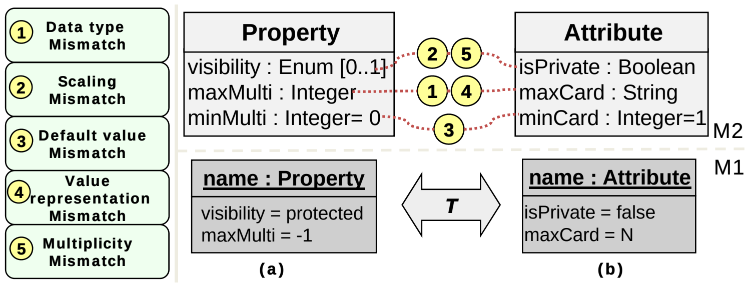


Figure 3.5: Attribute integration conflicts, (a) and (b) being different modeling languages.

- Default value: If default values are varying and the multiplicity of at least one attribute is optional, i.e., *zero-to-one* multiplicity, it must be ensured that the default values correspond to each other or a proper mapping function for the default values is provided.

  Example 13. *Mapping 3 in Figure 3.5 represents a default value mismatch for the attributes minMulti and minCard, whose default values are 0 and 1, respectively. As these defaults are not identical a separate mapping function has to be provided for the an integration task.*

- Multiplicity: An attribute has an upper and lower multiplicity which are typically *zero-to-one, one-to-one, zero-to-many, or one-to-many*. The default value for an attribute's multiplicity constraint is *one-to-one*. Conflicts may occur when two corresponding attributes have different upper or lower multiplicities. For example, when one attribute has a multiplicity *zero-to-one* and the other *one-to-one*, it must be ensured that the *one-to-one* attribute is set in any case, also when the *zero-to-one* attribute is not set.

  Example 14. *The attribute visibility in Figure 3.5(a) has a multiplicity of zero-to-one in contrast to isPrivate in (b), which is required.*

- Scaling: Attributes are often differently defined with respect to the meaning of scales. Furthermore, this category also comprises cases in which one domain is a subset of another one or domains are overlapping. Mappings between attributes having different scales can be expressed in terms of functions or lookup tables. However, this conflict is often responsible for applying injective instead of bijective mappings.

  Example 15. *The Visibility attribute of Property in Figure 3.5 is of type Enum, which may be any possible visibility feature such as private, public, protected and package. The attribute isPrivate of Attribute is of type Boolean and can therefore only represent a subset of possible values compared to the Visibility attribute.*

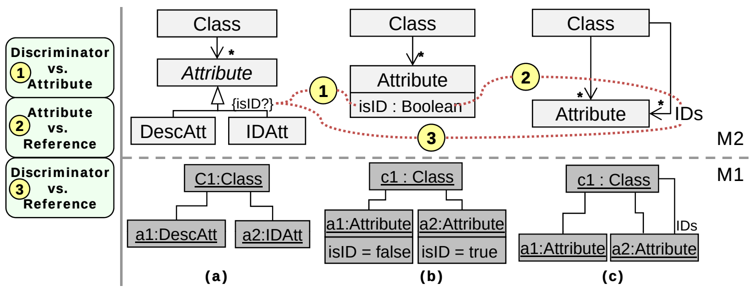


Figure 3.6: Property expressed as (a) discriminator, (b) attribute, (c) reference.

- Value representation: It is also possible that semantic equivalent attribute values are varying in their representation. This is typically the case when different symbols are used for the same concepts.

  Example 16. *The data type mismatch in mapping 1 of Figure 3.5 also has value representation mismatch as consequence as different symbols are used for specifying upper bounds of multiplicities. In (a) we could encode unbounded multiplicity by the value -1 whereas in language (b) we could use a * to allow for an arbitrary number of elements.*

3.2.2 Structural Conflicts

This category comprises integration conflicts arising when equivalent modeling concepts are expressed with different metamodel structures. Different structures mainly result on the one hand from properties of equivalent modeling concepts defined with different modeling elements (cf. Figure 3.6a property definition conflicts) and on the other hand from the fact that concepts may be represented by attributes or references instead of classes (cf. Figure 3.6b concept definition conflicts). In the following, we elaborate on these two conflict subcategories.

Property Definition Conflicts

Discriminator/Attribute/Reference Conflict: MOF allows many ways to define properties of modeling concepts. In practice, three distinct variants of property definitions are often occurring in metamodels. The first variant is that a property can be expressed via a discriminator of an inheritance branch and at runtime the property can be derived by looking up the instantiated class. The second variant is that the property is defined with an additional attribute, and finally the third variant is that the property can be expressed by using an additional reference.

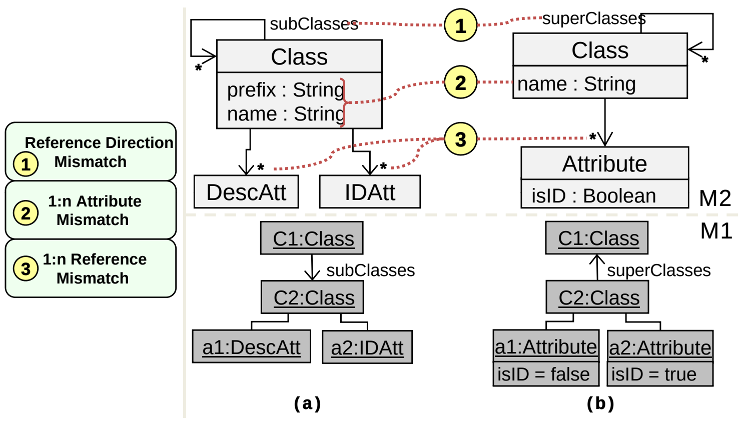


Figure 3.7: Reference direction and property set/subset mismatch.

Example 17. *In Figure 3.6 the mentioned variants are shown by a concrete example. Each variant represent the same information, namely that an attribute is either identifying or descriptive, however the metamodels have quite different structures. Figure 3.6(a) shows that a Class can have arbitrary Attributes whereas Attribute is an abstract class and only the concrete subclasses DescAtt and IDAtt can be instantiated. In Figure 3.6(b) an equivalent definition is illustrated by using the attribute isID in the class Attribute which is used as a flag, instead of using additional subclasses. Finally, Figure 3.6(c) uses an additional reference IDs to mark the set of identifiable attributes. Even though, each metamodel leads to different abstract syntaxes of the models, they represents the same information. Consequently, there exists an isomorphism for transformation models between these representations without loosing information.*

Domain/Range Conflict: This integration conflict results from the fact that in MOF only uni-directional references can be modeled, i.e., no bi-directional association as in UML are possible, and in most cases it does not make any difference, if a reference is modeled from the class A to the class B or vice versa. For example, when one wants to define the concept of inheritance between classes in a metamodel, this can be done by saying "a class has arbitrary subclasses" or alternatively "a class has arbitrary superclasses". It has to be mentioned that independent which alternative is chosen, it is possible to compute the inverse reference.

Example 18. *Figure 3.7 shows the aforementioned example of defining the inheritance concept for classes by means of references (cf. Figure 3.7 subClasses and superClasses) which are the inverse to each other. If one wants to transform models from the left to right hand side, the reference superClasses must be computed as by the following OCL constraint.*

```
Reference superClasses:= Class.allInstances()
->select(e|e.subClasses = obj1);
```

An analogous computation has to be done when moving form the right to left hand side for the reference subclasses as shown below.

```
Reference subClasses:= Class.allInstances()
->select(e|e.superClasses = obj1);
```

1:n Property Conflict

1:n Reference Conflict: This kind of conflict category represents the case that one reference of a metamodel corresponds to more than one references of another metamodel. The additional complexity of these conflicts is that when moving from one to n references, we have to split the single reference in several subsets by looking at attribute values or reference links of the referenced objects. Consequently, when moving form n references to one reference, the reference sets must be united to one set.

Example 19. *Figure 3.7 illustrates the mismatch of sets in mapping 3. In (a) a Class can have two different sets of attribute types, i.e., DescAtt and IDAtt, whereas in (b) Class just references Attributes. Due to mapping 3 these we have to merge the sets of DescAtts and IDAtts when transforming from (a) to (b) or split the set of Attributes when transforming from (b) to (a). Note, that in the latter case a proper constraint has to be defined on the Attribute of (b), which allows for splitting up a single set into two.*

1:n Attribute Conflict: An attribute value normally contain one information unit, however, sometimes, one attribute value represents several information units. This typically is a design decision if one attribute is used to store more information units or if for each information unit a separate attribute is modeled. If an attribute is used to hold more than one information units and corresponds to a set of attributes, the single value must be split into several values, which are then assigned to n attributes. Going the other way round, several values must be concatenated into one.

Example 20. *In Figure 3.7 an attribute conflict is shown in mapping 2. Here we have a two-to-one mapping from attributes prefix and name to name. Prefix and name must therefore be concatenated when moving from (a) to (b) and name must be split properly when moving from (b) to (a). For the latter case a meaningful mapping must also specify proper tokens or constraints, which allow to split up a single value.*

Property/Concept Conflicts

Attribute/Class Conflict: An attribute/class mismatch occurs, when a concept is modelled as an attribute in one metamodel and in another metamodel as a class, which contains the attribute. This means, on the instance level, the attribute values must be converted into objects and also the structure must be ensured by linking the two objects.

Example 21. *As an example consider mapping 1 in Figure 3.8, where on one side of the mapping attributes maxCard and minCard are contained in the "main" class Attribute itself (cf. a), but on the other side these attributes are separated into the class Multiplicity.*

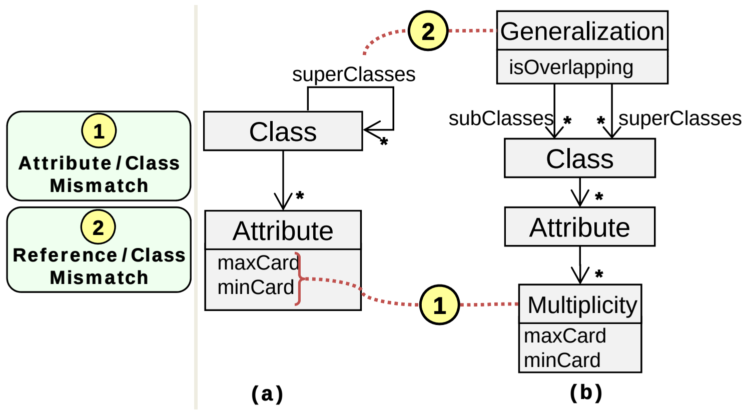


Figure 3.8: Concept definition conflicts.

Reference/Class Conflict: In one metamodel, only a reference is modeled and in another metamodel a class is defined, which can be seen similar to an association class in UML. This means, on the instance level, the reference link must be converted into an object and for ensuring the structure the object must be linked with the containing and referencing objects of the source reference.

Example 22. *Figure 3.8 illustrates a reference/class conflict for the concept of generalization of (a) and (b) in mapping 2. In (a) we modeled the the generalization concept as simple reference called superClasses. However, in (b) we introduced an additional class called Generalization to model inheritance graphs.*

3.2.3 Semantic Conflicts

Naming Conflicts

Under the category naming conflicts we subsume conflicts due to lexical representation of element names. This kind of conflict only complicates the discovery of semantically related elements, i.e., matching phase, however, after recognizing naming conflicts, no additional complexity results in the mapping phase.

Synonym Synonyms describe the same concept, however, they use different terms. In metamodels, attributes, references, and classes can be named differently, although they represent the same concept.

Homonym Homonyms use the same terms, however, they elements stand for different concepts. This means, in metamodels, elements with the same name can stand for different concepts.

Extension Conflicts

The extension of a set is its members or contents. This means, talking about metamodel classes, the extension defines all instances of a class. When comparing the extensions of two semantically related classes, the following situations may occur: *equal, distinct, subset, and overlapping*. The first two situations are no source for integration conflicts; however, the second two are.

No Conflict Cases *Equal:* This means, two concepts are semantically equivalent and also the extensions of the two concepts are the same. In such cases, a simple one-to-one correspondence is enough for describing the mappings, which typically do not lead to an integration conflict. *Distinct:* If the extensions of two concepts have no intersection, then no semantic relationship should exist between the two concepts. Consequently, no mappings should occur between the two concepts not leading to an integration conflict.

Conflict Cases *Subset:* Two concepts can be semantically related, however, the extension of one concept may be only a subset of the extension of the other concept. This means, a condition is necessary in order to identify the subset which is actually semantically equivalent.

Overlapping: This case is even more complicated than the subset case, because both sides require conditions to identify which parts actually match.

Intension

The intension of a set is its description or defining properties. This means, when we are talking about the intension of a class, we talk about the properties of the class. In MOF attributes and references represent properties of classes. When two classes which are semantically related are compared, also the attributes and the references of the classes should match. However, due to different viewpoints and slightly different modeling domains, not all attributes and references matches.

Missing Properties It may occur that two semantically related classes have different properties. Sometimes the properties can be derived from other information, like the type of the object (cf. c2) or other properties (derived properties), but often they cannot be automatically computed, because the information is simply not present in the metamodel. The resulting problem of missing properties is 0:1 mappings. The problem is that the values of

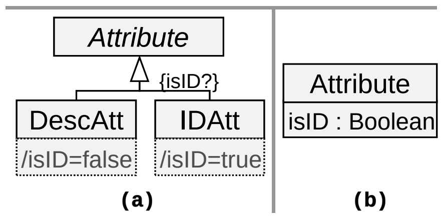


Figure 3.9: Implicitly missing property.

these 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. The first case is that the feature is optional leading to no problems because a null value can be assigned. The second case is the more problematic, namely if the UML attribute is mandatory. We can check if a default value is available for this attribute. If no default value is defined for the feature, the user must specify a value in the mapping model which is automatically assigned for this particular feature. In case, properties have standard values, these values can be implied. If this is not possible, then in the integration solution the user must give values for such integration scenarios.

Implicit Missing Properties When comparing two semantically equivalent classes, even if one class has one property missing compared to the other, often it is possible that this property can be inferred via the type and can be seen as implicitly available with a standard constant. This problem is related to Property Definition Conflicts of Section 3.2.1. Example: When we are comparing the metamodels in Figure 3.9(a) and (b), it seems that the class Class in Figure 3.9(b) has one additional property, namely isAbstract. However, this information can be expressed with the type information in Figure 3.9(a), thus it can be seen, that the two subclasses have an additional attribute with a constant value, e.g. DescAtt has a derived constant attribute isID with the value false.

3.3 Model Heterogeneities

In data engineering additionally to schema conflicts also conflicts on the data layer have been studied. Since the integration scenario was merging two or more schemas into one integrated schema as well as merging data from various sources into one representation, typing errors, outdated data, or different representations have been taken into consideration. Our primary tool integration scenario is the transformation of one model into another. More specifically, we have to create a new target model from an existing source model. Therefore, the reported data integration conflicts are not relevant. Nevertheless, some inte-

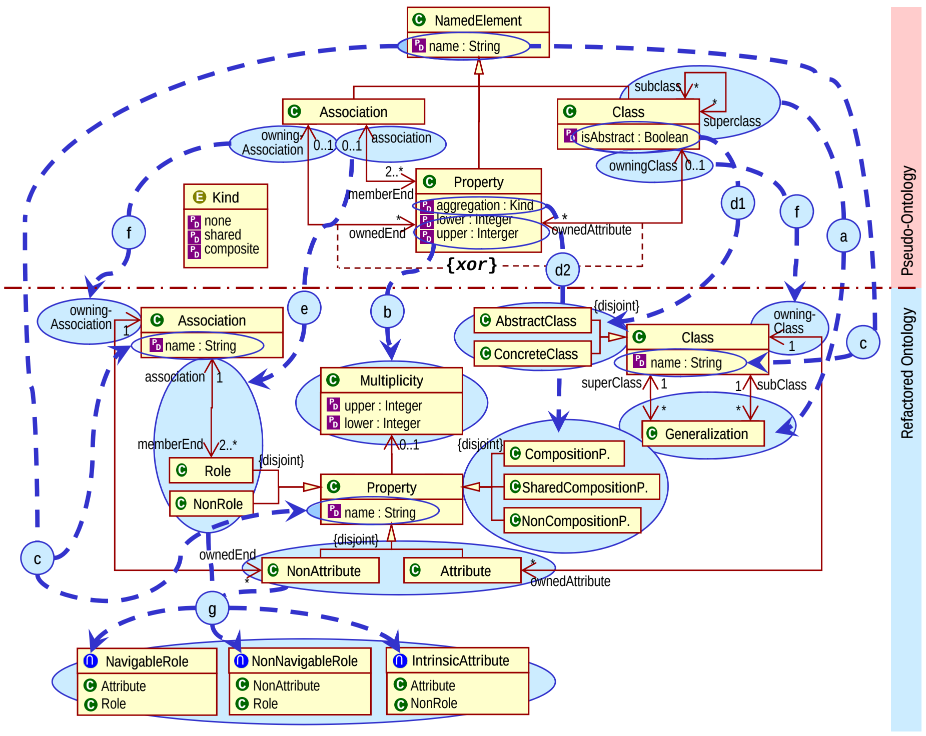


Figure 3.10: Part of the UML kernel as pseudo-ontology and as refactored ontology.

gration issues cannot be answered by looking at the metamodel layer only. Even if metamodels are identically defined, differences between languages are possible, which can only be determined with the help of instances of the metamodels, i.e., the models, and their use and interpretation. This is due to the fact that the semantics of the modeling languages are not defined within language definitions; instead they are only implemented in tool support such as code generators or simulation environments.

3.4 Refactoring Patterns

The aim of metamodeling lies primarily in defining modeling languages in an object oriented manner leading to efficient repository implementations. This means that in a metamodel not necessarily all modeling concepts are represented as first-class citi-zens. Instead, the concepts are frequently hidden in attributes or in association ends. We call this phenomenon concept hiding. In order to overcome this problem, we propose refactoring as a second step in the lifting process, which semi-automatically generates an additional and semantically enriched view of the conversion step's output, i.e., a what we call pseudo-ontology. For a detailed discussion on this conversion step and the term pseudo-ontology we refer the interested reader to [41]. However, the motivation for the shift from metamodels to ontologies is the available tool support for matching and the reasoning possibilities coming along with ontologies. Generally, we do not distinguish between metamodels and ontologies. We present the refactoring work we have done for ontologies in this section as the refactoring step can of course be directly applied to metamodels. Thus, refactoring also is of relevance to MTBE. First, with refactoring it is possible to solve heterogeneity problems. And second, one needs to be aware of possible refactorings when defining the CS, because many hidden concepts have a distinct representation in the CS, though.

Figure 3.10 gives an example of how concept hiding is achieved in metamodels. In the upper part it shows a simplified version of the UML metamodel kernel which is defined in the UML Infrastructure [19], represented as a pseudo-ontology. As we see in Figure 3.10 the pseudo-ontology covers twelve modeling concepts but uses only four classes. Hence, most of the modeling concepts are implicitly defined, only. To tackle the concept hiding problem, we propose certain refactoring patterns for identifying where possible hiding places for concepts in metamodels are and also how these structures can be rearranged to explicit knowledge representations. The refactoring patterns given in the following subsections are classified into four categories. The description of each pattern is based on [11] and consists of pattern name, problem description, solution mechanism, and finally, of an example based on the UML kernel. The kernel is shown in the upper part of Figure 3.10 as a pseudo-ontology (before applying the patterns) and in the lower part of Figure 3.10 as a refactored ontology (after applying the patterns). The numbers in the figure identify where a certain pattern can be applied and how that structure is refactored, respectively.

3.4.1 Patterns for Reification of Concepts

a) Association Class Introduction A modeling concept might not be directly represented by object properties but rather hidden within an association. In particular, it might be represented by the combination of both properties representing the context in which these object properties occur.

Refactoring: A new class is introduced in the ontology similar to an association class in UML to explicitly describe the hidden concept. Since there is no language construct for association classes in OWL, the association is split up into two parts which are linked by the introduced class. The cardinalities of the new association ends are fixed to one and the previously existing association ends remain unchanged.

Example 23. *The combination of the roles of the recursive relationship of Class, sub-class and superclass, occurs in the context generalization.*

b) Concept Elicitation from Properties In metamodels it is often sufficient to implement modeling concepts as attributes of primitive data types, because the primary aim is to be able to represent models as data in repositories. This approach is in contradiction with ontology engineering which focuses on knowledge representation and not on how concepts are representable as data.

Refactoring: Datatype properties which actually represent concepts are extracted into separate classes. These classes are connected by an object property to the source class and the cardinality of that object property is set to the cardinality of the original datatype property. The introduced classes are extended by a datatype property for covering the value of the original datatype property.

Example 24. *The properties Property.lower and Property.upper represent the concept Multiplicity which is used for defining cardinality constraints on a Property.*

3.4.2 Patterns for Elimination of Abstract Concepts

c) Abstract Class Elimination In metamodeling, generalization and abstract classes are used as a means to gain smart object oriented language definitions. However, this benefit is traded against additional indirection layers and it is well-known that the use of inheritance does not solely entail advantages. Furthermore, in metamodels, the use of abstract classes which do not represent modeling concepts is quite common. In such cases generalization is applied for implementation inheritance and not for specialization inheritance. However, one consequence of this procedure is a fragmentation of knowledge about the concrete modeling concepts.

Refactoring: In order to defragment the knowledge of modeling constructs, the datatype properties and object properties of abstract classes are moved downwards to their concrete subclasses. This refactoring pattern yields multiple definitions of properties and might be seen as an anti-pattern of object oriented modeling practice. However, the properties can be redefined with more expressive names (e.g. hyponyms) in their subclasses.

Example 25. *The property NamedElement.name is used for class name, attribute name, association name and role name.*

3.4.3 Patterns for Explicit Specialization of Concepts

d) Datatype Property Elimination In metamodeling it is convenient to represent similar modeling concepts with a single class and use attribute values to identify the particular concept represented by an instance of that class. This metamodeling practice keeps the number of classes in metamodels low by hiding multiple concepts in a single class. These concepts are equal in terms of owned attributes and associations but differ in their intended semantic meaning. For this purpose, attributes of arbitrary data types can be utilized but in particular two widespread refinement patterns are through booleans and enumerations.

d1) Refactoring for Boolean Elimination Concepts hidden in boolean attribute are unfolded by introducing two new subclasses of the class owning the boolean, and defining the subclasses as disjoint due to the duality of the boolean data type range. The subclasses might be named in an x and non-x manner but descriptive names should be introduced into the ontology by the user.

Example 26. *Class.isAbstract is either true or false, representing an abstract or a concrete class, respectively.*

d2) Refactoring for Enumeration Elimination Implicit concepts hidden in an enumeration of literals are unfolded by introducing a separate class for each literal. The introduced classes are subclasses of the class owning the attribute of type enumeration and are defined as disjoint, if the cardinality of the datatype property is one, or overlapping if the cardinality is not restricted.

Example 27. *Property.aggregation is either none, shared, or composite, representing a nonCompositionProperty, a sharedCompositionProperty or a CompositionProperty.*

e) Zero-or-one Object Property Differentiation In a metamodel the reification of a concept is often determined by the occurrence of a certain relationship on the in-stance layer. In such cases, the association end in the metamodel has a multiplicity of zero-or-one which implicitly contains a concept refinement.

Refactoring: Two subclasses of the class owning the object property with cardinality of zero-or-one are introduced. The subclass which represents the concept that realizes the relationship on the instance layer receives the object property from its superclass while the other subclass does not receive the object property under consideration. Furthermore, the object property of the original class is deleted and the cardinality of the shifted object property is restricted to exactly one.

Example 28. *Property.association has a multiplicity of zero-or-one, distinguishing between a role and a nonRole, respectively.*

f) Xor-Association Differentiation Xor-constraints between n associations (we call such associations xor-associations) with association ends of multiplicity zero-or-one restrict models such that only one of the n possible links is allowed to occur on the instance layer. This pattern can be used to refine concepts with n sub-concepts in a similar way like enumeration attributes are used to distinguish between n sub-concepts. Thus, xor-associations bind a lot of implicit semantics, namely n mutually excluding sub-concepts which should be explicitly expressed in ontologies.

Refactoring: This pattern is resolvable similar to the enumeration pattern by introducing n new subclasses, but in addition the subclasses are responsible for taking care of the xor-constraint. This means each class receives one out of the n object properties, thus each subclass represents exactly one sub-concept. Hence, the cardinality of each object property is fixed from zero-to-one to exactly one.

Example 29. *Property.owningAssociation and Property.owingClass are both object properties with cardinality zero-or-one. At the instance layer it is determined if an instance of the class Property is representing an attribute (contained by a class) or a nonAttribute (contained by an association).*

3.4.4 Patterns for Exploring Combinations of Refactored Concepts

Refactorings that introduce additional subclasses, i.e., patterns from category Specialization of Concepts, must always adopt a class from the original ontology as starting point since the basic assumption is that different concept specializations are independent of each other. Hence, in the case of multiple refactorings of one particular class, subclasses introduced by different refactorings are overlapping. In Figure 3.10 this is denoted using a separate generalization set for each refactoring. However, this approach requires an additional refactoring pattern for discovering possible relation-ships between combinations of sub-concepts.

g) Concept Recombination In order to identify concepts which are hidden in the ontology as mentioned above, the user has to extend the ontology by complex classes which describe the concepts resulting from possible sub-concept combinations.

Refactoring: User interactions are required for identifying the concepts behind the combination of concepts by evaluating the combinations in a matrix where the dimensions of the matrix are the overlapping generalization sets in consideration.

Example 30. *When studying the textual descriptions of the semantics of UML one finds out that some relationships between the different kinds of properties define additional concepts which are not explicitly represented in the ontology. In particular, the evaluation of role/nonRole and attribute/nonAttribute combinations leads to the additional intersection classes depicted in the lower part of Figure 3.10.*

Summarizing, the results of the refactoring step, either of ontologis or metamodels are characterized as follows:

- Only datatype properties which represent semantics of the real world domain (ontological properties) are contained, e.g. Class.className, Multiplicity.upper. This means no datatype properties for the reification of modeling constructs (linguistic properties) are part of the refactored ontology.
- Most object properties have cardinalities different from zero-or-one, such that no concepts are hidden in object properties.
- Excessive use of classes and is-a relations turns an ontology or a metamodel into a taxonomy.

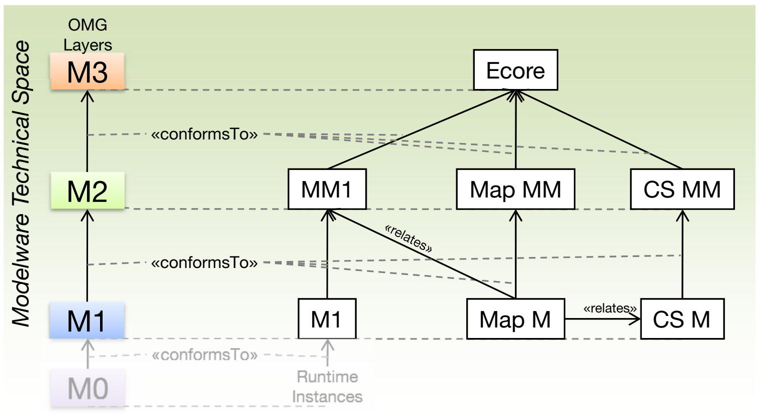


Figure 3.11: Modeling languages and their concrete syntax.

3.5 Concrete Syntax vs. Notation

So far we have been addressing the definition of the abstract syntax (AS) of a modeling language by means of metamodels, only. However, modeling languages often also comprise graphical concrete syntax. We then talk about visual modeling languages. In what follows we simply talk about concrete syntax (CS), but in general we mean graphical concrete syntax unless stated otherwise.

