TU Wien

Business Informatics Group
Institut für Softwaretechnik und Interaktive Systeme

Philipp Huber

The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches

Master Thesis

eingereicht bei o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
mitbetreuender Assistent: Univ.-Ass. Mag. Manuel Wimmer

Wien, May 14, 2008

Abstract

Model Transformations are a key prerequisite for Model Driven Engineering (MDE) and therefore represent an active research area. Various model transformation languages are available, whereas the languages can be categorized into different approaches. Depending on the particular situation, one model transformation approach might be better suited to accomplish the given task than another approach. Classifications of model transformation languages exist, which also include a taxonomy of general features a model transformation language may support. Based on this taxonomy, it is possible to compare model transformation approaches in order to find out how suitable they are for a given problem. Like in common object oriented-programming approaches, such as Java, there are some particular problems which appear repeatedly. For example, it is often necessary to transform an attribute value in the source model to an object in the target model. For such cases, it should be considered to solve these problems in a generic way for improving the reuse of model transformation definitions.

In this thesis, four model transformation languages, namely Atlas Transformation Language (ATL), SmartQVT, Kermeta, and ModelMorf are evaluated based on the taxonomy proposed by Czarnecki et al. [5]. These languages are chosen, because they represent the state of the art approaches for transforming models in the field of MDE. ATL is a hybrid language mixing declarative and imperative constructs. Kermeta and SmartQVT act as examples for an imperative language. More specifically, SmartQVT implements the operational part of the Object Management Group Query/View/Transformation (OMG QVT) standard. ModelMorf implements the Relational QVT language and therefore represents a purely declarative approach.

For the purpose of conducting the evaluation, several model transformation examples are defined and implemented using the aforementioned languages. These examples cover model transformation problems appearing in practice, in order to emphasize the advantages and limitations of the particular language. In the second part of this thesis, Kermeta is used to implement a library which solves common problems appearing in model transformations. This library can then be used in practical transformations for simplifying the development process. Finally, the results of the evaluation are discussed, in order to propose guidelines on which model transformation approaches are suitable for which problems.

Kurzfassung

Modelltransformationen sind die Grundlage für Model Driven Engineering (MDE) und stellen daher ein aktives Forschungsfeld dar. Es gibt verschiedenste Modelltransformationssprachen, welche in verschiedene Kategorien eingeteilt werden können. Abhängig von der jeweiligen Situation, kann ein bestimmter Modelltransformationsansatz geeigneter als ein Anderer sein, um eine bestimmte Aufgabe zu lösen. Klassifikationen von Modelltransformationssprachen existieren, darin beinhaltet ist auch eine Sammlung von allgemeinen Features, welche von Modelltransformationssprachen unterstützt werden können. Basierend auf dieser Klassifikation ist es möglich, verschiedene Modelltransformationsansätze zu vergleichen, um herauszufinden, wie geeignet sie für jeweils ein bestimmtes Transformationsproblem sind. Erste Untersuchungen zeigen, dass wie in üblichen objektorientierten Programmiersprachen, wie etwa Java, es bestimmte Problemstellungen gibt, die wiederholt auftreten. Zum Beispiel ist es oft notwendig, einen Attributwert im Quellmodell zu einem Objekt im Zielmodell zu transformieren. Für solche Fälle sollte man in Erwägung ziehen, diese Probleme generisch zu lösen, um die Wiederverwendbarkeit von Modelltransformationen zu verbessern.

Basierend auf der von Czarnecki et al. vorgeschlagenen Klassifikation [5] werden in dieser Arbeit vier Modelltransformationssprachen, nämlich Atlas Transformation Language (ATL), SmartQVT, Kermeta und ModelMorf evaluiert. Diese Sprachen wurden ausgewählt, da sie den momentanen Stand der Technik für Modelltransformationen im Bereich von MDE repräsentieren. ATL ist eine hybride Sprache und mischt sowohl deklarative als auch imperative Sprachkonstrukte. Kermeta und SmartQVT dienen als Beispiele für imperative Sprachen. Weiters implementiert SmartQVT den Operational Part des Object Management Group Query/View/Transformation (OMG QVT) Standards. ModelMorf implementiert den Relational QVT Standard und repräsentiert somit einen rein deklarativen Ansatz.

Für die Durchführung der Evaluierung, werden mehrere Beispieltransformationen definiert und mit den zuvor erwähnten Sprachen implementiert. Diese Beispiele decken Transformationsprobleme ab, die in der Praxis auftreten, um die Vorteile und Beschränkungen der einzelnen Sprachen aufzuzeigen. Im zweiten Teil der Arbeit wird mit Kermeta eine Bibliothek implementiert, die wiederkehrende Problemstellungen, die bei Modelltransformationen auftreten, löst. Diese Bibliothek kann dann in der Praxis genutzt werden, um den Entwicklungsprozess von Transformationen zu vereinfachen. Zuletzt werden die Ergebnisse der Evaluierung diskutiert, um Richtlinien vorzuschlagen, welche Modelltransformationsansätze für welche Probleme geeignet sind.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, 09.05.2008 Philipp Huber

Acknowledgements

Grateful acknowledgement is made to the following:

- Christa and Rudolf Huber, my parents, who constantly supported me until the end and also sponsored my education which led to the completion of this thesis.

- Güzide Selin Altan who provided me utmost moral support when ultimately needed.

- Last but not least, Gerti Kappel and Manuel Wimmer who supervised and guided me along the preparation process of this thesis.

Contents

1 Introduction

1.1 Motivation and Goal of this Thesis

The need for model transformation has risen considerably in the last years. Model transformation is a key technology in the area of Model Driven Engineering [3], in which models take the part of the central artifact generated in a software engineering process. Model transformation however can also be used in other fields than software engineering. A notable example for this is the transformation of business process models [1].

Model Driven Architecture (MDA) [19], as OMG's approach to Model Driven Engineering, also includes model transformation features. For this purpose, OMG issued a Query/View/Transformation (QVT) Request For Proposal, for which 8 different model transformation languages were developed and presented [15].

As can be seen by the responses to OMG's QVT Request For Proposal, the interest in model transformation is very high. Apart from that, many model transformation approaches have been developed independently. Because of the sheer quantity of different model transformation languages, it is difficult to get an overview. Several different main approaches to the problem are available, such as imperative, declarative, and graphical model transformation languages. These languages differ considerably in their features and overall usage.

Because of these factors, it is not easy to choose a model transformation language for a particular situation. For the efficient and cost-effective solution of a transformation problem, it is crucial to know the requirements for the transformation language. Once the requirements are clear, it is still difficult to choose a certain language, simply because so many transformation languages are available. In order to make a decision, several candidate languages have to be evaluated against the requirements needed.

In order to gain a better understanding of what is possible with model transformation languages and which features they offer, Czarnecki et al. proposed a taxonomy of model transformation language features [5]. Based on this taxonomy, four different model transformation languages are evaluated in this thesis. Their feature support is assessed and discussed. The goal of this thesis is to offer guidance in choosing a model transformation language for a certain transformation project.

1.2 Related Work

This thesis is set in the field of Model Driven Engineering with its most prominent protagonist Model Driven Architecure, with the focus set on model transformation. Several other papers and theses are also set in the field of model transformation languages.

- In [1], Altan discusses one particular model transformation approach, namely Triple Graph Grammars. The focus lies on the transformation of business process models.

- In [31], Schäfer compares several different transformation languages for their suitability for model based user interface design.

- In [39], Wimmer et al. discuss CAR mappings, including often occurring model transformation patterns. The transformation patterns discussed in Section 10 are based on the patterns introduced in this paper.

Although not set in the field of model transformation, de Jonge et al. [6] present a framework for program transformation. In this paper, *XT* is discussed, which combines several existing program transformation libraries into a reusable framework.

1.3 Structure of this Thesis

The subsequent chapters are structured as followed:

- In Chapter 2, an introduction to model transformation in general is given. The role of model transformation as part of MDA is explained. Further different applications and model transformation approaches are discussed.

- Chapter 3 contains the criteria catalogue, which is later used to evaluate the model transformation languages in this thesis. All features evaluated are explained, and possible values are listed.

- Chapter 4 explains the sample transformations which are later implemented using different model transformation languages. The transformation *ooclass2table* is an example for a simple transformation, whereas *epc2ad* is more sophisticated.

- Chapters 5 to 8 discuss the four model transformation languages evaluated in this thesis, namely ATL, SmartQVT, Kermeta and ModelMorf. First, an introduction to each language is given, followed by an explanation of the language's core features and user interface. After that, the implementation of the two sample transformations described in Chapter 4 is presented, followed by a overall summary on the respective language.

- Chapter 9 contains the evaluation results, based on the features of the criteria catalogue in Chapter 3. For every group of features, a table lists the feature support of all languages evaluated. Additionally, the results are discussed at feature group level. Finally, the lessons learned are discussed.

- Chapter 10 documents KLRT, the Kermeta Library for Reusable Transformations. An introduction and motivation on recurring model transformation patterns is given. The patterns implemented by KLRT are explained, followed by a description of the implementation of KLRT. Finally, a sample transformation is implemented using KLRT and plain Kermeta, and the differences are discussed.

- Chapter 11 provides a summary of this thesis and gives an outlook on further work.

2 Introduction to Model Transformation

2.1 Model Transformation as Part of MDA

MDA is an approach for using models in software development [28], thereby the model is seen as the primary artifact of the software development process. Model transformation is a key part of MDA, because it used to derive one model from another, and for closing the gap between problem-oriented description and its implementation.

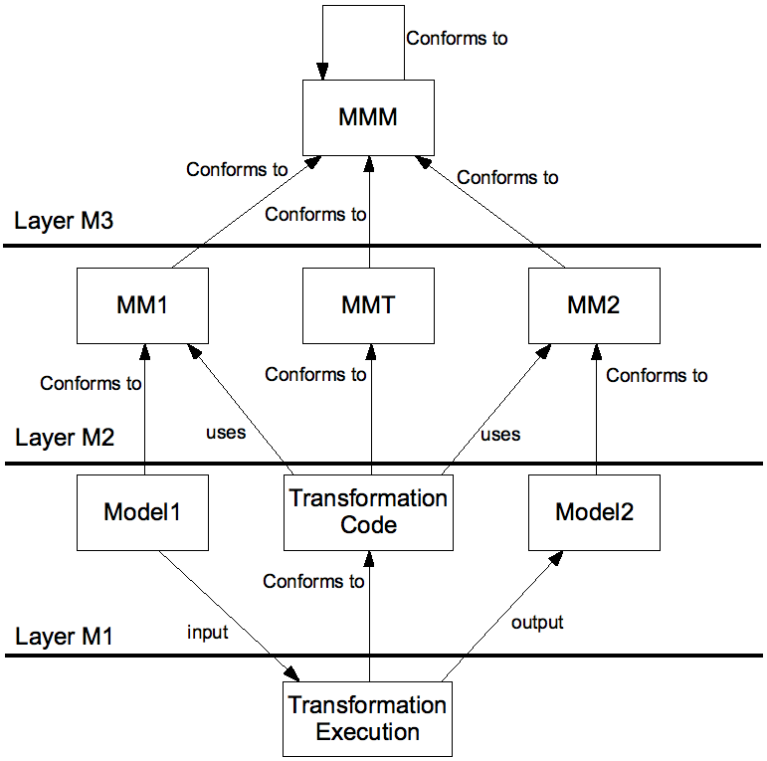

Figure 2.1: Model Transformation Pattern based on [25]

The basic idea and coherence of MDA and model transformation can be seen in Figure 2.1. The most basic entity is the meta-metamodel, as can be seen at the top of the illustration. This layer of the hierarchy is also called M3-layer. The meta-metamodel only contains basic elements, which are sufficient for specifying metamodels. In MDA, which is the model driven engineering approach by OMG, *MOF* [16] is used as meta-metamodel. The two packages, which together build

MOF, namely *Essential MOF* (EMOF) and *Complete MOF* (CMOF) are also self-specified, which means the EMOF is described using EMOF, and CMOF is described using CMOF [16].

Using a meta-metamodel like MOF, metamodels can be specified. Metamodels reside at the M2-layer of the MDA metamodeling stack (MM1 and MM2 in Figure 2.1) and are valid instances of M3-layer meta-metamodels. Metamodels are usually used as basis for creating models, which are meant to be representations of real-world entities. A widely known and used example for a metamodel is the UML metamodel, which consists of the *UML Infrastructure* [29] and *UML Superstructure* [30]. Besides these metamodels, another metamodel resides on layer M2, which is specific for model transformation, namely MMT. MMT denotes the metamodel of the model transformation language used. Although most transformation languages use textual syntax rather than MOF-based models, many model transformation languages still are specified using a meta-metamodel like MOF, like it is the case with QVT [17] and ATL [20]. In such cases, the textual representation is interpreted as concrete syntax.

On the layer M1, the models themselves, in this case Model1 and Model2, can be found. These models directly represent real-world (M0) entities. In practice this may be UML models, business process models etc. Models at the M1 layer are able to use constructs defined in the metamodels at level M2. From a model transformation point of view, a specific transformation also resides at this layer, because it is an instance of a model transformation language metamodel, and therefore uses the features provided by the metamodel. A model transformation uses models at the same layer as input and output models. For example, a model transformation written in QVT is conform to the QVT metamodel at layer M2 and operates on models at layer M1, which themselves again are conform to other metamodels at layer M2 (e.g. UML metamodel and relational database metamodel).

2.2 Applications for Model Transformation

In the following, different usage scenarios for model transformation are explained.

2.2.1 Horizontal Model Transformation

A horizontal model transformation is a transformation, where the source model and the target model belong to the same abstraction level [27]. An example for this particular type of transformation is refactoring, because the target model marks a change in internal structure compared to the source model, but without altering the abstraction level of the model.

2.2.2 Vertical Model Transformation

The opposite of horizontal model transformation is vertical model transformation. In this case, different levels of abstraction are used in the source and target

model [27]. It is also called refinement, because additional information is added, resulting in decreasing the abstraction level of the target model. On the other hand, information could be removed from the source model to create a more abstract target model.

The transformation of platform independent model to platform specific model [28] is a typical case of a vertical transformation, because both input and output model are on different levels of abstraction. Platform specific information is added to the target model (denoted by the blank rectangle in Figure 2.2) during the transformation, thus lowering the abstraction level. The platform specific information may also be called *platform description model*, or *PDM* abbreviated [14], especially if the platform specific information is placed in a separate model. In this case, a model transformation language which supports multiple source models and target models is mandatory. On the other hand, it is also possible that the platform specific information is contained within the transformation itself, and no additional input model besides the platform independent model is read.

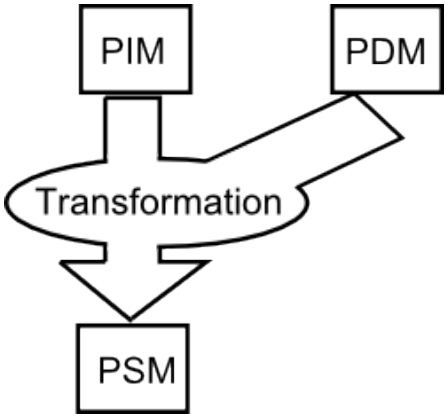

Figure 2.2: Transformation from PIM to PSM [28]

Code expressed in a specific programming language can also be interpreted as a model conforming to a metamodel(see [21]). From this perspective, code generation from models in MDA, also known as *Model2Code*, can also be seen as vertical model transformation, because abstract information contained in the source model is transformed to a more concrete model, namely the generated code.

2.2.3 Endogenous Model Transformation

Transformations which affect models expressed in the same language are called endogenous. Both source and target models conform to the same metamodel [27]. Examples for endogenous transformations include optimization or refactoring.

2.2.4 Exogenous Model Transformation

Contrary to endogenous transformations, exogenous transformations are expressed between models conforming to different metamodels. Exogenous transformations can also be called translations, because it is focused on the mapping of a model in different metamodels.

An important example for exogenous transformations is code generation, or its opposite, reverse engineering. During code generation, a model conforming to a specific metamodel is translated to its equivalent expressed in another metamodel. For example, a UML Class Diagram can be translated to Java code. The translation of Java code into UML Class Diagram is an example for reverse engineering.

2.2.5 Summary of Usage Scenarios

Horizontal/vertical transformations and endogenous/exogenous transformations to not contradict and exclude each other. In fact they can be considered as two orthogonal dimensions, as can be seen in Table 2.1. The table shows usage scenarios for all possible combinations between the two dimensions.

	Horizontal	Vertical
Endogenous	Refactoring	Formal refinement
Exogenous	Language migration	Code generation

Table 2.1: Horizontal/Vertical and Endogenous/Exogenous Transformations [27]

Refactoring

Refactoring is an instance of horizontal and endogenous transformation, because the same level of abstraction is kept in the source and the target model, and the language of both source and target models also stays the same. Refactoring is used to improve the quality of a certain model, without changing its behavior.

Formal Refinement

Formal refinement adds information to a given model. Therefore the level of abstraction lowers, because the information in the model gets more specific (vertical transformation). On the other hand, the language and metamodel of the target model remains unchanged in comparison to the source model (endogenous transformation).

Language Migration

In language migration, the source model is transformed to another language (exogenous transformation). The model semantics however are preserved, resulting in the same level abstraction as in the source model (horizontal transformation).

Code Generation

Code generation lowers the level of abstraction in the target model, because the code generated is more specific and verbose than its source model. (vertical transformation). Obviously, also a shift in the language between source and target model occurs (exogenous transformation).

2.3 Classification of Model Transformation Approaches

Numerous different model transformation languages exist nowadays. Many of them follow different approaches, but often share some similarities. Parallels to classic programming languages can be drawn, because all of them were developed for different purposes having different goals in mind. The same issue applies to model transformation languages.

For this reason, it is possible to establish a list containing major features which are useful in a model transformation setting. The following taxonomy of model transformation language features is based on [5].

Even at the highest level, several approaches to model transformation exist. The distinction among them can be compared to the different paradigms in programming languages. Because of the fundamental differences between the approaches, many lower-level features make more sense in one particular language than in another. In literature, no clear distinction has prevailed. Depending of the focus of the respective classification, other categories are choosen. Different top-level taxonomies can be found in [5], [24], or [27]. Additionally, some of the proposed approaches do not contradict each other. In some cases it is possible, that the basic idea of several approaches is similar, but the languages set the focus on different goals. For example, both graphical and textual model transformation languages can be declarative, as it is the case with Triple Graph Grammars [2] or ATL. Moreover, some approaches also incorporate techniques from others. A common example are declarative languages, which also support imperative statements. Because of the mixing of different paradigms, they are called *hybrid* approaches.

In the following, a top-level distinction of model transformation approaches is given. Only dedicated model-to-model transformation languages are taken into consideration. The categories reflect the most prominent language paradigm addressed by the respective language. The distinction is mainly based on [5].

2.3.1 Direct-Manipulation

Direct-manipulation techniques to model transformation consist of a general purpose programming language and an API which provides access to model and metamodel data. A popular example for this approach is Java in connection with Java Metadata Interface (JMI). JMI provides an MOF-based infrastructure

for the creation, storage, access, discovery, and exchange of metadata [33]. In fact, programming languages which are not primarily intended for model transformation can be used for that purpose if an API for accessing model and metamodel data is available [4].

When using a direct-manipulation approach, it is not necessary to get familiar with new development tools and a different language syntax. The complete transformation is written in the language given. Therefore all language constructs available can be used to perform the transformation. In most cases however, the language used is not primarily intended for model transformation. This results in many missing features typically used in a model transformation setting.

2.3.2 Imperative

Imperative (also called operational [5]) approaches to model transformation are very similar to the imperative paradigm in programming languages. From a model transformation point of view, the imperative approach is also similar to direct-manipulation . However, the languages in this category offer a more advanced support for model transformation features, without having to use an external API.

A well-defined control flow exists, which means that all statements in the transformation code are executed in order. Imperative model transformation techniques focus on *how* the transformation has to be executed, contrary to the *what* of declarative approaches [27].

Many constructs which exist in languages like Java are available in imperative model transformation languages, including for example statements which alter the program flow (conditions, loops), or subroutines.

Because of the similarity to conventional programming languages, imperative model transformation languages are easier to learn for experienced programmers. Many constructs and ways of thinking can be directly applied that way. Because of the widespread control the programmer has over the transformation execution, it is also possible to implement transformations in a very efficient way. Furthermore, complex transformations can be specified more easily than with other approaches. For this reason, some model transformation languages, such as ATL, provide imperative style constructs besides their main approach.

On the downside, imperative approaches suffer from more overhead code than other approaches. Many issues have to be accomplished in an explicit way. This however goes hand in hand with the flexibility and control the developer has over the transformation. Simpler parts of the transformation have to be written using more code than what is necessary for example in a declarative language, resulting in less readable code. If loading and saving of input and output models also has to be done in an explicit way, even more code has to be written. This can be observed in Chapters 6 and 7, where the code for loading and saving a

model in Kermeta needs approximately as many lines of code as nearly the whole transformation in SmartQVT.

2.3.3 Declarative

Declarative (also called relational [5]) approaches focus on *what* the transformation has to accomplish [27]. The developer specifies how the elements in the source and target model relate to each other. For this purpose, constraints, for example in OCL, can be specified as well [5]. Contrary to imperative approaches, control flow and rule order is not explicit. Therefore, procedural details of the transformation are hidden, which results in a smaller amount of code necessary. Because the transformation code is more compact, and consists of less how details, it is easier to comprehend. This applies especially for simpler transformation problems, which therefore are faster and easier to implement than in imperative model transformation languages.

Additionally, some declarative approaches support bidirectionality, which means that source and target models can be transformed in either way using the same transformation specification. This is hardly possible in imperative approaches.

On the other hand, more complex transformations are harder to specify in declarative model transformation languages, because they are not a expressive and give the developer less control as imperative languages.

Declarative model transformation languages can be further divided into several categories [27]. Functional approaches for example are oriented on functional-style programming languages. Also, graph transformation languages are often considered declarative as well.

2.3.4 Hybrid

All model transformation approaches described above have their strengths and shortcomings. Imperative approaches are very powerful, but also very verbose, and therefore harder to read and understand. Declarative languages on the other hand may not be suitable for complex transformation tasks. For this reason, some languages incorporate concepts of more than one model transformation approach. This makes it possible to use the advantages of more than one paradigm, and weaknesses can be compensated.

Despite the multi-paradigm idea, these languages often concentrate on one approach and only use other concepts to address situations, in which the main approach is not very suitable. A popular example is ATL, which primarily uses a declarative approach, and encourages the developer to use declarative style whenever possible. For transformation problems which can only be solved difficult (if at all), ATL also offers imperative code blocks.

