FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Supporting Model Extensions in RubyTL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Marc Dopplinger
Matrikelnummer 0525108

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Mitwirkung: Assistant Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Wien, 23. August 2018

Marc Dopplinger | Gerti Kappel

Supporting Model Extensions in RubyTL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Marc Dopplinger
Registration Number 0525108

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Assistance: Assistant Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Vienna, 23rd August, 2018

Marc Dopplinger	Gerti Kappel

Erklärung zur Verfassung der Arbeit

Marc Dopplinger
Johannagasse 5/28, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. August 2018

Marc Dopplinger

Kurzfassung

Model Engineering wird immer wichtiger, da die Nutzung von Modellen in der Software Entwicklung zu nimmt. Gleichsam bedeutend ist es das die Modelle leicht zu adaptieren und zu erweitern sind. Allerdings ist dies nicht immer Möglich. Dies kann zum Beispiel sein wenn das Modell nicht von einem selbst entwickelt wurde und es notwendig ist die Kompatibilität zu wahren. Deshalb wurden mehrere leichtgewichtige Erweiterungsmechanismen entwickelt. Zum Beispiel UML Profile für UML Diagramme oder EMF Profile für Standard Diagramme des Eclipse Modeling Frameworks (EMF). Diese erlauben es ein bestehendes Diagramm zu erweitern ohne das original Modell zu ändern. Aber leider haben diese leichtgewichtigen Erweiterungen einen Nachteil. Nur wenige Modell Transformationssprachen besitzen eine Unterstützung für diese. Die Atlas Transformation Language (ATL), zum Beispiel, kann nur über die unterliegende Java API darauf zugreifen und RubyTL bietet überhaupt keine Unterstützung an.

Im Rahmen dieser Arbeit soll eine Erweiterung entwickelt werden die es RubyTL erlaubt sowohl UML als auch EMF Profile zu verarbeiten. Dabei soll die Erweiterung nicht in den bestehenden Programmcode von RubyTL eingreifen. Um diese Anforderung zu erfüllen werden Modellprozessoren verwendet. Diese integrieren das Profil in das bestehende Diagramm wodurch das Profil dann zu einem kompletten Bestandteil des ursprünglichen Diagramms wird. Dadurch ist es der Transformationssprache möglich auf die Informationen des Profils zuzugreifen. Weiters soll es möglich sein die Modellprozessoren auch mit anderen Transformationssprachen, wie ATL, verwenden zu können.

Als Abschluss der Arbeit wird gezeigt wie man mit den Modellprozessoren bestehende Diagramme verarbeiten kann. Dabei wird gezeigt wie man die Prozessoren benutzt und wie die Transformationsregeln aussehen. Da ATL bereits UML Profile unterstützt, wird an Hand von ATL gezeigt wie die Modellprozessoren die Transformationsregeln vereinfachen können.

Abstract

Model Engineering gets more important in software development because of the increasing use of models. At the same it is important the adapt and extend existing models. But this is sometimes not possible. For example the model was developed from somebody else or the model is used in another project and it is necessary to keep the compatibility. Therefore several lightweight extension mechanism have been developed. For example UML profile for UML diagrams or EMF profiles for standard diagrams of the Eclipse Modeling Framework (EMF). They allow to extend an already existing model without changing the original one. But unfortunately they have some drawbacks. Only a few transformation languages have a support for lightweight extensions and if they do only very basic. ATL can only access the profile with the underlying Java API. With RubyTL it is not possible to process profiles at all.

This thesis covers the development of an extension which enables RubyTL to process EMF and UML profiles. Thereby should the extension be not integrated into the RubyTL code. This will be done with model processors. They integrate the profile into the existing model. Due to the circumstance that the profile is now a complete part of the diagram it is possible that the transformation language can access the stereotypes. Furthermore should it be possible to use the model processors for other transformation languages, like ATL. The goal is to enable the use of UML and EMF profiles also for other transformation languages. But they do not get integrated into the language. The model processors are used from a command line interface (CLI).