Definition *The graphical concrete syntax is part of a visual modeling language and defines in a formal way how abstract syntax (AS) elements are rendered to the screen. This formal definition for CS elements includes for example shape, appearance, layout and position.*

For an in depth discussion on the topic of CS see [31, 7, 30]. Also Fondement [30] discusses not only graphical concrete syntax but also textual concrete syntax, which we do not cope with in our MTBE approach. We want to embed the notion of CS within our modelware technical space, though. As already mentioned in the above definition we need a formal way to define our CS elements. The probably most intuitive way of defining these elements is by using again the well known meta modeling paradigm.

Figure 3.11 shows how the CS metamodel $CS\ MM$, which is again based on Ecore, is contained in our metamodeling stack. Because $CS\ MM$ acts as our modeling language for visually rendered elements we are able to define models that actually represent those visual objects. Model $CS\ M$ defines all CS elements for the modeling language $MM1$. But the definition of the CS is not enough for the implementation of a visual modeling language. What is left is the defintion of how metamodel elements of the modeling language and instances of the $CS\ MM$ relate to each other. The relationship of all those elements can be captured by a mapping model, which formally specifies model correspondences.

Definition *The mapping model relating AS and CS we call notation. The notation may be defined as follows:*

$$Triple := < as\_E, cs\_E, const(as\_E)? > \tag{3.1}$$

. In this basic definition we can relate one AS element as_E from MM1 with one CS element cs_E from the model CS. In addition one may give an optional constraint operating on as_E.

We have introduced a mapping mechanism for AS and CS and must finally provide the language definition for the mapping models, i.e., the notations, themselves. This is again done by a proper metamodel for mapping models that we call $Map\ MM$ in Figure 3.11. The references the notation must have to the AS and the CS are depicted as «$relates$» stereotyped references. So we have one meta layer crossing relationship between $MM1$ and $Map\ M$. Technically this can be achieved by importing Ecore in the $Map\ MM$ and directly use concepts of Ecore.

Notation in the Light of MTBE

In this subsection we show how we make use of the concept of notation in our MTBE approach. In Section 6.1 we show how the concept of notation has been implemented within state of the art modeling environments.

The primary idea of MTBE is to exploit the concrete notation of modeling languages, which is well known by the user, for defining mappings between semantically corresponding model elements on the M1 layer. In order to further discuss the by-example approach for model transformations, the interrelationships between the *abstract syntax*, *concrete syntax* and the *mapping* between them, which describes the notation of the modeling language, have to be clarified. In accordance with MMF [17] and GMF [23] Figure 3.12 depicts how these three parts of a formal language definition are interrelated in terms of an UML package diagram.

The package *abstract syntax* summarizes elements of the abstract syntax, i.e., the metamodel. For example for UML these would be concepts such as *property*, *class* and *association*. In contrast, the package *concrete syntax* covers graphical elements, e.g., ellipse, label, and rectangle, which can be further combined to more complex forms, e.g., *ClassRectangle*, *AttributeLabel*. Finally, how elements of the abstract syntax are mapped to elements of the concrete syntax is defined in the package *as_2_cs* which mainly consists of triples defined in Equation 3.1. The optional constraint part (*const(as_E)*) of the triple is the most relevant part for this work. In case no constraint is defined, there is a *one-to-one* mapping between an abstract syntax element and a concrete syntax element, i.e., the concept defined in the metamodel is directly represented by one concrete notation element. However, the other

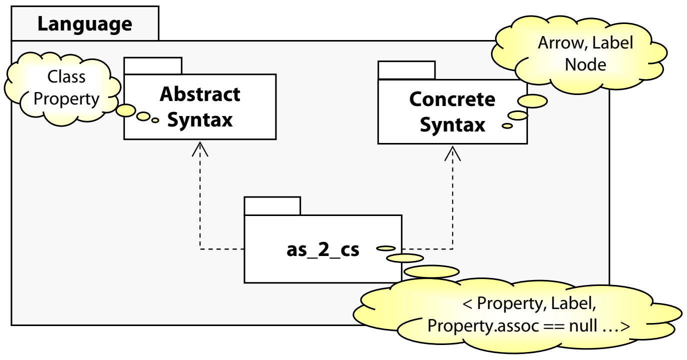


Figure 3.12: Relationship between abstract and concrete syntax.

case is the more interesting in context of solving the concept hiding problem. The presence of a constraint defines a new sub-concept for the notation layer, which is not explicitly represented by one of the metamodel classes. Consequently, when defining model transformations based on the abstract syntax, the constraints for these sub-concepts must be defined by the user in the query part or when going the other way round by setting the property values correctly in the generation part of the transformation rules. This is a tedious and error-prone task that requires excellent knowledge about the metamodel.

With MTBE this circumstance can be improved by incorporating the existing constraints defined in the triples (cf. Defintion 3.1) of the *as_2_cs* package (cf. Figure 3.12) into the model transformation generation process in order to minimize the effort for re-engineering and defining these constraints by hand.

3.6 Metamodel Metrics

Evaluation of models is a hard and ambiguous task. Whether a model can be asserted as semantically correct within a certain domain of discourse always depends on the viewpoint of the observer. In order to gather objective facts about models, the use of metrics is necessary. Metrics have a long history in software development as a quality measure. Measured value can be interpreted for itself, or it can be combined to create aggregated metrics for stating a more abstract conclusions. During the triumphal procession of Object Oriented Programming (OOP) in the last one and a half decades and the subsequent development of Object Oriented Modeling (OOM) techniques, a lot of effort was made to develop metrics for object-oriented models (OO-metrics). OO-metrics allow to make statements about the quality of software models [51][91][50].

With the advent of Model Driven Engineering (MDE) [75], the need of formal metamod-

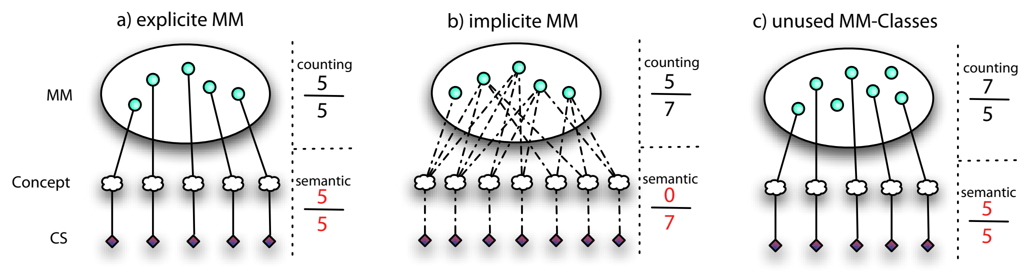


Figure 3.13: Three characteristics of the EM^2 metric.

els for modeling languages arises and consequently the need for metrics for metamodels. Basically a metamodel can be seen a s a model; it also consists of classes, relations, inheritance, etc. However, the intension of a metamodel (the AS) is different from that of most M1 models. A metamodel is an object oriented language definition. Therefore the instances of a metamodel are again models with a graphical notation, the CS. Nevertheless, metamodels can be treated as models for a specific domain, the domain of modeling language definition. Hence, most of the metrics for OOM can be applied to metamodels [49].

In this section we introduce a new metric, which discovers the *explicitness* of a metamodel. We define explicitness in the context of metamodels as the number of concepts in the modeling language that are first class concepts in the metamodel. This definition is based on the assertion that the number of first class concepts in the abstract syntax can differ from that in the concrete syntax. For example consider the modeling concept *Attribute* in the UML class diagram. In the UML 2.0 class diagram metamodel there exists no first class definition for *Attribute* . It is hidden in the class *Property* [41]. But in the concrete syntax of UML 2.0 class diagrams, there exists a notation for the concept *Attribute*.

3.6.1 Calculating the Explicitness of Metamodels

Modeling languages have specific modeling concepts, which can be expressed with its CS elements. The composition of modeling concepts is defined in the metamodel (the abstract syntax). These modeling concepts, which are used in the concrete syntax, do not necessarily have an exact counter part in the abstract syntax. Concepts in the metamodel are often reused with the help of attributes and associations to other concepts. We call this phenomenon concept hiding [41]. Our aim is to get a metric to estimate the explicitness of metamodels. The Explicitness of MetaModel (EM^2) metric can be calculated by counting all concrete classes of a metamodel and dividing it by the number of CS elements.

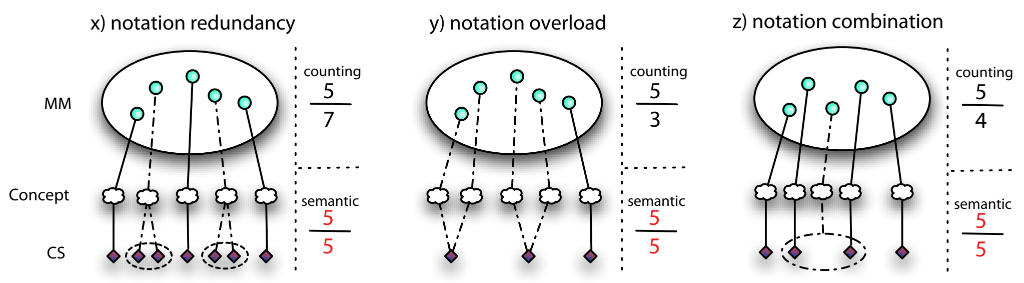


Figure 3.14: Side effects of the EM^2 metric.

$$EM^2(as, cs) = \frac{count(as.concreteClasses())}{count(cs.elements())}$$

A metamodel does not only consist of concrete classes but of abstract classes as well. Since abstract classes have no conceptual representation on the concrete syntax, they cannot be instantiated. For this reason abstract classes are not counted.

Interpretation of the Metric EM^2

In general, we can make three assertions about the ratio between the number of concrete classes in the metamodel and the number of CS elements.

EM^2 value	Interpretation
< 1	Concept deficit
$= 1$	Concept equilibrium
> 1	Concept redundancy | overload

A graphical representation is given in figure 3.13. The ellipses represent the metamodels with their first class concepts. The clouds represent the semantic concepts of the modeling language. The diamonds represent the CS elements. The links between these three elements depict the reference between the metamodel and its CS. The top right fraction (cf. *counting* in figure 3.13) represents the ratio by counting MM-concepts and CS elements without including further information. The bottom right fraction (cf. *semantic* in figure 3.13) represents the semantically correct ratio that describes the ratio of concepts and their representation on the concrete syntax. To determine the bottom right fraction, further information is necessary, that is included in the links between metamodel and CS. In the following we discuss the three aforementioned distinct cases of the EM^2 metric.

- *Concept deficit*: An EM^2 value smaller than one means that there exist more concepts in the concrete syntax than in the abstract one (see figure 3.13b). The reason for this is that the concepts are hidden in the metamodel. In the field of model transformations, which are defined on the metamodel, it is easier to map one metamodel to another if no hidden concepts exist in the metamodel. A balance smaller than one is an indicator for the need of refactoring [41].

- *Concept equilibrium*: An EM^2 value of one means that there are exactly as many concepts in the metamodel as in the concrete syntax (see figure 3.13a).
- *Concept overload*: An EM^2 value greater than one can have two reasons. The first one is a CS element with more than one concept in the metamodel. Two concepts in the metamodel with similar semantics is a non-realistic assumption and can be left out. The second reason are metamodel concepts, which do not represent a CS element (see figure 3.13c).

Side Effects of the Metric

Automatically applying the metric to a modeling language may entail some side effects. Only concrete classes of the metamodel and all CS elements that form the concrete syntax are counted. Depending on the metamodel, a concrete class must not necessarily have a CS element. This increases the denominator of the fraction and results in a higher EM^2 (the upper right fraction in figure 3.13c). This would be interpreted as a more explicit metamodel. The bottom right fraction in figure 3.13(c) depicts the semantic ratio between concepts, metamodel classes and its CS elements.

Counting metamodel classes with no representation in the concrete syntax can be prevented by following the mapping between a metamodel concept and its CS. If there is a link with no constraint, the metamodel class has a CS element. Otherwise, the class hides some information and is not a first class element for this concept. A constraint link is a mapping between a metamodel class and a CS element having further restrictions, e.g., that a association to another metamodel class ought to be set, or that an attribute of the metamodel class must have a specific value. With this further information, most of the side effects and the resulting misinterpretation of the balance can be avoided.

On the level of the concrete syntax, it is possible that a language concept has two or more CS elements, like the interface CS in UML. By counting all these CS elements, the numerator of the fraction increases and the balance becomes lower, see figure 3.14x. This would lead to interpret the metamodel as less explicit, with more hidden concepts. Evaluating the mappings between the CS and the metamodel, as aforementioned, can avoid this circumstance. If there exists a link without a constraint between a metamodel class and two CS elements this two CS elements are counted as one.

One and the same CS elements can be used for different concepts (3.14y). This result in a lower numerator of the fraction and leads to the interpretation of an overspecified metamodel. Again following the links between CS elements and metamodel elements is a solution to this problem.

CS elements represent a concept of the modeling language, but it is possible that the combination of two or more CS elements stands for a different concept which has no explicit CS. This changes the balance to a higher ratio if the concept is explicit in the metamodel but could not be counted on the concrete syntax because combined CS elements are not countable (3.14z). In this case following the links between metamodel concepts and CS elements provides no solution. An approach to cope with this problem is to analyze one or more concrete examples (M1 models) and analyze how CS elements can be combined.

3.6.2 Analysing UML 1.4 and UML 2.1

The UML metamodel 1.4.2 [66] and 2.1 [69] for class modeling were chosen as test cases for the EM^2 metric. Strictly counting concrete classes in the metamodel and predefined CS elements from the specifications, we found the following estimates for the EM^2 value.

Metamodel version	**EM^2 value**
UML 1.4.2	$\approx 0,77$
UML 2.1	$\approx 0,64$

These estimations have however to be taken with care because of possible inaccuracies in counting CS elements. There exist several CS elements where it is not clear whether to count them as one or each separately, as we did in our evaluation scenario. As an example for this kind of problem you might consider the *Dependency* relationship and its various stereotyped CS elements attached to it. Also we have not eliminated arising side effects, except for the *CS combination* side effect. Due to lack of space we only present a notation table for UML 2.1 in Figure 3.15, which helps reproduce the numbers of our metric.

Discussion of the Metric Results

Although the results are not totally unbiased they seduce that both metamodels rely on implicit concepts. Furthermore, we can say there is a greater implicitness incorporated into the UML 2.1 metamodel than in the UML 1.4.2 metamodel. The class *Attribute* represents just one concept in UML 1.4.2 that has been made implicit in UML 2.1. In the following we now go more into details and illustrate the characteristics and side effects of our metric.

As a potential source of implicitness of concepts we discovered the usage of enumerations, which are heavily applied in both metamodels. The definition of the EM^2 metric does not involve the count of enumerations and their literals. But these literals often represent a CS element, often in combination with some classes. For example, the enumeration *AggregationKind* is used in both metamodels to distinguish between three different CS elements, that is to say regular association, aggregation, and composition.

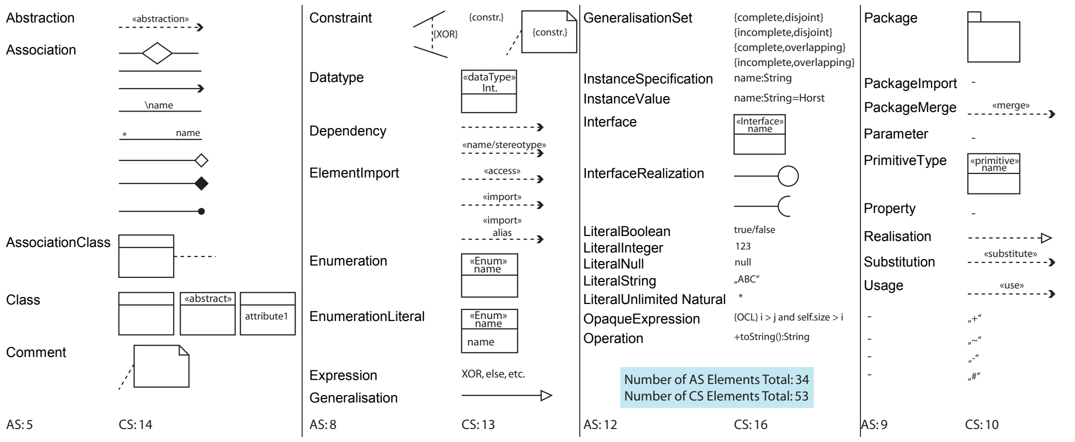


Figure 3.15: Results for UML 2.1.

The depicted side effect *notation overload* in figure 3.14, appeared in the UML 2.1 metamodel in the case of the meta classes *ElementImport* and *PackageImport*. Both classes make use of the same CS element. The meaning between the two can only be made unambiguous when considering the connected model elements, i.e., *Classes* or *Packages*.

Notation redundancy could also be recognized in both metamodels. Consider the interface class as an example, which can be graphically represented by two different means. Another example would be the various possible CS forms of an association (with a diamond in the middle or without, or with an arrow or without). The algorithm described above would filter such redundancies to eliminate the problem and concentrate on concepts instead of graphical representations.

We also encountered the problem of unused concrete meta classes that do not define a general notation. The responsibility is instead delegated to some other classes, that can be subclasses of the class under consideration. For example, the class *Parameter* in UML 2.1 has no direct link to any CS element. The class *Operation* therefore defines the CS for its parameters. Similar to the count of unused meta classes is the count of general CS elements that have no concrete class in the metamodel. The class *MultiplicityElement*, that is declared abstract, specifies a general CS for multiplicities, which can be further specialized in corresponding subclasses. The EM^2 metric takes the CS into account, but omits the abstract class, leading to a rare side effect, that we call *standalone notation*.

Combining CS elements is common practice in the UML metamodels. Take as an example the CS for a stereotype, that is composed of the CS of a simple class and the name of the stereotype within guillemets. When computing our metric for the two metamodels we counted each combined CS element as individual to avoid the side effect resulting from notation combination.

3.6.3 Metrics-Related Work

Best to our knowledge, there has been no work on metrics for explicitness of metamodels and our work is the first study on this topic. However, our work is mainly influenced by two orthogonal research directions, on the one hand by metrics for UML class diagrams and on the other hand by metrics for Ontologies.

Metrics for UML class diagrams are mostly based on metrics for OO programs. This is due to the close connection of UML class diagrams to OO programming languages like Java. In [91] six different metrics for UML class diagrams are analyzed and compared whereas the question arises which model elements, e.g., classes, attributes, and associations, have impact on the complexity of a class diagram. In [55] the metrics are more generically defined based on graph structures. Again, the metrics operate on quantitative characteristics, e.g., node count, edge count, and path length, and then these single metrics are combined to higher-order metrics.

Summarizing these proposed metrics for UML class diagrams mostly focus on the quantitative analysis of model elements, thus the metrics only measure the explicit definitions. Our work is different, because we look for implicit concepts which are hidden in combinations of model elements. In addition, we are analyzing language definitions and therefore we study the relation between modeling concepts, abstract syntax, and concrete syntax, which is certainly not applicable to UML class diagram models.

In [92] and [18] various metrics for ontologies are discussed which mainly measure the structural dimension in the same way as with OO models reflecting the fact that most ontologies are also represented in an object-oriented manner. Additionally to the structural measurement in Gangemi et al. [32] measurements for the functional dimension and usability, as well as a NLP-driven evaluation are introduced. Furthermore, the OntoClean approach [87] tries to detect both formal and semantic inconsistencies in an ontology. This perception goes along with our that counting the number of elements of certain types is not sufficient to specify the complexity of a model.

Our work is different to the proposed ontology metrics in that with our metric we are able to indicate how many concepts are implicitly represented which is due to the exploitation of the abstract to concrete syntax mapping which is metamodeling depending and not an ontology topic. Nevertheless, many ontology techniques are promising for the semantic evaluation of models and metamodels which is subject to future work.

The most related work is [38] in which OO metrics are applied to assess five versions of UML metamodels. The authors propose metrics for the stability of UML metamodels and for the design quality of UML metamodels such as reusability, flexibility, and understandability, which are computed from single measures.

Our work is different due to two facts. First, we also incorporate the CS of the modeling language, and second, we analyze which modeling concepts are missing in the abstract syntax as first class definitions. However, it is interesting that in [38] the computed value for understandability of UML 2 is much worse compared to its predecessor. Furthermore, it would be very interesting to compute the measurements for the design quality before and after applying our proposed refactoring patterns as introduced in [41].

3.7 Requirements for Example Models

So far we have not explained how our MTBE approach works in detail. However, we have to discuss the requirements we put on our example models in order to generate transformation code nearly without any user interaction, i.e., on a semi-automatic basis. Basically there is no limitation on the number of example models the user may want to specify for a specific transformation scenario between two different modeling languages. There can be either one big example or many small ones to foster clarity and traceability. Information from all examples is gathered and reasoned, though. A must in any case is the coverage of all relevant concepts available in the modeling languages. We also assume that tooling exists, which fully supports a modeling language. This means that a complete mapping between AS and CS is available by a proper notation as described earlier in Section 3.5. A concept on the modeling layer M1 must therefore be represented by a tool creating a CS element, which has a 1-to-1 or 1-to-many mapping to some AS elements. Special forms of implicit concepts on the modeling layer is discussed in Section 7.2. Besides this simple rule of using every concepts available in the example models we have experienced problems when using modeling structures made possible by the use of links between objects. Sometimes it is not sure how elements should be linked together or what linking structures should be supported. In the following example we deal with a nesting problem.

Example 31. *Figure 3.16 shows two examples of modeling languages simpleUML on the left and simpleER on the right side. Below these examples we show the corresponding metamodels that slightly differ in size. But excepts for this difference in meta elements used for modeling the two languages, we can in general model the same problem domains in either simpleUML or simpleER. As a consequence we do not loose any relevant information when transforming from simpleUML models to simpleER models and vice versa. By the blue dashed lines we depict the notation for the CS and AS elements of interest, only. We omitted to map the concepts of Attribute and also to define the Enum by a proper EEnum instance in the simpleUML metamodel. Notational mappings 3 and 4 can be seen as equivalent, as are the concepts of Class and Entity. So we can define or generate a rule, which transforms Classes into Entities with their corresponding name Attribute, see Listing 3.1, second ATL rule. In Ecore models we must have a model root element, which can be for example the name of a diagram. In our case we simply named it Root for both languages. There can be just one instance of Root containing all other modeling objcts. So the first challenge is to assign proper values to the containment references of these Root elements during transformation. We can use ATL's built-in resolve algorithms for this challenge. Line 7 demonstrates how reference sets can be correctly assigned and bypass instances of Package, which lie between a Root instance and some*

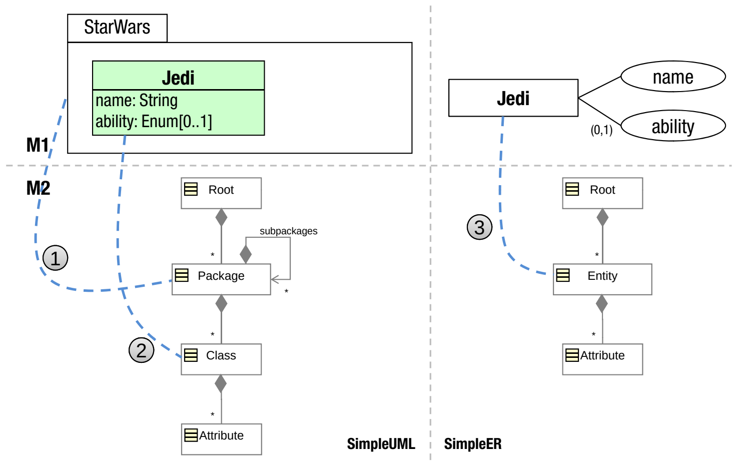


Figure 3.16: Demonstrating nesting problems: insufficient example.

Class instances in simpleUml. The output of this transformation is depicted in Listing 3.2, which shows the output simpleER model serialized as XMI. As can be seen we have transformed all classes from the source simpleUML model into Entities but the intended containment structure has been broken. This arises from the fact that we do not cope with subpackages and their child Classes.

Figure 3.17 shows an nearly identical example as in Figure 3.16. Though, they differ in the simpleUML model. In Figure 3.17 we provide an additional subpackage called Space within our basepackage StarWars on M1. The notation of this subpackage concept is given by mapping number 4. With this additional element in our example model we can derive proper ATL code, which includes the subpackage concept. We do not go into reasoning details in this section but in 5.2. Listing 3.3 shows the code to preserve the containment structure in Line 8. Note, that this now includes "second" level Packages not more. Recursive helper would be needed to include Classes of level n Packages, i.e. recursively nested Packages and Classes. In Listing 3.4 we finally show the correct output simpleER model of the transformation in Listing 3.3.

Listing 3.1: Incomplete ATL transformation for simpleUML to simpleER.

```
module uml2er; -- Module Template
create OUT : er from IN : uml;

rule root2root{
  from r : uml!Root
  to r2 : er!Root(
     entities <- r.packages -> collect(c | c.classes)
  )
}

rule class2entity{
  from c : uml!Class
  to e : er!Entity(
     name <- c.name
  )
```

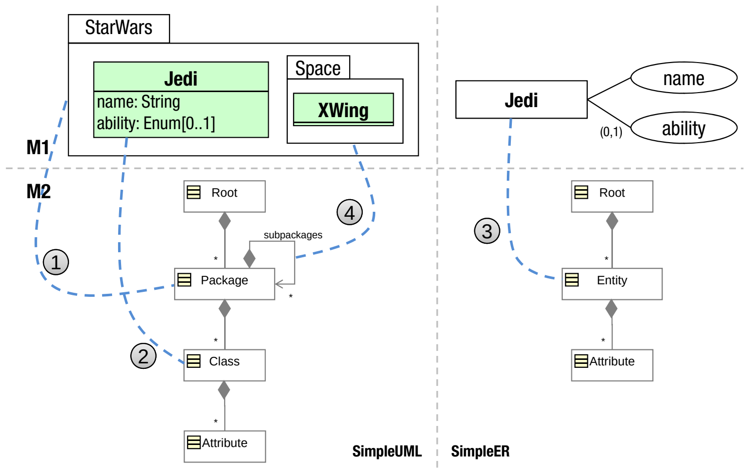


Figure 3.17: Demonstrating nesting problems: sufficient example.

```
}
```

Listing 3.2: XMI resulting from Listing 3.1.

```
<?xml version="1.0" encoding="ISO−8859−1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:er="http://er">
  <er:Root>
    <entities name="Jedi"/>
  </er:Root>
  <er:Entity name="XWing"/>
</xmi:XMI>
```

Listing 3.3: Correct ATL transformation for simpleUML to simpleER.

```
module uml2er; −− Module Template
create OUT : er from IN : uml;

rule root2root{
  from r : uml!Root
  to r2 : er!Root(
    entities <− r.packages −> collect(c | c.classes),
    entities <− r.packages −> collect(s | s.subpackages −> collect(c1 | c1.classes))
  )
}

rule class2entity{
  from c : uml!Class
  to e : er!Entity(
    name <− c.name
  )
}
```

Listing 3.4: XMI resulting from Listing 3.3.

```
<?xml version="1.0" encoding="ISO−8859−1"?>
```

```
<er:Root xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:er="http://er">
  <entities name="Jedi"/>
  <entities name="XWing"/>
</er:Root>
```

Chapter 4

Basic MTBE Concepts

Contents

The contribution of this chapter is to lay out basic concepts for MTBE for defining mappings on the M1 layer between concrete domain models. Based on these user mappings between concrete syntax elements and the notation included in the modeling languages we are able to derive model transformation code based on the M2 layer. In addition, challenges are discussed which are encountered, when generating model transformation code from the user defined inter-model mappings.