As seen in the example in Listing 2.1, ATL uses a *do* block to include imperative code. The attribute *name* is copied to the target model using usual ATL declarative syntax. The target model element also has an attribute *cnt*, which is unique for every element in the model, and therefore is incremented every time a model element is created. This is accomplished using the *do* block, which denotes imperative code in ATL.

Listing 2.1: Mixing Declarative and Imperative Code in ATL

```
1  helper def : cnt : Integer = 0;
2
3  rule Function2Activity {
4    from
5      f : EPC!Function
6    to
7      a : ActivityDiagram!Activity (
8        name <- f.name
9    )
10
11   do {
12     thisModule.cnt <- thisModule.cnt + 1;
13     a.cnt <- thisModule.cnt;
14   }
15 }
```

2.3.5 Graph Transformation

Graph transformation based approaches interpret models from a graph theoretical point of view. Model elements and their relationships are seen as graph vertices and edges. Graph transformation rules have a left hand side, and a right hand side, which correspond to the source and target model of the transformation [5]. This concept is similar to the relations in declarative approaches, therefore graph transformation is sometimes considered a subcategory of declarative approaches, for example in [27].

Graph-based does not necessarily mean that the transformation is specified in a visual way. As described above, *graph* refers to the theoretical foundation used in these approaches. However, many graph-based model transformation languages use a visual notation for specifying the transformation.

A particular example for graph-based transformation approach are Triple Graph Grammars (TGG), introduced by Andy Schürr [32]. Triple Graph Grammar rules use an additional third subgraph, called correspondence graph, which elements are linked to the source and target elements logically connected. From a model transformation point of view, the correspondence graph holds the tracing information of the transformation.

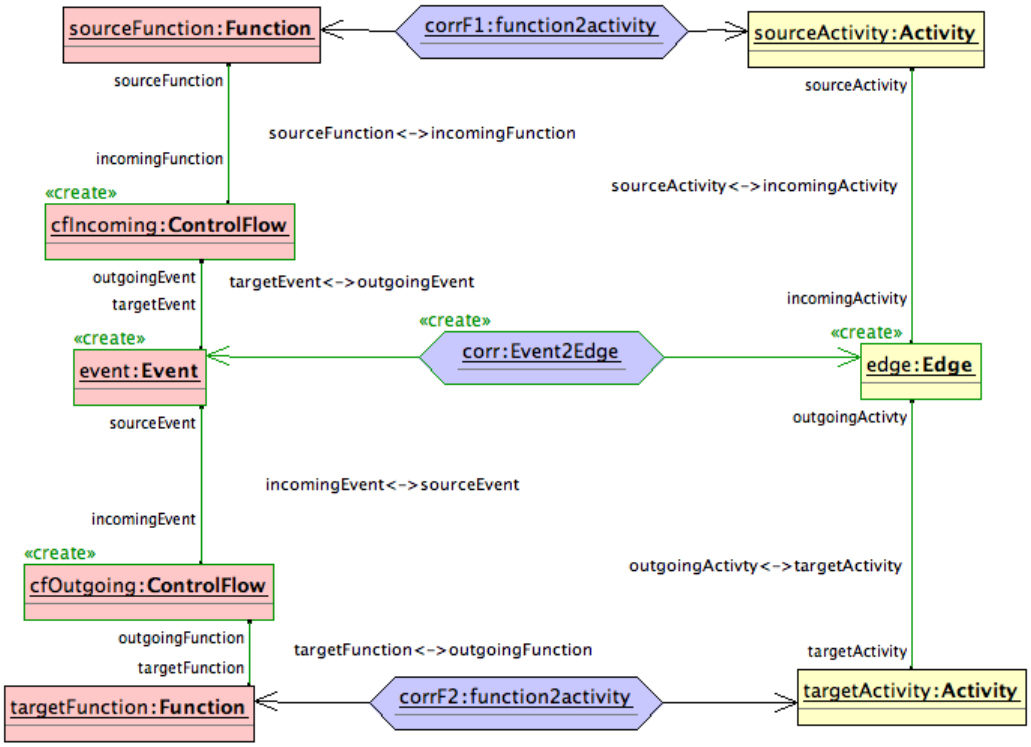

Figure 2.3: An Example Rule using Triple Graph Grammars

Figure 2.3 illustrates an example usage of graph-based transformations. This rule defines how *events* and *edges* relate to each other.

3 Criteria Catalogue

In this chapter, a criteria catalogue of model transformation language features is defined. It is based on the features presented in [5] and divided into several categories, which represent top-level model transformation features. These are then divided further.

Top level feature	Table
Transformation Rules	9.1
Rule Application Control	9.5 and 9.6
Rule Organization	9.7
Source-Target Relationship	9.8
Incrementality	9.9
Directionality	9.10
Tracing	9.11

Table 3.1: Top-level Features

3.1 Transformation Rules

Transformation rules can be understood as the smallest unit of a transformation [5]. Depending on the specific model transformation approach, a transformation rule can be a rewrite rule with a left-hand and right-hand side (declarative), but for example also a procedure (imperative). *Transformation rules* itself is not a criteria which can be met oder not, because their support in a language has to be included obviously. It is divided into several subcriteria, do describe how exactly a particular model transformation languages implements the concept *transformation rule*.

3.1.1 Syntactic Separation

A model transformation offering syntactic separation clearly separates the parts of a rule operating on different models [5]. In declarative approaches, this is the case in lefthand and righthand sides of a rule. In imperative approaches, however, the distinction between the models involved is not that clear.

Possible values: [Yes | No]

3.1.2 Multidirectionality

Multidirectionality describes the possibility to execute a rule in different directions. This criteria can hardly be fulfilled by imperative or direct-manipulation based approaches, because they describe the transformation step-by-step in an algorithmic way.

Possible values: [Yes | No]

3.1.3 Application Condition

An application condition describes conditions that can be attached to rules. The rule is only executed if the condition evaluates to *true*.

Possible values: [Yes | No]

3.1.4 Intermediate Structures

Intermediate structures are built by the model transformation language in order to accomplish the transformation. These informations are usually not part of the models transformed and can be persisted. Traceability links are an example for intermediate structures.

Possible values: [Yes | No]

3.1.5 Reflection

Reflection allows transformation rules to access the transformation themselves.

Possible values: [Yes | No]

3.1.6 Aspects

Aspects denote the support of concepts found in aspect oriented programming, for example weaving and crosscutting.

Possible values: [Yes | No]

3.1.7 Domain

Domains denote parts of a rule for accessing the models involved in the transformation. Usually, this means source and target models. In some cases, several domains are possible, for example model merging or model weaving [5].

Domain Language

Domain languages describe the structure of a domain and thus its models. For this reason, the domain language is the metamodel a certain model conforms to.

Possible values: domain languages (for example MOF, EMF)

Static Mode

This feature is interpreted differently as in [5]. No distinction between *in* and *out* is made, because usually both are available in model transformation approaches. Therefore the feature option *separate* is introduced, denoting that a given domain can only be *input* or *output*.

The static mode of a domain can have one of the following values:

- *separate*. Domains acts as either as input (source) or output domain.
- *in/out*. In multidirectional approaches, domains act both as input and output domains.

Dynamic Mode Restriction

Dynamic mode restriction enables the changing of the static modes described above. For example, a particular model transformation language could allow to restrict an *in/out* domain to only *in* at runtime.

Possible values: [Yes | No]

Typing

Typing refers to the way variables and types are classified in the respective model transformation language. It can either be *untyped*, *syntactically typed* or *semantically typed*. A model transformation language using syntactical typing has to detect at compile-time if values of incompatible data types are assigned to each other (for example a string value to an integer).

Possible values: [untyped | syntactically | semantically]

Body

The feature *Body* consist of three subcategories.

- *Variables*. Variables contain model elements of the models involved in the transformation.

  Possible values: [Yes | No]

- *Patterns*. Patterns are model fragments containing variables [5]. The *structure* of patterns can be either *String*, *Terms* or *Graphs*. The syntax can be *abstract* or *concrete*, where the last can be further divided in *textual* or *graphical*.

- *Logic*. Logic is divided in the following three sub-features:

- *Language Paradigm.* This describes the language paradigm the particular model transformation approach uses. Possible paradigms are *Object-oriented*, *Functional*, *Logic Procedural*, or *Hybrid*.
- *Value Specification.* Value specification can be either *Imperative assignment*, *value binding* or *constraint*.
- *Element Creation.* Elements in the target model can be created *implicit* or *explicit*.

3.1.8 Parameterization

The three subfeatures of parameterization denote, what kind of values can be specified as parameters.

Control Parameters

This is the simplest case, only values can be passed as parameters.

Possible values: [Yes | No]

Generics

Generics enable the passing of data types as parameters. This is very similar to generics in C++ or Java.

Possible values: [Yes | No]

Higher-order Rules

Higher-order rules allow the use of rules themselves as parameters.

Possible values: [Yes | No]

3.2 Rule Application Control

Rule application control is used to determine the rule application locations.

The rule application strategy can be either *deterministic*, *non-deterministic* or *interactive*. A deterministic rule application strategy uses for a example a traversal strategy. On the other hand, One-point non-deterministic strategy applies a rule to a non-deterministically selected location, whereas concurrent strategy applies the rule to all matching locations concurrently.

Rule scheduling defines in which order the respective rules are applied.

3.2.1 Form

The rule order can be specified in an implicit or explicit way. In an implicit rule scheduling order, the developer has no direct control over the order in which

the rules are applied. The opposite is the case with explicit rule scheduling. Explicit scheduling can be either *internal* or *external*. Using internal explicit rule scheduling, the developer can call other rules within rules. On the other hand, in external rule scheduling, the execution order of the rules can be specified separated from the rules themselves.

3.2.2 Rule Selection

Rule selection denotes how rules to be executed are selected. A particular model transformation approach can offer any of the following selection possibilities:

- Explicit Condition
- Non-deterministic
- Conflict resolution
- Interactive

3.2.3 Rule Iteration

Rule iteration can be either one of the following values:

- Recursion
- Looping
- Fixpoint Iteration

3.2.4 Phasing

Phasing is the ability of organizing the transformation into several phase, where each only certain rules can be executed.

Possible values: [Yes | No]

3.3 Rule Organization

Rule organization includes features which deal with composing and structuring of transformation rules.

3.3.1 Modularity Mechanisms

Modularity mechanisms allow to group several rules. This is similar to the packaging mechanism in UML and Java.

Possible values: [Yes | No]

3.3.2 Reuse Mechanisms

Reuse mechanisms address techniques which enable reusing of rules or modules. Some approaches for example allow rule inheritance, where one rule extends the behavior of another rule. Also, logical composition is possible.

3.3.3 Organizational Structure

Organization structure describes how rules are generally organized. For example, a source-oriented organizational structure organizes rules according to the source metamodel.

Possible values are:

- Source-oriented
- Target-oriented
- Independent

3.4 Source-Target Relationship

Source-Target relationship denotes how target model elements are created from source model elements.

3.4.1 New Target

Model transformation languages using the *new target* approach of source-target relationship always create a new target model from the source model.

3.4.2 Existing Target

Some model transformation approaches allow the manipulation of already existing target models. In this way, existing model elements can be updated by deleting and recreating them (*destructive*) or by *extension only*, where target model elements can not be deleted. In-place manipulation of existing targets element is used, when source and target model are both the same model [17].

3.5 Incrementality

3.5.1 Target-Incrementality

Target-incrementality enables the updating of the target model incorporating changes in the source model, without rebuilding the whole target model. This feature is the same as described in Section3.4.2.

3.5.2 Source-Incrementality

Source-incrementality aims to minimize how many source elements have to be rechecked upon an incremental model transformation. This feature can be compared to incremental compiling [5].

3.5.3 Preservation of User Edits in the Target

This feature denotes the ability to preserve changes made by users in the target model. In a code generation setting for example, the method code implemented by a developer is left untouched upon incremental code generation.

3.6 Directionality

The directionality of a model transformation language can either be *unidirectional* or *multidirectional*. Unidirectional languages allow only transformations from one particular source model to another particular target model. Multidirectional approaches however also allow the transformation back form the target model to the source model. This can be accomplished using rules which allowing a multidimensional mapping definition (see also Section 3.1), or by implementing the rule separately for each direction [5].

3.7 Tracing

The connection of source and target models in a model transformation is called tracing. It establishes links between a source model element and its corresponding target element. Not all model transformation languages offer built-in support for tracing. In most languages not support it, tracing can be implemented by the developer however.

In case of a dedicated tracing support, the language can either create the traceability information automatically (possibly tunable), or the developer has to establish the tracing links himself.

The tracing information created can be stored in the source or target model itself, or separately.

Possible values: [Yes | No]

4 Running Model Transformation Examples

Several sample model transformation scenarios are used to compare the transformation languages described. The metamodels are created as Ecore files using the EMF tree-based editor. For each ecore file, an EMF editor is generated, and also exported as Eclipse plugin. This is necessary for using the metamodels in Kermeta and SmartQVT.

4.1 Running Example 1: ooclass2table

Mapping UML-style, object-oriented class schemata to relational database models can be thought of the *Hello World* or *99 bottles of beer* of model transformation. This type of transformation is not too complex and can be done very quickly. On the other hand, it involves more than simple copying of classes and attributes. Therefore it is very handy to get a fast, first impression of how a given model transformation technology works.

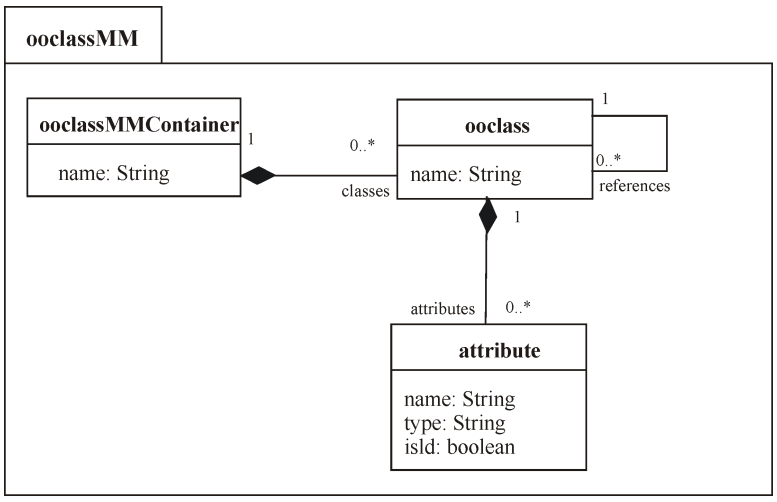

Figure 4.1: OoclassMM Metamodel

The ooclass2table transformation in this thesis is rather simple, compared to other, more sophisticated transformations found elsewhere. Classes are contained in an instance of the class *ooclassMMContainer*. Every class has several attributes, which have a name, a datatype and can be identifying. Classes can also have references to other classes. Advanced features, such as

user defined data types, generalization, containment etc. are not incorporated, for simplicity's sake. It is also notable, that it is always assumed that only one identifying attribute exists per class.

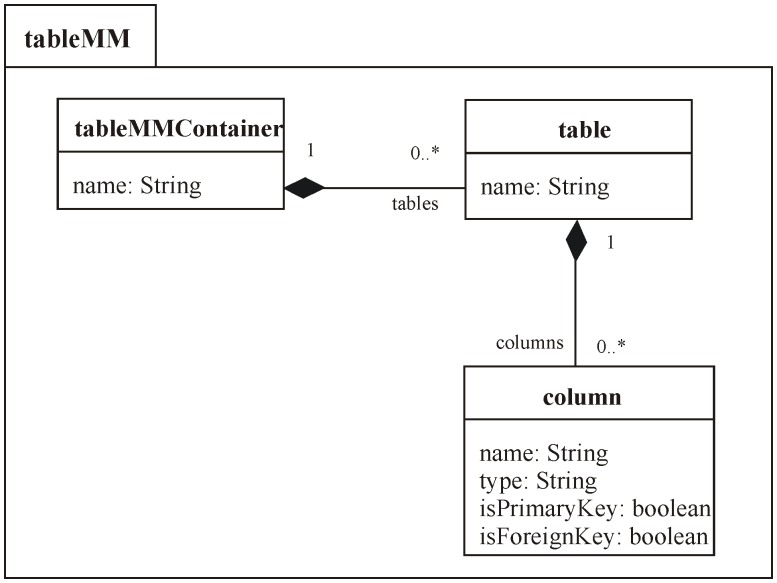

Figure 4.2: TableMM Metamodel

The transformed class model roots in an *tableMMContainer* element, which itself contains tables. Every table has a name and several columns. These columns are created in several different ways:

1. Non-identifying attributes of the source class are simply copied from the ooclass-model.
2. Identifying attributes get copied as well, but are furthermore marked as primary key in the table model.
3. References to other classes cause a foreign key column in the referenced class.

In other words, columns can not be created in a single step. It is necessary to create columns depending on the containing attributes of a class, but also depending on the references a class has. This can be solved for example by calling a rule for each reference a column has. This called rule has to find out the identifying attribute of the class given, which can generally be solved with OCL expressions.

Additionally, datatypes are mapped as well. Any given datatype expressed as string which is not included in this table, is simply mapped as-is. Table 4.1 describes the mapping.

ooclassMM	tableMM
String	VARCHAR(30)
Integer	INT
Double	DOUBLE
Date	DATE

Table 4.1: Mapping of Datatype in Ooclass2table

4.2 Running Example 2: epc2ad

The Mapping of event-driven process chains to UML activity diagrams is an example for a transformation of behavioral models, opposed to structural models like in ooclass2table. This transformation is more complex than ooclass2table.

Figures 4.3 and 4.4 show the respective metamodels used in the example transformation of this thesis. It should be noted, that they do not mean to reproduce an exact metamodel covering all features of the respective business process modeling language. Both only consider a subset of the model elements available. For example, the organizational and data view are missing in the event-driven process chain metamodel, and the activity diagram metamodel does not include activity partitions. Further, *activityFinalNode* and *initialNode* are subclasses of *activity*, which is not the case in reality. Additionally, the metamodels do not guarantee absolute correctness of the result models. For example, in event-driven process chains, functions and events always have to follow another alternating. This constraint is not included in the metamodel. In this thesis, it is modeled that way to simplify the transformation.

The aim of the design of both metamodels is to emphasize the challenging parts of a possible epc2ad transformation.

In both metamodels, a single container elements exists, which holds abstract *items*. All other elements of the respective metamodel are subclasses of item, and are therefore included in the model container element.

As can be seen in both metamodels for event-driven process chains and activity diagrams, there are more classes which have to be transformed, compared to the ooclass2table transformation described above. Some concepts of these diagrams can not be mapped in a simple 1:1 class mapping. A notable example is the mapping of *events* to *edges*. Events are nodes in event-driven process chains, but correspond to edges in activity diagrams. Therefore, the *control flow* elements are not needed to be transformed. They are only needed to establish the correct relationships between the activity nodes involved.

Another interesting aspect are *activityFinalNode* and *initialNode*. They do not

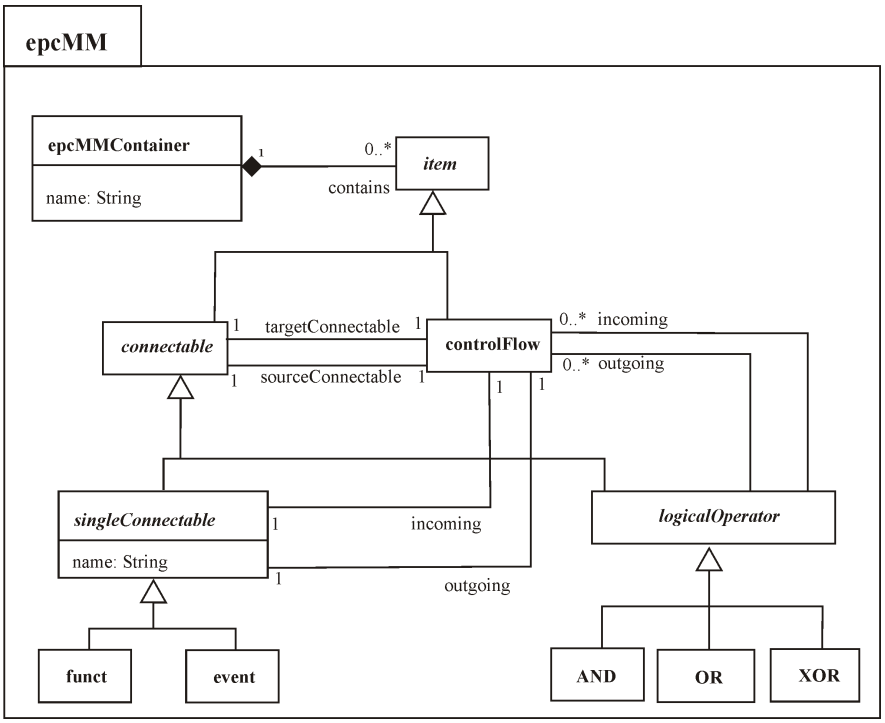

Figure 4.3: EpcMM Metamodel

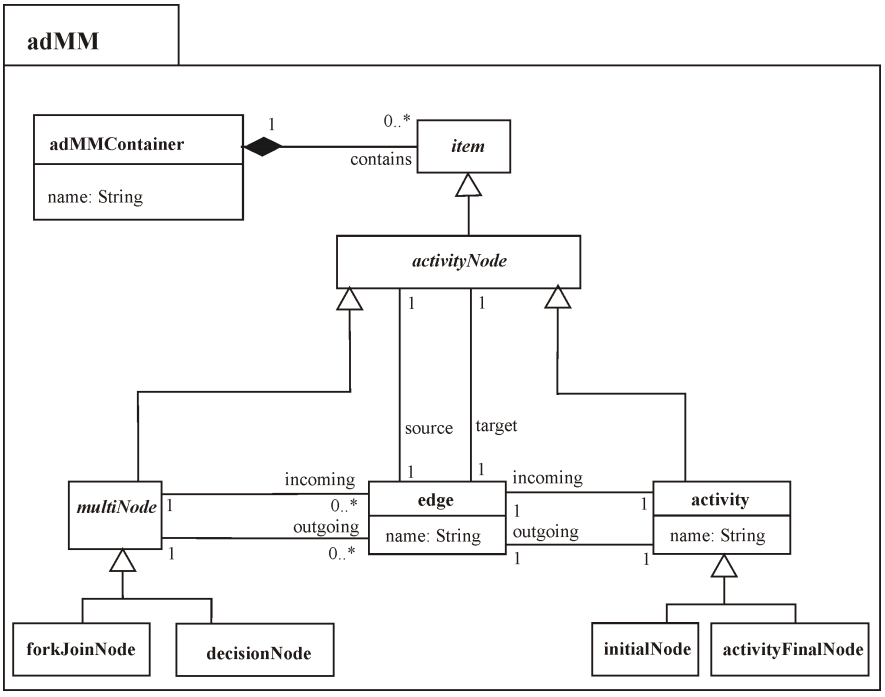

Figure 4.4: AdMM Metamodel

have corresponding metaclasses in the event-driven process chain metamodel, because event-driven process chains always begin and end with *events* and do not have such dedicated elements, which mark the beginning and the end of a diagram. During the transformation, events with no incoming or outgoing edge have to be identified. Further, an *activityFinalNode* or *initialNode* respectively has to be created as well.