The feasibility of the approach is demonstrated by using transformations to apply and read profile information in RubyTL and ATL. The resulting ATL transformations are also compared with ATL transformations using the basic ATL support based on the Java API for UML.

Contents

CHAPTER 1

Introduction

Model Driven Development (MDD) gets more and more used these days. Models are an abstract representations of real world objects. Those real world objects can be people, things or software systems. The abstract view allows to focus only on the important parts and leads to a better understanding of the real world. In case of software systems it will lead to better software quality and better maintenance. There are several ways to describe a model. The most used way is to describe the model in a graphical way. The most known representative is the Unified Modeling Language (UML). It is developed by the Object Management Group (OMG) and supports various types of diagrams for almost all parts and stages of a software development process, such as a class diagram to model data structures or an activity diagram to illustrate different stages of a process. Another way is the use of ECore to develop ones own metamodel. ECore is part of the Eclipse Modeling Framework (EMF). Metamodels are an abstraction of a domain and are the schema for the models. It enables the possibility to create models which are based on a specific domain.

Also a major part of the success of the MDD is the lightweight extension mechanisms. It is often not possible to adapt an existing model or modeling language to support new knowledge. Therefore were lightweight extensions developed. They allow to extend an existing metamodel without changing the original meaning. The OMG introduced UML profiles for UML models and allow to adopt every diagram to contain domain specific knowledge. This allows UML to cover even more domains. But UML profiles are not the only lightweight extension. Also ECore provides some support for lightweight extensions. One of them is called EMF profiles and was developed at the Vienna University of Technology and the Ecole de Nantes and are inspired by UML profiles. Both have in common that the extension is based on stereotypes which extending existing elements of the model. Stereotypes are metaclasses which contain tagged values. Tagged values are similar to attributes of classes and have a name and a data type.

Creating models [Kuh06] manually is not the only concern of the MDD. Another major concern is model transformation. Due the success of MDD it is important to provide some mechanism which allows to transform a model to another model. Those transformations are done with special transformation languages. There are many different languages [LS06], but the most popular is the Atlas Transformation Language (ATL) [JK06b, Atl] which is also the state-of-the-art language in the Eclipse Modeling Framework [Fra]. It contains rules which maps source elements to target elements. Another transformation language is RubyTL [CMT06] and is similar to ATL but entirely written in Ruby.

1.1 Problem statement

Due the importance of model transformation it is necessary that transformation languages are able to use the complete set of models. But unfortunately this is not the case. Many transformation languages lack support for lightweight extensions. Either their build-in support is challenging to use or they do not support them at all. ATL has such a build-in support, but it does not support them directly. The developer need to use the underlying Java API to access the stereotypes. This API usage causes several problems because its usage differs from the normal way of using ATL. Normally ATL uses a declarative programming style to map the elements from the source to the target. But the API forces the developer to use the imperative style. Additionally all tagged values are only addressable with normal strings. A typing error will cause an error during runtime and it is not guaranteed that the tagged value exist. Only an additional existence check can ensure a flawless transformation. But this will increases the complexity of the code and makes the code harder to read, harder to write and harder to maintain. The other transformation language, RubyTL, does not support any lightweight extensions.

1.2 Aim of the work

The aim of the work is to extend RubyTL to support UML profiles and EMF profiles. Not only reading stereotypes inside a transformation should be possible, also applying a profile to a target model. The result should be an implementation that takes profile enriched models and use them in a RubyTL transformation. A requirement is that the implementation uses the plugin mechanism or other extension mechanism to keep the changes to the RubyTL code at a minimum.

There are different ways to achieve that goal. One of them is the model processor technique. A model processor changes a model to extend it with a specific behaviour. In this case a model processor is used to merge the model and the profile into a single model. The additional knowledge of the profile is now part of the model and conforms with its metamodel. As a result, it is possible to use it inside a standard RubyTL transformation. Every stereotype is accessible like any other element of the model. This method has several benefits. Due the separation between the model processor and RubyTL it is possible to reuse the model processor in other transformation languages. Any other language can use their standard syntax and does not need to introduce new commands.