4.1 Shortcomings of Current Model Transformation Approaches

In the MDE research field various model transformation approaches have been proposed in the previous 5 years, mostly based on either a mixture of declarative and imperative rules such as ATL [40], or on graph transformations such as AGG [82], Fujaba [62], [61], and BOTL [11], or on relations such as MTF [37]. Moreover, the Object Management Group

(OMG) has published a first version of QVT [70] which should become the standard model transformation language. Summarizing all these approaches, it can be said that state of the art for defining model transformations is to describe model transformation rules between a source and a target metamodel (M2 layer), whereas the rules are executed on the model layer (M1 layer) for transforming a source model into a target model. Consequently, each of these approaches is based on the abstract syntax of modeling languages, i.e., on their metamodels, only, and the notation of the modeling language is totally ignored.

In collaboration with the Austrian Ministry of Defense and based on experiences gained in former integration scenarios [88], [74] we are currently realizing a system called ModelCVS [42] which aims at enabling tool integration through transparent transformation of models between metamodels representing different tools' modeling languages. Hence, we developed various model transformation examples for tool integration purposes using some of the aforementioned approaches, and in doing so, we discovered two main issues which prevent the user-friendly definition of model transformations. On the one hand there is a gap between how the modeler reasons about aligning two models and how the corresponding rules are defined in order to be executable by the computer, and on the other hand not all concepts of a modeling language supported by the concrete notation are explicitly represented in the metamodel. In the following we discuss these two issues in more detail.

Issue 1: There is a huge gap between the user's intention of aligning two languages and the way model transformation rules are defined for being automatically executable by the computer. Mostly, the user reasons on models representing real world examples shown by concrete notation elements and mappings between semantically corresponding model elements. However, this way of thinking is not appropriate for defining model transformations with currently available model transformation languages, because they support defining model transformation rules based on the abstract syntax, only.

Figure 4.1 illustrates this problem by an alignment scenario for UML and ER models. The upper half of Figure 4.1 depicts that for the user it is appropriate to reason on models representing real world examples expressed in concrete notation of the modeling language to find the semantic equivalent parts. In contrast, the lower half of Figure 4.1 shows the same domain model in abstract syntax visualized as an UML object model. As one can see, the abstract syntax is designed for the computer in order to process the models efficiently and not for the visualization of the domain knowledge in an easy understandable way. Hence, when trying to understand a domain model in abstract syntax one has to explore more model elements compared to the concrete notation representation, and furthermore, one has to know all relevant details of the metamodel, i.e., the language definition. Moreover, this problem is further aggravated by the following issue.

Issue 2: The aim of metamodeling lies primarily in defining modeling languages in an object-oriented manner leading to efficient repository implementations. This means that in a metamodel not necessarily all modeling concepts are represented as first-class citizens.

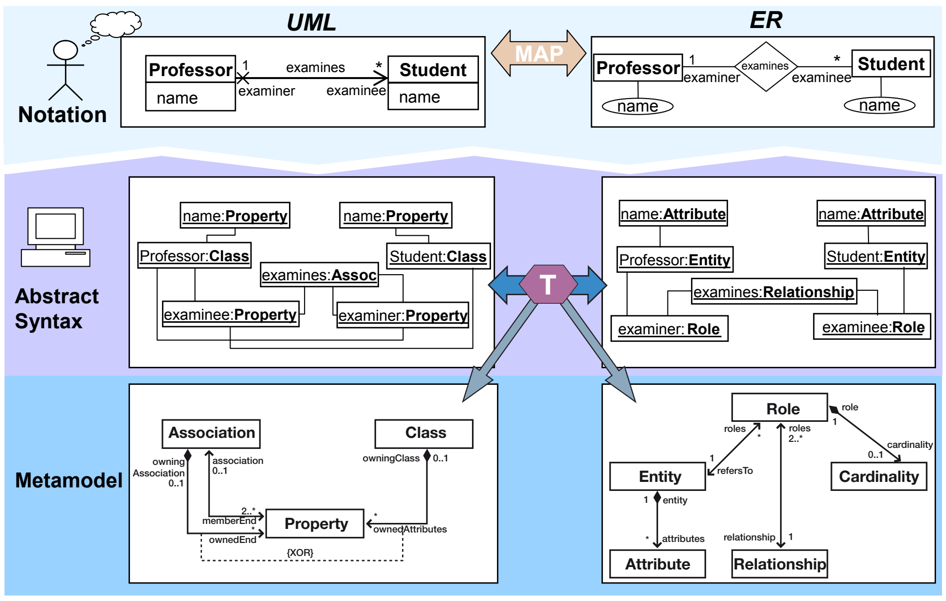


Figure 4.1: Gap between user intention and computer representation.

Instead, the concepts are frequently hidden in attributes or in association ends. We call this phenomenon *concept hiding*. For an in-depth discussion of concept hiding and how concepts can be hidden see [41].

As an example for concept hiding in metamodels consider Figure 4.2. In the upper part it shows a simplified version of the UML metamodel kernel which is defined in the UML Infrastructure [65]. In the lower part a domain model is shown in concrete UML syntax as defined by the notation tables in the UML Superstructure [67]. As one can see in Figure 4.2, the metamodel covers more than 10 modeling concepts but uses only four classes. Hence, most of the modeling concepts are implicitly defined, only. It is left as an exercise to the reader to find out *where* and *how* the concepts *attribute*, *navigable role*, *non-navigable role*, and *multiplicity* are defined in the metamodel.

These two issues mainly circumvent the user-friendly definition of model transformations. Therefore, we propose an orthogonal and extending approach to existing model transformation approaches for defining semantic correspondences in the concrete syntax of the models and the automatic generation of model transformations for the abstract syntax. This procedure allows a more user-friendly development of model transformations. Before going into details about the by-example approach we have to discuss which tasks are currently involved when model transformations are developed.

In general, before actually formalizing the model transformation rules in a model trans-

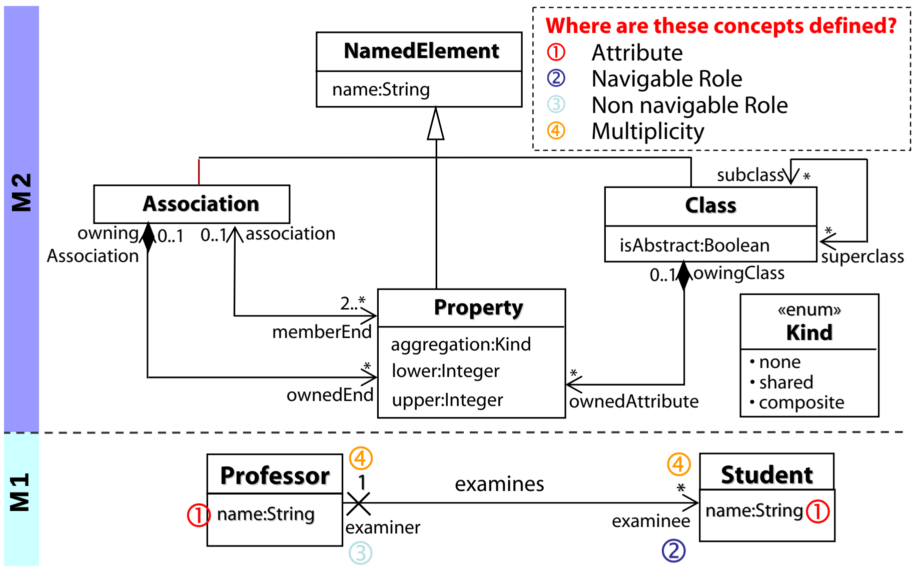


Figure 4.2: Concept hiding in metamodels.

formation language the user has to acquire knowledge about semantic correspondences between the concepts of the modeling languages as incorporated in their metamodels. One appropriate way to gain this knowledge is to start modeling the same problem domain with both modeling languages. By comparing the two resulting models the semantic correspondences between model elements can be easily found which again can be used to derive the correspondences between the metamodel elements. In addition, these models entail another benefit - they can be deployed for testing purposes as input for the expected model transformation and for comparing the output of the model transformation execution.

After clarifying all necessary semantic correspondences the user has to implement the gained mapping knowledge in the model transformation rules. For this task the user has to understand how the notation is represented in abstract syntax elements and how missing concepts in the abstract syntax can be reconstructed, e.g., by setting attribute values and links to other objects. Here comes MTBE into play. First, the mappings are explicitly definable between the domain models shown in concrete syntax which allows also the documentation of the semantic equivalences. Second, these mappings are a good starting point for automatically generating the required model transformation code which is more efficient in contrast to current approaches where the user has to implement all of them by hand.

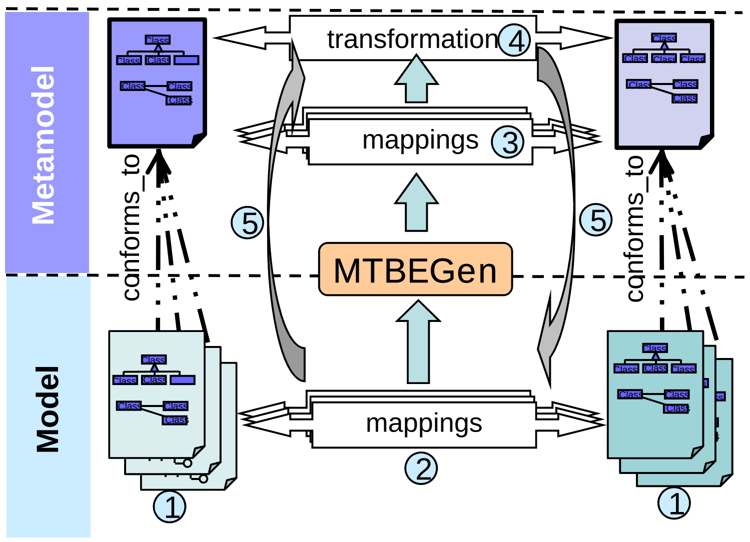


Figure 4.3: MTBE conceptual framework.

4.2 A Five Step Process for MTBE

This section discusses a conceptual five step process for MTBE at a glance. The key focus of this process is the automatic generation of transformation programs regarding semantic correspondences between two languages as can be seen in Figure 4.3. In this framework the model transformation generation process requires 5 steps, that are explained in the following. Each step is thereby marked properly in Figure 4.3.

- **Step 1.** The initial step is the definition of models of the same problem domain in both modeling languages (cf. left and right of the lower half of Figure 4.3 1). The user can decide if a single model, which covers all aspects of the languages, or several examples, each focusing on one particular aspect. Presumably the second approach is more preferable. The requirements on the models are twofold. First, certainly they must conform to their metamodels. Second, concerning completeness, all available modeling concepts of the modeling languages should be covered by the models. The second issue is closely related to the question of what appropriate test cases for model transformations are, which is e.g. discussed in [29]. However, the requirements for example models and what difficulties arise in conjunction with model and metamodel heterogeneities have been already discussed in Section 3.2.

- **Step 2.** The second step in the framework is that the user has to align the domain models (M1) by defining semantic correspondences (mappings) between model elements of the left and right side (cf. middle of the lower half of Figure 4.3). For simplicity, it is assumed that the models on the left and on the right side represent the same problem domain, as explained in step 1. In this Chapter of basic MTBE concepts we assume full equivalence mappings, only. However, in Chapter 5.1 we introduce other mapping language concepts, which allow for more expressiveness and dealing with more complex mapping scenarios.

Concerning the example models, general reference models for several modeling domains such as structural, interaction, and process modeling can ease the development of the required examples. However, this part is also subject to future work.

- **Step 3.** After finishing the mapping task, the third step is that the *MTBE Generator* (cf. MTBEGen in Figure 4.3) takes the user-defined mappings as input and produces equivalences between metamodel elements. The output of this particular task should be a complete mapping model between the two given metamodels.
- **Step 4.** Based on the generated metamodel mappings the MTBEGen component has to produce an executable transformation program, which is based on metamodel elements. The resulting model transformation programs can transform any source model, which conforms to the source metamodel, into a target model, which conforms to the target metamodel. However, as we explain later in this Chapter , the generation process may need some user interactions for resolving ambiguities in the mappings, arising from heterogeneities concerning the extend of the modeling languages. Another necessary condition for the transformation generation process is that the MTBEGen needs access to the package *as_2_cs* (cf. Figure 3.12 Section 3.5) in order to compute all necessary conditions for the query and property values for the generation parts of the transformation rules.
- **Step 5.** At last the generated transformation programs may need some user refinement for resolving ambiguities in the mappings, arising from heterogeneities concerning the extent of the modeling languages. Also it is possible to subsequently test the generated model transformation programs or transformation models on the the models, that were already used for defining the mappings between the two languages. This is another benefit of the by-example approach, as the input and output models for testing the model transformations are already available and no extra work for developing test cases is necessary. The target models generated by the transformation program can be compared to the already existing models. When some differences between the two models arise, the user can decide if the mappings on M1 should be revised and a newer version of the model transformation program should be generated or if the mappings on M1 are correct and the model transformation needs some fine-tuning directly in the transformation code. To act as a good test case is one of the

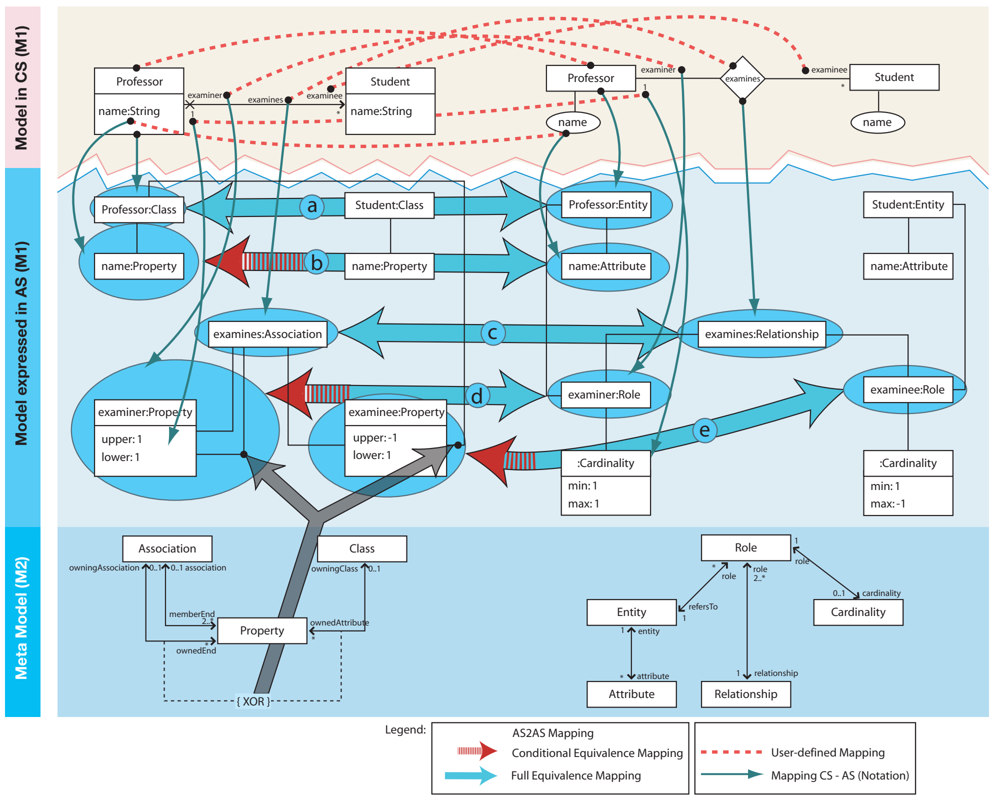


Figure 4.4: MTBE for UML2ER and vice versa.

requirements of good example models!

4.3 A Running Example

We exemplify the operating mode of MTBE by a concrete running example. In order to do so, consider the situation in which we have two UML classes *Professor* and *Student* as well as a one-to-many relationship between them. This simple UML class diagram is depicted in the upper left corner of Figure 4.4. In addition, the same problem domain is also modeled in terms of an ER diagram that can be found in the upper right of Figure 4.4. In the upper half both models are represented in concrete syntax, whereas the lower part of Figure 4.4 represent the same models in abstract syntax, which is the type of representation the computer uses for model transformation execution. For simplicity and higher readability the models in abstract syntax are represented as UML object diagrams. The example models shown in

Figure 4.4 are quite simple, however, they are sufficient to show the most important aspects of our proposed MTBE approach.

In the following subsections the steps 2 and 3 of the MTBE framework (cf. Figure 6.3) are discussed in more detail. Step 2 has to be carried out by users themselves and concerns the alignment of two domain models shown in concrete syntax (cf. subsection 4.4.2). Step 3 is split into 4 sub-steps, to give an in-depth discussion of the work the MTGen has to do. In particular, we explain how the abstract syntax is analyzed to collect all necessary data for the model transformations. Therefore, we interpret the models shown in abstract syntax as object models consisting of objects, attribute values and links, because these models can be seen as instances of the metamodel, which again can be seen as a simple class diagram. Consequently, we first explain the creation of objects (cf. subsection 4.5.1), then the placement of attribute values (cf. subsection 4.5.1), and finally the linking of objects (cf. subsection 4.5.1). By collecting the data of these three sub-steps, it is possible to derive all necessary information in order to define the query parts (e.g., the *from* part of ATL rules) and also the generation parts (e.g., the *to* part of ATL rules) of the model transformation rules (cf. subsection 4.5.3).

4.4 MTBE Frontend - The Black-Box View

By MTBE frontend we mean the graphical user interface, i.e., the workbench the user is confronted, when she decides to apply MTBE on some transformation scenario. Figure 4.5 shows the basic workbench vision that acts as our design prototype for an Eclipse-based implementation. We have identified four major views that have to be presented to the user in some perspective or view. The top level view and most important to the transformation designer is the model mapping view, which she can draw the mappings between graphical CS elements in an appropriate editor. After a reasoning process the produced metamodel mappings can be viewed in a separate editor. This not only allow the view of metamodel mappings but also their modification. The third part of the MTBE perspective shows the generated transformation code, i.e., the ATL code automatically produced from the metamodel mappings. And our last view component is associated with the models that have been the outcome of the generated model transformation program. This allows the testing of the ATL code and eases the correction of errors by manual refinements in one of the upper three view components depicted in Figure 4.5.

4.4.1 Model Mapping

The main purpose of our frontend is to allow the definition of model mappings, i.e., to map elements of different modeling languages.

But what is a model mapping? Naumann and Leser [48] define a schema mapping to

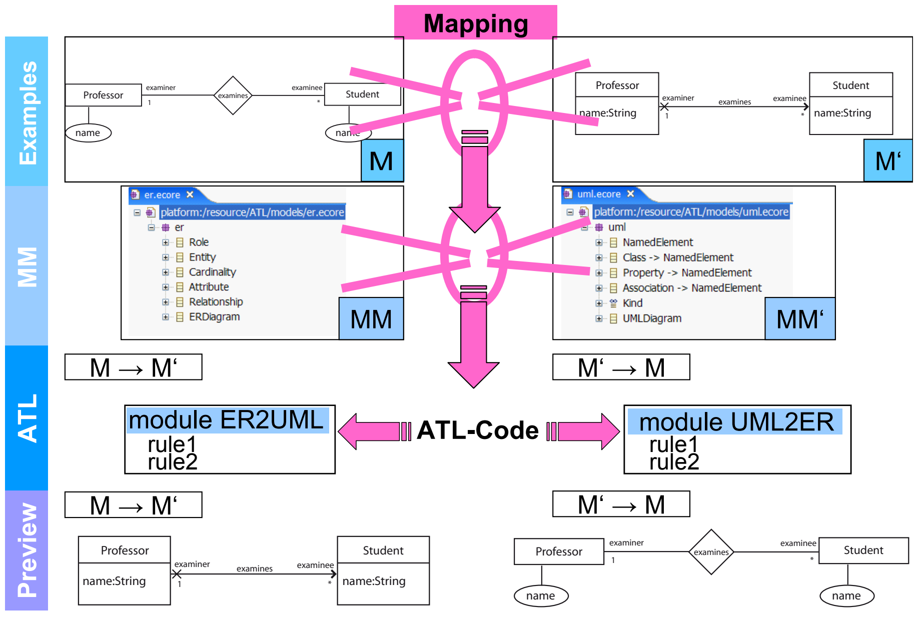


Figure 4.5: MTBE workbench vision.

be a set of correspondences between an arbitrary number of attributes of different schemas. Following this definition we can use the term correspondence and mapping equally. Furthermore, such mappings define a semantic correlation between mapped elements. There exist basically two ways to obtain mappings between models and schemas. One of these techniques is matching, which is an automatic task done by a tool following predefined matching strategies. See for example [6]. The second technique is defining mappings manually. This is the procedure we have chosen in our MTBE frontend. There are four different value correspondences available, i.e., $1:1$, $1:n$ (split), $n:1$ (merge) and $m:n$. Value correspondences can further be annotated with transformation functions, which can perform some calculations or string manipulations.

Naumann defines two kinds of usage for mappings in general that can be directly adopted for model mappings. First, we can use mappings for the representation of knowledge and thus for some model integration task. And second, we can use value correspondences as the basis for a more complex task, i.e., the transformation of models. Often a fundamental problem of correspondences stems from the lack of a semantic foundation. Without having the semantics of mappings defined tool support is often not reliable in terms of transformation correctness criteria.

In MTBE we basically face the same problem of lacking semantics for model mappings. But we can argue first that a model mapping has not always one specific semantic meaning

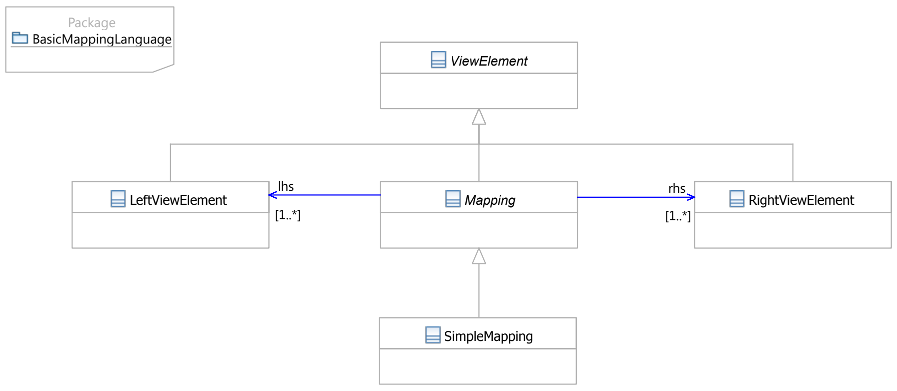


Figure 4.6: Basic model mapping language.

Meta Element	Concrete Syntax Element	Example
SimpleMapping	- - - - - - - - - - - - - -	See running example CS mapping professor:Class 2 professor:Entity

Figure 4.7: Notation table of basic model mapping language.

thus allowing for semantic variation points as in UML. And second the meaning of a mapping gets determined only when we perform some reasoning on these model mappings in order to deduce a proper mapping model on the M2 layer.
Abstract Syntax. Figure 4.6 defines a very basic mapping language for MTBE, which solely consists of one mapping operator, i.e., our *SimpleMapping*.

```
context SimpleMapping inv:
self.lhs->size()=1 and
self.rhs->size()=1
```

Concrete Syntax. Figure 4.7 shows the CS for our basic mapping language according to the AS defined above.

4.4.2 Mapping Definitions By-Example

The user has to define mappings between model elements of the two concrete domain models as shown in Figure 4.4 page 61. These mappings are illustrated by thin dotted lines between elements of the two models in Figure 4.4 page 61. For the sake of clarity, we omitted some of the mappings as this helps to focus on those mappings that are of special interest for our algorithms explained in the next subsections. As mentioned before mappings specified by users are solely full equivalence mappings, i.e. one-to-one mappings. Furthermore, these mappings can be regarded as bidirectional in contrast to other by-example transformation approaches, e.g., [47]. Hence, model transformation code can be generated for both directions, namely from UML to ER, and vice versa.

4.4.3 Validation of Transformation Code and Produced Output Models

The task of validation is very closely related to testing. First we have to ask: what can be validated? We can validate transformation code, i.e., the transformation model produced by MTBE, and the (meta)models involved in the transformaton process. However we can only check for syntactical correctness with models. Semantic validation is not possible as their exists in most cases no formal specification of metamodels yet. Most prominent example for a missing formal semantic specification is the UML. Hence, we can only check for syntactic correctness.

Validation can either be done manually by the user or automatically with tool support. We briefly discuss both ways, but won't go into detail as the validation and testing field for transformations and their are research areas of its own. However these topics are vital for the success of model transformation approaches and can be seen as major requirements.

Manual Validation. Generated transfromationcode can be reviewed by the programmers themselves and validated by executing the code with some input models and afterwards comparing with the output models. By again executing the transformation the other direction having now the output models as inputs one can aim at a full round trip. If information gets lost this is directly visible and the user can start with a refinement process, which may start at the model mapping layer and can finally reach the code layer, i.e., the transformation model derived by MTBE. A great benefit stems from the fact that test models are already available as they have been specified during step 1 of the MTBE process. Therefore the cost of time creating proper test cases for transformation code testing is reduced. However, additional critical test cases must be provided if they exist.

Automatic Validation. Automated testing is one of the key features in model transformations. Especially the graph grammar based approaches suffer from termination and uniqueness issues arising during rule execution. For an in depth discussion on syntactical correctness criteria, termination and confluence see for example [45]. In this work Küster presents a graph transformation approach and how validation on the transformation rules can be achieved. Sadilek et al. [73] are especially concerned with the testing of metamodels, but similar approaches acting upon models are possible. In Fleurey et al. [29], the authors discuss the usefulness and appropriateness of input models of some model transformation program. They define rules and a framework that help the user to design sound input models. These rules could be integrated into MTBE as supporting technology.

But there are several tasks the user can perform independently from these rather sophisticated approaches in order to test the transformation code. Input and output models must conform to their respective metamodels. If any model is produced that does not conform to the well-formedness rules of the metamodel we have indication for an error in the transformation code. Similarly the transformation model can be checked against the transformation metamodel or the textual concrete syntax can be parsed for syntax errors.

4.5 MTBE Backend - The White-Box View

In contrary to the frontend presented in the previous Section, we now take a closer look at what happens behind the curtain. It is about the MTBE "engine", that tries to generate executable transformation code. The aim of this Section is to layout how this automatic process works and what the basic ideas have been at the beginning of the development of our MTBE approach. The term white-box view refers intuitively to everything that is part of the program, the internal system.

The MTBE process outline in Figure 6.3 shows the backend related tasks or process steps 3 and 4. Step 3 can now be split up into 4 sub-steps, to give an in-depth discussion of the work the MTBEGen has to do. In particular, we explain how the abstract syntax is analyzed to collect all necessary data for the model transformations. Therefore, we interpret the models shown in abstract syntax as object models consisting of objects, attribute values and links, because these models can be seen as instances of the metamodel, which again can be seen as a simple class diagram. Consequently, we first explain the creation of objects (cf. subsection 4.5.1), then the placement of attribute values (cf. subsection 4.5.1), and finally the linking of objects (cf. subsection 4.5.1). By collecting the data of these three sub-steps, it is possible to derive all necessary information in order to define the query parts (e.g., the *from* part of ATL rules) and also the generation parts (e.g., the *to* part of ATL rules) of the model transformation rules (cf. subsection 4.5.3).