It is also challenging to create the edges and their relationships in a correct way, because this concept may occur in connection with different other model elements involved:

- Event followed by Function
- Function followed by Event
- Event, followed by Logical Operator and several Functions
- Function, followed by Logical Operator and several Events
- Several Events, followed by Logical Operator and one Function
- Several Functions, followed by Logical Operator and one Event

Variations which include the use of logical operators imply the creation of one or more additional edges.

5 ATL - Atlas Transformation Language

5.1 Introduction to ATL

ATL is the ATLAS INRIA & LINA research group's answer to the OMG QVT request [20]. It can be thought of a hybrid language, which incorporates both imperative and declarative paradigms. The preferred style however is declarative. Imperative code should be used to specificity mappings which are rather difficult to describe in a declarative way.

ATL is a component of the AMMA (Atlas Model Management Architecture) platform [10]. Other components of AMMA are AM3 (model management), AMW (model weaving), KM3 (metamodel specification language), and several others.

5.2 Core Features

ATL is notable for its hybrid approach to model transformation. Most parts of a transformation to be implemented can be specified in ATL's declarative style. Because declarative style code is not as expressive as imperative code, some model transformation problems are hard to implement by using a declarative-only approach. Therefore ATL offers also support for imperative code. Imperative code can be used in *do* blocks of transformation rules, or completely separated in helper rules.

ATL offers an updated compiler version, which is called *ATL 2006*. It offers several features additionally to the default compiler version. However these features are not well documented as of now. Some of the new features are (see [11]):

- several source pattern elements
- rule inheritance
- improved OCL support
- endpoint called rules

The ATL 2006 compiler does create bytecode which is compatible to the existing ATL virtual machine. Therefore, compiled ATL code using 2006 compiler features is still compatible with older virtual machine versions. To enable the ATL 2006 compiler, the following comment as seen in Listing 5.1 has to be placed at the first line of the respective ATL file:

Listing 5.1: Selecting the ATL 2006 Compiler

```
1 -- @atlcompiler atl2006
```

5.3 User Interface

ATL is available as Eclipse plugin. For a developer with experience in Eclipse, ATL tool support is therefore very familiar. Syntax highlighting, error indication, and integration with the Eclipse debugger are also very helpful.

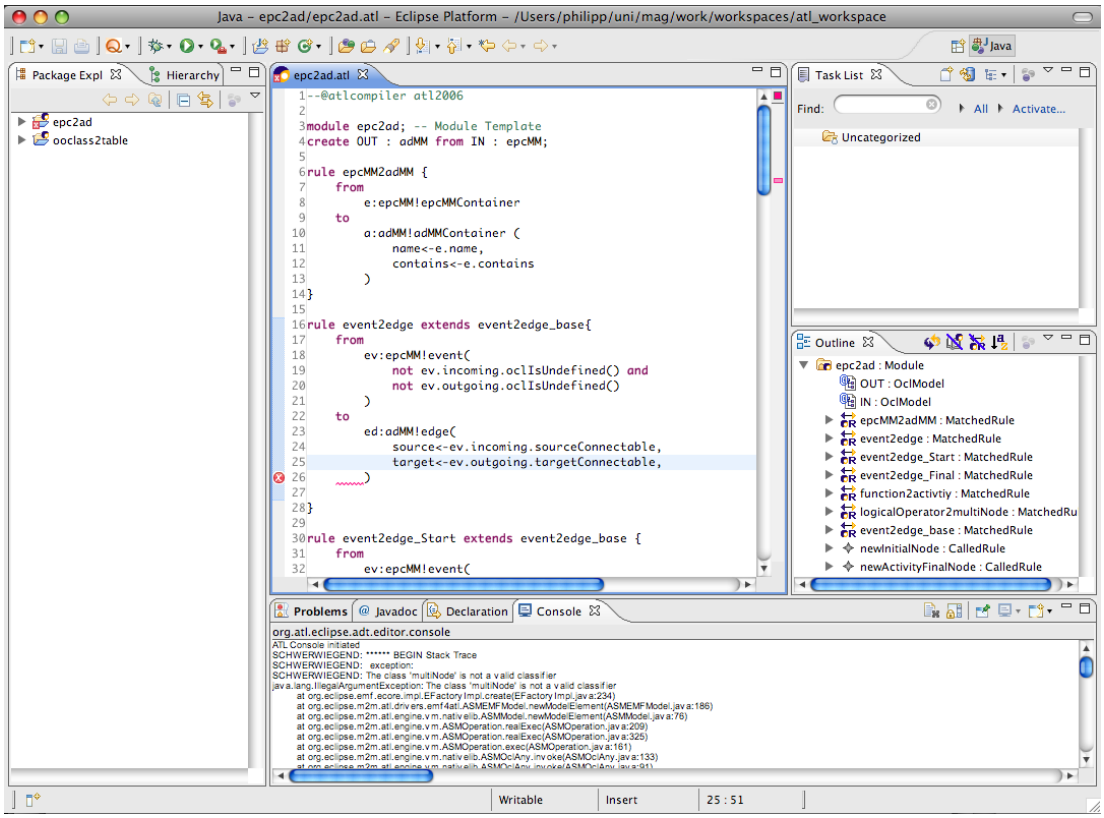

Figure 5.1: Editing an ATL File in Eclipse

As seen in Figure 5.1, the ATL development perspective in Eclipse is very similar to the Java perspective. On the left hand side, the package explorer is seen. Editor sub-windows are placed in the center pane. At the bottom, the ATL console is seen. If an error occurs during the runtime of a transformation, corresponding error messages are written there.

5.4 Sample Transformations

5.4.1 Solution for ooclass2table

Because of their similar approach, the transformation of *ooclass2table* is very similar in ATL and SmartQVT. Although ATL is a hybrid language and supports both imperative and declarative techniques, the use of declarative code is encouraged. This transformation follows that rule and uses mainly declarative style.

ATL automatically chooses the order in which the rules are executed. It is not necessary for the developer to specify an entry point. In most cases, it is sufficient to simply write all mapping rules. In the case of *ooclass2table*, ATL first executes the rule *ooclassMM2tableMM* (see Listing 5.2), because this rule applies to the root model element of the source model.

The root elements of both metamodels can be easily mapped, which is done using the declarative rule *ooclassMM2tableMM* in ATL.

Listing 5.2: ooclass2table in ATL: Rule ooclassMM2tableMM

```
1 rule ooclassMM2tableMM  {
2   from
3     c:ooclassMM!ooclassMMContainer
4   to
5     t:tableMM!tableMMContainer (
6       name <- c.name,
7       tables <- c.classes
8     )
9 }
```

Classes are basically copied to the target model without modification, however, additional columns are created using the lazy rule *ref2column*. As can be seen, ATL keeps already mapped columns and does not overwrite them during the second rule which creates columns. Therefore, ATL implicitly performs a union operation whenever the same target element is mapped multiple times, instead of overwriting already created target model elements.

Listing 5.3: ooclass2table in ATL: Rule ooclass2table

```
1  rule ooclass2table {
2    from
3      c:ooclassMM!ooclass
4    to
5      t:tableMM!table (
6        name <- 'tbl_'+c.name,
7        columns <- c.attributes,
8        columns <- c.references->collect(e|thisModule.
             ref2column(e))
9      )
10 }
```

The rule *ooclass2table* shown in Listing 5.3, also calles the lazy rule *ref2column* for creating the foreign key columns. For this purpose, all references are collected and the lazy rule is executed for each class which is referenced.

Attributes can also be easily mapped to columns, which is shown in Listing 5.4. The only difference between source and target model element is the attribute *type*. The datatype attribute is a string in both ooclass and table metamodels, however, they have different values (see Table 4.1). This mapping is done using the helper *dbType*.

Listing 5.4: ooclass2table in ATL: Rule attribute2column

```
1  rule attribute2column {
2    from
3      a:ooclassMM!attribute
4    to
5      c:tableMM!column (
6        name <- a.name,
7        isPrimaryKey <- a.isId,
8        type <- thisModule.dbType(a.type)
9      )
10 }
```

The rule *ref2column* (see Listing 5.5) is only executed when it is called (hence the name of the rule type *lazy*) and creates foreign-key columns for all attributes which are identifying in the referencing class. These identifying attributes are select with an OCL expression. A selection of all attributes containing the *isId* attribute is created. Because the metamodels in this transformation are assumed to only have exactly one identifying attribute, the result of this selection only contains this one attribute. Therefore, the OCL operation *first()* can be used to access the identifying attribute of the class.

The mapping of datatypes described in Table 4.1 is again done using the helper *dbType*, which translates a given input string to an output string.

Listing 5.5: ooclass2table in ATL: Lazy Rule ref2column

```
1  lazy rule ref2column  {
2    from
3      cl:tableMM!class
4    to
5      c:tableMM!column (
6        name <- 'FK_'+cl.name+'_'+cl.attributes->select(e|e.
             isId=true)->first().name,
7        type <- thisModule.dbType(cl.attributes->select(e|e.
             isId=true)->first().type),
8        isForeignKey <- true
9      )
10 }
```

5.4.2 Solution for epc2ad

The transformation of event-driven process chains to activity diagrams uses some features of ATL, which are only available in the 2006 version of the ATL-compiler. Especially, rule inheritance is only available in this version, which is used for the mapping of logical operators and events.

Listing 5.6: epc2ad in ATL: Abstract Rule logicalOperator2multiNode

```
1 abstract rule logicalOperator2multiNode {
2   from
3     lo:epcMM!logicalOperator
4   to
5     mn:adMM!multiNode(
6       incoming<-lo.incoming->collect(e|e.sourceConnectable)
            ,
7       outgoing<-lo.outgoing->collect(e|e.targetConnectable)
8     )
9 }
```

The abstract rule *logicalOperator2multiNode* as shown in Listing 5.6 maps the incoming and outgoing control flows. Because three different logical operators exist, which share common properties (namely *incoming* and *outgoing*), the mapping is created by two rules: the abstract rule *logicalOperator2multiNode* and a separate rule respectively *AND*, *OR* and *XOR*, which inherit from the abstract rule *logicalOperator2multiNode*.

Listing 5.7: epc2ad in ATL: Rule event2edge_Start

```
1  rule event2edge_Start extends event2edge_base {
2    from
3      ev:epcMM!event(
4        ev.incoming.oclIsUndefined() and
5        not ev.outgoing.oclIsUndefined()
6      )
7    to
8      ed:adMM!edge(
9        source<-initialNode,
10       target<-ev.outgoing.targetConnectable
11     ),
12
13     initialNode:adMM!initialNode(
14       outgoing<-ed
15     )
16   do {
17     adMM!adMMContainer.allInstances()->asSequence()->first
         ().contains<-initialNode;
18   }
19 }
```

A concrete rule which inherits from an abstract rule is shown in Listing 5.7. *event2edge_Start* extends the abstract rule *event2edge_base*. This rule is also notable for its *from* part, which includes a application condition. The rule only is

executed, if the condition evaluates to true. In this case, the event in question has to have a n outgoing control flow, but it is not allowed to have an incoming control flow. Therefore, for the correct mapping of this event, an *initalNode* has to be created as well. The imperative *do* part of the rule adds the created initialNode to the model container.

5.5 Summary on ATL

As mainly declarative approach, transformations in ATL can be implemented straightforward and fast. As a downside, despite ATL's imperative constructs, some transformation problems, like the mapping of logical operators in epc2ad, are relatively difficult to solve.

ATL's Eclipse integration offers a very flexible and useful development environment. Syntax highlighting, error indication, and debugging functionality is very helpful during the development of transformations. A slight shortcoming of ATL is, that output models are not checked for their metamodel conformance. This may lead to transformations which seem to run correctly, but the created models are erroneous.

6 SmartQVT

6.1 Introduction to SmartQVT

SmartQVT [36] is developed by France Telecom R&D. It is written in Java and available under the EPL open source license. SmartQVT aims to implement the QVT-Operational language, which is a subset of the Queries/Views/Transformations standard by OMG [17]. Therefore, SmartQVT follows the imperative model transformation approach. It is available as Eclipse-plugin.

6.2 Core Features

SmartQVT acts as a compiler in that sense, that given QVT-Code is compiled to Java source code. This is accomplished by a two-stage architecture [35]:

- The QVT Parser converts QVT textual syntax into corresponding representation in terms of the QVT metamodel.
- The QVT Compiler produces, from a QVT model, a Java program on top of EMF generated APIs for executing the transformation.

Because of this architecture, it is possible that the SmartQVT compiler can be used in connection with other tools which output a QVT model conforming to the QVT metamodel [35]. Additionally, serialized QVT transformations conforming to the QVT metamodel can be loaded and executed at runtime.

6.3 User Interface

Very similar to ATL, SmartQVT is an Eclipse plugin, as seen in Figure 7.1. Later versions (based on the Python QVT parser) did not offer error indication, but recent versions of SmartQVT support this feature, which is very convenient for transformation development. Additionally, syntax highlighting is available. Because SmartQVT compiles QVT code to Java code, no QVT-level debugger is available.

6.4 Sample Transformations

6.4.1 Solution for ooclass2table

Implementing *ooclass2table* in SmartQVT is possible in a simple way and consists of less code than for example the implementation in Kermeta. Although both

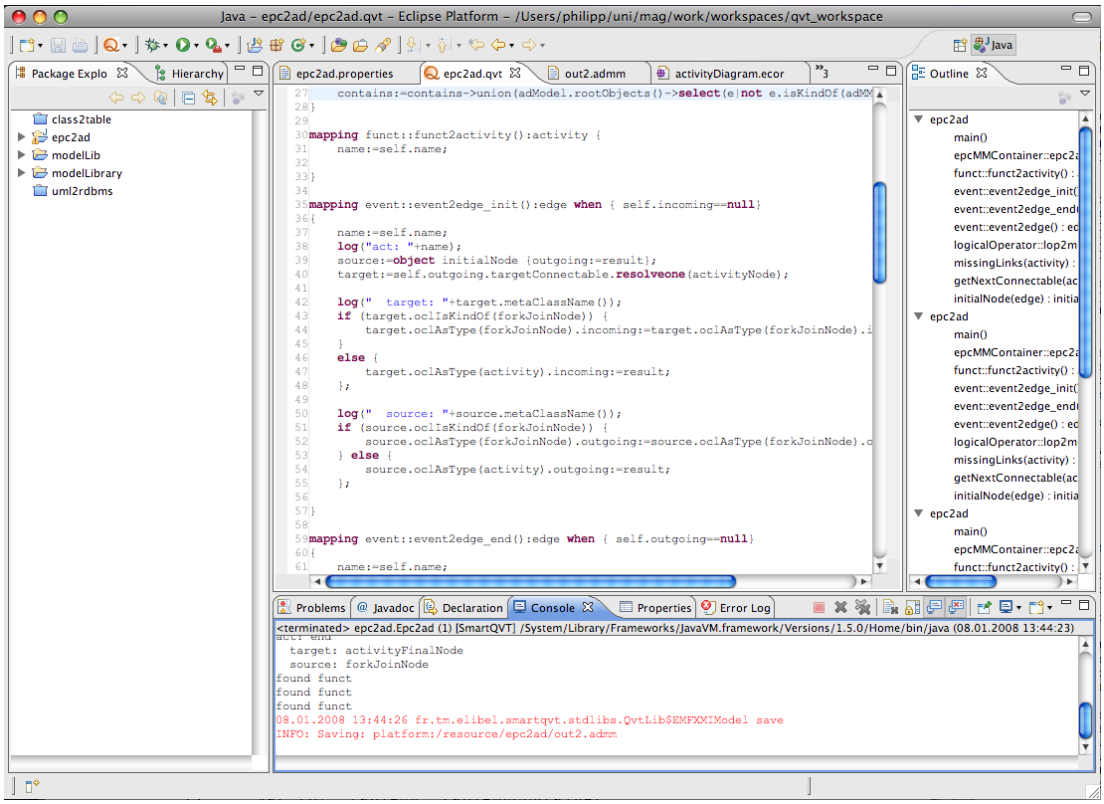

Figure 6.1: Editing a SmartQVT File in Eclipse

transformation languages follow a imperative approach, the amount of code used in SmartQVT is considerably smaller than in Kermeta.

An entry point is defined, which is always the rule *main()*, as seen in Listing 6.1. In this rule, the mapping rule *class2table* is called for all instances of *ooclass* in the given model, by using the *[]* brackets.

Listing 6.1: ooclass2table in SmartQVT: Main Rule

```
1 transformation class2table(in classModel:CLASSES,out
    tableModel:TABLES);
2
3 main() {
4   classModel.objects()[ooclass]->map class2table();
5 }
```

The target tables' columns are again created by two different rules: one rule copies the containing attributes to the target model, whereas the other rule creates foreign key columns based on class references. Both rules are called in a single statement, as described in Listing 6.2. If two sequential statements would be used, the second one would overwrite the result of the first statement. This is unlike ATL, where a union of both results is created implicitly. Because of this behavior, the result of both rule calls have to be joined explicitly. This is done

via OCL, using the *union()* expression.

Listing 6.2: ooclass2table in SmartQVT: Rule class2table

```
1 mapping ooclass::class2table() : table {
2   name := 'tbl_'+self.name;
3   columns := self.attributes->map attr2col()->union(self.
      references->collect(c|c->map class2fkey()));
4 }
```

Attribute can be mapped to columns in a simple way. For the mapping of the datatype, a helper function is again used. The complete rule is shown in Listing 6.3.

Listing 6.3: ooclass2table in SmartQVT: Rule attr2col

```
1 mapping attribute::attr2col() : column {
2   name := self.name;
3   type := self.type.dbType();
4   isPrimaryKey := self.isId;
5 }
```

Creating the foreign key columns shows similarity to how this problem is solved in ATL, as seen in Listing 6.4. Again, all attributes with the *isId* flag set to *true* are selected. Because of the assumption, that only one attribute is identifying, the selection can only return one element, which is accessed via the OCL statement *first()*.

Listing 6.4: ooclass2table in SmartQVT: Rule class2fkey

```
1 mapping ooclass::class2fkey(): column {
2   name := 'FK_'+self.name+'_'+self.attributes->select(a|a.
      isId==true).first().name;
3   type := self.attributes->select(a|a.isId==true).first().
      type.dbType();
4   isForeignKey := true;
5 }
```

6.4.2 Solution for epc2ad

The transformation of Event-driven Process Chains to UML Activity Diagrams is considerably more complex than the simplified transformation *ooclass2table* described in the previous section. The entry rule *main* is used to call the mapping rule for the model containers involved, which in turn then calls all subsequent transformation rules.

The rule *epc2ad_diag* in Listing 6.5 maps the container elements of the EPC and AD models used in the transformation. First, the EPC container's name is copied to the target container. After that, the mapping operations are called in the following order:

- Map all Functions to Activities (line 3)
- Map all Logical Operators (line 5)
- Map all Events with both incoming and outgoing control flows (line 6)
- Map the beginning Event of the model (line 7)
- Map the ending Event of the model (line 8)
- Create missing edges, which have not been created by the rules executed before (lines 10 - 12)

Listing 6.5: epc2ad in SmartQVT: Rule epc2ad_diag

```
1  mapping epcMMContainer::epc2ad_diag():adMMContainer{
2    name:=self.name;
3    contains:=self.contains[funct]->map funct2activity()->
       asOrderedSet();
4
5    contains.append(self.contains[logicalOperator]->map
       lop2multiNode());
6    contains.append(self.contains[event]->map event2edge());
7    contains.append(self.contains[event]->map event2edge_init
       ());
8    contains.append(self.contains[event]->map event2edge_end
       ());
9
10   result.contains[activity]->forEach(a) {
11     missingLinks(a);
12   };
13
14   -- rootobjects according to QVT specification, but
       rootObjects in SmartQVT
15   contains:=contains->union(adModel.rootObjects()->select(e
       |not e.isKindOf(adMMContainer)));
16 }
```

After the mapping rules have been executed, in line 15 all created elements which are not yet inside the target container element are moved there.

The rules *funct2activity* and *lop2multiNode* are very simple, because they do not handle relationships to other elements. Therefore their code is not shown here. They simply create the desired target elements, namely ActivityNodes and MultiNodes. Because they are created first, it is insured that later rules, which handle the creation of Edges and the corresponding relationships, do not have to take care of ActivityNodes and MultiNodes.

The rule *event2edge_end* in Listing 6.6 illustrates, how Events are mapped to edges, and how the creation of new elements is handled. This rule aims to create an activityFinalNode, and additionally sets the relationship between this newly

created node and the following ActivityNode. In its header, the rule includes a guard expression, which in this case means, that the rule is only executed, if the Event in question has no outgoing Control Flows and therefore is the last Event of the model. In its body, the rule first maps the Event's name and creates a new ActivityFinalNode using the *object* statement. The newly created ActivityFinalNode's incoming Edge is set to the Edge which is to be created by this rules, using the special variable *result*. In lines 7 - 11, the previous node's outgoing Edge is set to *result* as well. If the previous node is a ForkJoinNode, the Edge has to be added to the *outgoing* collection, whereas if it is a simple node, there is only one *outgoing* edge (not a collection), which can be set directly.

Listing 6.6: epc2ad in SmartQVT: Rule event2edge_end

```
1  mapping event::event2edge_end():edge when { self.outgoing==
       null}
2  {
3    name:=self.name;
4    target:=object activityFinalNode {incoming:=result};
5    source:=self.incoming.sourceConnectable.resolveone(
       activityNode);
6
7    if (source.oclIsKindOf(forkJoinNode)) {
8      source.oclAsType(forkJoinNode).outgoing:=source.
         oclAsType(forkJoinNode).outgoing->union(result);
9    } else {
10     source.oclAsType(activity).outgoing:=result;
11   };
12
13 }
```

Because not all Edges can be created using only the mechanism described above, the missing Edges have to be created separately. For this purpose, two helper rules are implemented, namely *missingLinks* and *getNextConnectable* (see Listings 6.8 and 6.7).

Listing 6.7: epc2ad in SmartQVT: Helper getNextConnectable

```
1  helper getNextConnectable(a:activity):activityNode {
2
3    var x:=a.invresolve(funct)->first();
4    --->first().oclAsType(singleConnectable).outgoing.
       targetConnectable;
5    return x.outgoing.targetConnectable.resolveone(
       activityNode);
6
7  }
```

The helper *getNextConnectable* is responsible for looking up the following ActivityNode for a given Activity. This functionality is needed later on to create all missing Edges. Because during the transformation, not all links in the target

Activity Diagram are guaranteed to be correct, the needed information is looked up in the source model. For this purpose, the helper function performs a inverse resolving operation, in order to get the corresponding Function to the Activity given. The Function's successor node is accessed via the outgoing Control Flow, and then resolved to the corresponding ActivityNode in the target model, which is returned eventually.