4.5.1 Basic Reasoning on User Mappings By-Example

Second, we need to know how the model elements shown by the concrete syntax correspond to the model elements shown by the abstract syntax. These definitions are provided by the package as_2_cs as described in section 3.5. The links between concrete syntax and abstract syntax are illustrated in Figure 4.4 page 61 as thin solid arrows for the right and for the left side, respectively. Again we left out some of the links to focus on the mapping definitions which are relevant for the following discussions.

Object Creation

As we defined semantic correspondences between model elements of the two domain models in the previous step, we can now move on to the object creation process. Assume first that we want to transform the UML class diagram into an ER diagram. Therefore, the algorithm has to analyze the abstract syntax of both models and additionally the user-defined mappings. In particular, the algorithm has to check if a certain type of object in the UML model is mapped to a certain type of object in the ER model. If this is the case, there is also a *full equivalence mapping* on the abstract syntax layer and a simple transformation rule without a condition can be generated for this object type. For example, objects of type *class* are mapped to objects of type *entity* (cf., mapping *a* in Figure 4.4 page 61), only. However, some objects of the same type are mapped to different object types depending on their attribute values and links. In this case, an additional mapping operator is available for the abstract syntax layer, namely *conditional equivalence mapping*. The conditions for the conditional equivalence links are derived from the *as_2_cs* package, i.e., the concept hiding is resolved, and finally these conditions manifests in the query part of the model transformation rules. For example, *property* objects of the UML class diagram are mapped to both *attribute* objects (cf., mapping *b* in Figure 4.4 page 61) and *role* objects (cf., mapping *d* in Figure 4.4 page 61) of the ER model. Taking the constraints *property.owningClass != null* and *property.association == null* of the *as_2_cs* package into account, we can assure that only an ER *attribute* is generated when the *property* actually represents an attribute in the UML class diagram. The same procedure can be applied for properties representing roles.

After completion of this step we have created all necessary objects for an ER diagram from a UML class diagram, which are the basis for our next steps to be performed. The same procedure can also be applied for a ER diagram to an UML class diagram transformation as our transformations can be generated in either direction.

Placement of Attribute Values

This step constitutes the placement of attributes values for the created objects. In contrast to the object creation step where primary the query parts of the transformation rules were relevant, this subsection focuses on the generation parts, i.e., how to set the attribute values. First of all, we have to differentiate between two different kinds of attributes which occur in metamodels, namely *ontological attributes* and *linguistic attributes*.

Ontological attributes represent semantics of the real world domain which can be incorporated by the user by setting the values explicitly in the concrete syntax. Examples for ontological attributes are *Class.name* and *Attribute.name*. In order to set the ontological attributes in the generation part of the transformation rules we use heuristics, e.g., string matching. In our example, we can conclude that the *name* of a *class* should be the *name* of an *entity* when considering the class *professor* and the entity *professor* (cf., mapping *b* in Figure 4.4 page 61), because these two attributes have the same value.

Linguistic attributes are used for the reification of modeling constructs which cannot be set explicitly by the user in the concrete syntax, e.g., *Class.isAbstract* or *Property.aggregation*.

Hence, these attributes have predefined ranges of values as they are fixed elements of the language definition. When dealing with linguistic attributes in context of MTBE we need to exploit the information stored in the as_2_cs mappings, because in these mappings the concepts become explicit by defining the required condition, i.e., how the values have to be set to fulfill the requirements for the sub-concept. For example, when transforming an ER attribute to an UML property, we also have to set the linguistic attributes of the property class (e.g. *Property.aggregation*) which can be done by incorporating the information stored in the as_2_cs mapping.

Linking Objects

Finally the links between the created objects have to be deduced from the metamodel, from our triples of the *as_2_cs* mappings containing OCL constraints, from the user-defined mappings, and user interaction as the last choice when the last three mentioned options are not sufficient. This part of the transformation step is obviously the most interesting one, as most difficulties arise at this stage. In particular, the user-defined mappings on the concrete syntax can result in ambiguous mappings, i.e., mappings that are controversial and it is not automatically decidable which case should be chosen for the general model transformation. Especially *0..1* associations in combination with *xor*-constraints in the metamodel are relevant in this context, as they might entail some hidden concepts. Another reason of unambiguous mappings is the heterogeneity of the expressiveness of the modeling languages.

In the following the creation of object links is described, whereby we classify some interesting cases regarding to multiplicity of the association ends of the metamodel, namely *1..1*, *0..1* and *0..1* in combination with *xor*-constraints.

Unambiguous Mappings Concerning unambiguous mappings, we discovered two interesting cases, namely association ends with multiplicity *1..1* and *0..1*.

- *1..1 association ends*: We encounter such association ends in our ER metamodel between *Entity* and *Attribute* as can be seen in the bottom of Figure 4.4 page 61. In addition, when looking at the middle of Figure 4.4 page 61, one can see, that each ER attribute is linked to an ER entity as this is the mentioned constraint of the ER metamodel. Furthermore, one can see that each UML attribute is linked to an ER attribute and that the containing UML class is linked to the containing ER entity, respectively. Consequently, if we transform an UML *property*, that is actually an attribute, into an ER *attribute*, we can automatically create the link between entity and attribute.
- *0..1 association ends*: This kind of association ends in the metamodel allows concept hiding, as is done in the UML metamodel for the class *Property*. A property can either be a special kind of role or an attribute belonging to a certain class. As we will

see, association ends of this kinds are not as easy decidable as 1..1 association ends are, because the links are not required on the abstract syntax layer and can vary, also within the same example. However, in case of unambiguous mappings, i.e., the link is always or never present on the abstract syntax layer, a general model transformation rule can be derived. For example, ER relationship has two links to its roles and UML association has two links to its properties. Furthermore, ER relationship is mapped to UML association (cf., mapping *c* in Figure 4.4 page 61) and the ER roles are mapped to the UML properties of the association (cf., mappings *d* and *e* in Figure 4.4 page 61). When going from ER to UML, we can deduce that each corresponding association should have links to the properties which correspond from the roles of the ER relationship. However, the second possible kind of link between association and role (cf. concerning association end *owningAssociation* in Figure 4.4 page 61) is not automatically decidable as we see in the next subsection.

Ambiguous Mappings In this part we describe an example that shows that especially for object linking some ambiguities can occur which have to be resolved by user interactions.

In Figure 4.4 page 61 two user-defined mappings are shown, which are the source for ambiguity mappings on the abstract syntax layer. In this example the role *examinee* in the ER model is mapped to the navigable role *examinee* in the UML class diagram, but the role *examiner* is mapped to the non-navigable role *examiner* in the UML class diagram. Now we want to discuss the impacts on the abstract syntax layer mappings. The problem arises that it is not decidable which general transformation rule should be derived, because one role of the ER model is mapped to an UML property, which has a link to an class object (cf., mapping *e* in Figure 4.4 page 61), and another role of the ER model is mapped to an UML property which has instead an link to an association object (cf., mapping *d* in Figure 4.4 page 61). This unambiguity results from the metamodel of UML where an *xor*-constraint exists between *owningAssocation* and *owningClass*, as can be seen in Figure 4.4 page 61, and from the fact that UML differentiates between navigable role and non-navigable role without supporting the general role concept. As our definition of the ER metamodel does not allow for two different kinds of roles as the UML metamodel does, we cannot derive an general transformation rule. Instead the user must decide on how to deal with roles from the ER model in the UML model. This example shows that in general it is not possible to automatically derive all model transformation rules, not even between modeling languages, which share the same modeling domain. Instead, for some rules the user has to interact and decide, which alternative is appropriate, such as in our example to generate navigable roles in the UML model for roles of the ER model.

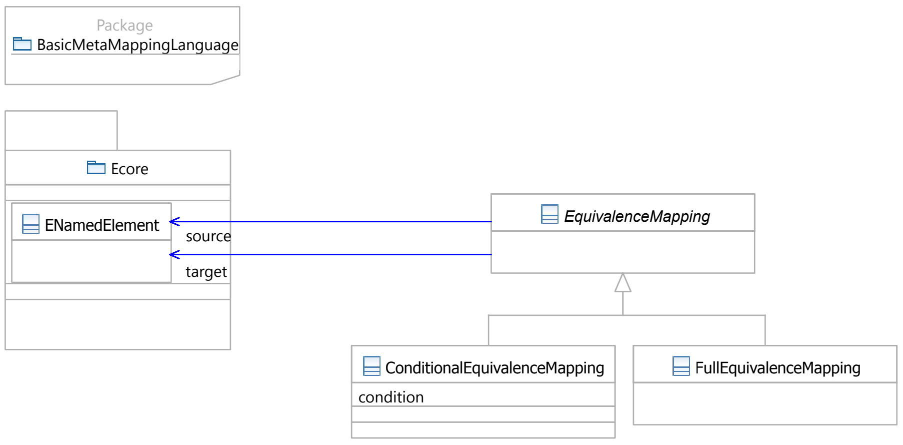


Figure 4.8: Basic metamodel mapping language.

4.5.2 Metamodel Mapping

Abstract Syntax. In the above we have already discussed how we can derive metamodel mappings based on the information given by the user and how these mappings look like. For the sake of completeness and we present the AS for our metamodel mapping language in Figure 4.8. Also we explain in the next chapter how this language can be extended to ease the generation of model transformation code. Figure 4.8 shows that the central mapping element is the abstract class *EquivalenceMapping*, which concrete mapping classes inherit from, i.e., *ConditionalEquivalenceMapping* and *FullEquivalenceMapping*. So this abstract root class acts as extension point for more specialized metamodel mapping operators. We also include the Ecore package here to be able to reference any AS element used in our metamodels.

4.5.3 Transformation Model Generation By-Example

At last all gathered information can be aggregated to generate proper ATL transformations. In the following, two examples are shown just to give an idea how the query parts and generation parts of the ATL transformations are generated.

The first example as presented in Listing 4.1 is a transformation from *ER attributes* to *UML properties*, which actually represent UML attributes in the concrete syntax. Note that the generation part of this rule is the most interesting part, because the attribute value assignments for ontological and linguistic attributes have been automatically generated.

Listing 4.1: ATL rule for Attribute2Property.

```
module ER2UML;
create OUT : UML from IN : ER;

rule A2P {
   from a : ER!Attribute
   to p : UML!Property (
       name <- a.name,
       aggregation <- 'none',
       ...
       owningClass <- a.entity
   )
}
```

Listing 4.2: ATL rule for Property2Attribute.

```
module UML2ER;
create OUT : ER from IN : UML;

rule P2A {
   from p : UML!Property (
        p.owningClass.oclIsUndefined()
        = false and
        p.association.oclIsUndefined()
   )
   to a : ER!Attribute (
        name <- p.name,
        entity <- p.owningClass
   )
}
```

The second example shown in Listing 4.2 is an ATL rule that incorporates the condition of the abstract to concrete syntax mapping for UML in its query part in order to produce *ER attributes* for *UML properties* which are actually representing *UML attributes* on the concrete syntax layer, only.

4.6 MTBE-Related Work

With respect to our approach of defining inter-model mappings between domain models (M1) and the derivation of model transformation code from these mappings we distinguish between three kinds of related work: first, related work concerning on linking model elements between models within a separate model (model weaving), second, declarative and example-based transformation rules mainly supported by graph transformations and third, related by-example approaches starting from their origin approach, namely query-by-example.

In general, our approach of defining similarities between modeling languages and models is related to model transformation. Model transformation in the context of MDE is a rapid emerging topic as can be seen in the model transformation workshop at the MoDELS/UML 2005 conference. One of the first and nowadays one of the most matured approaches is the ATLAS Model Weaver (AMW) [26] and the ATLAS Transformation Language ATL [40].

The idea behind model weaving is to define a relationship between a left model (or metamodel) and a right model (or metamodel) with certain kind of mapping operators which can also be user-defined. This approach is related to the mapping between two concrete domain models of two different modeling languages, however, the difference lies in the representation of the models and in the level of the mappings. AMW works with the abstract syntax representation of a model, while our approach works with mappings between models represented with the concrete syntax of the modeling languages. The benefit of mapping examples shown in concrete syntax is the absence of hidden concepts which occur quite often in metamodels. Our work is also different to the AMW in that the model transformation generation process of the AMW currently focuses on using mappings between metamodels (M2 mappings) and therefore based on the abstract syntax as input to derive ATL code [39], while our approach aims at generating model transformation code from M1 mappings. Hence, we have shifted the definition of the mappings from the abstract syntax to the concrete syntax and from the metamodel layer to the model layer.

Our proposed MTBE approach follows the main principles of the query by example (QBE) approach introduced in [93]. The aim of QBE is to have a language for querying and manipulating relational data. This is achieved by skeleton tables, which consists of example rows filled out with constants, constraints, and variables, combined with commands. Commands describe what to do with the selected tuples that match the defined queries, such as deletion or selection of the tuples. In order to operate on relational data stored in DBMS, real queries (e.g., SQL scripts) are derived from the skeleton tables and can be executed on relational models. Lechner et al. [47] follow this original approach of QBE, but with extensions for defining scheme transformers, which is demonstrated in the area of web application modeling with WebML [15]. Therefore, the original QBE approach is extended by introducing in addition to the query part (WebML model before transformation) also a generation part (WebML model after transformation) in the template definitions. Finally, XSLT code is generated to transform the WebML models which are represented within the accompanying tool *WebRatio* as XML files.

Our work reuses the main idea of the aforementioned by-example approach [47], but our work is different to this work in that first, we propose the use of real world examples instead of using abstract examples, second, we introduce bi-directional mappings in contrast to uni-directional template based examples, third, our domain for applying a by-example approach is the modeling technical space [46], while the others are based on relational data, and fourth, we also consider the abstract syntax to concrete syntax mappings to tackle the problem of implicitly defined modeling concepts and are therefore able to make them explicit.

Other by-example based approaches related to our proposed MTBE approach are programming by-example [72], [3], and [24] as well as XSLT style sheet generation by-example [71]. The objective of these approaches is to facilitate the end user to be able to perform tasks which normally need more knowledge, e.g., knowledge about programming languages like Visual Basic, Java or even XSLT. The way PBE tries to to achieve this objective is to record the users actions (e.g., by a trace model) maybe in more than one iteration, and generate a program from the trace models to automatically perform the afore manually performed task by the computer.

The difference to the programming by-example approaches is that we statically define the mappings between two models instead of the iterative adaptation of the examples to get the resulting code, in our case the ATL code.

Parallel to our MTBE approach Dániel Varró proposed in [84] a similar approach. The overall aim of Varró's approach is comparable to ours, but the concrete realizations differ from each other. With our basic MTBE approach we describe the definition of semantic correspondences on the concrete syntax, which are propagated to the abstract syntax. From these mappings ATL rules can then be derived. Varró's approach uses the abstract syntax to define the mappings between source and target models, only. The definition of the mapping is done with reference nodes leading to a mapping graph. To transform one model into the other, graph transformation formalisms [25] are used. However, both approaches generate model transformation rules semi-automatically leading to an interactive and iterative process.

4.7 Summary

In this chapter we have introduced a basic by-example approach for defining semantic correspondences between domain models shown in their concrete notation, that allows the derivation of model transformation code. This approach tackles concept hiding in metamodels, which results in complex query and generation parts of model transformation rules. Furthermore, the user can reason about semantic correspondences in a notation and with concepts the user is familiar with. Hence, metamodel details resulting from the need for efficient API and repository implementations are hidden from the user.

We have presented relevant issues concerning MTBE, however, various extensions of the presented concepts are discussed in the following chapters, e.g., application on larger modeling languages, also from other modeling domains and full elaboration of the so far gained insights. In particular, MTBE requires proper tool support and methods guiding the mapping and transformation code generation tasks in order to fulfill the requirements for the user-friendly application of MTBE. Therefore, Chapter 6 then elaborates on an implementation of a prototype in order to be able to evaluate our proposed approach in the large.

Chapter 5

Advanced MTBE Concepts

Contents

In the previous chapter we have introduced MTBE and how it is split up into five distinct task comprising the MTBE process. Nevertheless, extensions and advanced concepts for MTBE are needed to face more complex integration scenarios than the one considered in Section 4.3. In particular, we develop additional mapping operators, both on the M1 layer and the M2 layer to capture and store more integration knowledge for the final code generation using higher order transformations. Also, we do reasoning on models and metamodels by means of pattern matching in a heuristic way and by means of reasoning algorithms.

5.1 Adding Expressiveness to the Model Mapping Language

In this section, we present a refined version of our model mapping language by first introducing its AS and subsequently its CS.

Abstract Syntax. The metamodel for the model mapping language defines all mapping concepts that may be applied by the user to accomplish the task of bridging two modeling languages by means of mapping example models shown in their concrete syntax. The abstract root class *ViewElement*, depicted in Figure 5.1, is in fact only for implementation convenience and to visualize that inheriting classes have an associated element on the view. The central class is the abstract class *Mapping*, that serves as basis for all kinds of connections relating two graphical model elements of source and target languages. The design of mapping ends in the metamodel (cf. references *lhs* and reference *rhs*) allows for all kinds

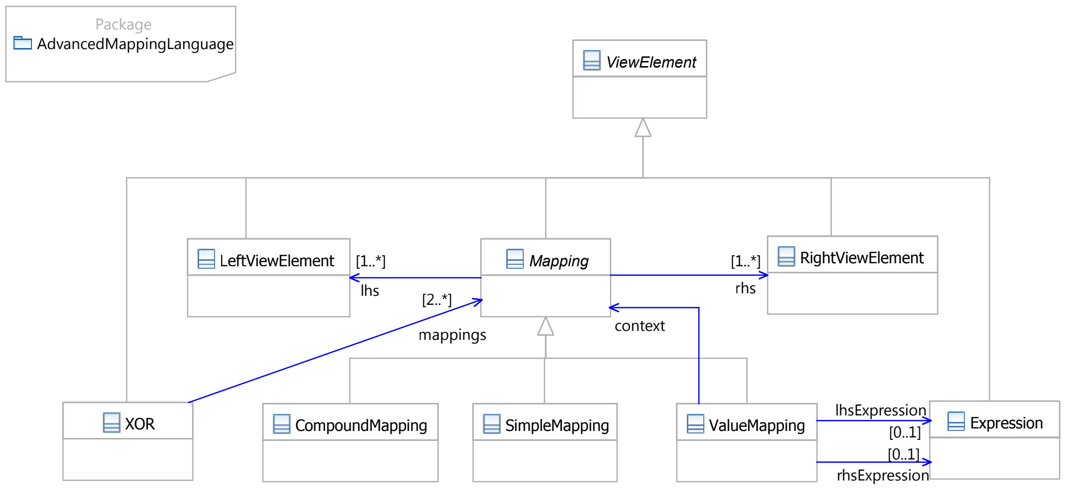


Figure 5.1: Extended model mapping language.

of mappings, i.e., *one-to-one*, *one-to-many*, and *many-to-many*. Elements of the languages to be integrated must have the corresponding abstract class *LeftViewElement* or *RightViewElement* as superclasses. How these requirements are realized within our MTBE approach is discussed in Subsection 6.2.2.

Actually, the remaining concrete classes form the bases for the concrete syntax of the mapping language, for which we defined a notation. However, the specification provided in Figure 5.1 is not sufficient to completely determine the abstract syntax of our mapping language. There exist further well-formedness rules for model which have to be defined with the Object Constraint Language (OCL) for each concrete subclass of class *Mapping*.

Simple Mapping. The concept that allows the user to draw simple *one-to-one* mappings between any two concrete syntax elements is represented by the *SimpleMapping* class. Additionally, to restrict on 1..1 multiplicities, the following constraint is necessary.

```
context SimpleMapping
inv: self.lhs -> size() = 1 and self.rhs -> size() = 1
```

Compound Mapping. To allow for *one-to-many* and *many-to-many* mappings, we introduced the *CompoundMapping* class. In order to complete the syntax specification the following constraint must hold for compound mappings.

```
context CompoundMapping
inv: self.lhs -> size() > 1 or self.rhs -> size() > 1
```

Value Mapping. The classes *ValueMapping* and *Expression* constitute what was introduced as string manipulation operator [81]. Whenever attribute values represented by labels are part of a mapping, it would be nice to have some sort of functions that can be applied to modify the participating attribute values appropriately. The container, which encapsulates the actual value mappings is the *ValueMapping* class, able to manage two lists whose elements point to a label. For each of these two lists, a function may be applied to. This function is stored within instances of *Expression* that supports e.g. the concatenation of values by accessing list elements through their index. A *ValueMapping* is however not self-dependent and thus must have a *context* specified, which can be either of type *SimpleMapping* or *CompoundMapping*.

XOR. The last operator we have to specify is the *XOR*. While experience has shown, that an explicit *XOR* operator on the concrete syntax layer is not desirable in common use cases as it can be derived on the metamodel layer automatically, we include it in the mapping language description for sake of completeness. The Role *mappings* of *XOR* must be unique. For xor-ed simple mappings we must further specify

```
context XOR
inv: self.mappings-> forAll(m | m.oclIsTypeOf(SimpleMapping))
inv: self.mappings.lhs-> asSet()-> size() = 1
xor self.mappings.rhs-> asSet()-> size() = 1
```

as a constraint. The first invariant says that this constraint can only be applied on mappings of type *SimpleMapping*. The second invariant is needed to further restrict the way XOR may be applied to *SimpleMappings*. More specifically we have to ensure that all *SimpleMappings* have on one side, i.e., either left-hand side or right-hand side, exactly one *ViewElement* in common.

Example 32. *An example of a wrong usage scenario of the XOR operator is given in Figure 5.2(b). Figure 5.2 also shows a valid application of XOR in (a). In (c) we depict the corresponding Object Diagram, which shows how serialization is done and gives a notion of how domain elements are mapped to ViewElements. In Chapter 6 we demonstrate how this mapping is done in tool support in practice.*

For the type *CompoundMapping* a similar constraint may be specified.

Concrete Syntax. Above we have described the AS of our model mapping language. Now we briefly present what the corresponding CS of the model mapping language looks like. Figure 5.3 depicts the notation tables for our mapping language. Each concrete class of our mapping language has a distinct CS element for defining model mappings. How these elements may be used in real world examples has been already presented in our previous work [81] for business process models. In this thesis, see Section 7.1, we present a concrete application of the model mapping language for bridging structural modeling languages as well as business modeling languages.

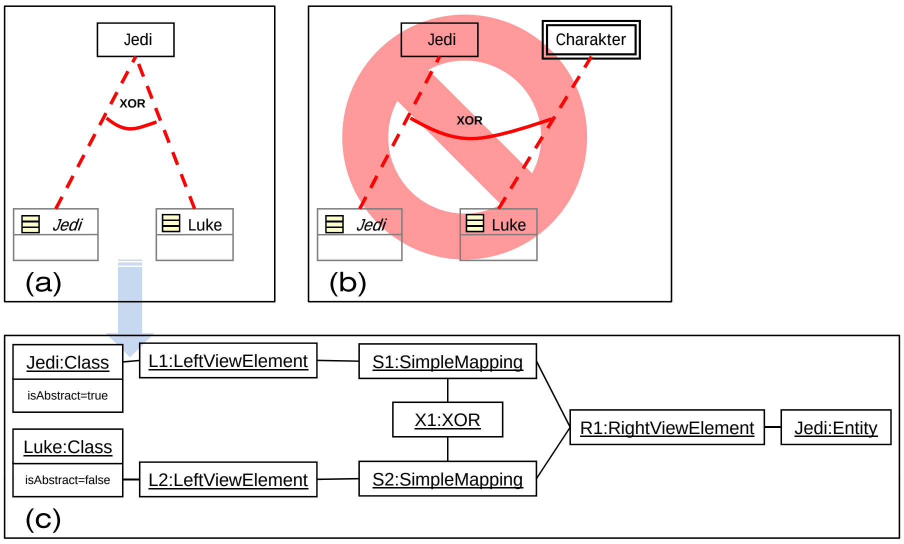


Figure 5.2: The XOR mapping operator, (a) Valid use, (b) Invalid use, (c) Object Diagram of (a).

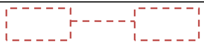
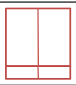
Meta Element	Concrete Syntax Element	Meta Element	Concrete Syntax Element
SimpleMapping		lhs, rhs (of ValueMapping)	🟥 {label}
CompoundMapping		Expression	{Value@Pos1}...{Value@Pos2}
ValueMapping		XOR	

Figure 5.3: Notation tables of the model mapping language.

Note this CS approach acts as a guideline. Implementation may slightly differ in some aspects. Also the *semantic* of our mapping language is only defined in natural language. A formal semantics is still missing and it is not clear yet in which way to define it.

5.2 Reasoning based Pattern Matching

This subsection covers the conceptual step number three outlined in Subsection 4.2. Model transformations operate on the model level but need to be defined having knowledge how metamodel elements semantically correspond to each other. This is why we have to perform a movement from model mappings defined by the user up to metamodel mappings. Unfortunately, user mappings are in general not as accurate as metamodel mappings have to be in order to be used as input for the generation of model transformations. Model mappings usually consist of ambiguities mainly because of various structural and semantical heterogeneities occurring between different modeling languages and due to the user-friendly model mapping language.

To cope with the shift in "mapping space", we propose reasoning based on pattern matching. By applying any kind of predefined or custom-defined model pattern, we aim to create metamodel mappings from model mappings on an automatic basis. These metamodel mappings can be made persistent in a so-called mapping model, which allows to relate all kinds of metamodel elements. In the following, we present six core patterns that are illustrated in Figure 5.4. The first three patterns operate on the model level (cf. $M \Leftrightarrow M'$ in Figure 5.4) and produce an initial mapping model, whereas the last three patterns are based on the metamodel level (cf. $MM \Leftrightarrow MM'$ in Figure 5.4) and aim at the refinement of the initial mapping model.

Initializer Pattern. The first pattern matches if classes, attributes, and references in metamodels are identical resulting in full equivalence mappings between metamodel elements. The basis for this pattern represents simple mappings between model elements and reasoning capabilities to check whether mapped objects (e.g., instance of class *A* and instance of class *B* in Figure 5.4) have equivalent attribute values and links. With the help of this pattern, all simple equivalence mappings between two metamodels can be found. However, with this pattern it is not possible to resolve structural or semantical heterogeneities between modeling languages. For resolving such problems, we propose the following five patterns.

Pathway Pattern. The second pattern poses a challenge as alternate paths have to be found if someone wants to set the link from an instance of A to an instance of B when transforming from M' to M. Analysis of the metamodel alone would not be sufficient, because metamodels might be nearly identical but the reference semantics are different. An analysis of both models represented in AS has to be performed to check for the existence of an equivalent path in M' between an instance of *C* and an instance of *D*.

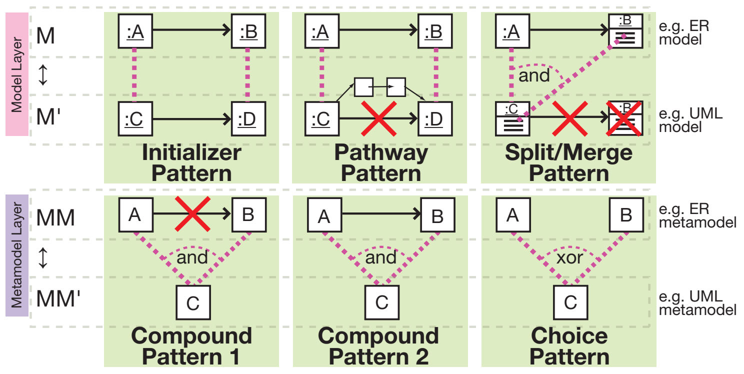


Figure 5.4: Core analyzer patterns.