Listing 6.8: epc2ad in SmartQVT: Helper missingLinks

```
1  helper missingLinks(a:activity):OclVoid {
2
3    if (a.outgoing == null) {
4      var next:=getNextConnectable(a);
5      a.outgoing:=object edge {
6               source:=a;
7               target:=next;
8             };
9      if (next.oclIsKindOf(forkJoinNode)) {
10       next.oclAsType(multiNode).incoming:=next.oclAsType(
             multiNode).incoming->union(a.outgoing);
11     } else {
12       next.oclAsType(activity).incoming:=a.outgoing;
13     }
14
15   };
16
17 }
```

The helper *missingLinks* is called for every Activity which has been created, as can be seen in Listing 6.5, lines 10 - 12. Listing 6.8 only shows the code relevant for adding outgoing Edges, incoming Edges are created the same way. First, the helper *getNextConnectable*, which is described before, is called. After that, a new Edge is created in lines 5 - 8. The successor ActivityNode is set as the new Edge's target node. Finally, the reverse property is set: the successor node's incoming Edge is set to the newly created Edge.

6.5 Summary on SmartQVT

Previous versions of SmartQVT (< 0.2.x), which were based on the QVT parser written in Python, were not all to comfortable to work with. No errors in the code where indicated during design time. Errors were only detected during compile time, and some errors, which could not be found by the parser, occurred at runtime.

In recent version, the Python parser has been replaced by a parser written in Java. Since version 0.2, error indication during editing is also available, and most errors in the transformation code are found during design time. These features considerably improve the development process with SmartQVT, which is now comparable to Kermeta and ATL.

Operational QVT itself is a very powerful language including advanced language features. Compared to ATL, QVT is somewhat harder to learn, but also more expressive because of its imperative approach. Kermeta on the other hand, which also follows the imperative approach, is missing some features which are very useful and more or less obligatory in a model transformation environment. For example, tracing support has to be implemented by the developer in Kermeta, but is offered out of the box by SmartQVT. In general, it takes some time to understand and to be able to use QVT's features, but as soon the developer is proficient with QVT, many model transformation problems can be solved in an elegant and efficient way.

7 Kermeta

7.1 Introduction to Kermeta

Kermeta can be described as a *metamodeling language which allows describing both the structure and the behavior of models* [12]. It incorporates several features and ideas of other technologies in the model driven architecture domain. Therefore Kermeta is not limited to model transformation. For example, it can be used to define metamodels, or add constraints and behavior to them.

Kermeta's imperative approach to model transformation results in a different transformation code in comparison to ATL and SmartQVT, which follow mainly the declarative paradigm. Instead of rules, Kermeta uses operations, which are basically very similar to operations or methods in object-oriented programming languages, such as Java.

Listing 7.1: Load and Save a Model using built-in EMF Support

```
1  operation main() : Void is do
2
3      var oocl: ooclassMM::ooclassMMContainer
4      var repository : EMFRepository init EMFRepository.new
5      var resource : EMFResource
6
7      // load ooclassMM model via EMF
8      resource ?= repository.createResource("test2.ooclassmm"
           , "ooclassMM.ecore")
9      resource.load
10     oocl ?= resource.instances.one
11
12     // instantiate target model
13     var tbl: tableMM::tableMMContainer
14     tbl:=tableMM::tableMMContainer.new
15
16     // transformation entry point
17     tbl:=addTables(oocl,tbl)
18
19     // write output file
20         var out : Resource init repository.createResource("
               out.tablemm", "tableMM.ecore")
21     out.instances.add(tbl)
22     out.save()
23
24     end
```

It should also be noted that in Kermeta, the input and output of metamodel and model data has to be taken care of by the programmer. Therefore, every Kermeta program has to load and save data explicitly by itself. In ATL and SmartQVT on the other hand, the input and output metamodels and models can be specified outside the transformation code, using Eclipse Run Configurations. However, Kermeta supports EMF, which results only in little extra code, as can be seen in Listing 7.1.

7.2 Core Features

Kermeta's imperative approach and advanced language features make it nearly as powerful as conventional programming languages. It offers some features, which are not available in the other languages evaluated in this thesis, for example, reflection, genericity, exception handling or aspect orientation [12]. Similar to the Black-box mechanism in QVT, it is possible to call external Java code. Therefore nearly all transformation problems can be solved in Kermeta. On the other hand, some features typical for model transformation languages are missing. Kermeta has no built-in tracing support, and loading and saving of model files is somewhat complicated compared to other languages.

7.3 User Interface

Like SmartQVT and ATL, Kermeta is also available as Eclipse plugin. It is well integrated into the development environment and provides a debugger, syntax highlighting and error indication during the editing of code.

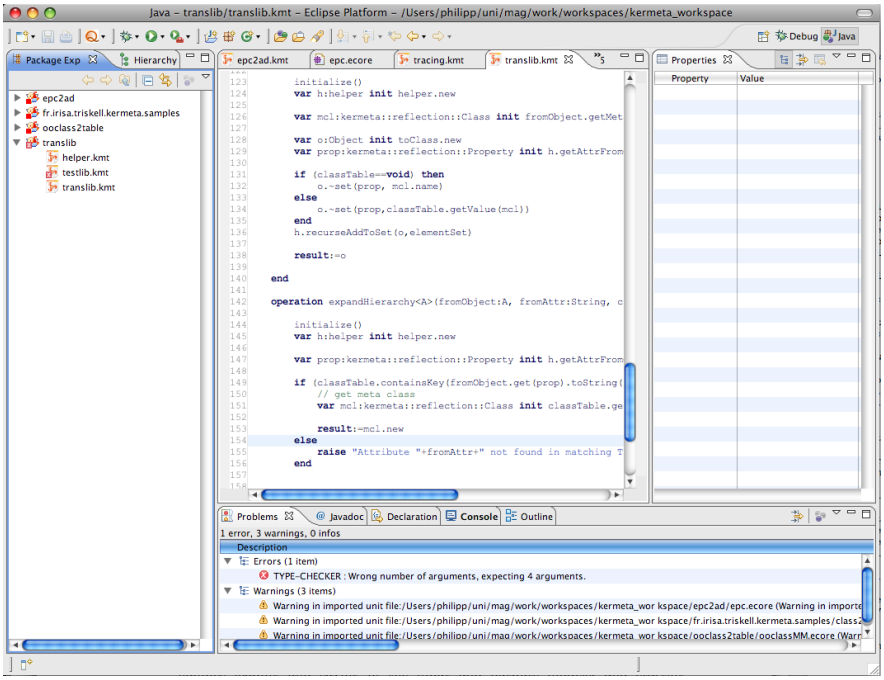

Figure 7.1: Editing a Kermeta File in Eclipse

7.4 Sample Transformations

7.4.1 Solution for ooclass2table

The ooclass2table transformation is solved by creating a main method, which handles loading and saving of the input and outputs models, and starting the transformation itself. This is done via the operation *addTables*, which takes an ooclassMMContainer object as input and returns the transformed tableMMContainer object as result.

The operation *addTables* calls two other operations, namely *addColumnsToTable* and *createRefCol*. These two operation are responsible for creating the column objects. *addColumnsToTable* basically only copies the attribute data over to a newly instantiated column object. Despite being rather simple, this operation nicely illustrates imperative-style transformations in contrast to declarative techniques: it is necessary to iterate over a collection of multiple elements (*each* statement in Listing 7.2). Furthermore, the target objects have to be instantiated and added to the parent object. More code has to be written to accomplish this task, which results in less readable and less concise code, at least for simple transformations.

Listing 7.2: ooclass2table in Kermeta: Operation addTables

```
1  operation addTables(cls : ooclassMM::ooclassMMContainer) :
       tableMM::tableMMContainer is do
2
3    var tbl: tableMM::tableMMContainer
4    tbl:=tableMM::tableMMContainer.new
5
6    cls.classes.asBag().each { c|
7      // instantiate new table and store tracing information
8      var table:tableMM::table init tableMM::table.new
9
10     // copy name, create direct columns
11     table.name := c.name
12     table:=addColumnsToTable(c,table)
13     tbl.tables.add(table)
14
15     // resolve references
16     c.references.asBag().each { ref|
17       createRefCol(ref,table)
18     }
19   }
20   result :=tbl
21 end
```

The operation *addColumnsToTable* seen in Listing 7.3 is similar to *addTables* in the sense, that it also iterates over multiple elements (in this case attributes). It also instantiates the target model elements and copies data from the source elements to the target elements.

Listing 7.3: ooclass2table in Kermeta: Operation addColumnsToTable

```
1  operation addColumnsToTable(cls : ooclassMM::ooclass, tbl:
       tableMM::table): tableMM::table is do
2
3    cls.attributes.asBag().each { a |
4      var col:tableMM::column init tableMM::column.new
5
6      col.name:=a.name
7      col.type:=mapType(a.type)
8
9      if a.isId==true then
10       col.isPrimaryKey:=true
11     end
12     tbl.columns.add(col)
13   }
14   result:=tbl
15 end
```

createRefCol as seen in Listing 7.4 on the other hand makes heavy use of OCL statements. For this reason, this part of the transformation does not differ a lot to the implementation using ATL or SmartQVT.

Listing 7.4: ooclass2table in Kermeta: Operation createRefCol

```
1  operation createRefCol(cls : ooclassMM::ooclass, tbl:
       tableMM::table): Void is do
2
3    var col:tableMM::column init tableMM::column.new
4    col.name:="FK_"+cls.name+"_"+cls.attributes.select{ o | o
       .isId==true }.first().name
5    col.type:=mapType(cls.attributes.select{ o | o.isId==true
       }.first().type)
6    col.isForeignKey:=true
7
8    tbl.columns.add(col)
9
10 end
```

7.4.2 Solution for epc2ad

The transformation of Event-driven Process Chains to UML Activity Diagrams in Kermeta consists of considerably more code than the *ooclass2table* transformation. Again, some parts of the code are needed to cover features, which are supplied by other model transformation languages, but which are absent in Kermeta. For the *epc2ad* transformation, model loading and saving, and tracing support have to be implemented.

The main operation of the *epc2ad* transformation seen in Listing 7.5 is very similar to the main operation in *ooclass2table*, because the loading and saving of

the involved models is handled there.

Listing 7.5: epc2ad in Kermeta: Operation mapContainer

```
1  operation mapContainer(epcContainer: epcMM::epcMMContainer)
       : adMM::adMMContainer is do
2
3    var res: adMM::adMMContainer
4    res:=adMM::adMMContainer.new
5
6    res.name:=epcContainer.name
7
8    epcContainer.contains.select{x|x.isKindOf(epcMM::funct)}.
       each {f|
9      res.contains.add(mapFunction(f.asType(epcMM::funct)))
10   }
11
12   epcContainer.contains.select{x|x.isKindOf(epcMM::AND)}.
       each {l|
13     res.contains.add(mapAND(l.asType(epcMM::AND)))
14     createLinks(l,res)
15   }
16
17   epcContainer.contains.select{x|x.isKindOf(epcMM::event)}.
       each {f|
18     res.contains.add(mapEvent(f.asType(epcMM::event)))
19     createLinks(f,res)
20   }
21
22   result:=res
23
24 end
```

Listing 7.5 shows the operation *mapContainer*, which is responsible for creating the target model container and calling all additional mapping operations. First, in lines 3 - 6, the Activity Diagram container element is instantiated, and its name is set to the source model container's name. After that, the mapping operations for Functions, Logical Operators and Events are called and the thereby created elements are added to the target model container element. In lines 14 and 19, the helper operation *createLinks* is called, which creates missing links for the newly created elements.

Listing 7.6: epc2ad in Kermeta: Operation mapFunction

```
1  operation mapFunction(fun: epcMM::funct): adMM::activity is
       do
2
3    var act: adMM::activity
4    act:=adMM::activity.new
5
```

```
6     act.name:=fun.name
7
8     tracingLinks.storeTrace(fun,act)
9
10    result:=act
11
12  end
```

As shown in Listing 7.6, the mapping of Functions to Activities is relatively simple. The mapping of Events to Edges and the mapping of Logical Operators works also exactly the same. The target Activity element is instantiated, and the source element's name is copied. The newly created element and its corresponding source element is stored in the tracing framework. Finally, the created Activity is returned. No relationships are created in this operation, this will be handled later on during the transformation.

Listing 7.7: epc2ad in Kermeta: Operation createLinks

```
1  operation createLinks(it: epcMM::item, ad:adMM::
       adMMContainer) is do
2
3    if it.isKindOf(epcMM::event) then
4      var ev:epcMM::event init it.asType(epcMM::event)
5      var ed:adMM::edge init tracingLinks.getTargetElem(ev).
           asType(adMM::edge)
6
7      if ev.incoming!=void then
8        if ev.incoming.sourceConnectable.isKindOf(epcMM::
             funct) then
9          ed.source:=tracingLinks.getTargetElem(ev.incoming.
               sourceConnectable).asType(adMM::activityNode)
10         ed.source.asType(adMM::activity).outgoing:=ed
11       else
12         ed.source:=tracingLinks.getTargetElem(ev.incoming.
               sourceConnectable).asType(adMM::multiNode)
13         ed.source.asType(adMM::multiNode).outgoing.add(ed)
14       end
15     end
16     if ev.outgoing!=void then
17       // do the same for outgoing
18     end
19
20     if ev.incoming==void then
21       var in:adMM::initialNode init adMM::initialNode.new
22       ad.contains.add(in)
23       ed.source:=in
24       in.outgoing:=ed
25     end
26     if ev.outgoing==void then
27       // do the same for outgoing
```

```
28      end
29    end
30
31    if it.isKindOf(epcMM::AND) then
32      var lo:epcMM::AND init it.asType(epcMM::AND)
33      var fjn:adMM::forkJoinNode init tracingLinks.
           getTargetElem(lo).asType(adMM::forkJoinNode)
34
35      lo.incoming.each {i |
36        var andIncoming:adMM::edge init adMM::edge.new
37        andIncoming.source:=tracingLinks.getTargetElem(i.
             sourceConnectable).asType(adMM::activity)
38        andIncoming.source.asType(adMM::activity).outgoing:=
             andIncoming
39        andIncoming.target:=fjn
40        ad.contains.add(andIncoming)
41        fjn.incoming.add(andIncoming)
42      }
43    end
44
45  end
```

As can be seen in Listing 7.7, the creation of relationships between the model elements create before is the most complex part of the transformation. For easier comprehension, similar tasks have been commented out. Basically the operation differs between Events (lines 3 - 29) and Logical Operators (lines 31 - 43). In the first case, the edge belonging to the input Event, which has been created by the operation *mapEvent*, is retrieved by using the tracing information. After that, in lines 7 and 16, it is decided whether the originating Event has incoming and outgoing Control Flows. If this is the case, the Edge's source and target have to be set to the corresponding Activity Diagram element of the Events predecessor and successor elements. In case the predecessor or successor element respectively is a Function, the reverse property can simply be set to the currently processed element (lines 8 - 10). If it is a Logical Operator, the newly created edge has to be added to the corresponding multiNode's incoming or outgoing collection (lines 12 - 13). If on the other hand, the current Event has no incoming or outgoing Control Flow, it is the beginng or ending Event of the Diagram. In this case an InitialNode or ActivityFinalNode has to be created and its relationships have to be set (lines 20-25). In lines 31-42, the creation of links in case of a Logical Operator is impemented. This is basically the same as described before for Events. However, more than one incoming Control Flow may exist, therefore it is necessary to iterate over the *incoming* collection to create all missing links.

7.5 Summary on Kermeta

Kermeta's model transformation abilities are actually only a subset of what can be accomplished with the Kermeta language. It is intended as core language of a model oriented platform [12] and may be used as language for metamodel-

ing, specifying model constraints and model semantics, and as transformation language. In this thesis, the focus of the attention lies on Kermeta's model transformation support.

Kermeta is best suited as model transformation language, if the transformation to implement is relatively complex. Because of its imperative approach, the developer can implement any behavior, especially compared to declarative approaches. On the downside, many features supported by most other model transformation languages are not as convenient to use in Kermeta, for example loading and saving of models, or may be completely missing, such as automatic tracing support. This results in transformation code, which is usually longer and harder to read than code written in other model transformation languages.

8 ModelMorf

8.1 Introduction to ModelMorf

ModelMorf [34] is an implementation of the Relational QVT standard issued by OMG [17]. An other implementation of Relational QVT is medini QVT [23]. Tools with support for Triple Graph Grammars, like Fujaba [7] and MOFLON [9] are also QVT compliant, by implementing relational QVT's graphical syntax.

Currently, not all features of Relational QVT are supported in ModelMorf. Most notably, there is no support for [18]:

- Incremental transformation execution
- Transformation extensibility
- Graphical Syntax

8.2 Core Features

The main feature of ModelMorf is its general model transformation approach. As implementation of the Relational QVT standard, it is purely declarative. For some transformation problems, a declarative transformation language offers some advantages over other approaches. For example, if multidirectionality or target incrementality is mandatory for a particular model transformation setting, ModelMorf is the only choice out of the four transformation languages compared in this thesis.

8.3 User Interface

ModelMorf offers no development environment or graphical user interface. Transformation code may be created using any text editor. Transformations are executed by calling the executable with parameters, as can be seen in Listing 8.1:

Listing 8.1: Executing a ModelMorf Transformation

```
1 sh /Applications/ModelMorf/modelmorf -m ooclassMM -mf
    ooclassMM.xml -m tableMM -mf tableMM.xml -c test.qvt -u
    ooclass -f test2.ooclassmm -u table -f out.admm -t
    ooclass2table -d table -q enforce
```

The connection between the names of metamodels and their physical representation in XML files is done using the *-m* and *-mf* parameters. The parameter *-c* denotes the file, which contains the transformation code. The involved input and output domains and model files are specified using the parameters *-u* and *f*. The desired direction, in which the transformation has to be executed, is specified with the parameter *-d*.

Because no integrated development environment is provided, the development of transformation code is not as comfortable as in other approaches. Errors in the code are only detected when the transformation is executed. However, in case of an error, useful error messages are given.

8.4 Sample Transformations

8.4.1 Solution for ooclass2table

The transformation *ooclass2table* is relatively simple to implement using a declarative approach like ModelMorf. First, the transformation header and several key attributes are defined, as illustrated in Listing 8.2.

Listing 8.2: ooclass2table in ModelMorf: Header and Keys

```
1  transformation ooclass2table(ooclass: ooclassMM; table:
       tableMM)
2  {
3
4  key ooclassMM::ooclassMMContainer{name};
5  key tableMM::tableMMContainer{name};
6  key ooclassMM::ooclass{name};
7  key tableMM::table{name};
8
9    ...
10 }
```

The transformation header in line 1 identifies the models used in this transformation. The mapping to the respective metamodel files is established using command line parameters during the start of the transformation. The key definitions in lines 4 - 7 denote, which attribute uniquely identifies an instance of a given class. This is necessary, so that the respective elements are only created once. If these key definitions were omitted, a separate container element would be created around every class.

After that, the rules which make up the transformation itself are defined. Basically, two types of rules, in this case called *relations*, are used: top relations and normal relations. Top relations are called at the beginning of the transformation, whereas other relations have to be called by other rules using the *where* or *when* clauses. In *ooclass2table*, two top relations are used, which are shown in

Listings 8.3 and 8.5.

Listing 8.3: ooclass2table in ModelMorf: Container Mapping

```
1  top relation classCont2tableCont {
2    n:String;
3
4    enforce domain ooclass
5    classCont:ooclassMMContainer {
6      name=n,
7      classes=cl:ooclass{}
8    };
9
10   enforce domain table
11   tableCont:tableMMContainer {
12     name=n,
13     tables=tbl:table{}
14   };
15
16   where {
17     class2table(cl,tbl);
18   }
19 }
```

The relation *classCont2tableCont* maps the container elements of both models involved. In line 2, the String variable *n* is defined, which is used to hold the container name, as seen in line 6 and line 12. The lines 4 - 8 describes the structure of the class container element. First, the membership to the domain *ooclass* is established in line 4. They keyword *enforce* denotes, that during a transformation from tables to classes, the class domain elements described here have to be created. In line 7, the variable *cl* is created, which holds all elements in the *classes* relationship. Lines 10 - 14 describe the table model accordingly. Eventually, the relation *class2table* is called in the where clause, using the classes and tables as parameters.

Listing 8.4 shows the mapping rule *class2table*, which is called by the top level rule described beforehand. Its basic appearance is very similar to the relation *classCont2tableCont*, with the exception that the elements involved are described nested, according to their relationships to each other as defined in the metamodels. In lines 9 - 11 and 17 - 19, the attribute/column names of the class and table accordingly are set. Additionally, in the *table* domain, the attribute *isForeignKey* is set to *false*. This part nicely illustrates, how transformations in Relational QVT are defined for multidirectional execution. If a class model is translated to a table model, the attribute *isForeignKey* is set to false. On the other hand, if the transformation is executed from a table model to a class model, columns are only processed by this rule, if their attribute *isForeignKey* has the value *false*. This relation also is responsible for calling the *attr2col* relation.

Listing 8.4: ooclass2table in ModelMorf: Class to Table Mapping

```
relation class2table {

  n:String;
  a_n:String;

  enforce domain ooclass
  cl:ooclass {
    name=n,
    attributes=attr:attribute{
            name=a_n
          }
  };

  enforce domain table
  tbl:table {
    name=n,
    columns=col:column{
          name=a_n,isForeignKey=false
        }
  };

  where {
    attr2col(attr,col);
  }

}
```

In Listing 8.5, the second top relation, namely *ref2col* is shown. This rule is again very similar to the rules discussed before. In this case, the nesting goes across three levels.

Listing 8.5: ooclass2table in ModelMorf: Mapping of References

```
top relation ref2col {

  n,an:String;

  enforce domain ooclass
  cl:ooclass {
    name=n,
    references=refs:ooclass{
      attributes=attr:attribute{
          name=an,
          isId=true
        }
      }

  };

  enforce domain table
  tbl:table {
```

```
19      name=n,
20      columns=col:column {
21        name=an,
22        isForeignKey=true
23      }
24    };
25
26
27  }
```

8.4.2 Solution for epc2ad

The *epc2ad* transformation could not be solved completely using ModelMorf. Only the mapping of the basic elements, namely functions, events, activities and edges of the metamodels could be implemented. Control flows and the linking between the model elements on the other hand is much more complex and very hard to solve using a declarative-only model transformation approach. However, the *epc2ad* transformation has been implemented by Altan [1] using Triple Graph Grammars, which are very similar to Relational QVT.

Listing 8.6: epc2ad in ModelMorf: Mapping of Functions to Activities

```
1  top relation funct2activity {
2
3    n:String;
4    c_n:String;
5    cf_name:String;
6
7    enforce domain epc
8    f:funct {
9      name=n,
10     contained=ce:epcMMContainer{
11       name=c_n
12     }
13   };
14
15   enforce domain ad
16   a:activity {
17     name=n,
18     contained=ca:adMMContainer{
19       name=c_n
20     }
21   };
22
23 }
```

Even for the working parts of the transformation, the given metamodels have to be adjusted. Additionally to the *contains* relationship, an opposite relationship is introduced in *item*, in order to express the containment relationship in QVT

rules, as illustrated in Listing 8.6. The elements *funct* and *activity* respectively include the attribute *contained*, which is the opposite of the *contains* relationship in the unmodified metamodels.

Additionally, debugging QVT rules with ModelMorf is hardly possible, because no insight on the rule execution is given. For example. the rule *e2e_links*, shown in Listing 8.7 is executed, with no errors reported by ModelMorf, but the attribute *source* of the element *edge* is not created.

Listing 8.7: epc2ad in ModelMorf: Rule e2e_links

```
1  relation e2e_links {
2
3    cf_n:String;
4
5    enforce domain epc
6    ev:event {
7      incoming=cf:controlFlow{
8        sourceConnectable=f:funct{
9          name=cf_n
10       }
11     }
12   };
13
14   enforce domain ad
15   ed:edge {
16     source=a:activity{
17       name=cf_n
18     }
19   };
20
21 }
```

Although the desired transformation may be completely implemented using ModelMorf, *epc2ad* still serves as an example for a particular transformation which is difficult to solve using a declarative approach.

8.5 Summary on ModelMorf

Both example transformations presented in this thesis illustrate the advantages and shortcomings of declarative-only approaches, as in this case Relational QVT implemented by ModelMorf. Simple transformations, such as *ooclass2table* can be developed very fast, using elegant and easy to read code. On the other hand, more complex transformations, like *epc2ad* are much more difficult to solve compared to hybrid or strictly imperative transformation approaches.

Before choosing a declarative approach, one should make sure that the transformation to implement is solvable conveniently with a declarative approach. If this is the case, ModelMorf is a good choice, because the transformation code

is efficient and easy to read. Also for some specific requirements, such as multidirectionality and target incrementality, there is hardly tool support in other languages, which makes ModelMorf very attractive for such situations.

9 Evaluation

In this chapter, the results of the feature support evaluation for the criteria catalogue [5], which has been discussed in Chapter 3, are documented and discussed.

9.1 Transformation Rules

9.1.1 ATL

Considering its hybrid approach, ATL basically supports *syntactic separation* of the involved models. This is achieved using the *from* and *to* parts of a rule, where the first applies to the source model elements and the latter to the target model elements.

ATL supports *application conditions* using boolean expressions in the *from* part. The rule is only executed if the condition specified evaluates to true. In the example in Listing 9.1, the rule is only executed if the node *event* has an incoming and outgoing control flow.

Because ATL includes built-in traceability support, it also uses *intermediate structures*.

Listing 9.1: Conditional Rule in ATL

```
1  rule event2edge extends event2edge_base{
2    from
3      ev:epcMM!event(
4        not ev.incoming.oclIsUndefined() and
5        not ev.outgoing.oclIsUndefined()
6      )
7    to
8      ed:adMM!edge(
9        source<-ev.incoming.sourceConnectable,
10       target<-ev.outgoing.targetConnectable
11     )
12
13 }
```

Domain Features

ATL can use both MOF and Ecore as *domain languages* for the metamodels involved in the transformation. As a consequence of the unidirectional rule execution approach of ATL, domains can only be specified as either *input* or *output*, but not combined. Further, *dynamic mode restriction* is not supported

for the very same reason.

ATL uses *syntactical typing*, based on the types defined in the metamodels involved in the transformation. However, the typing is only performed at run-time, which means that for example the assignment of a String value to an attribute of type Integer is not detected during the compilation of the transformation, but during the execution of the transformation.

ATL uses *patterns* with term-based structure. Syntax is represented in a *concrete*, *textual* way.

ATL only supports *control parameters* for parameterization. Additionally, higher-order rules are supported, because ATL is able to load ATL files as model, and transform them.

9.1.2 Kermeta

Kermeta does not support syntactic separation. This is a result of the imperative approach Kermeta is based on. Basically, Kermeta is very similar to traditional programming languages, therefore the code frequently mixes parts of all model domains involved. The missing support for multidirectionality is also based on Kermeta's imperative approach.

Transformation rules in Kermeta can be interpreted as operations, because the transformation behavior is specified within them. However, no conditions can be specified, which have to be fulfilled in order for the operation to be executed.

Kermeta does not maintain intermediate structures. However, it is possible for the developer to create such structures manually. This can be useful for example for building a traceability framework. Kermeta also supports both reflection, which is heavily used by the Kermeta library for reusable transformations (see Chapter 10), and basic aspect-oriented features.

Domain Features

Kermeta supports Ecore as domain language. Models expressed in Ecore can be imported to any Kermeta transformation. Also, saving into Ecore models is possible. Additionally, Kermeta is also a metamodelling language, therefore metamodels and models can be expressed in Kermeta itself. This can be achieved by writing classes in Kermeta, which implement the desired metamodel. However, serialization of models only works with Ecore, Kermeta models and metamodels can only be created at runtime.

Syntactical typing is used in Kermeta, and is also checked during compile time. Variables of a specific type can only hold data of this type, otherwise

the program can not be compiled. This is comparable to statically-typed programming languages like Java, which check type correctness also at compile-time.

Like ATL, Kermeta uses term-based structure and a concrete, textual syntax. Kermeta uses an imperative approach, which results also in imperative assignment of values and explicit element creation. This is illustrated by the operation *mapFunction* in Listing 9.2. Lines 2 and 3 explicitly create a new *activity* object. In line 6, a value is assigned to the *name* property of the object created before. Additionally, in the first line, the parameterization of Kermeta is shown as well: an object of the type *epcMM::funct* is passed on to the operation.

Listing 9.2: Kermeta Operation *mapFunction*

```
1  operation mapFunction(fun: epcMM::funct): adMM::activity is
       do
2
3    var act: adMM::activity
4    act:=adMM::activity.new
5
6    act.name:=fun.name
7
8    tracingLinks.storeTrace(fun,act)
9
10   result:=act
11
12 end
```

9.1.3 SmartQVT

As a mainly imperative approach, SmartQVT does not support clear syntactic separation. Although basically the left hand side of an expression usually addresses target elements, with a right hand side based on source model elements, this is not always the case. For example, the *result* variable within a QVT mapping rule refers to the object created by that rule. Multidirectionality is also not supported in Operational QVT [17].

Listing 9.3: Application conditions in SmartQVT

```
1 mapping event::event2edge_{end}():edge
2  when  { self.outgoing==null}
3  where { name<>"Begin" }
4 {
5   name:=self.name;
6
7 }
```

Application conditions are supported via the *when* clause, which acts as a guard conditions. The condition specified in the *when* clause must evaluate to true in order for the mapping rule to be executed. Additionally, post-conditions also exist in the form of the *where* clause. If the *where* condition is not *true*, an

exception is raised and the program is interupted [38]. The example in Listing 9.3 illustrates the usage of application conditions, more specifically pre- and postconditions, in SmartQVT.

Intermediate structures are built up by SmartQVT mainly for its tracing support. It is also possible to add intermediate metamodel elements during run time.

Domain Features

SmartQVT offers variables and uses a term-based structure. Its basic language paradigm is imperative. Value specification is achieved using imperative assignments.

As for parameterization, SmartQVT supports simple control parameters. Generics are not supported. However, SmartQVT is able to load and transform QVT-compliant models, which results in a support for higher-order rules. It is further possible to load a model conforming to the QVT metamodel, and then use this model as transformation.

9.1.4 ModelMorf

Relational QVT's and therefore ModelMorf's declarative approach enables clear syntactic separation. Both the input and output model side have separate corresponding parts in the QVT rules. Also, as consequence of the used transformation language approach, multidirectionality is possible in ModelMorf. There is no need for separate code for different execution directions, one rule file can be used to perform a transformation in any direction. Application conditions are also supported by ModelMorf, via the *when* and *where* clauses of QVT rules. ModelMorf also builds up intermediate structures, for example for saving the tracing information. Reflection and aspect-oriented constructs are not available.

Domain Features

According to its documentation [8], metamodels and models have to be specified in the XMI format. Static modes are *in/out*, as consequence of multidirectionality support. The mode of domains, for example whether a certain domain is input or output, can only be specified at the start of the transformation execution, but not inside the transformation code itself. Therefore, dynamic mode restriction is not possible with ModelMorf.

Similar to most other transformation languages, ModelMorf uses term-based structure and concrete textual syntax. Its main language paradigm is declarative, with imperative assignment for value specification, and implicit element creation. For parameterization features, ModelMorf only supports control parameters, by which rules can be called with model element variables as parameters. ModelMorf does not implemented the graphical syntax of Relational QVT.

9.1.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Syntactic Separation	Y	N	N	Y
Multidirectionality	N	N	N	Y
Application Conditions	Y	N	Y	Y
Intermediate Structures	Y	N	Y	Y
Reflection	N	Y	Y	N
Aspects	N	Y	N	N
Domain (see Table 9.2)				
Parameterization (see Table 9.4)				

Table 9.1: Results for Transformation Rules

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Domain				
Domain language	MOF,Ecore	MOF,Ecore	Ecore	MOF,Ecore
Static Mode	separate	separate	separate	in/out
Dynamic Mode Restriction	N	N	N	N
Typing	synt.	synt.	synt.	synt.
Body (see Table 9.3)				

Table 9.2: Results for Transformation Rules: Domain

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Body				
Variables	Y	Y	Y	Y
Patterns				
Structure	Term	Term	Term	Term
Syntax				
Abstract	Y	N	Y	N
Concrete				
Textual	Y	Y	Y	Y
Graphical	N	N	N	N
Logic				
Language Paradigm	Hybrid	Imperative	Imperative	Declarative
Value Specification	Imperative Assignment	Imperative Assignment	Imperative Assignment	Imperative Assignment
Element Creation	Implicit	Explicit	Explicit	Implicit

Table 9.3: Results for Transformation Rules: Domain, Body

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Parameterization				
Control Parameters	Y	Y	Y	Y
Generics	N	Y	N	N
Higher-order Rules	Y	N	Y	N

Table 9.4: Results for Transformation Rules: Parameterization

9.2 Rule Application Control

9.2.1 ATL

ATL determines the location of a rule application in a *deterministic* way. This happens during the source model elements matching phase (see [20]) of the transformation. Called rules are an exception for this behavior, because they are only executed when explicitly called in another rule.

9.2.2 Kermeta

Most of the feature support of Kermeta in this section is a consequence of its imperative approach. Because a *rule* is interpreted as a operation in Kermeta terminology, some features would not make sense in this context. The developer is responsible of which operation is executed at a specific point in the transformation, therefore no rule application strategy is available.

Features in *rule scheduling* are also affected by Kermeta's imperative approach. The scheduling of the rules is created by the developer, therefore it is explicit. No rule selection mechanisms exist. However, rules can make use of recursion.

Kermeta does not offer built-in phasing mechanisms, but it is possible for a developer to organize the transformation process into different phases explicitly.

9.2.3 SmartQVT

Because SmartQVT follows the imperative approach, the rule application is primarily specified by the developer. Therefore, the *rule application strategy* features are not applicable. The execution order of the rules is the execution flow of the transformation code, which is typical for imperative approaches.

The features included in *rule scheduling* also reflect SmartQVT's imperative approach. A notable feature supported by SmartQVT despite its imperative approach are the different sections a mapping operation can consist of. This is marked as the supported feature *phasing*, although it only works on operations,

not on whole transformation. The sections of a certain operation are executed in order when the operation is called. The following three sections are possible [17]:

- *init*. Code in this section is executed before the declared output elements are instantiated.
- *population*. Code in this section is used to populate the output elements.
- *end*. Code in this section is executed before the operation exits.

9.2.4 ModelMorf

ModelMorf uses a deterministic rule application strategy. First, all *top*-level rules are executed, followed by the rules which are specified in the *when* clauses of the top-level rules. The rule scheduling may be both implicit and explicit. The execution order of the top-level rules cannot be specified by the developer, which makes their execution order implicit. On the other side, any rule can *call* another rule in *when* and *where* clauses. This enables the developer to explicitly specify the execution order of these rules. The concurrence of implicit and explicit rule scheduling form is illustrated in the transformation skeleton in Listing 9.4. The top-level rules *classCont2tableCont* and *ref2col* are executed first, but no order between these two rules is specified. On the other hand, *classCont2tableCont* calls *class2table* using the *where* clause, which itself calls *attr2col* again using the *where* clause. In this way the execution order (classCont2tableCont > class2table > attr2col) is explicitly given by the developer.

Listing 9.4: Implicit and Explicit Execution Order in ModelMorf

```
1  transformation ooclass2table(ooclass: ooclassMM; table:
       tableMM)
2  {
3    top relation classCont2tableCont {
4      ...
5      where { class2table(cl,tbl);  }
6    }
7
8    relation class2table {
9      ...
10     where { attr2col(attr,col); }
11   }
12
13   relation attr2col {
14     ...
15   }
16
17   top relation ref2col {
18     ...
19   }
20 }
```

9.2.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Rule Application Strategy				
Deterministic	Y	N	N	Y
Non-Deterministic	N	N	N	N
Concurrent				
One-Point				
Interactive	N	N	N	N

Table 9.5: Results for Rule Application Control: Location Determination / Rule Application Strategy

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Rule Scheduling				
Form				
Implicit	Y	N	N	Y
Explicit	Y	Y	Y	Y
Rule Selection				
Explicit Condition	N	N	N	N
Non-deterministic	Y	N	N	N
Conflict Resolution	N	N	N	N
Interactive	N	N	N	N
Rule Iteration	recursion	recursion	recursion	
Phasing	Y	N	Y	Y

Table 9.6: Results for Rule Application Control: Rule Scheduling

9.3 Rule Organization

9.3.1 ATL

ATL offers several features in the area of rule organization. Often used ATL helpers can be included in libraries, which then can be used in ATL modules. Despite the name, modules themselves can not be reused. In ATL terminology, a module denotes the code for one model-to-model transformation (see [20]). Within a module, rule inheritance is available, if the 2006 version of the ATL compiler is used. This is described in Section 5.2.

The organizational structure of ATL modules is loosely based on the structure of the source metamodel. For one source model element, several target elements can be specified. However, ATL 2006 adds the ability to define more than one source element for each rule.

9.3.2 Kermeta

As full-featured programming language, Kermeta offers features for organizing code and therefore rules. The transformation can be organized in classes. Unlike ATL, the transformation code can also be split into several different files. Kermeta also supports multiple inheritance, which means that one class can be a subclass of more than one superclass (see [12]). Strictly spoken, the inheritance feature as found in Kermeta is not *rule inheritance*, because it acts on classes, not on operations. However, operations of a class are included in inheritance relationships. A subclass can therefore take advantage of the operations defined in the superclass.

The organizational structure of written Kermeta code is bound neither to the source nor the target metamodel. It lies entirely in the developer's responsibility how the transformation is structured.

9.3.3 SmartQVT

In the area of rule organization, SmartQVT offers several features. Modularity in terms of whole transformations are supported by the *access* and *extend* during the transformation definition, as illustrated in Listing 9.5. They keyword *access* in line 2 denotes that the operations available in the referenced transformation *UmlCleaning* are also available in this transformation. This is illustrated in line 6, where a new instance of the UmlCleaning transformation is created, and its *transform()* operation is called. As can be seen, the *access* mechanism is similar to import mechanisms in other programming languages. On the other hand, the *extends* mechanism is similar to inheritance in object-oriented programming languages.

Listing 9.5: Modularity Mechanisms in SmartQVT

```
1  transformation CompleteUml2Rdbms(in uml:UML,out rdbms:RDBMS
       )
2    access transformation UmlCleaning(inout UML),
3    extends transformation Uml2Rdbms(in UML,out RDBMS);
4
5  main() {
6
7      var tmp: UML = uml.copy();
8      var retcode := (new UmlCleaning(tmp))->transform(); //
           performs the "cleaning"
9      if  (not retcode.failed())
10          uml.objectsOfType(Package)->map packageToSchema()
11     else raise "UmlModelTransformationFailed";
12
13 }
```

Additionally, *rule inheritance* is also supported, which is very similar to rule inheritance in ATL. The inherited rule's code is executed first, followed by the

code of the inheriting rule. Further, merging of rules is also possible. Contrary to rule inheritance, here the code of the referenced rules is executed after the merging rule's code.

In general, the organizational structure of transformation code in SmartQVT is independent. However, because the mapping operations usually take source model elements as input parameter and target model element as output values, therefore the organization of the rules mostly reflects the organizational structure of the models involved.

9.3.4 ModelMorf

As all other languages, ModelMorf supports basic modularity mechanisms, in terms of importing rules from different source files. Using the *import* statement, another QVT transformation can be imported, and its rules can be used in *when* and *where* clauses. Besides that, no reuse mechanisms, such as rule inheritance are supported. The organizational structure of a rule file is independent from both source and target model.

9.3.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Modularity Mechanisms	Y	Y	Y	Y
Reuse Mechanisms				
Inheritance	Y	Y	Y	N
Logical Composition	N	N	N	N
Organizational Structure	source, independent	independent	independent	independent

Table 9.7: Results for Rule Organization

9.4 Source-Target Relationship

9.4.1 ATL

ATL can write both new target elements or use existing target elements. This is accomplished by two different execution modes [20]. In *default mode*, ATL creates new target elements as specified by the developer. In *refining mode*, the source elements are completely copied over to the target model, as long as no rule exists for the respective source element. This mode is useful for transformations between very similar metamodels, because the basic copying is done automatically by ATL.

9.4.2 Kermeta

Because Kermeta does not offer a mechanism for automatic model element creation, it does not support any features in this section. The developer has to allocate and create target elements manually. The missing built-in support for these features does not imply that such behavior is impossible in Kermeta. In fact, the developer can implement the desired behavior himself. For example, the manual creation of target model elements, as described in Section 9.1.2, can be interpreted as manual implementation of the feature *new target*. On the other side, it is also possible that the transformation simply performs changes in a source model loaded before, and then saves the updated model as output file.

9.4.3 SmartQVT

SmartQVT does not offer the possibility to update already created output models in any way. Change propagation as described by the features in Table 9.8 is only a part of the Relational QVT standard [17]. General change propagation only makes sense in relational transformation approaches, because imperative-based approaches only describe how a new output model is created from a given input model in a step-by-step manner. Similar as discussed in Section 9.4.2 for Kermeta, it may be possible to implement the features listed in Table 9.8 using SmartQVT.

9.4.4 ModelMorf

ModelMorf can create a new target model, but it is also possible to update existing model files. In-place updates are also supported by ModelMorf, as documented in the QVT standard [17].

9.4.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
New Target	Y	N	N	Y
Existing Target	Y	N	N	Y
Update	N	N	N	Y
In-Place	Y	N	N	Y

Table 9.8: Results for Source-Target Relationship

9.5 Incrementality

9.5.1 ATL

ATL does not support any of the features concerning incrementality. During every transformation run, the whole source model is read, and the target model is completely recreated. User created changes in the target model are not preserved if the transformation is run again.

9.5.2 Kermeta

Kermeta also does not offer mechanisms like source- or target incrementality by default. When loading external files containing model data, Kermeta always loads the whole file, which makes it impossible to implement source-incrementality. However, target incrementality could be simulated by loading the existing target model at the beginning of the transformation, and only update necessary target elements. The same way, preservation of user edits could be implemented as well.

9.5.3 SmartQVT

SmartQVT does not offer any incrementality features. Similar to Kermeta, some features may be implemented by the transformation developer by first loading the existing target model, and then perform the updates as desired.

9.5.4 ModelMorf

ModelMorf is the only transformation language evaluated in this thesis, which supports any features concerning incrementality. Target incrementality in fact is a part of the relational QVT standard, therein called *change propagation* [17]. Source incrementality and preservation of user edits in the target model are not supported.

9.5.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Target-Incrementality	N	N	N	Y
Source-Incrementality	N	N	N	N
Preservation of User Edits in the target	N	N	N	N

Table 9.9: Results for Incrementality

9.6 Directionality

9.6.1 ATL

ATL only offers unidirectional transformations.

9.6.2 Kermeta

Kermeta's extendable imperative approach allows a developer to exactly specify *how* a transformation is executed. Therefore it is thinkable that a multidimensional transformation can be implemented in Kermeta. In such case, the transformation has to be specified separately for each transformation direction needed, which means that multidirectionality is not achieved using exactly one logical set of transformation rules. In fact, there would be a set of rules for each transformation direction. Because of this, multidimensional transformations in Kermeta are not very practical. Although they can be realized, a transformation language with dedicated support for multidirectionality would be more feasible.

9.6.3 SmartQVT

Because of its imperative approach, SmartQVT basically only supports unidirectional model transformations. SmartQVT transformations specify their input and output model types in their header, therefore a simulation of a multidirectional transformation, like in Kermeta, is not possible.

9.6.4 ModelMorf

As an implementation of the OMG Relational QVT standard, multidirectionality is one of the key features of ModelMorf. As described before, transformations are defined in a declarative way, which does not contain any information on how the models involved are transformed in a step-by-step way. Instead, the relations of the model elements of the source and target models are described. Using this description, ModelMorf can execute the transformation based on the same rule file in any direction.

9.6.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Directionality	uni	uni (multi)	uni	multi

Table 9.10: Results for Directionality

9.7 Tracing

9.7.1 ATL

ATL includes dedicated tracing support. The tracing information is created automatically, as soon as a specific source model element is matched to a specific target model element. This is documented in the follow listing. In line 7, all elements in the *contains* relationship of the source model are copied to their traced counterparts in the *contains* relationship in the target model. ATL automatically resolves the tracing information, as long as the target element is the default target element in the rule in which it was created. An example for this behavior is seen in Listing 9.6. The default target element of a rule is always the first element in the *to* block of a matched rule (see [20]).

Listing 9.6: ATL Rule using Implicit Tracing

```
1 rule epcMM2adMM {
2   from
3     e:epcMM!epcMMContainer
4   to
5     a:adMM!adMMContainer (
6       name<-e.name,
7       contains<-e.contains
8     )
9 }
```

Additionally, ATL also offers the *resolveTemp* operation, which returns the created target model element from a given source model element and its target pattern name. resolveTemp only has to be used, if the target element wanted is not the default target element of a rule.