Split/Merge Pattern. The third pattern illustrates the case, where two concepts are represented as two classes explicitly in one language, whereas these two concepts are nested within one class in the other language. As an example consider the UML class *Property*, that specifies the concept multiplicity as attributes, whereas in the ER metamodel the concept multiplicity is expressed explicitly as a class. This means, in transformations there is either a merge of objects and values or a split into separate objects. In principal, there is no restriction on the number of classes that need to be merged or splitted as long as they are somehow related and connected through references. Note that a merge of classes leads to concept hiding whereas a split makes concepts explicit.

Compound Pattern 1 – no structural connection. This pattern reasons about metamodel mappings which have been produced from compound mappings of the model level. In case no structural connection between instances of class *A* and class *B* can be found in the example model, then we simply define two independent classes *A* and *B* to be equivalent with one class *C*. This is the most trivial form of *one-to-many* mappings and leads to transformation rules which simply create two unrelated instances from one instance of class *C*.

Compound Pattern 2 – structural connection. This pattern also reasons about metamodel mappings produced for compound mappings of the model level and represents the opposite case of *Compound Pattern 1* in the sense that a structural connection between instances of class *A* and class *B* can be found in the example model. Consequently, this pattern produces metamodel mappings which lead to transformation rules for creating two linked instances from one instance of class *C*.

Choice Pattern. We encountered the case in which two distinct classes, such as class *A* and class *B*, are mapped to the one class *C*. This kind of metamodel mappings is produced for simple model mappings pointing from instances of one class to instances of several classes. Whenever this pattern is matched, further reasoning on the model represented in

AS in combination with the CS definition is needed trying to distinguish between the concepts which are hidden. Again, consider our simple UML to ER integration scenario of Figure 4.4. Instances of class *Property* represent on the one hand attributes when no link to an association instance is set and on the other hand roles when a link is set. This distinction is also represented in the CS of UML, thus the constraints can be reused to build the xor constraint between the metamodel mappings. If the feature comparison in combination with the CS definition does not yield any result, the user has to decide and adjust the metamodel mappings or transformation code manually.

The application of these core patterns is vital for the generation and refinement of the metamodel mapping model. The metamodel mapping language, see Figure 5.5, allows at the moment only for full or conditional equivalence mappings. However, for model transformation generation purposes this metamodel can be extended with additional mappings to be able to contain further information generated by the analyzer component.

5.3 Adding Expressiveness to the Metamodel Mapping Language

After applying the pattern matching on the model layer and the metamodel layer, we have to provide some way to store the retrieved information about semantic correspondences between metamodel elements. Conceptually, we have however spotted three possibilities to move from user mappings to executable transformation code:

1. Generate model transformation code in the course of pattern matching using a *template* based approach as supported by code generation frameworks.
2. Apply a *Higher Order Transformation (HOT)* [27] containing the pattern matching capabilities for analyzing the model mappings and generate a transformation model.
3. Run the pattern matching on model mappings and produce an intermediate *mapping model* upon a HOT is executed for producing a transformation model.

We believe that an intermediate mapping model capturing the derived correspondences between metamodel elements is well suitable for MTBE due to the following reasons.

- Existing implementations using mapping models between metamodels (e.g., HOTs and graphical editors) can be reused.
- Using a HOT ensures that we do not leave the *"modeling technical space"*.
- A mapping model between metamodels allows to keep track of the actual model transformation code generation. Thus, debugging is made easier.

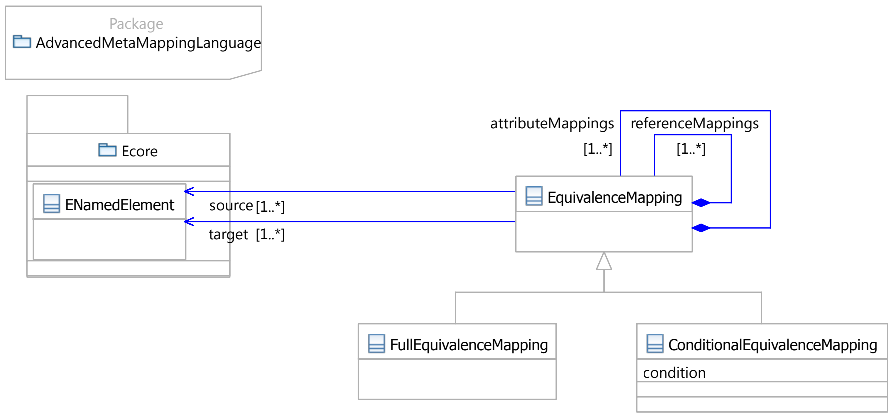


Figure 5.5: Extended metamodel mapping language.

- Complexity is reduced by separation of concerns by splitting the task moving from model mappings to model transformation code into two separate tasks.

- Customized HOTs may be easily applied leading to extensibility.

Figure 5.5 shows our basic mapping language for metamodels. The central concepts in this metamodel are represented by the classes *ConditionalEquivalence* and *FullEquivalence* used to distinguish between conditional equivalence mappings and full equivalence mappings. Equivalence mappings can additionally contain other mappings for specifying which reference mappings and attribute mappings belong to a certain class mapping. Note, that we do not categorize the mappings according to the types they reference. The task to interpret and act properly according to the types the mappings reference, is carried out by the HOT in a subsequent step.

Furthermore, the metamodel for metamodel mappings is quite different in its structure compared to the model mapping metamodel shown in Figure 5.1. The metamodel mapping language needs not to incorporate any usability and user-friendliness issues and can therefore contain any relevant information concerning the transformation model generation. For example, complex OCL conditions are also contained in the metamodel mapping model, cf. attribute *condition* in Figure 5.5.

In fact, one has to make sure that no information reasoned during the pattern matching process is lost. The metamodel in Figure 5.5 is to be seen as a core mapping language specification open for extension if necessary. This can be simply achieved by introducing additional subclasses of the abstract class *EquivalenceMapping*.

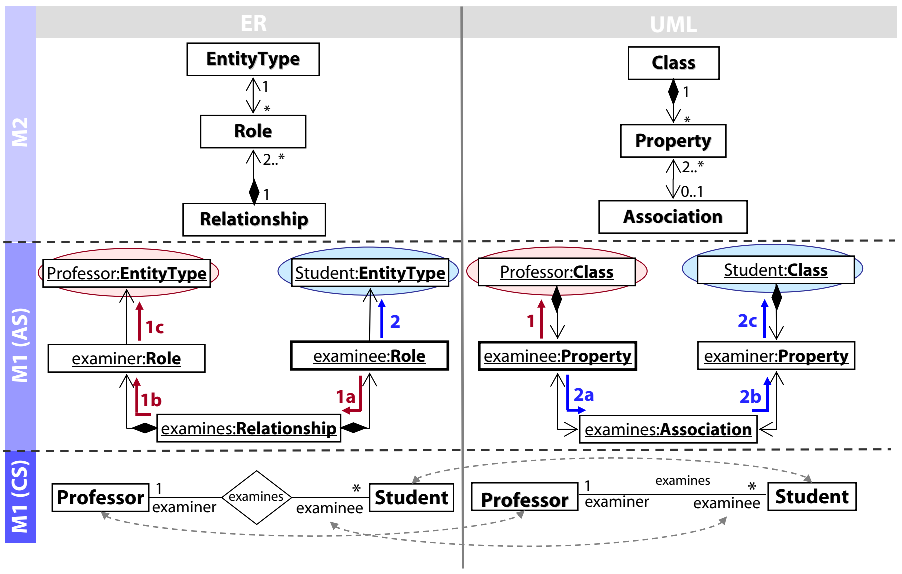


Figure 5.6: Reasoning about derived associations.

5.4 Advanced Reasoning Algorithms

There exist rudimentary reasoning mechanism for deriving simple *object*, *attribute*, and *association equivalences* from the abstract syntax of the models, the user defined mappings on the concrete syntax, and the mappings between the abstract and concrete syntax definitions. However, in the meantime we discovered, that these basic mechanisms are not applicable for all abstract syntax hetereogenities. Especially for association equivalences further reasoning mechanisms are required - also for models which have nearly the same metamodel structures. This topic has only be touched in Section 5.2 very briefly by the **Pathway Pattern**. Hence, in this section we discuss one instance of the additional reasoning algorithms, namely for finding equivalent queries for describing derived associations based on the example depicted in Figure 5.6.

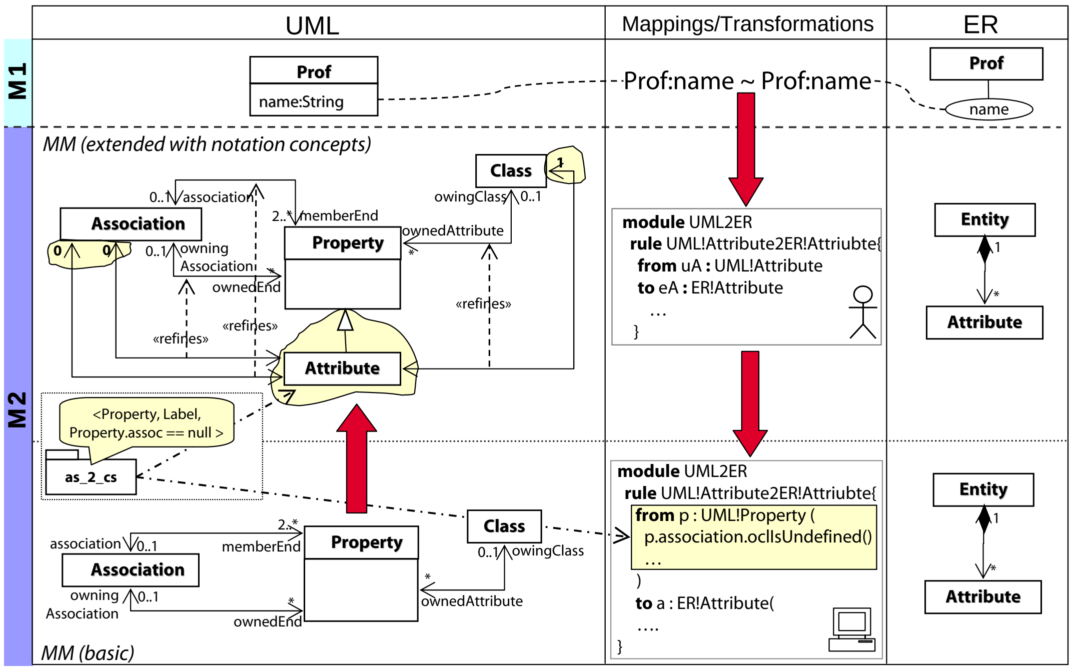
The upper part of Figure 5.6 (M2 layer) illustrates simple metamodels for *ER* and *UML*, respectively. As one can see, the metamodel structures are nearly identical. The lower part of Figure 5.6 (M1 layer) shows possible ER and UML models in abstract (*M1(AS)*) and concrete syntax (*M1(CS)*). In addition, in the concrete syntax layer the user defined mappings (grey dashed lines in Figure 5.6), which are shown to explain how the reasoning algorithm works. Again, the structures of the models are nearly identical, but one important difference in the abstract syntax can be identified, which is not directly visible in the metamodel layer. In ER an *EntityType* is linked to its adjacent *Role* in contrast to UML where a *Class* is linked to its opposite *Property*, e.g, *examinee:Role* is linked to *Student:EntityType* in the ER model, but in the UML model the corresponding element *examinee:Property* is linked to *Professor:Class* and not directly to *Student:Class* which is the corresponding element for *Student:EntityType*. This linking convention is not accurately defined in both metamodel structures, however, some hint may be given by association end names such as *owningClass* in the UML metamodel or *type* in the ER metamodel. However, in order to explicitly define such linking conventions some informal natural languages description are necessary, for example as is done in the UML 2 specification [67]. To tackle the automatical recognition of such linking conventions, which have an impact on finding equivalent associations in our MTBE approach, we have to introduce an advanced reasoning mechanism for finding derived associations. In the following we describe this mechanism based on the introduced example.

Assume we move from ER to UML and the user mapped *examinee:Role* to *examinee:Property* in the concrete syntax. This mapping is directly transferable to the corresponding abstract syntax elements, because no constraint is defined in the abstract to concrete syntax mapping of the modeling language. When transforming *examinee:Role* to *examinee:Property* we have to set the link to *Professor:Class* (cf. arrow *1* in Figure 5.6). But in the ER model there is no direct link between *examine:Role* and *Professor:EntityType*. At this point we have to start a search in the ER model to find out if it is possible to indirectly navigate from *examinee:Role* to *Professer:EntityType*. As it is illustrated in Figure 5.6 there is a path *1a* → *1b* → *1c* which can be used to derive the required information for generating the desired UML model. The same procedure is applicable to produce the link between *examiner:Property* and *Student:Class* in the UML model. Furthermore, when moving from UML to ER we also need to find derived associations, e.g., when generating the link between *examine:Role* and *Student:EntityType* (cf. arrow *2* in Figure 5.6). This link is derivable from the path *2a* → *2b* → *2c* in the UML model.

The above mentioned example shows the potential of an example based approach for deriving model transformations. The reasoning on the abstract syntax of models (M1) allows to detect syntactic heterogeneities which are not explicitly visible on the metamodel level and furthermore provides the requisite information for a solution (mechanism). Another benefit of this procedure is that the reasoning is totally transparent for the user who has to map domain models in concrete syntax only. However, we believe that it is important to search for other reasoning mechanisms and to provide a set of reasoning mechanisms from which the user may choose.

5.5 A Two Step Transformation Process

As we have already discussed in the previous chapter the refinement of model transformation code is part of the MTBE process. This refinement step is necessary in transformation scenarios where fully automatic generation of transformation code is not possible. There-



```
module UML2ER
 rule UML!Attribute2ER!Attriubte{
  from uA : UML!Attribute
  to eA : ER!Attribute
   …
 }
```

```
module UML2ER
 rule UML!Attribute2ER!Attriubte{
  from p : UML!Property (
   p.association.oclIsUndefined()
   …
  )
  to a : ER!Attribute(
   ….
}
```

Figure 5.7: 2-step transformation generation.

fore the user-friendly adaption of generated model transformation code is a requirement for a more advanced MTBE approach. The MTBE framework in its current state has one major drawback concerning the one-step model transformation generation based on the abstract syntax when the user needs to adapt the generated transformation by hand. This drawback is due to hidden concepts in the metamodel, that are explicit in the concrete syntax. Hence, the user has to deal with the used constraints from the notation when adapting ATL rules. We are to tackle this problem by applying higher-order transformations as introduced in [85] in combination with using models of models whereas each particular model has a particular purpose as introduced in [10].

In particular, we combine these two techniques in a two-step model transformation generation process with an intermediate layer as illustrated in Figure 5.7.

Step 1: Starting from the mappings between concrete domain model elements, in a first step, a model transformation is generated in which the concepts available in the notation are explicitly represented to hide complexities of the original metamodel from the user. For this intermediate step, a metamodel has to be generated from the original metamodel which in addition to concepts from the original metamodel covers concepts introduced by the notation. Hence, the purpose of this generated metamodel is explicit knowledge representation allowing easier development of model generation code. The generation of the extended metamodel is realized by automatically transforming the original metamodel combined with mapping conditions of the package as2cs into a new metamodel which explicitly represents the concepts.

Step 2: In a second step, the transformation code adapted with additional user extensions is transformed into a model transformation which operates on the abstract syntax. For this step we adopt the fact that also a transformation is a model, which allows the transformation, of a transformation to reduce to model transformation. In the transformation of the transformation, the sub-concepts introduced by the notation are reduced to their super-concepts and expressed in the transformation rules with complex conditions in the query parts.

Chapter 6

Implementation

Contents

So far, the MTBE approach has been introduced on the conceptual level, only. Its practical relevance may only be explored with proper tool support. The contribution of this chapter is to explain how MTBE concepts have been integrated into existing state-of-the-art graphical modeling and model transformation frameworks. We conclude with a critical discussion on our implementation approach.

6.1 Overview of the Graphical Modeling Framework

How GMF Works

With the existence of the EMF the need for constructing visual editors used to be met with the Graphical Editing Framework (GEF). The effort it takes to build a visual editor for one's own from scratch is remarkable and the learning curve coming along with the GEF API is quite steep. This is why the Graphical Modeling Framework (GMF) project was set up. It aims at automatically generating basically GEF code from certain modeling artifacts. This generation functionality is encapsulated into the GMF Tooling components. Besides GMF Tooling there exist runtime components, that provide common editor properties and a general look and feel. Related to MTBE primarily the tooling components are of interest to us,

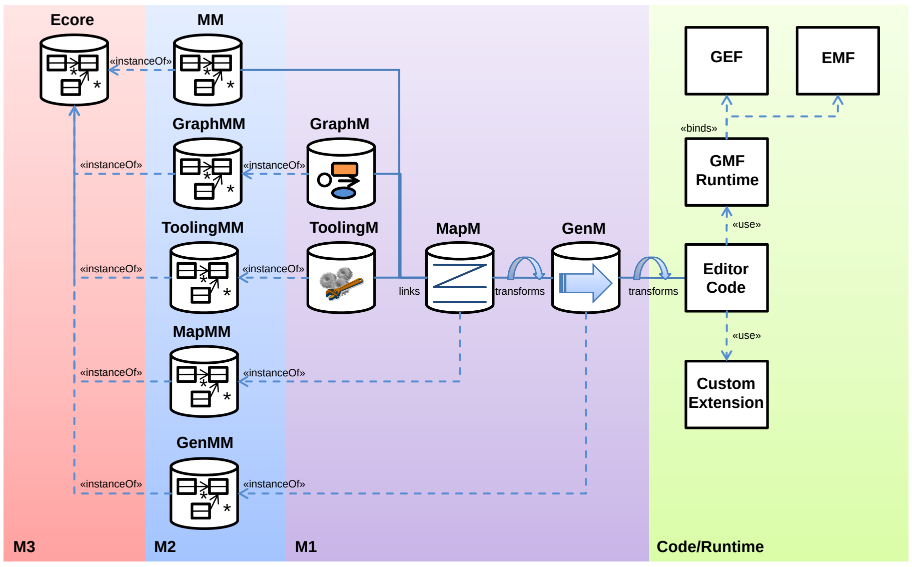


Figure 6.1: Overview of the Graphical Modeling Framework.

as they are enabling the MTBE visual frontend as will be explained later in this section. In Figure 6.1 we give a brief outline of the modeling artifacts defined and used by GMF.

For a better understanding of how smoothly GMF fits into model engineering paradigm we have categorized each modeling artifact according to its position in the 4-layer architecture proposed by the OMG's MDA approach. On the meta-metamodeling layer M3 we have the ECORE meta-metamodel introduced by EMF and based on a subset of MOF, i.e. EMOF. On M2 we find five metamodels, which the first one (*MM*) can be any user specified language based on ECORE and the other four metamodels are specified by GMF as they are needed in the process of editor code generation. The *GraphMM* defines any view elements and how they can be related to while the *ToolingMM* sets possible language constructs for toolbar specification. Similarly, *MapMM* and *GenMM* define mapping and generation model constructs. In order to generate an GMF based visual editor the user has to create three separate models, i.e. the *GraphM*, the *ToolingM* and the *MapM* located on M1. Most interesting however is the *MapM*, whose intention is to link the graphical definition, the tooling definition and the existing user metamodel elements together. The mapping model simply states what tool creates which domain element and how it looks like on the canvas. After the definition of these model element correspondences a transformation takes care of the generation of a so called GenModel (Generation Model). The GenModel then serves as input for the code generator templates (model-to-code transformation), that produce fully functional editor code. The GMF runtime is however designed such that the generated code

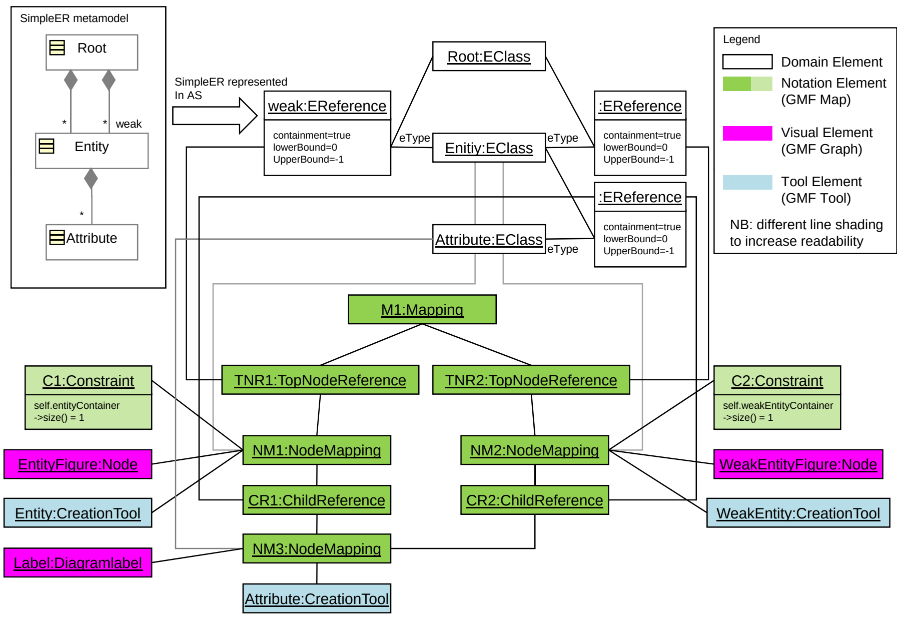


Figure 6.2: How GMF deals with notation and concrete syntax.

can be customized by various extension mechanisms.

MTBE Related Notation Issues

In Section 3.5 we have already discussed the difference between notation and CS and how the notation can be formally defined. GMF has chosen a very similar approach to capture the notation and thus keep AS and CS elements separated from each other to maintain flexibility. As mentioned above GMF maintains a mapping model called GMFMap referred to as *MapMM* in the formal specification in the previous subsection. This metamodel defines proper concepts to align domain elements, graphical view elements and tooling elements. Basically there exist 5 major mapping elements, i.e., *LabelMapping*, *CanvasMapping*, *NodeMapping*, *CompartmentMapping* and *LinkMapping*. Furthermore, there exist two concrete meta classes called *TopNodeReference* and *ChildReference* for the task of mapping references from the metamodel to the view.

Figure 6.2 gives an example of how the notation is implemented in GMF. In the left upper corner we show a very simple SimpleER metamodel with only three concepts and a standard EMF root element. These concepts are *Entity*, *Attribute* and weak entity, which is implemented through a containment reference called *weak*. The way this latter concept is implemented here can be regarded as concept hiding or metamodel heterogeneity.

To the left we depict the same metamodel in AS using the UML object diagram notation. The reason for this shift in presentation of the SimpleEr language is that we have now all meta concepts explicitly represented as UML instance specifications, which we can refer to from the GMFMap model. This means that our *EReferences* in the metamodel are now explicitly expressed as instances maintaining links to other instances, which they are connecting. Links to other objects may also reflect the role names of the corresponding *EReferences* in the metametamodel, i.e., Ecore. In most cases role names are negligible but to distinguish between source and target objects of an *EReference* we labeled one link with *eType*. If we had some *EAttributes* specified in our SimpleER metamodel their instance specifications would then also be easy to link with our example GMFMap model.

At the bottom of Figure 6.2 all GMF related elements are drawn as colored instances of their corresponding GMF tooling models explained in the previous subsubsection. For the sake of clarity we omit some details like the compartment mapping for *Attributes* and some graphical definitions. The point we want to emphasis in this object diagram is the way GMF deals with ambiguities in the AS and hence in the CS. The concept of weak entity is only reflected by an additional containment reference from *Root* to *Entity* and shares the same metaclass with the concept *Entity*. GMF defines *NodeMapping* as the view relevant concept. Both instances *NM1* and *NM2* are linked to the same *EClass* making it for the GMF runtime impossible to distinguish between these two distinct view elements. To solve this problem GMF provides for the definition of OCL constraints to handle ambiguities on the CS level. In our example we therefore specify *Constraint* instances *C1* and *C2* containing the proper OCL constraints, which make the differences in the AS visible, i.e., the different *EReferences* to the container metaclass *Root*.

Model Operators

The automatic MTBE editor code generation needs to have the following modeling artifacts available: $\{MM_1, MM_2, GraphM_1, GraphM_2, ToolingM_1, ToolingM_2, MapM_1, MapM_2\}$. Afterwards four distinct model operations have to be carried out.

1. $JointMM = MM_1 \cup MM_2 \cup MM_ML = merge(MM_1, MM_2, MM_ML)$.
2. $JointMapM = MapM_1 \cup MapM_2 = merge(MapM_1, MapM_2)$ having $JointMM, GraphM_1, GraphM_2, ToolingM_1, ToolingM_2$ available for reference.
3. $JointGenM = transform(JointMapM)$.
4. $Code = transform(JointGenM)$.

This is of course seen from a high level perspective and shall only briefly give an overview how modeling artifacts of GMF are handled to foster the aim of having tool support for the visual, concrete syntax part of MTBE. The transform operations in step 3 and 4 are a model transformation using Java and model to code transformation using templates respectively. These two steps are basically determined by GMF itself. Steps 1 and 2 have been implemented by us using simply Java as transformation language. We could have also decided to use ATL instead and define model transformation rules in a declarative way. The next two paragraphs describe how we implemented steps 1 and 2 in detail.

6.2 An Eclipse Based Implementation for MTBE

Our MTBE framework comprises several components. As can be seen in Figure 6.3(1), our implementation heavily relies on major Eclipse projects, providing essential functionalities our components are based upon. These components are the Eclipse Modeling Framework (EMF) [12], the Graphical Modeling Framework (GMF) and Eclipse serving as IDE. The components within Figure 6.3(2) comprise the user front-end and match with conceptual *Step 2* of the MTBE process. In Figure 6.3(3) all components facilitating the task of metamodel mapping modeled in *Step 3* are depicted. Figure 6.3(4) components correspond to *Step 4*, i.e., the code generation. Note that we omitted dependencies and interfaces among non-MTBE components to preserve readability. In what follows we will give a short description on each of the modules and will focus on the user front-end in subsequent sections.

Model Mapping Language. The *MappingLanguage* component provides a specification of several alignment operators in terms of an Ecore based metamodel, cf. *ML MM* in Figure 6.3. The implementation of this component allows the user to map different kinds of visual model elements as described in Subsection 5.1. The definition of this mapping language is central to the MTBE framework as it is directly or indirectly used by other components.

Merge Components. To be able to define mappings between model elements in a graphical way certain artifacts have to be available a priori in one or the other way. Therefore, it is assumed that at least a language definition in terms of a metamodel, a graphical definition of the concrete syntax, and a mapping between these two exist for every modeling language. To allow for mapping two metamodels within the Eclipse environment we decided to merge existing artifacts and build a single editor for both languages. This merging procedure is described in Subsection 6.2.2. The *MapMerger* component also takes care of the *MTBE Mapping Editor* component generation with the help of *GMF Tooling*.

MTBE Mapping Editor. Our graphical editor prototype uses the *GMF Runtime* and is built solely from *GMF Tooling* at the moment.