Listing 9.7: Usage of resolveTemp in ATL

```
1  rule r1 {
2    from
3      a_a:MM1!A
4    to
5      b_b:MM2!B(x<-a.name),
6      b_c:MM2!C(y<-a.size)
7  }
8
9  rule r2 {
10   from
11     f:MM1!F
12   to
13     g:MM2!G(h<-resolveTemp(f.a,b_c))
14 }
```

Listing 9.7 illustrates the usage of resolveTemp. First, the rule *r1* creates two target elements, which are instances of the classes B and C respectively. The

default target element of this rule is *b_b*, because it is listed first. The element *b_c* therefore is a non-default target element. In the rule *r2*, the attribute *h* of the target element *g* has to be set. Because the needed information for this in the source model is *f.a*, an instance of the class *A*, two target elements for this source element have been created beforehand. In this case, the second one is needed (*b_c*, as created in rule *r1*), which is a non-default target element. To access the tracing information for this, *resolveTemp* has to be called with the source model element (*f.a*) and the target element name in its creation rule (*b_c*).

9.7.2 Kermeta

Kermeta offers no dedicated tracing support, but it is possible to implement it. Listing 9.8 shows a simple tracing implementation, which is used in the *epc2ad* transformation.

Listing 9.8: Simple Tracing Framework in Kermeta

```
1  require kermeta
2  using kermeta::utils
3  using kermeta::standard
4
5  class Tracing<S,T>
6  {
7    reference src2tgt : Hashtable<S,T>
8
9    operation create() is do
10     src2tgt := Hashtable<S,T>.new
11   end
12
13   operation getTargetElem(src : S) : T is do
14     result := src2tgt.getValue(src)
15   end
16
17    operation storeTrace(src : S, tgt : T) is do
18     src2tgt.put(src, tgt)
19    end
20
21 }
```

This is a very simple implementation. It only supports exactly one target element for each source element. Also, backwards tracing from a target element to the corresponding source element is not included. These features could be implemented as well in a similar manner.

9.7.3 SmartQVT

SmartQVT has dedicated tracing support. The tracing information is built automatically, based on the input and output elements specified in the header of a mapping operation. According to the QVT standard [17], tracing information can be retrieved using one of three different resolving operations:

- *resolveone*. Looks for a target object created from a given source object.
- *invresolve*. Reverse resolve, looks for a source object created from a given target object.
- *resolveIn*. Looks for target objects created from a source object by a unique mapping operation.

Additionally, SmartQVT also supports late resolve, which behaves just like described above, with the exception that the resolving operation is performed at the end of the transformation.

9.7.4 ModelMorf

ModelMorf includes dedicated tracing support.

9.7.5 Summary

Feature	ATL	Kermeta	SmartQVT	ModelMorf
Dedicated Support	Y	N	Y	Y
Creation	Automatic		Automatic	Automatic
Storage Location	Separate		Separate	Separate

Table 9.11: Results for Tracing

9.8 Lessons learned

ATL supports many features listed in this evaluation. This is clearly a result of its hybrid approach, which incorporates declarative and imperative constructs. Most features which are needed for an average model transformation are implemented. Besides that, ATL is relatively easy to learn, because it just offers the most needed model transformation features in a relatively straight-forward way, and on other hand leaves out more sophisticated features, for example different rule selection algorithms or QVT-blackbox like mechanisms. In conclusion, ATL is able to cover a broad territory of model transformations, but complex transformations are harder to implement than in SmartQVT or Kermeta.

Kermeta as model transformation language supports a bit less features than the other languages out of the box. However, most features can be implemented in Kermeta itself. For larger transformations, Kermeta offers mechanisms for improved code reuse. Because of its imperative approach, tricky transformation problems can be solved easier than with ATL or ModelMorf. Also because of this, Kermeta is easier to learn for most developers compared to declarative languages. Narrowed down only to transformation languages, there may be more convenient languages than Kermeta. Because Kermeta is not only a

transformation language, but a general platform for model driven development, it may be a good choice for a complete MDE environment nevertheless.

SmartQVT is a very powerful transformation languages. It supports many features of this evaluation, and also covers mechanisms and techniques, which are not discussed in this thesis. The QVT blackbox mechanism for example allows a transformation to call arbitrary external code, which could be used to incorporate database access into a transformation. Another advanced feature of SmartQVT is the possibility to dynamically load, modify and execute transformations. As a downside of this, it takes a while for a developer to get used to SmartQVT. After that learning phase however, a developer can implement almost any transformation in SmartQVT in an elegant and efficient way.

ModelMorf, as implementation of the Relational QVT standard, follows a strictly declarative approach. Therefore, developers proficient in imperative programming, may need more training time compared to the other languages of this evaluation. ModelMorf's declarative approach has some advantages. Transformations can be specified elegantly in a very short time. Also, it is the only language in this evaluation which supports multidirectional model transformation. As a downside, more complex transformations, like *epc2ad*, are very difficult to implement. Therefore, ModelMorf is suited best for equivalent representation models which have to be transformed in more than one direction.

In general, no transformation language evaluated in this thesis offers considerably more features than any other language. Also, it would not make any sense to identify a *winning* candidate by summing up all features supported, because some features are more important than others. Other features simply denote the approach a specific language chooses. For example, Kermeta and SmartQVT do not support many features in *rule scheduling*, because of their overall imperative approach. Different rule selection mechanisms simply do not make sense in an imperative language.

Additionally, some features are not supported by any language evaluated. For example, no language supports source incrementality or dynamic mode restriction. Other features, which are not supported by a language may be implemented as part of the transformation, if they are necessary to develop the desired transformation. This is especially the case for Kermeta. Kermeta misses some typical model transformation language features, most notably support for tracing, but because of Kermeta's imperative approach, these features can easily implemented by the developer.

Despite the fact that no model transformation language is clearly overall better than another, still some languages are more suitable for a specific task compared to others. In general, if the transformation to implement includes structurally similar metamodels, declarative approaches may be a good choice. Most classes and relationships have directly corresponding model elements in the target model and can therefore easily be transformed. An example for this is model

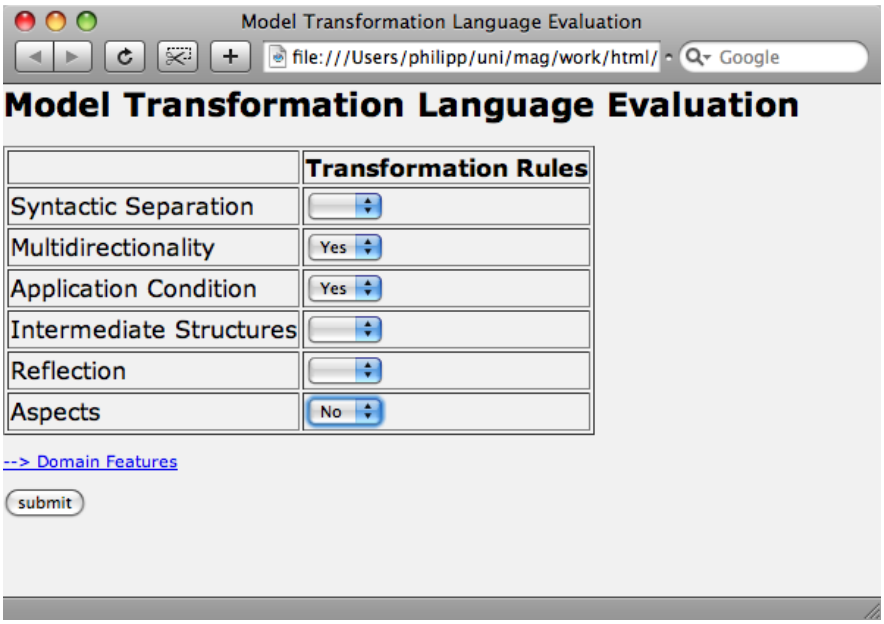

Figure 9.1: Choosing Features on the Language Evaluation Webpage

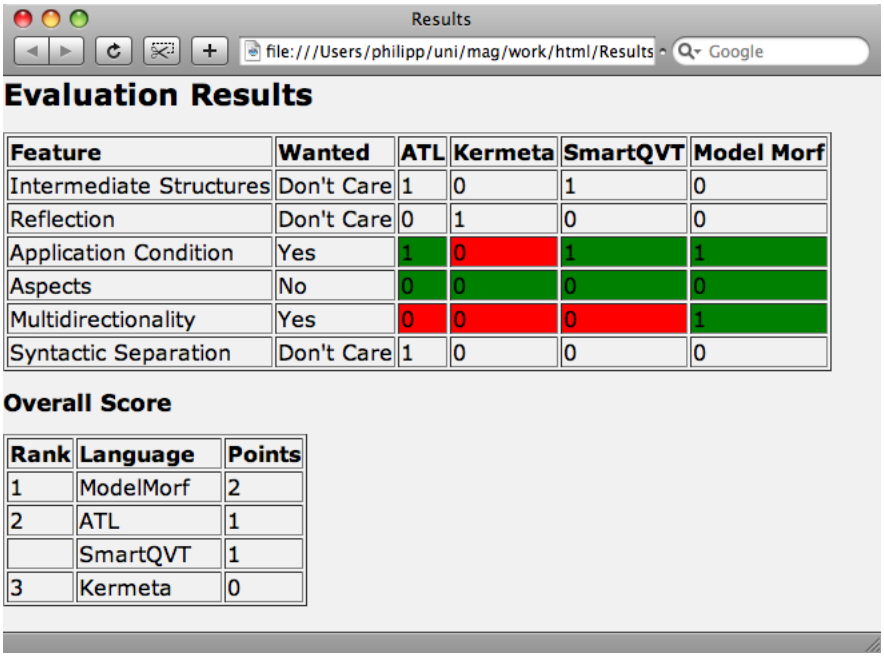

Figure 9.2: Results

refactoring, because the source and target metamodels are structurally equivalent. Metamodels which differ structurally, where concepts from one metamodel do not have direct equivalents in the other metamodel, are more complex to transform. For such situations, imperative approaches are more suitable, because they are more expressive than declarative approaches. Language migration (like transforming Event-driven Process Chains to Activity Diagrams) is an example usage scenario which is suited well for imperative transformation languages, because different languages may support certain features in a very different way, which has to be taken into account for the transformation. Hybrid approaches on the other hand aim to combine the advantages of imperative and declarative approaches. Hybrid approaches are therefore most suitable for transformations, which include a certain amount of structural similarity, but still include concepts which cannot be mapped to each other directly. For hybrid aproaches, formal refinement represents a suitable usage scenario, because the metamodels involved are structurally very similar, but the target model includes additional information, which in many cases has to be computed or compiled by the transformation.

In the end, the *best* model transformation language has to be chosen depending on the requirements of the given transformation and the infrastructure of the MDE environment used. In order to support the decision process, a website is provided [22]. All features evaluated in this thesis can be marked depending on the requirements, as shown in Figure 9.1. The support for the selected features in ATL, Kermeta, SmartQVT and ModelMorf is then shown in order to help picking out the most suited transformation language, which supports most of the desired features. In this case, the user is interested in a transformation language offering support for Application Conditions and Multidirectionality, but explicitly not Aspect-orientation. Other features included in the feature classification may be chosen on different subpages, as indicated by the *Domain Features* link seen in Figure 9.1. Figure 9.2 shows what the result for the query defined before looks like. A detailed overview of the languages' respective feature support is given. Features which are wanted by the user and supported by the language in question are marked green. If a wanted feature is missing in a transformation language, the respective field is colored red. If a feature is explicitly unwanted by the user, the fields indicating the languages' support is colored green if the feature is not supported in the respective language and red otherwise. Below the detailed result table, an overall score is given. For every supported feature, which is wanted by the user, the language is awarded one point. The points for every language are summed up. The language awarded the most points offers the best feature support according to the user's requirements. In the example shown in Figures 9.1 and 9.2, ModelMorf has the best coverage of the user's requirements because it supports both Application Conditions and Multidirectionality.

10 KLRT - Kermeta Library for Reusable Transformations

In this chapter, the Kermeta Library for Reusable Transformation patterns is explained.

10.1 Motivation

Many problems occurring in model transformation situations are very similar. For example, the deaggregation of one class in the source model to two classes with a relationship to each other in the target class is a very often used technique. Despite the frequency of occurrence of these problems, model transformation languages do not incorporate built-in support for solving this problems in an easy and elegant way. Developers have to implement solutions for these common transformation schemes by themselves, which leads to additional development effort, error-proneness and incompatibilities across transformations.

Kermeta is chosen as platform for KLRT, because it offers powerful language features, which are needed to implement such a library. Reflection and genericity for example are necessary to handle model elements with unknown properties at compile time. Furthermore, Kermeta is lacking some features, which are found in most other transformation languages, most notably support for tracing.

The overall goal of KLRT is to simplify model transformation development with Kermeta. Kermeta code using KLRT aims to be shorter, less verbose, and easier to understand.

10.2 Recurring Problems in Model Transformation

The transformation problems described here are based on [39], with tracing support added as additional pattern.

10.2.1 Tracing Support

Kermeta offers no built-in support for tracing. However, most model transformation problems need support for tracing. As a consequence, a tracing framework has to be implemented by the developer. Because this framework is most likely very similar for several transformations, it is very suitable for inclusion in KLRT.

10.2.2 Aggregation

Aggregation is the process of combining two distinct classes in the source model. Both classes may also have a relationship to each other. As result, the target model includes one class, which incorporates features of both source model classes.

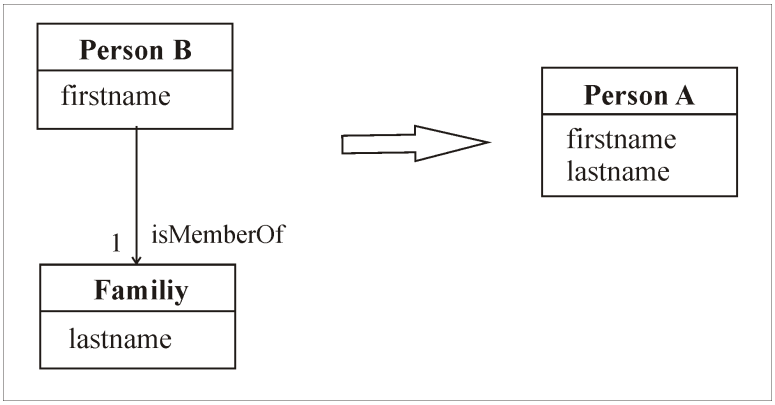

Figure 10.1: Aggregation

As shown in Figure 10.1, two classes exist in the source model. The class *PersonB* denotes a single person, with its first name as attribute. The other class, namely *Family* contains the family's name as attribute. An instance of *PersonB* refers to exactly one instance of *Family*. These two classes are transformed into one single class *PersonA*, which both includes the person's first name and the name of the family the person belongs to.

10.2.3 Deaggregation

Contrary to aggregation, deaggregation splits one source model class into two classes. The two resulting classes may also again have a relationship to each other.

Figure 10.2 illustrates the idea of deaggregation. One class, in this case *PersonA* has two attributes. In the target model, this results into two distinct classes, with one attribute copied from the source class respectively, namely *PersonB* and *Family*. Additionally, the togetherness of the transformed objects is indicated by the *isMemberOf* relationship. If multiple *PersonA* instances belonging to the same family are to be transformed, the *Family* object must only be created once. The different *PersonB* instances then have to reference the same *Family* object. In order to fulfill this requirement, the implementation of the transformation has to keep track of all transformed objects.

10.2.4 Collapse Hierarchy

A common technique used in object-oriented modeling is the usage of inheritance hierarchies. Given a specific metamodel which includes an inheritance relationship, another metamodel may specify the information given by the inheritance

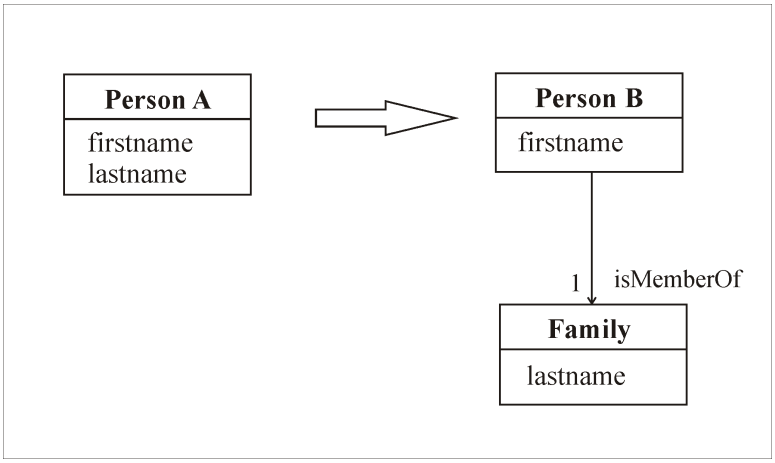

Figure 10.2: Deaggregation

relations as attribute of the mapped super class. In other words, instead of a super class and several subclasses, a model can contain one single class, with an attribute which specifies what subtype an instance of this class belongs to.

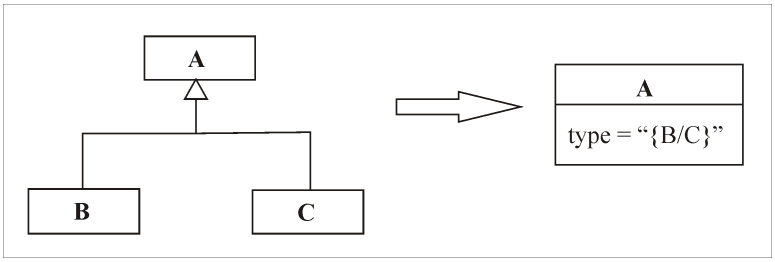

Figure 10.3: Collapse Hierarchy

Figure 10.3 illustrates the mechanism of collapsing hierarchies. The superclass A has two subclasses, B and C. In the target model, only the superclass A exists. The exact type of the object is specified by the attribute *type*.

10.2.5 Expand Hierarchy

Expanding hierarchies is the reverse operation to collapsing hierarchies. In this case, a subclass of a superclass is created in the target model, depending on the value of a certain attribute in the source model.

Figure 10.4 shows how the expanding of hierarchies in model transformations works. In the source model, the class A has a attribute *type*. Depending on the value of this attribute, an object is transformed either to an instance of subclass B or C during the transformation.

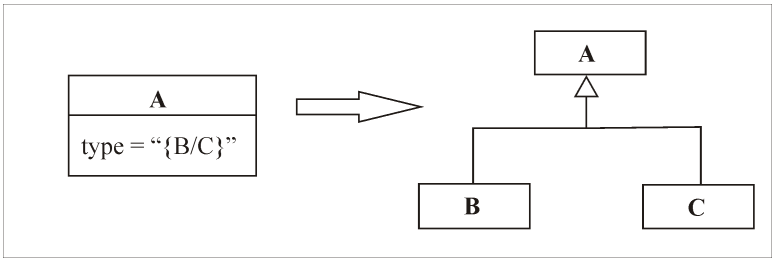

Figure 10.4: Expand Hierarchy

10.2.6 Reverse Property

Reverse property is another frequently occurring pattern in model transformations. In this case, the relationship between two model elements in the source model is reversed in the target model. No reverse operation exists for reverse property, because the same operation can be applied to opposing metamodel fragments.

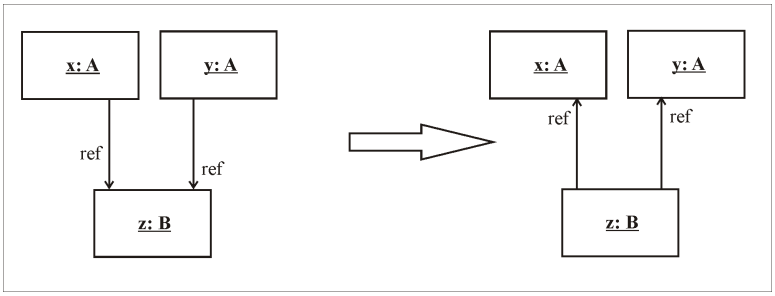

Figure 10.5: Reverse Property

For example, as shown in Figure 10.5, in the source model an object of class A has a reference to an object of class B, where multiple A's may reference one B. In the target model on the other hand, an object of class B itself references to multiple instances of class A.

10.3 Language Requirements

In order to implement the aforementioned transformation schemes, the model transformation language of choice must support several language features.

10.3.1 General Support for Implementing Libraries

The chosen transformation language has to offer the possibility to create user-defined libraries, which can easily be used by different transformations. Most languages support this feature. For example, in ATL it is possible to create separate library files including helper rules, which can be called by other transformations. Other approaches, such as Triple Graph Grammars or QVT

Relational do not offer such possibilities.

Kermeta supports implementing libraries by importing Kermeta files into other Kermeta files. This is illustrated by the following example in Listing 10.1.

Listing 10.1: Trivial Kermeta library (File trivlib.kmt)

```
1  package at::adore::kermeta::trivlib;
2
3  require kermeta
4
5  class t
6  {
7    operation f():Void is do
8      ...
9    end
10 }
```

Listing 10.2: Using Trivlib in a Transformation

```
1  package at::adore::kermeta::test;
2
3  require kermeta
4  require "trivlib.kmt"
5
6  using at::adore::kermeta::trivlib
7
8
9  class Main
10 {
11   operation main() : Void is do
12
13     var x: t init t.new
14     t.f()
15
16   end
17 }
```

As can be seen, the file *trivlib.kmt* defines a package named *at::adore::kermeta::trivlib*. The usage of packages in Kermeta is not mandatory, but recommended, because in this way, the library namespace is clearly separated from the rest of the transformation. Inside the library package, the class *t* is defined, containing the operation *f()*.

A transformation in Kermeta, like the transformation shown in Listing 10.2, can now access the defined class and its operation by including the Kermeta file containing the library code, namely *trivlib.kmt*. Additionally, the package has to be imported with the *using* statement. After that, the class *t* is instantiated and the operation *f()* is called on its newly created instance.

10.3.2 Access to Meta Classes/Reflection

In most cases, the exact metamodel classes used in transformations are not known at the design time and compile time of the library. Otherwise, it would not be possible for the same code, namely the library, to operate on different classes. Therefore, a reflection mechanism is needed which provides access to meta classes, their attributes and relationships. For example, it might be necessary to set an attribute value, given only the attribute name as string and the desired value as generic object.

Genericity as technique is also useful, but in most cases may be substituted by using reflection meta classes. In general, genericity may be used if already existing objects of a certain class are passed to an operation, whereas reflection is useful if a new object has to be instantiated.