Analyzer Component. The Analyzer takes the output model of the GMF editor, i.e., the user defined model mappings, as well as the corresponding metamodels and tries to match various kinds of patterns as presented in Section 5.2. The user can decide which of the available patterns shall be applied in order to translate the manual mappings into an AMW [26] weaving model conforming to the MTBE weaving metamodel, which basically captures the concepts presented in Subsection 5.3. This module will be designed in such a

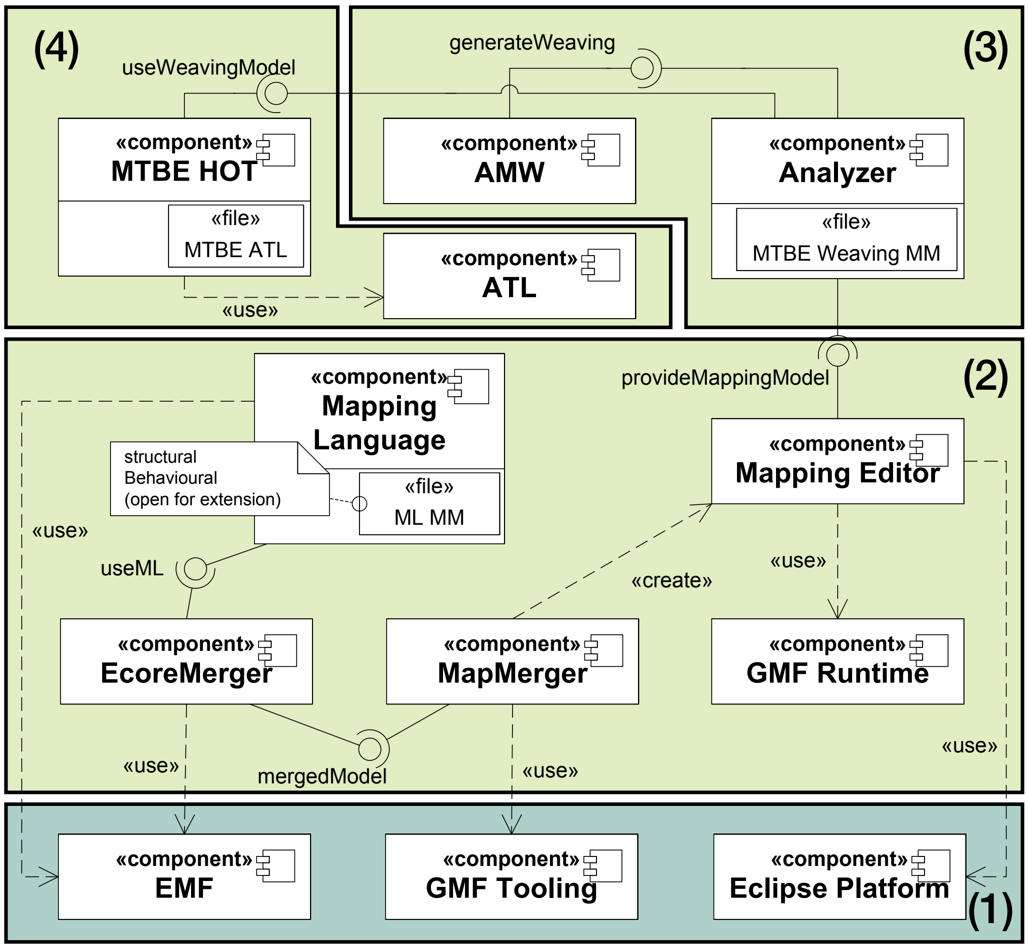


Figure 6.3: Framework architecture, (1) Underlying frameworks, (2) User front-end, (3) Pattern matching and creation of weavings, (4) Transformation model generation.

way to allow for several pattern extensions.

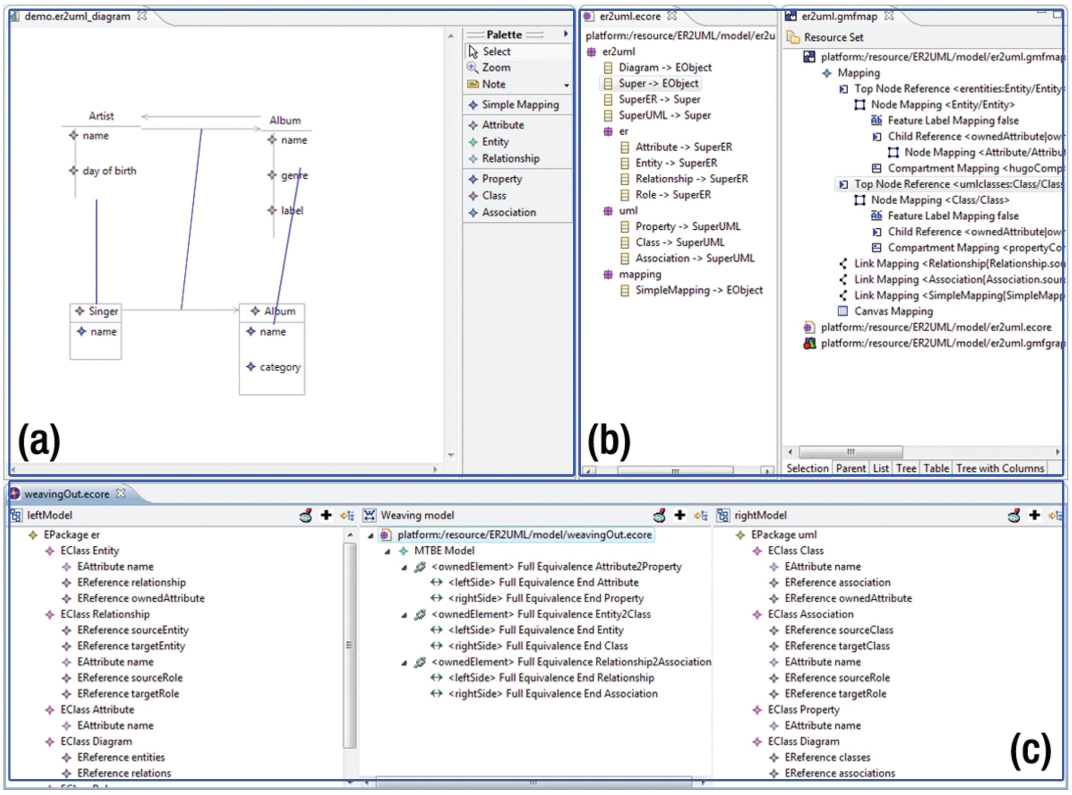
MTBE HOT Component. On top of our framework lies the *MTBE HOT* component. This component takes the generated weaving model as input and produces an ATL model conforming to the ATL 2006 metamodel [39]. The built in ATL transformation model can be used to generate transformations in both directions. After generation of the transformation models the AM3 extractor can be run to get the user readable ATL code. The advantage of this approach stems from the fact that all artifacts used in the code generation are models. No hand-coded templates or Java programs are involved to produce the ATL code. The output of the Analyzer module shall serve as input for several kinds of transformation code generators, depending on the use case, so we are not limited to ATL.

6.2.1 MTBE Workbench

The MTBE prototype is implemented as Eclipse plug-in. The workbench is to consists of four different views. The first view is composed with our *MTBE Mapping Editor*, that operates on the notation and lets the user define mappings. In the second view, the bridging between the two metamodels is presented to the user with the help of the Atlas Model Weaver, which is extended by our MTBE weaving metamodel. The third view presents an editor, that shows all so far generated ATL code as textual representation. It's left to the user whether to refine the code or not. The last view can then be utilized to test the transformation code and see its results immediately.

6.2.2 Integration of GMF

Our MTBE approach poses a novel idea as it moves the task of developing model transformation away from the complex structures of abstract syntax to the well known notation elements of modeling languages. To achieve this increased usability for model engineers we decided to integrate the Graphical Modeling Framework (GMF) to be able to generate a graphical editor. The advantage of this decision is that we can use the capabilities of an emerging framework, supported by a rather big community. In order to apply GMF to create an editor out of a simple generation model we have to provide for some model merging components in our MTBE framework. In the following we describe the merging process in more detail and explain how the generated editor can be used to define (appropriate) mappings between notation elements. In the context of MTBE we assume that a graphical editor for some modeling language can be fully generated from GMF (and EMF models). We do not cope with any editor specific changes made to the generated code. Therefore we rely on the declarative power of GMF models and their limitations.



er2uml.ecore

platform:/resource/ER2UML/model/er2u
- er2uml
  - Diagram -> EObject
  - Super -> EObject
  - SuperER -> Super
  - SuperUML -> Super
  - er
    - Attribute -> SuperER
    - Entity -> SuperER
    - Relationship -> SuperER
    - Role -> SuperER
  - uml
    - Property -> SuperUML
    - Class -> SuperUML
    - Association -> SuperUML
  - mapping
    - SimpleMapping -> EObject

er2uml.gmfmap

Resource Set
- platform:/resource/ER2UML/model/er2uml.gmfmap
  - Mapping
    - Top Node Reference <erentities:Entity/Entity
      - Node Mapping <Entity/Entity>
        - Feature Label Mapping false
        - Child Reference <ownedAttribute|ow
          - Node Mapping <Attribute/Attribu
        - Compartment Mapping <hugoComp
    - Top Node Reference <umlclasses:Class/Class
      - Node Mapping <Class/Class>
        - Feature Label Mapping false
        - Child Reference <ownedAttribute|ow
        - Compartment Mapping <propertyCo
    - Link Mapping <Relationship{Relationship.so
    - Link Mapping <Association{Association.sour
    - Link Mapping <SimpleMapping{SimpleMapp
    - Canvas Mapping
- platform:/resource/ER2UML/model/er2uml.ecore
- platform:/resource/ER2UML/model/er2uml.gmfgra

weavingOut.ecore

leftModel
- EPackage er
  - EClass Entity
    - EAttribute name
    - EReference relationship
    - EReference ownedAttribute
  - EClass Relationship
    - EReference sourceEntity
    - EReference targetEntity
    - EAttribute name
    - EReference sourceRole
    - EReference targetRole
  - EClass Attribute
    - EAttribute name
  - EClass Diagram
    - EReference entities
    - EReference relations

Weaving model
- platform:/resource/ER2UML/model/weavingOut.ecore
  - MTBE Model
    - <ownedElement> Full Equivalence Attribute2Property
      - <leftSide> Full Equivalence End Attribute
      - <rightSide> Full Equivalence End Property
    - <ownedElement> Full Equivalence Entity2Class
      - <leftSide> Full Equivalence End Entity
      - <rightSide> Full Equivalence End Class
    - <ownedElement> Full Equivalence Relationship2Association
      - <leftSide> Full Equivalence End Relationship
      - <rightSide> Full Equivalence End Association

rightModel
- EPackage uml
  - EClass Class
    - EAttribute name
    - EReference association
    - EReference ownedAttribute
  - EClass Association
    - EReference sourceClass
    - EReference targetClass
    - EAttribute name
    - EReference sourceRole
    - EReference targetRole
  - EClass Property
    - EAttribute name
  - EClass Diagram
    - EReference classes
    - EReference associations

Figure 6.4: MTBE Prototype, (a) Mapping editor, (b) Merged Ecore model and merged mapping definition model, (c) Automatically produced weaving model.

Ecore File Sharing

The GMF mapping definition model, that constitutes the mapping from AS to CS, relies on an Ecore based metamodel besides a graphical as well as a tooling definition model. The latter two we do not have to cope with in our merging process as they represent highly reusable elements. But for the Ecore model our MTBE framework needs a merging component which we call *EcoreMerger*, as shown in Figure 6.3. This component takes two Ecore models, representing different modeling languages, as input and generates a single Ecore file (cf. Figure 6.4b on the left). Each language is represented by its own package, which is a child of a common root package. The root package itself has to contain a single class that owns all required containment references to classes from the original metamodels. These required references can easily be extracted from the two Ecore files, as these also have to consist of a single class acting as a container, which is a specialty of EMF. We introduced another type of class in the root package, because of GMF limitations, the *GMFElement{language name}*. This class serves as supertype from which all classes of a modeling language have to inherit. Our EcoreMerger therefore has to modify all classes in these language packages accordingly. However, this design decision then allows us to have mappings defined between every two classes in the two modeling languages. To define these mappings in our graphical editor we also have to add mapping meta classes to our merged Ecore metamodel. This is done in the *mapping* package, that includes the *SimpleMapping* class with source and target references of type of the superclass *GMFElement{name}*. After the work of the EcoreMerger is completed, the created Ecore file can be used to generate the EMF generation model as well as the EMF model and edit code, that are prerequisites for the GMF diagram code.

Mapping Definitions Merging

Similar to the joint Ecore file we have to provide a component that merges the two mapping definition models (GMFMap models) to a single model, the GMF generation part can handle to create diagram code, see Figure 6.4(b) on the right. The algorithm for solving this task simply copies the different kinds of mapping elements from both mapping models into a single one and adjusts the XPath expressions for domain model elements to point to the newly created Ecore file. XPath expressions for tooling and graphics can be reused. After execution of our *MapMerger* (cf. Figure 6.3) tool the GMF generation model is produced from the merged mapping model to facilitate diagram code generation.

6.3 Critical Discussion

6.3.1 Implementation Status

Currently the development of our first experimental prototype is completed. So far we have implemented a mapping language for general mapping scenarios only. Based on our *Mapping Language* we fully implemented the *EcoreMerger* as well as the *MapMerger*. After the user finishes the merging processes a wizard guides through the generation of all required *Mapping Editor* plug-ins. The plug-ins are placed directly into the plug-ins folder of eclipse and are ready to use after a workbench restart. The generated *Mapping Editor* possesses all modeling features of the two input editors as long as the latter have been solely created with GMF Tooling. A detailed description and a user manual can be found in [57]. The prototype can be found and downloaded at www.modelcvs.org.

6.3.2 Discussing the GMF Approach

For every two modeling languages a user wants to define model transformations, a unique editor with its own domain, graphics, tooling and mapping models has to be generated. There is no possibility to reuse already generated code. Before one can begin to define mappings on the concrete syntax, the merging and generation components of the MTBE framework have to be executed. In order to reduce the cost of time we have provided for an implementation that can do things nearly without user interaction. Another drawback of the current approach is that we are not able to cope with custom code in an available editor, which limits our design possibilities. For example UML roles are contained in a *Class* and not directly shown at an *Association* in our simple UML GMF editor.

In order to fully support our MTBE approach we would like to be able to map labels, such as role names of associations explicitly. GMF does not support such a mapping scenario. In most cases such mappings are given implicitly and can be deduced by simple string comparison algorithms or available matching tools. But there might be situations in which the user wants to explicitly define mappings.

Also GMF suffers from minor bugs at the moment, which primarily have an impact on the editor view and how elements are rendered or not. But these are well known bugs that will most likely be removed in future versions of this very promising framework.

6.3.3 Alternative Implementation Approaches

MTBE in its described implementation lacks to take custom GMF editor code into account. It is in most cases however not satisfactory to have only GMF tooling and its declarative models available. It is also not the aim of GMF to provide modeling artifacts that are powerful enough to replace programming languages completely. The aim of GMF is to reduce

the cost of time to build a comprehensive modeling editor. Consequently, we have been thinking of alternatives capable of handling custom code refinements and still use the GMF framework as well as the proposed MTBE framework design for model alignment and code generation.
Separation of Editors. Basically there is no need to have only one editor allowing to draw correspondences between CS elements between two different modeling languages. We could also have two separate editors active at a time and only put those CS elements on the canvas, which are equivalent and therefore should be mapped. This mapping of elements can then however be done implicitly by just storing the elements on the canvas in some XMI serialization and establishing the mapping automatically. We do lose the visualization of mappings this way and have to cope with some other problems, e.g., we cannot draw connections without corresponding nodes on the canvas, but we could use editors having defined custom code sections.
The Aspect Way. Another way to reuse already implemented GMF modeling editors is to use aspects and the OSGI component model to relate various plug-ins among each other as described by [35, 58]. Their work is based on an aspect-oriented programming language called Object Teams/Java and the Eclipse implementation of the OSGI standard Equinox. The basic idea is to have an arbitrary number of plug-ins that can be handled as aspects for other plug-ins through proper binding methods. Following this approach we could implement one central GMF editor that handles only the user mapping language. The two other editors enabling the model creation can then be simply bound as aspects to the base plug-in, i.e., our GMF mapping language editor. This would be done dynamically by the user and because of this simple binding custom code would not get lost. There are some difficulties concerning the binding of different GMF editors because there are some classes, which have to be handled with care. According to [58] these issues will be overcome in the near future.

6.4 Summary

In this chapter we have introduced our prototype for MTBE, which has been implemented as Eclipse plug-in. This prototype allows for defining semantic correspondences between domain models (M1) shown in their concrete notation, from which model transformation code can be generated. Our framework is based on emerging technologies, such as the Graphical Modeling Framework, which are based on Eclipse and the Eclipse Modeling Framework. Our early prototype has shown that it possible to realize our framework based on the current versions of the underlying technologies. However, one important drawback of the proposed framework must be mentioned, namely the strong dependency on the success and the usability of the underlying technologies.

Chapter 7

Applications of MTBE

Contents

The aim of this chapter is to prove the practical relevance of MBTE with the help of our prototypical implementation presented above. For this reason we have conducted two different case studies in the domains of structural and behavioral modeling. The results of the two case studies are critically reflected and directions to improve the implementation of MTBE in future work are presented.

7.1 Application to Structural Modeling Languages

In this section, the functionality of our prototype is demonstrated by a small case study. The aim is to be able to transform UML models into Entity Relationship (ER) diagrams and vice versa. Note, that these two modeling languages have already been used in [89] to exemplify our MTBE approach. However, we had no implementation at hand to underpin our conceptual findings. With our prototype we are now in the position to demonstrate general MTBE concepts in practice. The MTBE plug-in as well as various artifacts and examples can be downloaded from our ModelCVS project site[1].

[1]http://modelcvs.org/prototypes/mtbe.html

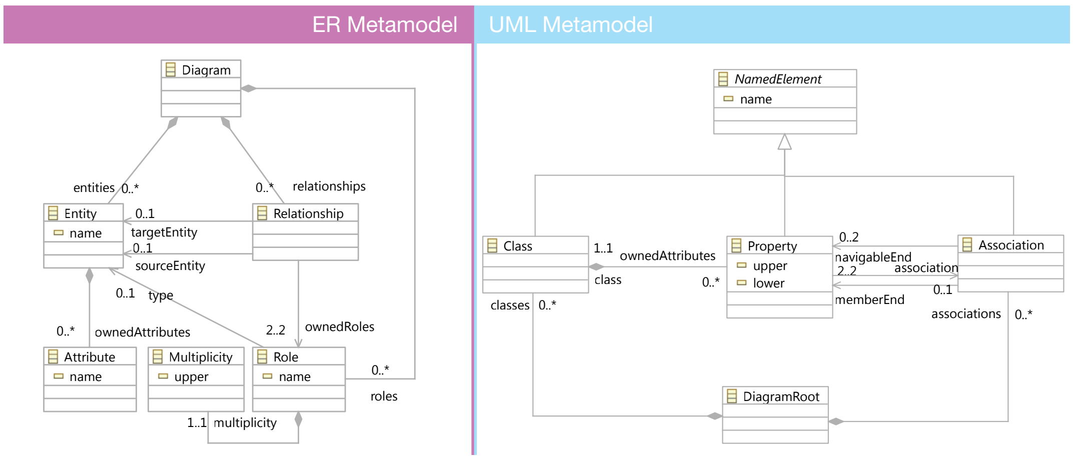


Figure 7.1: Simplified ER and UML metamodels.

7.1.1 Integration Problem: ER2UML

Figure 7.1 illustrates two simplified metamodels of the ER diagram and UML class diagram, respectively. Generally speaking, both languages are semantically nearly equivalent and thus the corresponding metamodels cover the same modeling concepts. Note that the metamodels for the two languages are designed such that their concrete syntax can be fully specified by declarative GMF components. Most important to us is that we can define the notation by means of the GMFMap model.

ER. The ER metamodel covers the basic concepts of *Entity*, *Relationship*, *Attribute*, *Role*, and *Multiplicity*. *Diagram* acts as the basic root element and is an implementation-specific of EMF. *Entities* are connected via *Relationships* through their *sourceEntity* and *targetEntity* references. *Entities* can further contain an arbitrary number of *Attributes*. *Relationships* can be assigned two distinct *Roles* through their *ownedRoles* reference. *Roles* are not contained in their corresponding *Relationship* but in the root element itself. Furthermore, a *Role* must be assigned to a certain *Entity*, which is done through the *type* reference. *Roles* are further enforced to contain a *Multiplicity* that consists of a single attribute called *upper* specifying the upper bound multiplicity of a role.

UML. The UML metamodel in turn consists of *Classes*, *Properties* and *Associations*. The abstract class *NamedElement* is for convenience only. The root element *DiagramRoot* is equivalent to *Diagram* in the ER metamodel. We introduced concept hiding in the UML metamodel by representing attributes and roles by the same class, namely by the class *Property*. One can only distinguish between these two concepts by the optional reference *association*, whose inverse reference is *memberEnd*. More specifically, an instance of class *Property* represents a role when the reference *association* is set. In case the reference *association* is not set

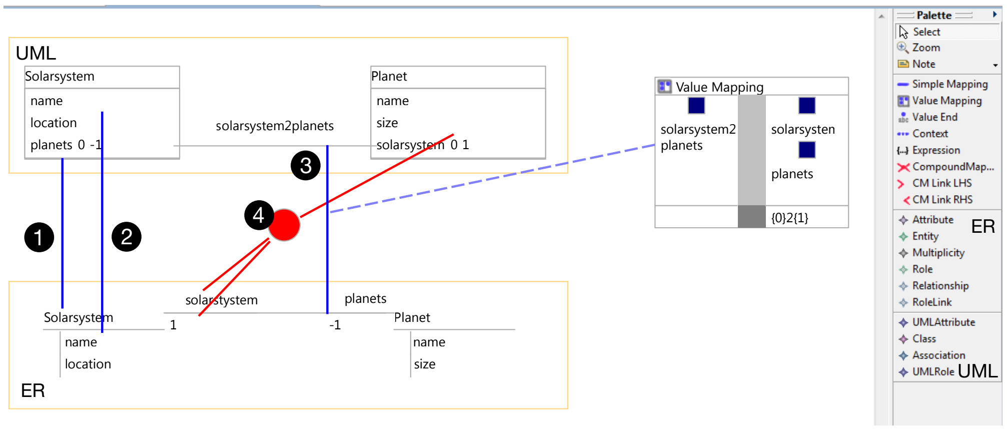


Figure 7.2: Model mappings between ER and UML example models.

by an instance of class *Property*, it represents an attribute. The class *Property* also comprises the attributes *lower* and *upper* in order to cover the concept multiplicity. Relationships between classes are achieved via roles and the *memberEnd* feature of *Association*. The feature *navigableEnd* of *Association* indicates whether a role is navigable or not.

Mostly all concepts presented in the ER metamodel are also covered in the UML metamodel. Although, the two metamodels can be considered semantically equivalent, there exist structural heterogeneities between them, that would complicate the manual creation of model transformations. These structural heterogeneities entail further reasoning upon the model mappings in order to generate a semantically richer mapping model, which is the basis for the model transformation generation.

7.1.2 User defined Model Mappings

Figure 7.2 depicts the mappings between the ER example model and the UML example model established by the user. Each GMF tool provided in the palette, see right side of Figure 7.2, has been used and therefore all concepts are covered by the models. Table 1 summarizes the mappings specified by the user. In particular, we have mostly used simple mappings, however, for mapping roles in combination with multiplicities of ER models to properties of UML models, we employ a compound mapping. Furthermore, we need to attach a value mapping to the simple mapping between relationship and association in order to set the name of an association (note that in our ER metamodel the relationship itself does not have a name).

Table 7.1: Summary of model mappings.

Mappings	UML Model Elements	Mapping Kind	ER Model Elements
(1)	Solarsystem: Class	*Simple Mapping*	Solarsystem: Entity
(2)	name: Property	*Simple Mapping*	name: Attribute
(3)	solarsystem2planets: Association	*Simple Mapping (+Value Mapping)*	: Relationship
(4)	solarsystem: Property	*Compound Mapping*	solarsystem: Role 1: Multiplicity

Table 7.2: Summary of metamodel mappings.

MAP	UML MM Elements	ER MM Elements	Mapping	Comment	Pattern	Model MAP
C1	DiagramRoot	Diagram	*Full Equiv.*	Reasoned via GMF properties	**X**	**X**
C2	Class	Entity	*Full Equiv.*		Pattern 1	(1)
C3	Association	Relationship	*Full Equiv.*		Pattern 1	(3)
C4	Property	Role ⋈ Multiplicity	*Cond. Equiv.*	Property.association == null	Pattern 3 + Pattern 6	(4)
C5	Property	Attribute	*Cond. Equiv.*	Property.association != null	Pattern 6	(2)
A1	Class.name	Entity.name	*Full Equiv.*			
A2	Property.name	Attribute.name	*Full Equiv.*			
A3	Property.name	Role.name	*Full Equiv.*			
A4	Association.name	*Function*	*Full Equiv.*	Exp Annotation		(3)
A5	Property.upper	Role.multiplicty.upper	*Full Equiv.*			
A6	Property.lower	**X**	**X**	Default Value: 0		
R1	Class.ownedAttributes	Entity.ownedAttributes + User Interaction	*Cond. Equiv.*	Roles have to be collected (cf. C4, C5)		
R2	Association.navigableEnd	Relationship.ownedRoles	*Full Equiv.*	Convention		
R3	Association.memberEnd	Relationship.ownedRoles	*Full Equiv.*			
R4	DiagramRoot.associations	Diagram.relationships	*Full Equiv.*			
R5	DiagramRoot.classes	Diagram.entities	*Full Equiv.*			

7.1.3 Reasoning and Metamodel Mappings

Based on the model mappings, the Analyzer generates a mapping model capturing as many metamodel mappings as possible. Concerning our core patterns presented in Section 5.2, the Analyzer component can apply pattern 1, 3, and 6 shown in Figure 5.4 for finding equivalent classes as is summarized in Table 2. Pattern 1 generates the mappings *C2* and *C3*, while mapping *C1* has been additionally reasoned from GMF configurations to find the equivalent root classes which represent the modeling canvas. Pattern 3 is used to reason about the compound mapping (cf. model mapping (4)) which results in mapping *C4* from *Property* to the join of *Role* and *Multiplicity*. Furthermore, pattern 6 is able to generate the conditions for splitting properties into attributes and roles. From these class mappings, most of the attribute and reference mappings can be derived which are necessary for the model transformation. Due to brevity reasons, we only show the reference mappings necessary for transforming ER models into UML models. As one can see in Table 2, only one user interaction is necessary for completing the model transformations, namely the *Class.ownedAttributes* reference must be split in two subsets, one can be reasoned, however, the other has to be defined by the user.

7.1.4 ATL Code Generation

Based on the metamodel mappings, we show how our HOT produces valid ATL model transformations. The ATL code for transforming ER models into UML models depicted in Listing 7.1 comprises both, the automatically generated code by our HOT implementation and some user refined code fragments.