10.4 Usage

In order to use KLRT, a transformation has to include the library's main file and import its package namespace, as illustrated by Listing 10.3:

Listing 10.3: Importing KLRT into a Transformation

```
1  package at::adore::kermeta::translib::test;
2
3  require kermeta
4  require "klrt/klrt.kmt"     // require library file
5
6  using at::adore::kermeta::translib   // import package
7
8
9  class Main
10 {
11   ...
12 }
```

From the point of view of a transformation developer, the library is accessed only by instantiating one specific class, namely *at::adore::kermeta::translib::transformations*. This class contains all reusable transformation schemes, which can be used by calling the respective operations. Because in many cases, more than one result object may be returned, KLRT uses a OrderSet as result value, which includes multiple objects. The individual objects can be accessed by using the method *at(index)*. Because the resulting OrderedSet may only include *Objects*, the method *asType(type)* has to be used in order to cast the result object to the desired type.

Listing 10.4 illustrates how the operation *deaggregate* can be used in order to transform a *PersonA* into a *PersonB* and a *Family*. First, the transformation library and the initial person are instantiated. For PersonA, both a firstname and lastname are set. After that, the result *OrderedSet* is instantiated, and the operation *deaggregate* is called. The parameters of this operation are as followed:

Listing 10.4: Calling an operation of KLRT

```
1  operation f():Void is do
2    // instantiate lib
3    var lib: transformations init transformations.new
4
5    // instantiate a person
6    var x1: PersonA init PersonA.new
7    x1.firstname="Jack"
8    x1.lastname="White"
9
10   // result variable
11   var result1: OrderedSet<Object>
12
13   // call deaggregate
14   result1:=lib.deaggregate(x1,"lastname",PersonB,"
        isMemberOf",Familie,"name")
15
16   // read results
17   var y1:PersonB init result1.at(0).asType(at::adore::
        kermeta::translib::test::PersonB)
18   var z1:Familie init result1.at(1)asType(at::adore::
        kermeta::translib::test::Familie)
19
20
21 end
```

- *x1.* The object, which is to be deaggregated, in this case the *PersonA* with both a *firstname* and *lastname*.
- *lastname.* Attribute of the source class, which is copied to a new instance of the second target class.
- *PersonB.* The first target class. A new object is created from this class, which will be the first object in the resulting OrderedSet.
- *isMemberOf.* A relationship attribute of the first target class which points to the newly created instance of the second target class.
- *Familie.* The second target class. A new object ist created from this class, which will be the second object in the resulting OrderedSet.
- *name.* The identifying attribute of the second target class. The value of the second parameter will be copied to this attribute.

10.5 Implementation

In this sections, the implementation of the transformation problems described in Section10.2 and additional helper operations and classes are explained.

10.5.1 Helper Operations - helper.kmt

The file *helper.kmt* includes several operations, which are called by the main library operations. Its layout is similar to the main library, as it only includes one class, which then contains the needed operations. The helper functions are not intended to be called by a user transformation.

Operation getAttrFromClass

In order to get and set a property of an object when the property is only known at runtime, Kermeta's reflection features have to be used. Both *get* and *set* operations in Kermeta need the wanted property's meta-class as parameter, as illustrated by Listing 10.5.

Listing 10.5: Get and Set Methods of Kermeta [37]

```
1 method get(~property : kermeta::reflection::Property) :
    kermeta::reflection::Object from kermeta::reflection::
    Object
2
3 method ~set(~property : kermeta::reflection::Property,
    element : kermeta::reflection::Object) : Void from
    kermeta::reflection::Object
```

The tilde character ∼ is used to escape keywords in Kermeta, so they can be used as operation names or variable names [12].

Listing 10.6: Operation getAttrFromClass

```
1 operation getAttrFromClass(cl:kermeta::reflection::Class,
    attr:String):kermeta::reflection::Property is do
2
3   cl.ownedAttribute.each {a|
4     if (a.name.equals(attr)) then
5       result:=a
6     end
7   }
8
9 end
```

Because KLRT often wants to set properties, which are only known by their name as *String*, the operation *getAttrFromClass* (see Listing 10.6) is provided, which returns the *kermeta::reflection::Property* object of a given class and the property's name as String value.

The operation steps through all owned attributes of the class *cl*. If a property is found, which has the same name as given in the parameter *attr*, this property is returned. If no matching property is found, *void* is returned implicitly. Using the *getAttrFromClass* operation, it is easily possible to to call the *get* and *set* methods provided by Kermeta.

Operation findObject

KLRT keeps a set of objects, which were already created by the library. This is necessary to avoid the double creation of elements, especially when using deaggregation. For example, if two persons of the same family are deaggregated, they both have to refer to the same newly created family object.

Listing 10.7: Operation findObject

```
1  operation findObject(cl:kermeta::reflection::Class, attr:
       String, attrVal:Object, elementSet:Collection<Object>):
       Bag<Object> is do
2
3    var b:Bag<Object> init Bag<Object>.new
4    var prop:kermeta::reflection::Property init
       getAttrFromClass(cl, attr)
5
6    elementSet.select{e|e.getMetaClass==cl}.each {a|
7
8      if (a.get(prop).getMetaClass.name=="ReflectiveSequence"
         ) then
9        var rs:kermeta::standard::Collection<kermeta::
           reflection::Object>
10       rs?=a.get(prop)
11
12       rs.each {item|
13         if (item.equals(attrVal) or item==attrVal) then
14           b.add(item)
15         end
16       }
17     else
18       if (a.get(prop)==attrVal or a.get(prop).equals(
           attrVal)) then
19         b.add(a)
20       end
21     end
22   }
23
24   result:=b
25
26 end
```

The operation *findObject*, as seen in Listing 10.7, looks up the given Collection of elements for an object of the class *cl*, with an attribute named *attr* with the value *attrVal*. Additionally, KLRT detects whether the given property to search is a collection (line 8). In that case, every matching item in this collection is returned. This behavior is needed by the implementation of *reverseProperty*.

Operation getInstance

The operation *getInstance* (see Listing 10.8) is a wrapper operation, which calls the aforementioned operation *findObject*. If no object matching the criteria given is found, a new object is instantiated. The idea of this mechanism is similar to singletons, although at a different level. Instead of only one object of a given class, several objects of the same class are allowed. However, only one object with a certain attribute value is permitted.

Listing 10.8: Operation getInstance

```
1  operation getInstance(cl:kermeta::reflection::Class, attr:
       String, attrVal:Object, elementSet:Collection<Object>):
       Object is do
2
3    var b:Bag<Object> init findObject(cl,attr,attrVal,
         elementSet)
4    var o:Object
5
6    if (b.size==0) then
7      o:=cl.new
8    else
9      o:=b.one
10   end
11
12   result:=o
13
14 end
```

Operation recurseAddToSet

As mentioned before, KLRT keeps a collection of already transformed objects. The operation *recurseAddToSet*, as seen in Listing 10.9, is intended to be called to add newly created elements to the collection of all elements. The operation recursively follows all referenced objects and adds them to the set.

Listing 10.9: Operation recurseAddToSet

```
1  operation recurseAddToSet(o:Object, elementSet:Collection<
       Object>): Void is do
2
3    if (o!=void) then
4
5      var val:Object
6      if (o.getMetaClass.name=="ReflectiveSequence") then
7        var rs:kermeta::standard::Collection<kermeta::
             reflection::Object>
8        rs?=o
9        rs.each {i|
10         recurseAddToSet(i,elementSet)
11       }
12     end
```

```
13      elementSet.add(o)
14
15      o.getMetaClass.ownedAttribute.each {a|
16        val:=o.get(a)
17
18        if (val!=void) then
19          var type:String init val.getMetaClass.name
20
21          if (type!="String" and type!="Integer" and type!="
               Boolean" and not elementSet.contains(val)) then
22            recurseAddToSet(val,elementSet)
23          end
24        end
25      }
26    end
27
28  end
```

First, the operation checks, whether the given object is some type of collection, which can be found out using *.getMetaClass.name=="ReflectiveSequence"*. If this is the case, *recurseAddToSet* is called recursively on all elements contained in this collection. After that, the special handling of collections is finished and the object given itself is added to the element collection. After that, all attributes of the object's class are iterated over. If the thereby referenced object is not an instance of a basic class, namely String, Integer, or Boolean, and the object is not contained in the element collection, the operation is called recursively. The not-containment of the object in question in the element collection is the main termination condition of the recursion. However, this also means that elements with updated references are not searched.

Operation createTracingHelper

Listing 10.10: Operation createTracingHelper and Wrapper Class

```
1  operation createTracingHelper(b:Bag<Object>):TracingHelper
      is do
2      result:=TracingHelper.new
3      result.b:=b
4  end
5
6  ...
7
8  class TracingHelper {
9
10   attribute b:Bag<Object>
11
12 }
```

KLRT uses a simple Hashtable-based tracing framework. *Bags* are used as keys and values for the hashtable, where each bag may contain multiple elements.

In this way, support for *m:n* tracing information is introduced. Due to a bug in Kermeta, it is not possible to store *Bag* objects as hashtable values. Therefore, KLRT provides a wrapper class, which is actually used for keys and values in the tracing hashtable. For the convenient creation of wrapper elements, the operation *createTracingHelper* (see Listing 10.10) is provided.

As can be seen, the wrapper class *TracingHelper* only contains one attribute of the type *Bag<Object>*. The operation *createTracingHelper* simply instantiates a new object of the wrapper class and sets the bag attribute accordingly to the given parameter.

10.5.2 Main Library - klrt.kmt

Tracing Support

Tracing support in KLRT is implemented in the file klrt.kmt, by the class *Tracing*, as shown in Listing 10.11.

Listing 10.11: Class Tracing

```
1  class Tracing<S,T>
2  {
3    reference src2tgt : Hashtable<S,T>
4
5    operation create() is do
6      src2tgt := Hashtable<S,T>.new
7    end
8
9    operation getTargetElem(src : S) : T is do
10     result := src2tgt.getValue(src)
11   end
12
13
14    operation storeTrace(src : S, tgt : T) is do
15     src2tgt.put(src, tgt)
16    end
17
18 }
```

This is basically a very simple implementation of tracing facilities. It only supports exact 1:1 mappings by using a Hashtable. Although this class may be instantiated and used by transformations, transformation developers are encouraged to use the wrapper methods provided in the class *transformations*. All KLRT operations automatically store their tracing information using the class described here. The limitation of only be able to store 1:1 mappings is circumvented by using *Bag*s of objects as keys and values for the Hashtable. In this way, several objects of both source and target model may correspond to each other. Transformation developers may use the operations *resolve* and *invresolve* of the class *transformations* in order to access tracing information.

Listing 10.12 shows the implementation of the operation *resolve*. The inverse operation, *invresolve* is implemented exactly the same way, only keys and values are swapped in their usage. Notably, the resolving feature of the underlying tracing framework is not used, because it can not resolve elements inside the key and value bags. Therefore, *resolve* directly accesses the tracing hashtable, and searches every key bag for an object matching to the parameter *o*. If a match is found, the whole target bag is returned.

Listing 10.12: Operation resolve

```
1  operation resolve(o:Object):Bag<Object> is do
2
3    var ht:Hashtable<TracingHelper,TracingHelper> init
        tracingLinks.src2tgt
4    ht.keys.each {i|
5      i.b.each {b|
6        if (b==o) then
7          result:=ht.getValue(i).b
8        end
9      }
10   }
11
12 end
```

Aggregation

The concept of aggregation is already described in Section 10.2.2. The operation *aggregate* implements this behaviour in KLRT (see Listing 10.13). The operation takes the following parameters:

- *fromClass1: A*. The first source object, of type *A*.
- *fromAttr1: String*. The attribute of class A which is copied to *toAttr1* in the target class.
- *fromClass2: B*. The first source object, of type *B*.
- *fromAttr2: String*. The attribute of class B which is copied to *toAttr2* in the target class.
- *toClass: Class*. The target class. The operation creates a new instance of this class.
- *toAttr1: String*. First target attribute.
- *toAttr2: String*. Second target attribute.

Listing 10.13: Implementation of Aggregation

```
1  operation aggregate<A,B>(fromClass1:A, fromAttr1:String,
      fromClass2:B, fromAttr2:String, toClass:Class, toAttr1:
      String, toAttr2:String): OrderedSet<Object> is do
2
3    initialize()
4
5    var h:helper init helper.new
6
7    var to:Object init toClass.new
8
9    var from_a1:kermeta::reflection::Property init h.
       getAttrFromClass(A,fromAttr1)
10   var from_a2:kermeta::reflection::Property init h.
       getAttrFromClass(B,fromAttr2)
11
12   var to_a1:kermeta::reflection::Property init h.
       getAttrFromClass(toClass,toAttr1)
13   var to_a2:kermeta::reflection::Property init h.
       getAttrFromClass(toClass,toAttr2)
14
15   to.~set(to_a1,fromClass1.get(from_a1))
16   to.~set(to_a2,fromClass2.get(from_a2))
17
18   h.recurseAddToSet(to,elementSet)
19
20   result:=OrderedSet<Object>.new
21   result.add(to)
22
23 end
```

As with nearly all operations of the main library, first the operation *initialize()* is called, in order to ensure the element collection of the library has been instantiated. After that, the helper class is instantiated, because some of its operations are used later on in this operation. In line 7, a new instance of *toClass* is created, which will later be filled and returned. In lines 9 to 13, the operation *getAttrFromClass* is called to retrieve the property objects, which are needed to get and set the desired attribute values. Lines 15 and 16 read the from-attribute values of the from-objects, and set these values accordingly in the target class. Finally, the created element is added to the library's element collection, and the result value is constructed.

Deaggregation

Deaggregation in KLRT is also implemented using both reflection and genericity. The resulting OrderedSet contains two objects. The operation *deaggregate* takes the following parameters:

- *fromClass: A*. The source object, of generic type A

- *attr: String.* Attribute of the source class, which corresponds to the identifying attribute of the second target class
- *toClass: Class.* First target class
- *ref: String.* Name of the property of the first target class, which refers to the second target class
- *attrToClass: Class.* Second target class. If no instance of this class with the value contained in the attribute *attr* of the source object exist, a new instance of this class will be created. Otherwise, the already existing instance will be referenced and returned.
- *attrTo: String.* Identifying attribute of the second target class

Listing 10.14: Implementation of Deaggregation

```
1  operation deaggregate<A>(fromClass:A, attr:String, toClass:
       Class, ref:String, attrToClass:Class, attrTo:String):
       OrderedSet<Object> is do
2
3    initialize()
4    var h:helper init helper.new
5    var a:kermeta::reflection::Property
6
7    a:= h.getAttrFromClass(A,attr)
8
9    var attrToProp:kermeta::reflection::Property init h.
       getAttrFromClass(attrToClass,attrTo)
10   var refProp:kermeta::reflection::Property init h.
       getAttrFromClass(toClass,ref)
11
12   var attrToClassInstance:Object init h.getInstance(
       attrToClass, attrTo, fromClass.get(a), elementSet)
13
14   var toClassInstance:Object init toClass.new
15
16   toClassInstance.~set(refProp,attrToClassInstance)
17   attrToClassInstance.~set(attrToProp,fromClass.get(a))
18
19   h.recurseAddToSet(toClassInstance, elementSet)
20   h.recurseAddToSet(attrToClassInstance, elementSet)
21
22   result:=OrderedSet<Object>.new
23   result.add(toClassInstance)
24   result.add(attrToClassInstance)
25
26 end
```

The implementation of *deaggregation* is shown in Listing 10.14. Like in all other main operations of KLRT, the library is ensured to be initialized by calling

the operation *initialize*. In lines 5 and 7, the wanted attribute of the source class is prepared to be read by getting its reflective *Property* object. In lines 9 and 10, the needed attributes for both target classes are instantiated the same way. After that, the instance of the second target class is prepared by calling the helper operation *getInstance*, and additionally an instance of the first target class is created. In lines 16 and 17, the reference of the first target class to the second target class and the identifying attribute of the second target class are set. Eventually, both created objects are added to the library's element collection and the resulting OrderedSet is constructed and returned.

Collapse Hiearchy

Because the collapsing of hierarchies only operates on one specific instance of a class, it is sufficient that only one transformed object is returned. Therefore, the return value of the operation simply is *Object*, and not *OrderedSet*. The operation *collapseHierarchy* takes the following parameters:

- *fromObject:Object*. The object in the source model.
- *toClass:Class*. The target class, of which a new instance will be created.
- *toAttr:String*. The target attribute, which will contain the type.
- *classTable:Hashtable<kermeta::reflection::Class,String>*. A Hashtable containing the mapping of which source class corresponds to which target attribute String value. If this parameter is *void*, the target attribute will contain the source class name as String value.

Listing 10.15: Implementation of Collapse Hierarchy

```
1  operation collapseHierarchy(fromObject:Object, toClass:
       Class, toAttr:String, classTable:Hashtable<kermeta::
       reflection::Class,String>): Object is do
2
3    initialize()
4    var h:helper init helper.new
5
6    var mcl:kermeta::reflection::Class init fromObject.
       getMetaClass
7    var o:Object init toClass.new
8    var prop:kermeta::reflection::Property init h.
       getAttrFromClass(toClass, toAttr)
9
10   if (classTable==void) then
11     o.~set(prop, mcl.name)
12   else
13     o.~set(prop,classTable.getValue(mcl))
14   end
15   h.recurseAddToSet(o,elementSet)
16   result:=o
17 end
```

Listing 10.15 shows the implementation of *collapseHierarchy*. In lines 3 and 4, the library is initialized, and the helper class is instantiated. Line 6 creates a variable containg the meta class of the source object, which will be needed later on. In lines 8 and 9, the target object is instantiated as object of the given target class. In lines 11 - 15, the target attribute is set, either to the string value of the source class, or to the corresponding value specified in the Hashtable parameter.

Expand Hierarchy

The operation *expandHierarchy* performs the opposite to the operation *collapseHierarchy*. It is implemented using both reflection techniques and genericity. The parameters are as follows:

- *fromObject:A*. This is the source object, of the generic type *A*.
- *fromAttr:String*. The attribute containg the subtype information.
- *classTable:Hashtable<String, Class>*. A Hashtable containg the mapping between the String value of the attribute *fromAttr* and the corresponding classes. Contrary to *collapseHierarchy* this parameter must not be *void*.

Listing 10.16: Implementation of Expand Hierarchy

```
1  operation expandHierarchy<A>(fromObject:A, fromAttr:String,
       classTable:Hashtable<String, Class>): Object is do
2
3    initialize()
4    var h:helper init helper.new
5
6    var prop:kermeta::reflection::Property init h.
       getAttrFromClass(A,fromAttr)
7
8    if (classTable.containsKey(fromObject.get(prop).toString
       ())) then
9      var mcl:kermeta::reflection::Class init classTable.
         getValue(fromObject.get(prop).toString())
10
11     result:=mcl.new
12   else
13     raise "Attribute␣"+fromAttr+"␣not␣found␣in␣matching␣
         Table"
14   end
15
16 end
```

Listing 10.16 shows the implementation of the operation *expandHierarchy*. Lines 3 and 4 again initialize the library and instantiate the helper class. In Line 6, the property object for the type attribute of the source class is retreived. Line 8 makes sure, that the value of the type attribute has a corresponding class in the mapping Hashtable. If so, in line 9 an object is created from this class, and

is set as result value for the operation in line 11.

Contrary to *collapseHierarchy*, the Hashtable parameter is not optional in *expandHierarchy*, because it is not possible to create an instance of a class, which is only known by its name as String. The reverse operation, getting a classes' name as String is easily possible via *object.getMetaClass.name*.

Reverse Property

The implementation of reverse property uses both reflection and genericity. The parameters are as follows:

- *fromObject:A*. Source object, which is referenced by one or more other source objects of type *otherClassSrc*.
- *fromRef:String*. The reference property's name of class *otherClassSrc*.
- *otherClassSrc:Class*. The class, which objects refer to *fromObject*.
- *toClass:Class*. Target class corresponding to *fromObject*.
- *toRef:String*. The reference property's name of class *toClass*.
- *otherClassTgt:Class*. The class, which objects are refered by the new instance of *toClass*.

Listing 10.17: Implementation of Reverse Property

```
1  operation reversePropery<A>(fromObject:A, fromRef:String,
      otherClassSrc:kermeta::reflection::Class,toClass:kermeta
      ::reflection::Class,toRef:String,otherClassTgt:kermeta::
      reflection::Class):OrderedSet<Object> is do
2
3    initialize()
4    var h:helper init helper.new
5
6    var res:OrderedSet<Object> init OrderedSet<Object>.new
7    var src_bag:Bag<Object> init Bag<Object>.new
8    var tgt_bag:Bag<Object> init Bag<Object>.new
9
10   var ref:kermeta::reflection::Property init h.
        getAttrFromClass(otherClassSrc,fromRef)
11
12   var b:Bag<Object> init h.findObject(otherClassSrc,fromRef
        ,fromObject,sourceModel)
13
14   src_bag.add(fromObject)
15   src_bag.add(b)
16   var new1:Object init instantiate(toClass,fromObject)
17
18   var ref2:kermeta::reflection::Property init h.
        getAttrFromClass(toClass,toRef)
```

```
19
20    tgt_bag.add(new1)
21    res.add(new1)
22
23    var revSet:kermeta::standard::Collection<kermeta::
          reflection::Object>
24    revSet?=new1.get(ref2)
25
26    b.each {item|
27      var new2:Object init instantiate(otherClassTgt,item)
28      tgt_bag.add(new2)
29      res.add(new2)
30      revSet.add(new2)
31    }
32
33    h.recurseAddToSet(new1,elementSet)
34    tracingLinks.storeTrace(h.createTracingHelper(src_bag),
          createTracingHelper(tgt_bag))
35    result:=res
36
37  end
```

In lines 3-8 of Listing 10.17, the library is initialized, the helper class instantiated, and the collection objects for tracing and the operation result are created. After that, the source reference property object is fetched in line 10. In line 12, the operation *findObject* is used to retrieve all objects of the element set, which point to *fromObject* via their specified reference property. Tracing information is built up in lines 14 and 15. The first target object is created in line 16, and its reference property is fetched after that. In line 21, the newly created target object is added to the resulting OrderedSet, in order to make sure it is always the first item of the collection. In lines 23-31, a new instance of *otherClassTgt* is created for every item which references to *fromObject* in the source model. A reference from the first created object to every secondary object is created in line 30, by adding the newly created element to the collection *revSet* containing the referenced objects of the new instance of *toClass* via the reference *toRef*. Eventually, all newly created items are saved in the library's element set and the created tracing links are stored.

10.6 KLRT in Action

In order to evaluate how useful KLRT may be in practice, an example transformation is provided in this section. The metamodels used in this transformation are seen in Figures 10.6 and 10.7. As can be seen, no real world entities are used in these metamodels, only placeholder classes. Despite being relatively small, the transformation from *testSource* to *testTarget* uses three of the aforementioned transformation patterns:

- *Aggregation.* The classes A and B are aggregated to the class A in the target model.

- *Collapse hierarchy.*The source metamodel uses generalization to distinguish different types of C's, namely the subclasses E and F. In the target metamodel, only the class C exists. It uses the attribute *type* in order to identify different subtypes.
- *Reverse property.* In the source metamodel, Cs may reference to multiple Ds, whereas in the target metamodel Ds reference Cs.

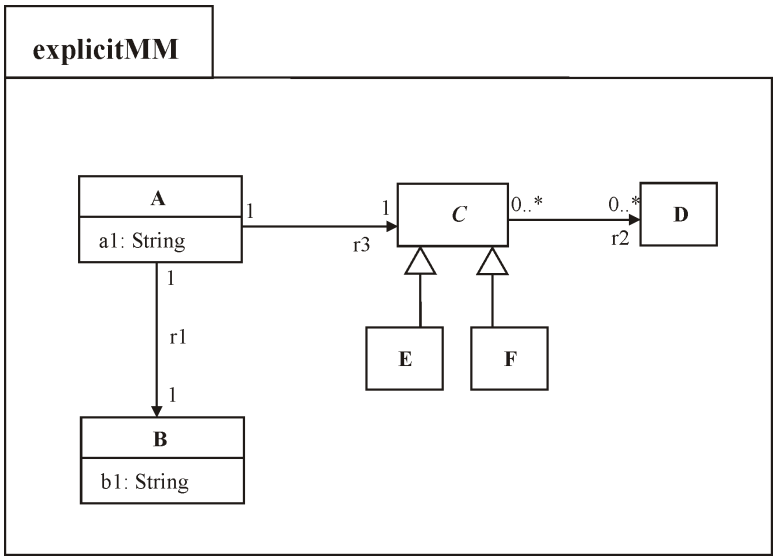

Figure 10.6: Language A Explicit Metamodel

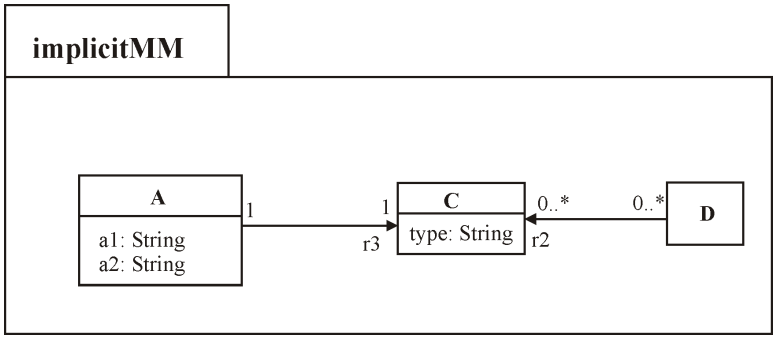

Figure 10.7: Language A Implicit Metamodel

Apart from using the patterns, the transformation requires only little transformation logic. Only the relationship *r3* is not covered by any of the transformation patterns, although the inspection of tracing information is necessary. Also the other transformation direction is implemented and discussed, because the reverse transformation patterns are used.

In the following, the transformation is implemented twice for both directions. First, plain Kermeta is used to transform *testSource* to *testTarget*. After that,

the same transformation is implemented using KLRT. Additionally, the direction from *testTarget* to *testSource* is also implemented using both Kermeta and KLRT. Only the transformations themselves are discussed, because loading and saving of models is done exactly the same way in all variations.

10.6.1 Case Study 1: From Explicit to Implicit Representation

Implementation with Plain Kermeta. Listing 10.18 shows the implementation of the sample transformation using plain Kermeta. As basic pattern, *select* and *each* is used to iterate over all elements which are transformed the same way. All new target elements have to be instantiated by the developer. In order to transform the relationship *r2* correctly, tracing information has to be kept. This is done using a simple Hashtable, because no *m:n* source-target mappings are needed in this transformation.

Listing 10.18: Explicit2Implicit: Implementation using plain Kermeta

```
1  operation transformPlain(src:testSource::container):
       testTarget::container is do
2
3    var resCont:testTarget::container init testTarget::
       container.new
4    var ht:Hashtable<Object,Object> init Hashtable<Object,
       Object>.new
5
6    src.items.select{x|x.isInstanceOf(testSource::A)}.each {i
       |
7      var a:testTarget::A init testTarget::A.new
8      a.a1:=i.asType(testSource::A).a1
9      a.a2:=i.asType(testSource::A).r1.b1
10     resCont.items.add(a)
11     ht.put(i,a)
12   }
13
14   src.items.select{x|x.isInstanceOf(testSource::C)}.each {i
       |
15     var c:testTarget::C init testTarget::C.new
16     c.type:=i.getMetaClass.name
17     resCont.items.add(c)
18     ht.put(i,c)
19   }
20
21   src.items.select{x|x.isInstanceOf(testSource::D)}.each{i|
22     var d:testTarget::D init testTarget::D.new
23     resCont.items.add(d)
24     ht.put(i,d)
25   }
26
27   src.items.select{x|x.isInstanceOf(testSource::A)}.each{i|
28     var a:testTarget::A
29     var c:testTarget::C
```

```
30      a:=ht.getValue(i).asType(testTarget::A)
31      c:=ht.getValue(i.asType(testSource::A).r3).asType(
            testTarget::C)
32      a.r3:=c
33    }
34
35    src.items.select{x|x.isInstanceOf(testSource::C)}.each{i|
36      var d:testTarget::D
37      var c:testTarget::C
38      c:=ht.getValue(i).asType(testTarget::C)
39      d:=ht.getValue(i.asType(testSource::C).r2.at(0)).asType
            (testTarget::D)
40      d.r1.add(c)
41    }
42
43    result:=resCont
44
45  end
```

Implementation with KLRT. Listing 10.19 shows how the demonstrative transformation may be implemented using KLRT. In lines 3-5, the library and the result object are instantiated and initialized. The following code blocks illustrate, how KLRT is typically used. All source elements, which are needed as source objects for the desired patterns are selected and iterated over (lines 8, 13, 22). After that, the mapping operation is called, and the created target elements are added to the target model container. The transformation parts, which are not covered by any pattern supported by KLRT are executed at the end in lines 29-35. In this case, the relationship *r3* has to be transformed. This is done by iterating over all *A*s in the source model. For every *A*, the counterpart *A* in the target model is resolved (line 32). The corresponding *C* at the other end of the relationship *r3* is also found by inspecting the tracing information using *resolve*. At line 34, the relationship in the target model is set.

Listing 10.19: Explicit2Implicit: Implementation using KLRT

```
1  operation transform(src:testSource::container): testTarget
       ::container is do
2
3    var lib: transformations init transformations.new
4    var resCont:testTarget::container init testTarget::
         container.new
5    lib.setSourceModel(src)
6
7    var c:testTarget::C
8    src.items.select{x|x.isKindOf(testSource::A)}.each {i|
9      var result1: OrderedSet<Object> init lib.aggregate(i.
           asType(testSource::A),"a1",i.asType(testSource::A).
           r1.asType(testSource::B),"b1",testTarget::A,"a1","a2
           ")
10     resCont.items.add(result1.at(0).asType(testTarget::A))
```

```
11    }
12
13    src.items.select{x|x.isKindOf(testSource::E)}.each {i|
14      var result1: Object init lib.collapseHierarchy(i.asType
          (testSource::E),testTarget::C,"type",void)
15      resCont.items.add(result1.asType(testTarget::C))
16    }
17
18    src.items.select{x|x.isKindOf(testSource::F)}.each {i|
19      var result1: Object init lib.collapseHierarchy(i.asType
          (testSource::F),testTarget::C,"type",void)
20      resCont.items.add(result1.asType(testTarget::C))
21    }
22    src.items.select{x|x.isInstanceOf(testSource::D)}.each{i|
23      var res:OrderedSet<Object> init lib.reversePropery(i.
          asType(testSource::D),"r2",testSource::C,testTarget
          ::D,"r1",testTarget::C)
24      res.each{r|
25        resCont.items.add(r.asType(testTarget::item))
26      }
27    }
28
29    src.items.select{x|x.isInstanceOf(testSource::A)}.each{i|
30      var a1:testTarget::A
31      var c1:testTarget::C
32      a1:=lib.resolve(i).one().asType(testTarget::A)
33      c1:=lib.resolve(i.asType(testSource::A).r3).one().
          asType(testTarget::C)
34      a1.r3:=c1
35    }
36
37    result:=resCont
38
39  end
```

10.6.2 Case Study 2: From Implicit to Explicit Representation

Implementation with Plain Kermeta. The reverse sample transformation using plain Kermeta is shown in Listing 10.20. As can be seen, more lines of code are needed to accomplish the transformation in this direction. This is mainly because the patterns in this transformation direction rather expand the model. The transformation target model contains 6 classes, therefore more code is needed for instantiation purposes. Additionally, checking whether searched instances of B already exist and keeping of tracing information has to be done manually.

Listing 10.20: Implicit2Explicit: Implementation using plain Kermeta

```
1  operation transformPlainRev(src:testTarget::container):
      testSource::container is do
2
3    var resCont: testSource::container init testSource::
       container.new
4    result:=resCont
5    var ht:Hashtable<Object,Object> init Hashtable<Object,
       Object>.new
6
7    var bs:Set<testSource::B> init Set<testSource::B>.new
8    src.items.select{x|x.isInstanceOf(testTarget::A)}.each{i|
9      var a:testSource::A init testSource::A.new
10     a.a1:=i.asType(testTarget::A).a1
11     ht.put(i,a)
12     var b:testSource::B
13     if (bs.exists{e|e.b1==i.asType(testTarget::A).a2}) then
14       b:=bs.detect{e|e.b1==i.asType(testTarget::A).a2}
15     else
16       b:=testSource::B.new
17       b.b1:=i.asType(testTarget::A).a2
18       a.r1:=b
19       resCont.items.add(b)
20     end
21     resCont.items.add(a)
22   }
23
24   src.items.select{x|x.isInstanceOf(testTarget::C)}.each{i|
25     if (i.asType(testTarget::C).type=="E") then
26       var e:testSource::E init testSource::E.new
27       resCont.items.add(e)
28       ht.put(i,e)
29     else
30       var f:testSource::F init testSource::F.new
31       resCont.items.add(f)
32       ht.put(i,f)
33     end
34   }
35
36   src.items.select{x|x.isInstanceOf(testTarget::D)}.each{i|
37     var c1:OrderedSet<testTarget::C>
38     c1:=i.asType(testTarget::D).r1
39     c1.each{x|
40       var c:testSource::C
41       var d:testSource::D init testSource::D.new
42       c:=ht.getValue(x).asType(testSource::C)
43       c.r2.add(d)
44       resCont.items.add(d)
45     }
46   }
```

```
47
48    src.items.select{x|x.isInstanceOf(testTarget::A)}.each{i|
49      var a1:testSource::A init ht.getValue(i.asType(
          testTarget::A)).asType(testSource::A)
50      var c1:testSource::C init ht.getValue(i.asType(
          testTarget::A).r3).asType(testSource::C)
51      a1.r3:=c1
52    }
53
54  end
```

Implementation with KLRT. The reverse transformation using KLRT is illustrated in Listing 10.21. The lines of code used is almost exactly the same as in the *source2target* direction. Differences only occur in the transformation of C's, where the differentiation of E and F is handled differently. The other patterns use basically the same code, by only calling the reverse transformation pattern.

Listing 10.21: Implicit2Explicit: Implementation using KLRT

```
1  operation transformRev(src:testTarget::container):
     testSource::container is do
2
3    var resCont: testSource::container init testSource::
       container.new
4    result:=resCont
5
6    var lib: transformations init transformations.new
7    lib.setSourceModel(src)
8
9    src.items.select{x|x.isInstanceOf(testTarget::A)}.each{i|
10     var res:OrderedSet<Object> init lib.deaggregate(i.
         asType(testTarget::A),"a2",testSource::A,"r1",
         testSource::B,"b1")
11     resCont.items.add(res.at(0).asType(testSource::A))
12     resCont.items.add(res.at(1).asType(testSource::B))
13   }
14
15   var ht:Hashtable<String,kermeta::language::structure::
       Class> init Hashtable<String,kermeta::language::
       structure::Class>.new
16   ht.put("E", testSource::E)
17   ht.put("F", testSource::F)
18   src.items.select{x|x.isInstanceOf(testTarget::C)}.each{i|
19     var res:Object init lib.expandHierarchy(i.asType(
         testTarget::C),"type",ht)
20     resCont.items.add(res.asType(testSource::item))
21
22     var res2:OrderedSet<Object> init lib.reverseProperty(i.
         asType(testTarget::C),"r1",testTarget::D,testSource
         ::C,"r2",testSource::D)
```

```
23      resCont.items.add(res2.at(0).asType(testSource::item))
24      resCont.items.add(res2.at(1).asType(testSource::item))
25    }
26
27    src.items.select{x|x.isInstanceOf(testTarget::A)}.each{i|
28      var a1:testSource::A
29      var c1:testSource::C
30      a1:=lib.resolve(i.asType(testTarget::A)).one().asType(
          testSource::A)
31      c1:=lib.resolve(i.asType(testTarget::A).r3).one().
          asType(testSource::C)
32      a1.r3:=c1
33    }
34
35  end
```

10.7 Discussion

KLRT aims to simplify Kermeta transformations by providing support for often occurring model transformation patterns. In order to make a statement on how helpful KLRT is in practice, a sample transformation has been implemented using KLRT and plain Kermeta in the previous section.

Advantages of KLRT are smaller resulting transformation code and automatic tracing support. On the other hand, transformation code using KLRT is not easier to read then plain Kermeta code, because many transformation operations need several parameters, which result in very long code lines.

A comparison of the resulting transformation code shows, that the transformation using KLRT uses about 10-20% less lines of code. However, the sample transformation is relatively small (6 classes in the source model, 3 classes in the target model), and therefore the advantages of using KLRT can not be proved certainly. It is also notable that patterns which use more output classes than input classes benefit more of KLRT because less instantiation work has to be done by the developer. For a more comprehensive conclusion, a more complex and bigger transformation has to be implemented, preferably with further transformation patterns supported by KLRT.

As already mentioned, Kermeta code using KLRT is harder to read than conventional Kermeta transformation code, because the pattern operations need several parameters in most cases. For more readable code in general, a different approach to the library implementation would be necessary. For example, a domain specific language describing the transformation patterns could be designed. However, this is very hard to do in Kermeta. A possible way of simplifying the API of KLRT would be the usage of Fluent Interfaces [13], which eases the creation of objects of an API. Although Fluent Interface is a design pattern, it has similarities to a domain specific language [13]. Future work may evaluate how the concept of

KLRT can be implemented in another way, which is more comfortable and easier to use than KLRT. For example, in [26], Kappel et al. discuss a framework for building mapping operators for resolving structural heterogeneities.

11 Conclusion and Outlook

11.1 Conclusion

The main goal of this thesis was to evaluate different model transformation approaches concerning their support of the features documented in [5]. Four different transformation languages have been chosen as representation of different overall transformation approaches: Kermeta is an example for an imperative model transformation language. SmartQVT also follows the imperative approach, but offers more built-in model transformation features. ModelMorf, an implementation of the QVT relational standard, serves as example language for purely declarative model transformation approaches. ATL on the other hand is an example for a hybrid model transformation language.

As the results of the evaluation show, imperative model transformation languages offer generally more powerful features and therefore more flexibility than other approaches. However, this does not necessarily mean that they are superior to other transformation approaches. Most notably, transformations in relational transformation approaches are harder to specify for complex transformations, but on the other hand, only relational approaches offer multidirectionality, which may by crucial for some model transformation projects.

For every model transformation project, the requirements to be fulfilled by the transformation language have to be made clear. Choosing a transformation language without knowing about the requirements may result in inefficient, or in the worst case impossible to implement transformation problems. The feature support documented in this thesis, and the corresponding web site help choosing an appropriate transformation language for transformation projects.

The second part of this thesis is a result of the fact, that although imperative transformation languages are more flexible and powerful than declarative languages, more code is needed to implemented the desired transformation. Therefore, a Kermeta library is presented. The goal of this library is to implement often occurring transformation problems in a generic way. In imperative languages, it is generally necessary to write more code in order to accomplish the same tasks, compared to relational transformation languages. In order to reduce the amount of repetitive code, and to increase code reuse, KLRT implements several transformation patterns, which may then be used in real transformation settings. This way, 10-20 % less code is needed to implemented a transformation.

To sum up, imperative and declarative approaches have their advantages and disadvantages, which have to be addressed when choosing a transformation lan-

guage. In order to increase code reuse when using an imperative language, like Kermeta, KLRT may be used.

11.2 Outlook

In this thesis, four different model transformation languages have been evaluated. Concerning future work, additional transformation languages may be evaluated as well. Further, any transformation language may also be evaluated more specifically for certain features and additional features not covered in this thesis. For example, declarative approach languages may be evaluated for their incrementality and directionality support, which are typical features only found in declarative languages.

KLRT currently only supports a limited set of transformation patterns. In future work, more patterns may be implemented in KLRT. Because KLRT code is harder to maintain than plain Kermeta code, reusable transformation patterns may be implemented using a domain specific language or a fluent interface based API, which would increase code readability considerably. Because KLRT is currently only available for Kermeta, an equivalent library to KLRT could be implemented for another transformation language.

List of Figures

List of Tables

List of Listings

Bibliography

[1] S. Altan. On the usability of triple graph grammars for the transformation of business process models - an evaluation based on FUJABA. Master's thesis, Technical University Vienna, 2008.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proceedings of the 2nd European Conference of Model Driven Architecture - Foundations and Applications*, 2006.

[3] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2), 2005.

[4] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

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

[6] M. de Jonge, E. Visser, and J. M. W. Visser. XT: a bundle of program transformation tools; system description. In *Proceedings of the 1st Workshop on Language Descriptions, Tools and Applications*, 2001.

[7] Fujaba development team. Fujaba website, 2007. available at `http://www.fujaba.de/`.

[8] ModelMorf development team. Modeling with EMF editor, 2007. available at `http://www.tcs-trddc.com/ModelMorf/index.htm`.

[9] MOFLON development team. MOFLON website, 2007. available at `http://www.moflon.org/`.

[10] Eclipsepedia. AMMA, 2007. available at `http://wiki.eclipse.org/AMMA`.

[11] Eclipsepedia. ATL 2006, 2007. available at `http://wiki.eclipse.org/ATL_2006`.

[12] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahe. *Kermeta language reference manual*, 2007. available at `http://www.kermeta.org/documents/manual/`.

[13] M. Fowler. Fluent interface, 2008. available at `http://www.martinfowler.com/bliki/FluentInterface.html`.

[14] M. Gervais. Towards an MDA-oriented methodology. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, 2002.

[15] Object Management Group. *Request for Proposal: MOF 2.0 Query / View / Transformation RFP*, 2002. available at `http://www.omg.org/docs/ad/02-04-10.pdf`.

[16] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*, 2004. available at `http://www.omg.org/cgi-bin/doc?ptc/03-10-04`.

[17] Object Management Group. *MOF QVT Final Adopted Specification*, 2005. available at `http://www.omg.org/docs/ptc/05-11-01.pdf`.

[18] Object Management Group. OMG document on ModelMorf, 2007. available at `http://www.omg.org/docs/bc/07-08-13.txt`.

[19] Object Management Group. MDA website, 2008. available at `http://www.omg.org/mda/`.

[20] ATLAS group LINA & INRIA. *ATL User Manual*, 2005. available at `http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual%5Bv0.7%5D.pdf`.

[21] V. Gruhn, D. Pieper, and C. Röttgers. *MDA: Effektives Software-Engineering mit UML 2 und Eclipse*. Springer-Verlag Berlin Heidelberg, 1 edition, 2006.

[22] P. Huber. Model transformation language comparison website, 2008. available at `http://www.adore.at/mtlcomp/`.

[23] IKV. Medini QVT website, 2007. available at `http://projects.ikv.de/qvt`.

[24] INRIA. Classification of model transformation approaches, 2007. available at `http://modelware.inria.fr/rubrique21.html`.

[25] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006.

[26] G. Kappel, H. Kargl, T. Reiter, W. Retschitzegger, W. Schwinger, M. Strommer, and M. Wimmer. A framework for building mapping operators resolving structural heterogeneities. In *Proceedings of the 7th International Conference on Information Systems Technology and its Applications*, 2008.

[27] T. Mens, K. Czarnecki, and P. Van Gorp. A taxonomy of model transformations. In *Proceedings of the 1st International Workshop on Graph and Model Transformation*, 2005.

[28] Object Management Group. *MDA Guide Version 1.0.1*, June 2003. available at `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[29] Object Management Group. *UML 2.0 Infrastructure Specification*, 2004. available at `http://www.omg.org/docs/ptc/03-09-15.pdf`.

[30] Object Management Group. *UML 2.0 Superstructure Specification*, 2004. available at `http://www.omg.org/cgi-bin/doc?formal/07-11-01`.

[31] R. Schaefer. A survey on transformation tools for model based user interface development. In *Proceedings of the 12th International Conference on Human-Computer Interaction. Interaction Design and Usability*, 2007.

[32] A. Schuerr. Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, 1995.

[33] Sun. Java metadata interface specification, 2002. available at `http://java.sun.com/products/jmi/index.jsp`.

[34] TCS. ModelMorf - a model transformer, 2007. available at `http://www.tcs-trddc.com/ModelMorf/index.htm`.

[35] SmartQVT Development Team. *SmartQVT Detailed Description*, 2007. available at `http://smartqvt.elibel.tm.fr/doc/Detailed_description/architecture.html`.

[36] SmartQVT Development Team. SmartQVT - a QVT implementation, 2008. available at `http://smartqvt.elibel.tm.fr/`.

[37] Triskell Team. Kermeta API documentation. 2007. available at `http://www.kermeta.org/docs/KermetaFramework/framework.km.html`.

[38] France Telecom. *SmartQVT Documentation*, 2007. available at `http://smartqvt.elibel.tm.fr/doc/fr.tm.elibel.smartqvt.doc/SmartQVT_documentation.html`.

[39] M. Wimmer, H. Kargl, M. Seidl, M. Strommer, and T. Reiter. Integrating ontologies with CAR mappings. In *Proceedings of the 1st International Workshop on Semantic Technology Adoption in Business (STAB'07)*, 2007.

Appendix A

The CDROM enclosed contains the following:

- *eclipse/*: The Eclipse installation used in this thesis, containing all plug-ins used.
- *metamodels/*: The metamodels described in Chapter 4 in Ecore format.
- *transformations/*: The source code of the sample transformations in ATL, SmartQVT, Kermeta and ModelMorf.
- *klrt/*: The source code and example code for KLRT.
- *readme.txt*: Further information concerning the files on the CDROM.