Listing 7.1: ER2UML transformation including user refined code.

```
-- @atlcompiler atl2006
module ER2UML; -- Module Template
create OUT : UML from IN : ER;

rule diagram2diagramRoot{
    from d : ER!Diagram
    to dr : UML!DiagramRoot(
        classes <- d.entities ,
        associations <- d.relationships
    )
}

rule entity2class{
    from e : ER!Entity
    to c : UML!Class(
        name <- e.name,
        ownedAttributes <- e.ownedAttributes
    )
}

rule attribute2property{
    from a : ER!Attribute
    to p : UML!Property(
            name <- a.name
        )
    }

    rule relationship2association{
        from rel : ER!Relationship
        to a : UML!Association(
            name <- rel.ownedRoles.first().name+'_2_'+rel.ownedRoles.last().name,
            memberEnd <- rel.ownedRoles -> collect( t |
                thisModule.resolveTemp(Tuple{r = t, m = t.multiplicity}, 'p')
            ),
            navigableEnd <- rel.ownedRoles
        )
    }

    rule role_multiplicity2property{
        from
            r : ER!Role,
            m : ER!Multiplicity (
                    r.multiplicity = m
            )
        to p : UML!Property(
            name <- r.name,
            class <- ER!Relationship.allInstances()
                -> select(x|x.ownedRoles -> collect(y|y.type)-> includes(r.type))
                -> collect(y|y.ownedRoles).flatten()
                -> select(y|y.type <> r.type).first().type,
            upper <- m.upper
        )
    }
```

Listing 7.2: UML2ER transformation including user refined code.

```
-- @atlcompiler atl2006
module UML2ER; -- Module Template
create OUT : ER from IN : UML;

-----------------------------HELPER BEGIN----------------------------------------------------------

helper def : getRelationship (role : ER!Role) : ER!Relationship =
  ER!Relationship.allInstances()-> select(e|e.ownedRoles
    -> includes(role)).first();

-----------------------------HELPER END------------------------------------------------------------

rule diagramRoot2diagram{
  from dr : UML!diagramRoot
  to d : ER!Diagram(
    entities <- dr.classes,
    relations <- dr.associations,
    roles <- dr.classes -> collect(x|x.ownedAttributes).flatten()
      -> select(x| not x.association.oclIsUndefined())
  )
}

rule class2entity{
  from c : UML!Class
  to e: ER!Entity(
    name <- c.name,
    ownedAttribute <- c.ownedAttributes
      -> select(x|x.association.oclIsUndefined())
  )
}
```

```
31
32  rule property2attribute{
33    from p : UML!Property (p.association.oclIsUndefined())
34    to a : ER!Attribute(
35      name <- p.name
36    )
37  }
38
39  rule association2relationship{
40    from a : UML!Association
41    to r : ER!Relationship(
42      ownedRoles <- a.memberEnd,
43      sourceEntity <- a.memberEnd.last().class ,
44      targetEntity <- a.memberEnd.first().class
45
46    )
47  }
48
49  rule property2role{
50    from p : UML!Property (not p.association.oclIsUndefined())
51    to r : ER!Role(
52      name <- p.name,
53      multiplicity <- m,
54      type <- p.association.memberEnd -> collect(x|x.class)
55        -> select(x|x <> p.class).first()
56    ),
57    m : ER!Multiplicity(
58      upper <- p.upper
59    )
60  }
```

In general, the generation of transformation code dealing with object and value creation is rather simple. What complicates our automatic transformation model generation are links between the objects, especially when the metamodels have different structures due to structural heterogeneities. The first three ATL rules shown in Listing 7.1 can be derived fully automatically. Rule *relationship2association* comprises a tricky part in lines 32 to 34. Because we deal with multiple source pattern matching in rule *role_multiplicity2property*, we have to use the *resolveTemp* construct to query produced properties. Therefore, this reference assignment looks complicated in ATL, but may be generated out of the metamodel mappings. An issue we have to deal with manually is depicted in lines 47 to 50 of the last rule. As for *roles* in ER, we also have to set the container for the corresponding *properties* in UML. However, the concept of a role is mirrored among ER and UML and therefore it was not possible to automatically produce a metamodel mapping which is also depicted in Table 2 mapping *R1*. Therefore, a rather complicated query has to be defined by the user, which assigns *properties* to *classes*.

For the sake of completeness we provide the ATL code for the other transformation direction in Listing 7.2 where similar user adjustments have been made to obtain a complete and correct transformation output.

7.2 Application to Behavioral Modeling Languages

Instead of focusing on the domain of *structural modeling languages*, what has been done in previous investigations [89], [84], in this section we concentrate on *behavioral modeling languages*. More specifically, we apply MTBE on the domain of business process modeling (BPM), which, up to our best knowledge, has not yet been subject to the MTBE approach. The definition of requirements for MTBE in the context of business process modeling and how they can be met in terms of proper generation of transformation rules comprise the main contribution of this section. Therefore, we present *main challenges encountered in business process (BP) model transformations*, and how these challenges can be tackled by using our extended MTBE *mapping operators* and *reasoning algorithms* in order to allow a more sophisticated model transformation code generation. Furthermore, we present a case study in which two prominent BP modeling languages are used, namely the UML Activity Diagram and Event Driven Process Chains.

7.2.1 Models for Business Processes

Business process models are in use for quite a long time and continue to gain importance as support from the software engineering field is improving significantly. Particularly model engineering fosters research in the area of BPM. There exist several metamodels for existing languages in order to raise there acceptance and tool interoperability. Due to this growing interest in BPM and proper tool support, we believe MTBE can be advantageous for specifying model transformations between BP models. Usually BP models cover various perspectives as e.g. described in [19]. The following two BP modeling languages we choose to use in our case study presented in Section 7.2.3, however, cover only the behavioral perspectives of BPM.

UML 2.1 Activity Diagram

The UML 2.1 Activity Diagram (UML AD) [69] is a specification of the Object Management Group. The metamodel of Figure 7.3 depicts an excerpt of the UML AD language, namely the basic control flow elements which are used for modeling BP models, as well as the concrete syntax.

The central element is the *Opaque Action*, which is used to model the activities within a process. The *Call Behavior Action* represents the concept of a sub process call. *Control Nodes* are used to structure the process. More specifically, a *Fork Node* and a *Join Node* express a concurrent flow as well as a *Decision Node* and a *Merge Node* to express an alternative flow. The *Initial Node* marks the begin of a process model. The UML AD differs between two final nodes, the *Flow Final Node* (*FFN*) and the *Activity Final Node* (*AFN*). The *FFN* is used to mark the final of a distinct flow, that means if it is reached the remaining tokens in the process

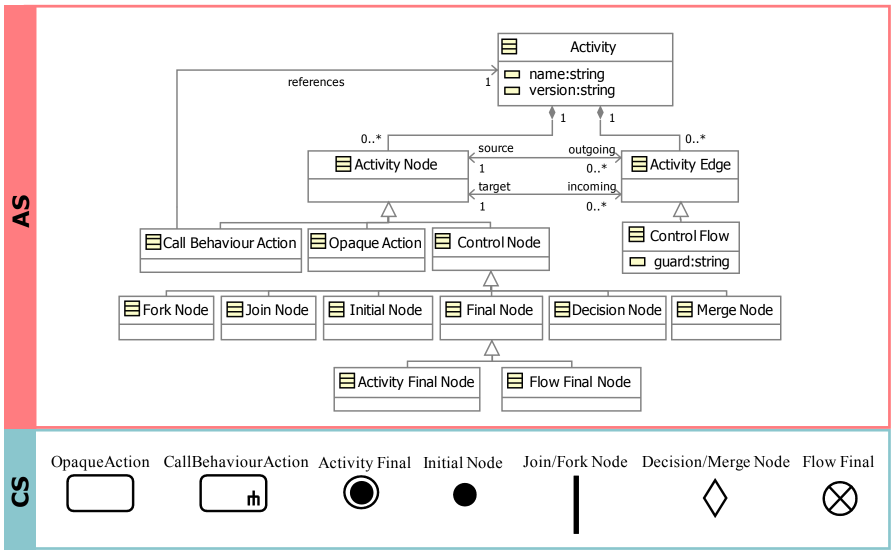


Figure 7.3: Parts of the UML 2.1 AD metamodel and its concrete syntax.

proceed. Whereas the *AFN* marks the end of the whole process which means if it is reached the remaining tokens in the process are killed immediately. The only kind of *Activity Edge* we consider in this work is the *Control Flow*, which is used to connect the *Activity Nodes* to form the flow of control of a process.

Event-driven Process Chains

Event-driven Process Chains (EPCs) [43] have been introduced by Keller et al in 1992 as a formalism to model processes. We focus on the main elements, which are used to model the control flow aspect of a BP model. The metamodel and concrete syntax of EPCs are illustrated in Figure 7.4.

The *Function* represents an activity. It creates and changes information objects within a certain time. The *Event* represents a BP state and is related to a point in time, it could be seen as passive element compared to the *Function* as an active element [52]. To model a sub process call the *Complex Function* is used. The *Logical Operators* elements are used to structure the proceed of the BP model.

When dealing with EPCs some special modeling restrictions must be considered which are not directly represented in the metamodel. EPCs do not provide a specific element to indicate the begin and the end of a BP model, instead the *Event* is used. *Event* elements are not allowed to be in front of an *OR* and *XOR* element. *Function* and *Event* elements must alternate in the proceed of the BP model and are connected via the *Control Flow*. This feature

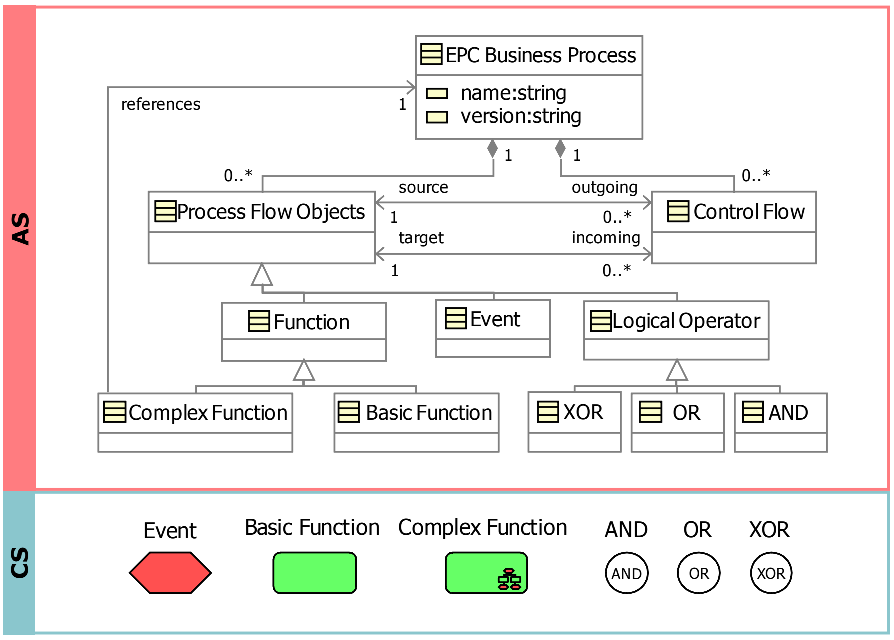


Figure 7.4: Parts of the EPC metamodel and its concrete syntax.

of the EPC language is in fact a static semantic constraint, which is not specified in the metamodel illustrated in Figure 7.4. Another restriction in EPCs is that parallel branches as well as alternative branches must be split and merged with the same kind of *Logial Operator*. Again we have to face a static semantic constraint in the context of *Logical Operators*, when it comes to specifying model transformations.

7.2.2 Dealing with Heterogeneities in Business Procss Models

During our investigation of BP models we discovered, that there are considerable differences compared to structural models concerning the requirements for MTBE. To transform structural models, one has to be familiar with the notation and hidden concepts in the metamodels, especially when dealing with UML diagrams. Resulting ambiguities on the metamodel layer have to be solved either by reasoning algorithms or user input, as we described in detail in our previous work. Now, with the task of transforming BP models we have to deal with quite different issues, in order to apply our MTBE approach. A lot of interesting aspects concerning the heterogeneity of BP models have been identified in [59].

One of the special requirements coming along with BP models has its root in the mapping from concrete to abstract syntax layer (notation) and the number of modeling elements involved on each layer.

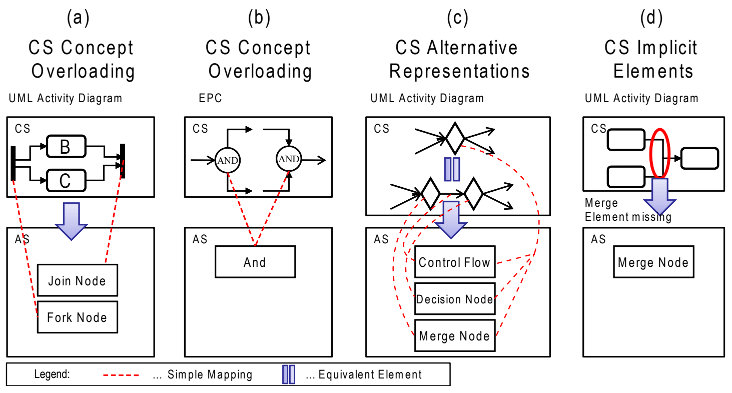


Figure 7.5: Overview of BP model heterogeneities.

In UML AD we have for example the notation:

$$< \{MergeNode, ControlFlow, DecisionNode\}, \{DecisionMergeFigure\}, \{\} >$$

as is illustrated in Figure 7.5(c) for the CS modeling element on the very top. Note, that the used modeling construct is here just an abbreviation on the CS layer and could be equivalently expressed by the following pattern of notation triples:

$$< \{DecisionNode\}, \{DecisionFigure\}, \{\} >$$
$$< \{ControlFlow\}, \{ConnectionFigure\}, \{\} >$$
$$< \{MergeNode\}, \{MergeFigure\}, \{\} >$$

We also observed several heterogeneities between modeling languages, which pose further requirements for MTBE. Figure 7.5 gives four examples for the peculiarities we found in the two BP modeling languages we introduced in Section 7.2.1. Examples (a) and (b) in Figure 7.5 depict the case of so called CS overloading in UML AD and EPC. In example (a) we encounter no problems because with the help of the notation we can distinguish between the two concepts join and fork despite the CS overloading. In example (b) CS overloading represents a real challenge for MTBE as two equal CS elements, but in fact featuring two different meanings, are mapped to the same AS element.

When we have to deal with alternative representations in the CS, see Figure 7.5(c), we can use the notation in MTBE to find them. The challenge arises not until we have to map two languages, where one consists of such variation points in the CS. Example (d) in Figure 7.5 shows the possibility in UML AD to merge parallel flows implicitly by omitting a merge/join node, i.e., we have no mapping from the AS to the CS.

In the following we apply our advanced mapping operators, which have indeed been influenced and inspired by BP models, and transformation heuristics which resolve heterogeneities, as expressed in the examples (a),(b), and (c), in Figure 7.5 are faced. Unfortunately, up to now we are not able to cope in MTBE with implicit elements as shown in example (d). The problem here is twofold. First we have to address the question how to map these implicit elements on the concrete syntax layer. And second we have to adjust the code generation process accordingly.

7.2.3 Integration Problem: UML AD and EPC

Our MTBE approach for the domain of Business Process modeling can be best explained in a by-example manner. Therefore, we use the two BP languages EPC and UML AD described in Section 7.2.1. For demonstration purposes we show what the generated code would look like in ATL. Although the example given in Figure 7.6 is rather simple, it still covers a lot of interesting aspects for MTBE.

For the case study we assume that on the concrete syntax layer in EPC's *Events* and *Basic Functions* to always occur pairwise connected through a *Control Flow* edge. Furthermore, in UML AD modeling it could be possible to omit a *Join* node and therefore model joins implicitly. However, in our first MTBE approach for BPM we do not jet cope with implicit joins or merges.

7.2.4 Model Mappings and Metamodel Mappings

As a first step one has to define manual mappings between two languages, which the transformation model shall be derived from. In the example in Figure 7.6 we specified six mappings that capture all concepts being used in the two sample models. Mappings *a,b,c,d,f,* and *g* are of type simple mapping.

Mapping *e* is of type compound mapping with multiplicity 1:3. Consequently, whenever the pattern *Event*, *Control Flow*, *Basic Function* is matched this corresponds to a single *Opaque Action*. We also marked the *Basic Function C* in our compound mapping as anchor element, which has implications specific to transformation code generation. In our case the ATL code generator would use this *Basic Functions* metamodel element as single source pattern element instead of using multiple source pattern elements. During our implementation attempts we realized, that an anchor feature can be desirable in some transformation scenarios.

Mapping *h* in our example takes care of the labels used in *Events*, *Basic Functions* and *Opaque Actions*. To maintain usability this string manipulation operator is used in a separate modeling element and references the involved labels. To define string equivalences one can use only unidirectional mappings, which are applied transforming from one set of labels to another. An optional expression allows us for example in mapping *h* to apply a *toLowerCase()* operation on the first mapping of the right hand side set of labels.

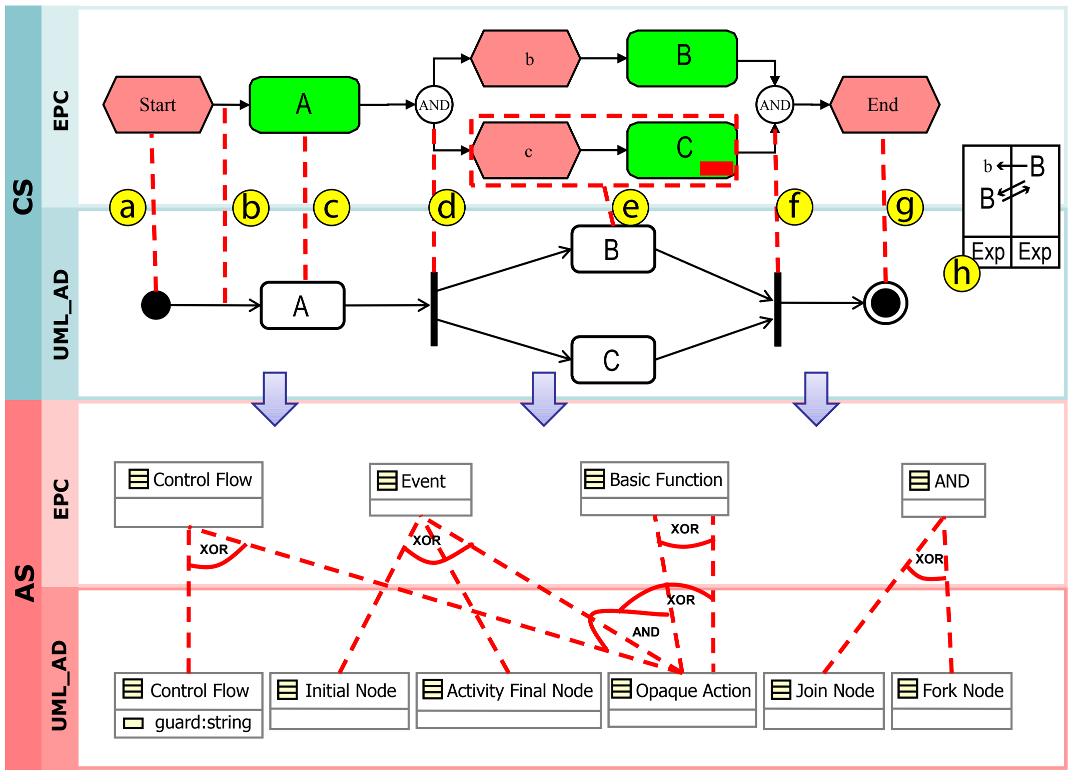


Figure 7.6: Mapping EPC and UML activity diagram - CS + AS perspectives.

Table 7.3: Summary of model mappings for BP models.

Mappings	UML AD	Mapping Kind	EPC
(a)	: InitialNode	*Simple Mapping*	Start: Event
(b)	: ControlFlow	*Simple Mapping*	: ControlFlow
(c)	A: OpaqueAction	*Simple Mapping*	A: BasicFunction
(d)	: ForkNode	*Simple Mapping*	: AND
(e)	B: OpaqueAction	*Compound Mapping*	c: Event : ControlFlow C: BasicFunction
(f)	: JoinNode	*Simple Mapping*	: AND
(g)	: ActivityFinalNode	*Simple Mapping*	End: Event
(h)	B: name	*Value Mapping*	b: name B: name +[0][0].toLowerCase

In EPC's there are no distinct metamodel elements nor distinct concrete syntax elements for start and end nodes, although these concepts are used in the modeling language implicitly. In UML AD we do have explicit concepts for start and end nodes both, in the model and the metamodel. If a transformation from EPC2UML_AD has to be performed the transformation model must know how to distinguish between start and end nodes even without having these concepts specified in EPC. We will elaborate on this issue in 7.2.4. Table 7.2.4 summarizes the above described user mappings in a compact and clear manner.

To keep our illustration in Figure 7.6 transparent and clear we omitted the mappings between CS and AS. Also these mappings are quite straightforward to define, as there are no constraints specified in the notation.

At last the mappings between the two metamodels can be derived from the user mappings and the notation. To highlight the existence of a compound mapping in the metamodel we marked the three involved mappings with an *and* operator. On the metamodel mapping level we now make use of our new XOR operator we introduced in Section 7.2.2. To keep the mapping task user-friendly the XOR between mappings can be reasoned automatically based on information in the metamodels. Whenever a meta class contains at least two outgoing mapping edges, an XOR relation can be set in an implicit way. A complete list of metamodel mappings including *EAttribute* and *EReference* mappings is shown in Table 7.2.4. Note, that XOR mappings are expressed in our weaving model by means of conditional equivalence mappings with proper OCL conditions, which will be explained later in this section.

How to Make Mappings Executable

As the automatic generation of transformation rules is a difficult task, we do not claim to support fully automatic rule generation. Instead we believe in a semi-automatic approach. To face the new domain of business process models we implemented a methodology, which

Table 7.4: Summary of metamodel mappings for BP models.

MAP	UML AD MM Elements	EPC MM Elements	Mapping	Comment	Pattern	Model MAP
C1	Activity	EPCBusinessProcess	*Full Equiv.*	Reasoned via GMF properties	**X**	**X**
C2	InitialNode	Event	*Cond. Equiv.*	Event.incoming -> size() = 0	Pattern 6	(a)
C3	ActivityFinalNode	Event	*Cond. Equiv.*	Event.outgoing -> size() = 0	Pattern 6	(g)
C4	ControlFlow	ControlFlow	*Cond. Equiv.*	Property.association == null	Pattern 1+6	(b)
C5	OpaqueAction	BasicFunction	*Cond. Equiv.*	BasicFunction that succeeds a "StartEvent" or OpaqueAction that succeeds InitialNode	Pattern 6	(c)
C6	ForkNode	AND	*Cond. Equiv.*	AND.incoming -> size() = 1 and AND.outgoing -> size() > 1	Pattern 6	(d)
C7	JoinNode	AND	*Cond. Equiv.*	AND.incoming -> size() > 1 and AND.outgoing -> size() = 1	Pattern 6	(f)
C8	OpaqueAction	Event ⋈ ControlFlow ⋈ BasicFuncition	*Cond. Equiv.*	Event.incoming -> size() <> 0 and Event.outgoing -> size() <> 0	Pattern 5	(e)
A1	Activity.name	EPCBusinessProcess.name	*Full Equiv.*			
A2	Activity.version	EPCBusinessProcess.version	*Full Equiv.*			
A3	InitialNode.name	Event.name	*Full Equiv.*			
A4	ActivityFlowFinal.name	Event.name	*Full Equiv.*			
A5	OpaqueAction.name	BasicFunction.name	*Full Equiv.*			
A6	OpaqueAction.name	Event.name ⋈ BasicFuncition.name	*Full Equiv.*	Exp Annotation		(h)
R1	ControlFlow.outgoing	ControlFlow.outgoing	*Full Equiv.*			
R2	ControlFlow.incoming	ControlFlow.incoming	*Full Equiv.*			
R2	ActivityNode.source	ProcessFlowObjects.source	*Full Equiv.*			
R4	ActivityNode.target	ProcessFlowObjects.target	*Full Equiv.*			

can be best compared to Architecture-Centric MDSD [80]. First of all we have implemented correct ATL transformation code, which acts as reference implementation. Thereby we have avoided imperative code sections and concentrate on coding in a declarative fashion.

In the next step we have developed the mapping operators described in Section 7.2.2. During this step we have turned our attention to the user-friendliness.

Next we have looked at the example models, the user mappings and the metamodels and tried to deduce the reference implementation. Code segments that could not be deduced automatically then lead to further refinement of the underlying heuristics. After refinement we tried again to deduce the reference implementation. This process can be seen as an iterative way to deduce heuristics on how to generate ATL transformation rules from a given set of models, metamodels and user mappings. The aim of this process is to optimize the relation between user-friendly mapping operators and the ability to generate executable transformation rules.

ATL Code Generation

Due to space limitations we will not expand on every aspect of the ATL code generation for the example in Figure 7.6. Instead we focus on the most interesting and challenging parts, only. The three ATL code snippets presented in the following paragraphs transform from EPC models to UML ADs. However, the example mappings provided by the user, also allow for UML AD 2 EPC transformation code generation.

Event2InitialNode and Event2FlowFinal. We already mentioned that we somehow have to distinguish between *Events*, that can be either normal *Events*, *Start Events* or *End Events*, to properly generate elements in UML Activity models. In our previous work we tried to overcome mapping and thus generation problems by means of reasoning on the metamodel layer. For business process models it seams to be more appropriate to do reasoning on the model layer.

Listing 7.3: ATL rule for Event2InitialNode and Event2FlowFinal.

```
rule StartEvent2InitialNode {
    from
            s : EPC!Event ( s.incoming->size() = 0 )
    to
            i : Activity!InitialNode (...)
}

rule EndEvent2FlowFinal {
    from
            e : EPC!Event ( e.outgoing->size() = 0 )
    to
            f : Activity!FlowFinalNode (...)
}
```

When the user maps two elements that are completely identical in the metamodel in one language, but correspond to two different elements in the other language, reasoning algorithms have to examine the graph structure in the example model. In our *Event* example the algorithm would have to determine, that *Start Events* do not have any incoming *Control Flows* and that *End Events* do not have any further outgoing *Control Flows*. Listing 7.3 shows the corresponding ATL rules with proper conditions in the source pattern. This addresses mappings *a* and *g* in Figure 7.6.

In ATL it is not possible to match an element more than once, i.e., to have more than one rule applied. Therefore, whenever a metamodel element occurs in at least two source patterns, we have to make sure that only one rule is matched. In the example above this would only be possible, if the user would model an *Event* without any *Control Flow* connected to it. Of course this would already violate some validity constraint. However, the OCL constraint to check for multiple matching would look like in ATL as follows:

$$\begin{aligned} EPC!Event.allInstances()- > select(e|e.incoming- > size() = 0)- > asSet() \\ - > intersection(EPC!Event.allInstances()- > select(e|e.outgoing- > size() = 0)) \\ - > size() = 0 \end{aligned} \tag{7.1}$$

EventControlFlowFunction2OpaqueAction. Now we want to cope with the user mapping *e* of Figure 7.6. From the model itself and especially the metamodel we know, that in EPC there are three distinct concepts involved whereas in UML Activity diagrams only one concept is affected. For this reason we use a new feature coming along with ATL 2006, i.e., the matching of multiple source pattern elements, see Listing 7.4. Note that the returned set of elements from matched multiple source pattern elements corresponds to the cartesian product.

Listing 7.4: ATL rule for EventControlFlowFunction2OpaqueAction.

```
rule EventControlFlowFunction2OpaqueAction {
    from
            ev : EPC!Event,
            c  : EPC!ControlFlow,
            f  : EPC!BasicFunction (
                c.target = f and c.source = ev and
                ev.incoming->size() <> 0 and
                ev.outgoing->size() <> 0
            )
    to
            o : Activity!OpaqueAction (
                name <- f.name,
                parent <- f.parent,
                incomming <- ev.incoming
            )
}
```

This is why we have to give a guard clause ($c.target = f$ *and* $c.source = ev$) to select only those elements we are interested in. This is similar to a join in SQL. To generate this "join" condition automatically we have to assume that elements in a compound mapping are always connected through proper link elements. A reasoning algorithm can check for the existence of links and build conditions that must hold for the pattern to match.

There are two more conditions given in Listing 7.4 that must evaluate to true if this rule shall be executed. This condition originates from the XOR constraint we face in the metamodel between the mappings *Event_InitialNode*, *Event_ActivityFinalNode* and *Event_OpaqueAction*, which is actually part of a compound mapping indicated by an *and*. To avoid matching a rule twice we can just take the conditions we have deduced in the previous two ATL rules and insert their negation, i.e. $ev.incoming->size() <> 0 \; and \; ev.outgoing->size() <> 0$. The idea of inserting the negation of already existing conditions in other rules can be seen as general heuristic.

And2Fork and And2Join. In Figure 7.5 *b* we referred to the problem of concept overloading in the CS, which we face in the transformation from an EPC to a UML AD model. We know, that the simple mappings *d* and *f* are actually in an XOR relationship, which is determined from the deduced mappings between the metamodel elements. This transformation difficulty was also the reason why we introduced the XOR operator. The user mappings together with the derived XOR constraint are not yet sufficient to provide for a heuristic capable of generating valid transformation code. What we need in this special case of concept overloading is an algorithm performing "local reasoning" on a specific node and compare the results with the ones from another one. The differences in the properties found between these nodes are then used to distinguish between them. In our example we determined for the class *And* mapped to class *Fork Node* there has to be only one incoming *Control Flow* and at least two outgoing *Control Flows* on the CS layer if the rule And2Fork shall be applied. For the class *And* mapped to class *Join Node* the opposite has to be true if the rule And2Join is supposed to match. Both rules are given in Listing 7.5

Listing 7.5: ATL rule for And2Fork and And2Join.

```
rule And2Fork {
    from
            an : EPC!AND (
                an.incoming->size() = 1 and
                an.outgoing->size() > 1
            )
    to
            fn : Activity!ForkNode (...)
}

rule And2Join {
    from
            an : EPC!AND (
                an.incoming->size() > 1 and
                an.outgoing->size() = 1
            )
```

```
        to
                jn : Activity!JoinNode (...)
    }
```

7.3 Critical Discussion

ER2UML

During our case study and experimenting with our prototype we discovered some limitations of our implementation.

HOT. Our basic Higher Order Transformation works well and produces model transformations in both directions. There are however some metamodel weavings that we do not fully support in our HOT, i.e., weavings that contain an OCL expression originating either from the notation or from some analyzer pattern. The problem is that OCL conditions are stored in weaving models as strings. But in the HOT we have to compute the equivalent abstract syntax trees for producing the ATL transformation model. As an alternative we consider a template based approach producing ATL code out of weaving models, where we could use plain OCL strings and need not handle abstract syntax trees.

UML_AD2EPC

The code for the examples above is generated in a heuristic way and we believe that in many cases and languages there is no great effort for code refinement necessary. However, there are limitations of MTBE we want to briefly discuss. In our language definition of EPC we assumed, that every *Basic Function* is directly preceded by an *Event*. But it may be possible in our example to place the *Events b* and *c* as one *Event* in front of the *And* split. This is another static semantic constraint one can only capture in a natural language description of EPC. As an example we refer to Figure 7.7(a).

Due this alternative way of positioning concrete syntax elements our rule defined in Listing 4.2 would no longer match any of the elements in such small models. To solve this problem MTBE could again be applied on this new EPC example model and map the concepts of interest again. For the example given in Figure 7.7(a) we would have to map the *Basic Function* located between the two *And* elements to an *Opaque Action* in our EPC 2 UML AD mapping scenario. Because of the XOR constraints later derived in the metamodels a heuristic could be applied to prevent multiple rule matching and select the rules properly. The mapping of the *Event a* remains however an open issue.

In Section 7.2.2 we presented the heterogeneity of alternative representations in CS. In UML AD we could model a join followed by a fork the way shown in Figure 7.7(b1). This representation is just an abbreviation, which we want to map in our example to EPC, where this form of abbreviation is not possible. From the users point of view it is sufficient to

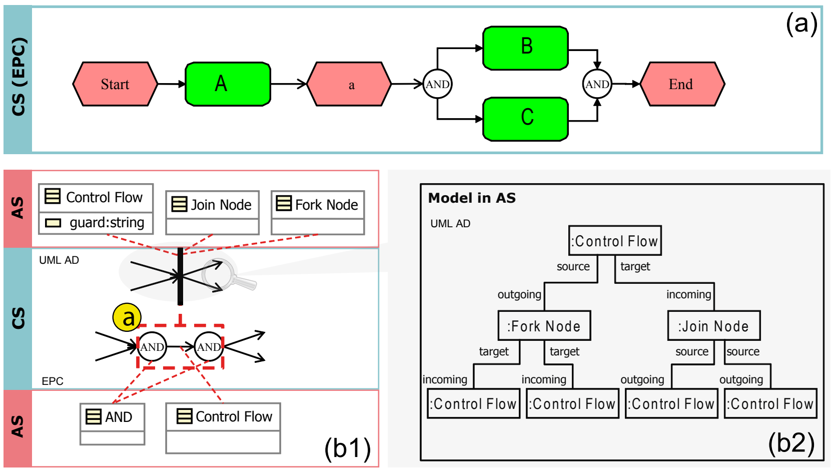


Figure 7.7: Challenges for MTBE in business process models.

simply draw the compound mapping *a*. For both modeling constructs we have again drawn the corresponding metamodel elements and also their mapping to the CS (notation). As one can easily see it is not possible to determine from these notations and the compound mapping *a* how the elements in the metamodel shall be mapped from one language to the other. Again we can apply local reasoning algorithms operating on the model expressed in AS to find out what elements possibly go together. The UML AD example model given in Figure 7.7(b1) is also illustrated in AS (see Figure 7.7b2) , modeled in UML object diagram concrete syntax. We can now reason on this representation of the model and try to find out how the single metamodel elements have to be mapped to the elements in EPC. For example we learn from this graph that *Fork* and *Join Nodes* have a single *outgoing* and *incoming* edge, respectively. The heuristic is similar to the one that copes with mappings *d* and *f* in the first example, cf. Figure 7.6.

Chapter 8

Open Issues and Future Work

In this thesis we have described our MTBE approach, including basic as well as advanced concepts. Additionally, MTBE has been implemented within the Eclipse platform. However, our evaluation by means of two case studies in two different modeling domains has revealed open issues we need to tackle in future work. In the following we emphasize major directions of future work.

1) Extension of the mapping language: The mapping language developed so far is able to solve the basic transformation problems in the domains of structural and behavioral modeling. Usually each specific modeling domain raises a set of additional transformation problems. This requires a customization of the mapping language by further mapping operators. For example, in the domains considered we recognized the need for conditional equivalence mapping, and nested mapping. Therefore, we have to look at further domains to derive an extensive requirements catalog for a more powerful MTBE mapping language.

2) Web modeling languages: So far we have only studied the domains of structural and behavioral modeling languages. However, we believe applying MTBE to the domain of web modeling languages would provide further interesting results for the MTBE approach. Therefore, we apply our current MTBE approach to hypertext models, such as WebML [16], which represent the navigation and the data flow between hypertext nodes via links and link parameters. We hope that the area of web modeling languages reveals new mapping problems and allows the evaluation of MTBE in more detail.

3) Development of reference models in specific modeling domains: Having also participated in the ModelCVS project we recognized the power of building reference models. In ModelCVS, the reference models support the integration task by delivering solutions for transformation problems of a specific domain. These reference models may serve as domain models in our by-example approach. This means the user is able to specify the mappings between two reference models in different modeling languages. Hence, she does not need conceive domain models from scratch. Instead the reference models are used as starting point for the most prominent modeling languages and modeling domains. Reference models are designed to designed all or at least most of the concepts of a certain domain.

4) Proving MTBE: Usually, models evolve in the course of time. For example, a model

A at time t0 progresses to A′ at time t1. A transformation between two models is always specified in a given point in time. For example, the transformation T specifies the mapping between A and B at time t0. In future work we experiment on the consequences on the existing transformation T when models progress to new versions A′ and B′. We classify different characteristics in the delta between A and A′ (as well as B and B′) to derive further requirements for the automatic model transformation code generation.

5) Model Heterogeneities: In this thesis we have mainly focused on heterogeneities that have their origin in the metamodels of different modeling languages. We have given a rather comprehensive categorization of metamodel heterogeneities, which has been underpinned with real world examples. However, our case study of business process modeling languages has revealed challenges in the area of model heterogeneities. These model heterogeneities appear in cas of underspecified metamodels or of hidden concepts in the metamodels. For example the implicit merge of *OpaqueActions* in UML activity diagrams is not specified by means of metamodeling concepts. To handle such model heterogeneities special reasoning algorithms and patterns as well as proper example models have to be provided. The topic of model heterogeneities is therefore closely related to the requirements for example models, see Section 3.7. Also the automatic generation of valid models from metamodels would be helpful for finding heterogeneities on the instance level.

6) Metrics Implementation: We have demonstrated on a conceptual level how model metrics can support the MTBE approach by helping to measure the explicitness of metamodels. Thereby, the metric assists in finding metamodel heterogeneities. Our metamodel metric however has not been integrated yet within our MTBE framework. It is beneficial, though, to have a measure at hand supporting the software engineer to make complex notation related issues visible. By using GMF, we build on a framework that features the same notation concepts as the ones we incorporate in measuring hidden concepts in metamodels. The graphical model of GMF holds all CS elements, the GMF mapping model denotes the concepts, i.e., the notation, and the Ecore-model captures all AS elements. A special view for visualizing hidden concepts and guiding the user should be developed in order to better cope with metamodel heterogeneities.

Bibliography

[1] 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 2004). Satellite workshop of ICGT 2004, Rome, Italy*, 2004.

[2] Freddy Allilaire and Tarik Idrissi. ADT: Eclipse Development Tools for ATL. In *Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2). Canterbury, UK*, 2004.

[3] Robert St. Amant, Henry Lieberman, Richard Potter, and Luke Zettlemoyer. Programming By Example: Visual Generalization in Programming By Example. *Communications of the ACM*, 43(3):107–114, 2000.

[4] ATLAS Group. *ATL (ATLAS Transformation Language)*. http://www.eclipse.org/m2m/atl/, Last Visit: May 2008.

[5] ATLAS Group, INRIA, and LINA. *ATL User Manual*. http://www.eclipse.org/m2m/atl/doc/, Last Visit: May 2008.

[6] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and Ontology Matching with COMA++. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 906–908, New York, NY, USA, 2005.

[7] Thomas Baar. Correctly Defined Concrete Syntax for Visual Modeling Languages. In *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, Springer LNCS 4199, pages 111–125, Genova, Italy, 2006.

[8] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling*, 4(2):171–188, 2005.

[9] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, Springer LNCS 4199, pages 440–453, Genova, Italy, October 2006.

[10] Alexander Borgida and John Mylopoulos. Data Semantics Revisited. In *Proceedings of the Second International Workshop on Semantic Web and Databases (SWDB 2004)*, pages 9–26, Toronto, Canada, 2004.

[11] Peter Braun and Frank Marschall. Transforming Object Oriented Models with BOTL. *Electronic Notes in Theoretical Computer Science*, 72(3), 2003.

[12] Frank Budinsky, David Steinberg, Ed Merks Raymond, Ellersick Timothy, and J. Grose. *Eclipse Modeling Framework*. Addison Wesley, August 2003.

[13] Jean Bézivin. In search of a Basic Principle for Model Driven Engineering. *UPGRADE*, 5(2):21–24, 2004.

[14] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In *Proceedings of the OOPSLA Workshop on Generative Techniques in the context of MDA*, Anaheim, California, 2003.

[15] S. Ceri, P. Fraternalia, A. Bongio, M. Bramilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan-Kaufmann, 2003.

[16] Stefano Ceri, Piero Fraternali, and Maristella Matera. Conceptual Modeling of Data-Intensive Web Applications. *IEEE Internet Computing*, 6(4):20–30, 2002.

[17] Tony Clark, Andy Evans, Stuart Kent, and Paul Sammut. The MMF Approach to Engineering Object-Oriented Design Languages. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA2001)*, 2001.

[18] Valerie Cross and Anindita Pal. Metrics for Ontologies. In *Proceedings of the Annual Meeting of the North American Fuzzy Information Processing Society*, pages 448–453, June 2005.

[19] Bill Curtis, Marc I. Kellner, and Jim Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, 1992.

[20] Allen Cypher. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Massachusetts, USA, 1993.

[21] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[22] Eclipse Foundation. *Eclipse Modeling Framework (EMF) - Ecore*. http://www.eclipse.org/modeling/emf/javadoc/, Last Visit: May 2008.

[23] Eclipse Foundation. *Graphical Modeling Framework (GMF)*. http://www.eclipse.org/modeling/gmf/, Last Visit: May 2008.

[24] Jonathan Edwards. Example Centric Programming. *SIGPLAN Not.*, 39(12):84–91, 2004.

[25] Hartmut Ehring, Gregor Engels, Hans-Jörg Kreowsky, and Grzegorz Rozenberg. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.

[26] 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 (IDM05)*, 2005.

[27] Marcos Didonet Del Fabro and Patrick Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, pages 963–970, Seoul, Korea, 2007.

[28] Richard Felder and Linda Silverman. Learning and Teaching Styles in Engineering Education. *Electronic Notes in Theoretical Computer Science*, 78(7):674–681, February 1988.

[29] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying Input Test Data for Model Transformations. *Software and Systems Modeling*, 2007.

[30] Frédéric Fondement. *Concrete Syntax Definition For Modeling Languages*. PhD thesis, EPFL, Lausanne, France, 2007.

[31] Frédéric Fondement and Thomas Baar. Making Metamodels Aware of Concrete Syntax. In *European Conference on Model Driven Architecture (ECMDA)*, Springer LNCS 3748, pages 190 – 204, Nuremberg, Germany, 2005.

[32] Aldo Gangemi, Carola Catenacci, Massimiliano Ciaramita, and Jos Lehmann. A Theoretical Framework for Ontology Evaluation and Validation. In *Proceedings of the 2nd Italian Semantic Web Workshop on Semantic Web Applications and Perspectives (SWAP)*, Trento, Italy, 2005.

[33] Jan Hendrik Hausmann and Stuart Kent. Visualizing Model Mappings in UML. In *Proceedings of the ACM 2003 Symposium on Software Visualization*, pages 169–178, San Diego, California, USA, 2003.

[34] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, February 2006.

[35] Stephan Herrmann and Marco Mosconi. Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity. *Journal of Object Technology*, 6(9):105–125, October 2007.

[36] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[37] IBM. Model Transformation Framework. http://www.alphaworks.ibm.com/tech/mtf, Last Visit: May 2008.

[38] Yanbing Jiang, Weizhong Shao, Lu Zhang, Zhiyi Ma, Xiangwen Meng, and Haohai Ma. On the Classification of UML's Meta Model Extension Mechanism. In *Proceedings of the 7th International Conference on the Unified Modelling Language: Modelling Languages and Applications (UML 2004)*, Springer LNCS 3273, pages 54–68, Lisbon, Portugal, 2004.

[39] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track*, Dijon, Bourgogne, France, 2006.

[40] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium*, pages 128–138, Montego Bay, Jamaica, 2005.

[41] 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 *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, Springer LNCS 4199, pages 528–542, Genova, Italy, 2006.

[42] 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*, GI LNI 82, pages 11–27, Innsbruck, Tirol, Austria, March 2006.

[43] Gerhard Keller, Markus Nüttgens, and August-Wilhelm Scheer. Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)". Technical Report Heft 89, Institut für Wirtschaftsinformatik Universität Saarbrücken, January 1992.

[44] David Kolb. *Experiential Learning: Experience as the Source of Learning and Development*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[45] Jochen Küster. Definition and Validation of Model Transformations. *Software and Systems Modeling*, 5(3):233–259, September 2006.

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

[47] Stephan Lechner and Michael Schrefl. Defining Web Schema Transformers by Example. In *Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA 2003)*, pages 46–56, Prague, Czech Republic, 2003.

[48] Ulf Leser and Felix Naumann. *Informationsintegration*. Dpunkt Verlag, Heidelberg, Deutschland, 2007.

[49] Haohai Ma, Weizhong Shao, Lu Zhang, Zhiyi Ma, 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)*, Springer LNCS 3273, pages 12–26, Lisbon, Portugal, October 2004.

[50] Esperanza Manso, Marcela Genero, and Mario Piattini. No-redundant Metrics for UML Class Diagram Structural Complexity. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, Springer LNCS 2681, pages 127–142, Klagenfurt, Austria, June 2003.

[51] Coral Calero Marcela Genero, Mario Piattini. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4(9):59–92, November-December 2005.

[52] Jan Mendling and Markus Nüttgens. EPC Modelling based on Implicit Arc Types. In *Proceedings of the 2nd International Conference on Information Systems Technology and its Applications (ISTA 2003)*, GI LNI 30, pages 131–142, Kharkiv, Ukraine, 2003.

[53] Tom Mens. On the Use of Graph Transformations for Model Refactoring. In *Proceedings of Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, Springer LNCS 4143, pages 67–96, Braga, Portugal, July 2005.

[54] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[55] Tom Mens and Michele Lanza. A Graph-Based Metamodel for Object-Oriented Software Metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[56] Dragan Milicev. On the Semantics of Associations and Association Ends in UML. *IEEE Transactions on Software Engineering*, 33(4):238–251, 2007.

[57] Abraham Müller and Gerald Müller. Model Transformation By-Example: An Eclipse based Framework. Master's thesis, Vienna University of Technology, Austria, 2008 (forthcoming).

[58] Marco Mosconi. Durchgängige Modularität in der modellgetriebenen Entwicklung domänenspezifischer Modellierungssprachen mit Hilfe aspektorientierter Programmierung. In *Proceedings of Modellierung 2008*, Berlin, Deutschland, March 2008.

[59] Marion Murzek and Gerhard Kramler. Business Process Model Transformation Issues. In *Proceedings of the 9th International Conference on Enterprise Information Systems*, Madeira, Portugal, 2007.

[60] Isabel Briggs Myers. *Manual: The Myers-Briggs Type Indicator*. Consulting Psychologists Press, Palo Alto, CA, 1962.

[61] Jörg Niere and Albert Zündorf. Testing and Simulating Production Control Systems Using the Fujaba Environment. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, pages 449–456, Kerkrade, The Netherlands, 1999.

[62] Jörg Niere and Albert Zündorf. Using FUJABA for the Development of Production Control Systems. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, Springer LNCS 1779, pages 181–191, Kerkrade, The Netherlands, 1999.

[63] Jürg Nievergelt and André Behr. Die Aussagekraft von Beispielen. *Informatik Spektrum*, 29(4):281–286, 2006.

[64] Object Management Group (OMG). *MDA Guide Version 1.0.1* . http://www.omg.org/docs/omg/03-06-01.pd, Last Visit: May 2008, June 2003.

[65] Object Management Group (OMG). *UML 2.0 Infrastructure Specification*. http://www.omg.org/docs/ptc/03-09-15.pdf, Last Visit: May 2008, September 2003.

[66] Object Management Group (OMG). *Unified Modeling Language Specification Version 1.4.2*. http://www.omg.org/cgi-bin/doc?formal/04-07-02, Last Visit: May 2008, July 2004.

[67] Object Management Group (OMG). *UML Superstructure Specification 2.0*. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf, Last Visit: May 2008, August 2005.

[68] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Core Specification*. http://www.omg.org/docs/formal/06-01-01.pdf, Last Visit: May 2008, January 2006.

[69] Object Management Group (OMG). *UML Superstructure Specification 2.1*. http://www.omg.org/docs/ptc/06-04-02.pdf, Last Visit: May 2008, April 2006.

[70] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. http://www.omg.org/docs/ptc/07-07-07.pdf, final adopted specification 1.1 edition, Last Visit: May 2008, July 2007.

[71] Kouichi Ono, Teruo Koyanagi, Mari Abe, and Masahiro Hori. XSLT Stylesheet Generation by Example with WYSIWYG Editing. In *Proceedings of the 2002 Symposium on Applications and the Internet (SAINT 2002)*, pages 150–161, Washington, DC, USA, 2002.

[72] Alexander Repenning and Corrina Perrone. Programming By Example: Programming by Analogous Examples. *Communications of the ACM*, 43(3):90–97, 2000.

[73] Daniel A. Sadilek and Stephan Weißleder. Towards Automated Testing of Abstract Syntax Specifications of Domain-Specific Modeling Languages. In *Proceedings of the Workshop on Domain-Specific Modeling Languages (DSML-2008)*, Berlin, Germany, March 2008.

[74] 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 2006)*, Palo Alto, California, July 2006.

[75] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[76] Andy Schürr and Alexander Königs. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, 2006.

[77] David Canfield Smith. Building Personal Tools by Programming. *Communications of the ACM*, 43(8):92–95, 2000.

[78] David Canfield Smith, Allen Cypher, and Jim Spohrer. KidSim: Programming Agents without a Programming Language. *Communications of the ACM*, 37(7):54–67, 1994.

[79] David Canfield Smith, Allen Cypher, and Larry Tesler. Programming by Example: Novice Programming comes of Age. *Communications of the ACM*, 43(3):75–81, 2000.

[80] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung*. Dpunkt Verlag, March 2005.

[81] Michael Strommer, Marion Murzek, and Manuel Wimmer. Applying Model Transformation By-Example on Business Process Modeling Languages. In *Proceedings of Advances in Conceptual Modeling - Foundations and Applications, ER 2007 Workshops CMLSA, FP-UML, ONISW, QoIS, RIGiM,SeCoGIS*, pages 116–125, Auckland, New Zealand, November 2007.

[82] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, Springer LNCS 1779, pages 481–488, Kerkrade, The Netherlands, 1999.

[83] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, pages 446–453, Charlottesville, VA, USA, October 2003.

[84] Dániel Varró. Model Transformation By Example. In *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, Springer LNCS 4199, Genova, Italy, October 2006.

[85] 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: Modeling Languages and Applications (UML 2004)*, Lisbon, Portugal, October 2004.

[86] Dániel Varró, Gergely Varró, and András Pataricza. Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[87] Chris Welty, R. Kalra, and Jennifer Chu-Carroll. Evaluating Ontological Analysis. In *Proceedings of the ISWC-03 Workshop on Semantic Integration*, Sanibel Island, Florida, October 2003.

[88] Manuel Wimmer and Gerhard Kramler. Bridging Grammarware and Modelware. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, Springer LNCS 3844, pages 159–168, Montego Bay, Jamaica, 2005.

[89] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By-Example. In *Proceedings of the 40th Hawaii International International Conference on Systems Science (HICSS-40 2007)*, Big Island, HI, USA, 2007.

[90] Patrick Winston. Learning Structure Descriptions from Examples. *The Psychology of Computer Vision*, pages 157 – 209, 1970.

[91] Tong Yi, Fangjun Wu, and Chengzhi Gan. A Comparison of Metrics for UML Class Diagrams. *SIGSOFT Software Engineering Notes*, 29(5):1–6, 2004.

[92] Haining Yoa, Anthony Mark Orem, and Letha Etzkorn. Cohesion Metrics for Ontology Design and Application. *Journal of Computer Science*, 1(1):107–113, 2005.

[93] Moshé M. Zloof. Query By Example. In *Proceedings of National Compute Conference*, pages 431–438. AFIPS Press, 1975.

Curriculum Vitae

Michael Strommer
Schliemanngasse 9/6
1210 Wien
Date of Birth: 27.01.1980
Nationality: Austria

Contact Information

Web: http://www.big.tuwien.ac.at/staff/mstrommer.html
E-Mail: strommer@big.tuwien.ac.at

Education

PhD studies in "Business Informatics" (Wirtschaftsinformatik) **Vienna University of Technology, Austria** Advisors: Gerti Kappel, Christian Huemer Degree: Dr. rer. soc. oec.	10/06–06/08
Degree program, MSc in "Business Informatics" (Wirtschaftsinformatik) **University of Vienna and Vienna University of Technology, Austria** Thesis: Ökonomische Aspekte der Entwicklungsproblematik "Dritter Welt Länder" Advisor: Bernhard Böhm Degree: Mag. rer. soc. oec.	10/00–10/05
Vienna Business School	09/94–06/99
Bundesrealgymnasium, Franklinstraße 21	09/90–06/94
Volksschule, Prießnitzgasse 1	09/86–06/90

Teaching Experience

Teaching assistant:

- Courses on Model Engineering (Winter Term 2006, Summer Term 2007)
- Course on Introduction of the Semantic Web (Winter Term 2006)

Lecturer at University of Applied Science (FH Campus Wien):

- Course on Programming in C/C++ (Winter Term 2007)
- Course on Algorithms and Data Structures (Summer Term 2008)

Tutor (2002-2005):

- Courses on Programming Basics (Java)
- Courses on Web Engineering

Additional Competence

- Adviser and Referent at the HochschülerInnenschaft an der TU Wien (HTU) (2002 - 2006)

Publications

1. M. Strommer, M. Wimmer: A Framework for Model Transformation By-Example: Concepts and Tool Support. 46th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'08), Zurich, Switzerland, July 2008.

2. G. Kappel, H. Kargl, T. Reiter,W. Retschitzegger,W. Schwinger, M. Strommer, M.Wimmer: A Framework for Building Mapping Operators Resolving Structural Heterogeneities. 7th International Conference on Information Systems Technology and its Applications (ISTA'08), Klagenfurt, Austria, April 2008.

3. M.Wimmer, A. Schauerhuber, M. Strommer, J. Flandorfer, G. Kappel: HowWeb 2.0 can leverage Model Engineering in Practice. Workshop on Domänenspezifische Modellierungssprachen (DSML'08), in conjunction with Modellierung 2008, Berlin, Deutschland, March 2008.

4. M. Wimmer, A. Schauerhuber, M. Strommer, W. Schwinger, G. Kappel: A Semi-automatic Approach for bridging DSLs with UML. 7th OOPSLA Workshop on Domain-Specific Modeling, in conjunction with OOPSLA'07, Montreal, Canada, October 2007.

5. M. Wimmer, H. Kargl, M. Seidl, M. Strommer, T. Reiter: Integrating Ontologies with CAR-Mappings, First International Workshop on Semantic Technology Adoption in Business (STAB'07), Vienna, Austria, 2007.

6. M. Strommer, M. Murzek, M. Wimmer: Applying Model Transformation By-Example on Business Process Modeling Languages, 3rd International Workshop on Foundations and Practices of UML (ER 2007), Auckland, New Zealand, November 2007.

7. G. Kappel, H. Kargl , G. Kramler , A. Schauerhuber , M. Seidl, M. Strommer, and M. Wimmer. Matching Metamodels with Semantic Systems - An Experience Report. 12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web, Aachen, March 2007.

8. H. Kargl, M. Strommer, M. Wimmer. Measuring the Explicitness of Modeling Concepts in Metamodels. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006), Workshop on Model Size Metrics, Genova, Italy, October 2006.

9. M. Wimmer, M. Strommer, H. Kargl, G. Kramler. Model Transformation Generation By-Example. HICSS-40 Hawaii International Conference on System Sciences, Hawaii, USA, January 2007.

10. M. Strommer: Ökonomische Aspekte der Entwicklungsproblematik "Dritter Welt" Länder, Diploma Thesis, Vienna University of Technology, October 2005.