Another requirement is a seamless integration in RubyTL. This means that a transformation with a profile enriched model should be done in one step. No other external tool or manual interventions should be necessary. The integration should take all required models as input, run the model processor and execute a normal RubyTL transformation. The only step for the developer should be to configure the transformation just like any other RubyTL transformation. Only the additional profile and profile application are necessary. But the RubyTL integration should not be the only way to use the model processor. A Command Line Interface (CLI) allows the usage of the model producer in other transformation languages, like ATL. It will only create the appropriate models which can be used as source for transformations. The model processors and the transformation need to be used manually.

Another part of this thesis is the evaluation of the model processors. It should point out the advantages and the limitations of the implementation. This will be done with some quantitative metrics, like the Line of Code (LOC) or the performance in terms of runtime speed. Also the CLI should be evaluated with ATL as the evaluation transformation language. The same metrics, as for RubyTL, will be used for the CLI. Afterwards there is a comparison between both results to determine the advantages of the model processors over the native implementation of ATL.

1.3 Structure of the work

This master thesis is structured in the following chapters.

Chapter 2 gives an overview about the general topics of the Model Driven Development (MDD). It starts with the key principles of MDD, followed by a general introduction to modeling languages. It covers UML and the basic of EMF modeling. Also part of this chapter is the explanation of the two lightweight extension, UML profiles and EMF profiles. The model extension section also explains the different extension elements, like stereotypes and tagged values, in detail. The end of the chapter is dedicated to transformation languages with the focus on RubyTL and the Atlas Transformation Language (ATL).

Chapter 3 is focusing on the state of the art of the extension support in transformation languages. Due the complete lack of profile support in RubyTL, this chapter only covers the ATL support. It shows the difficulty to perform a transformation using plain ATL. Another section describes three different approaches to extend an existing transformation language to transform profile enriched models.

Chapter 4 describes the used approach from Chapter 3 in detail. Another part of this chapter are generic examples showing how the approach works in theory.

Chapter 5 contains the actual solution and its implementation. The implementation is split up into a UML and EMF profile part. The next part describes two ways to start a transformation with profile enriched models. The last part contains a practical example how the solution can be used to transform profile enriches models. RubyTL and ATL are used as examples.

Chapter 6 is focusing on the evaluation of the two implementations. The evaluation will answer three major question about the quality of the solution presented in this thesis. They cover the question if the solution works with all introduced lightweight extension along with its benefits and limitations.

Chapter 7 describes the related work. It is focusing on transformation languages, as well on model extension approaches. Also it takes a closer look on related solutions to the approach of this work.

Chapter 8 is the final chapter and contains the conclusion and future work.

CHAPTER 2

Model Driven Development

This chapter describes the key principles of Model Driven Engineering (MDE) [Bé05b]. Nowadays data structures and processes tend to get larger and more complex with the result that software development is getting more complicated. MDE is trying to remove some of this complexity. The reality gets represented in a graphical way. This is called *model*. The traditional purpose is documentation but today it is not bound to that purpose alone. It is possible to abstract some parts of a software to a model to give an overview about the system. Many different possibilities and tools were developed to use such models. They can be summarized into 4 different categories. Chapter 2.1 shows an overview of these categories.

There are several approaches to specify MDE:

- Computer Aided Software Engineering (CASE)
- Model Driven Software Development (MDSD)
- Model Driven Architecture (MDA) [WKB03][MSUW02]

They are all similar but differ in approach, goals and area of use. They also have different features and tools. The most known MDE is the Model Driven Architecture from the Object Management Group (OMG)[1]. MDE tries to achieve *Interoperability* and *portability*. Interoperability means that a model should work with different tools and technologies. This is important because different tools support different features. Portability allows the migration of a system from one environment to another without requiring manual changes.

[1] `http://www.omg.org/mda/`

2.1 Key principles of model driven engineering

Model Driven Engineering can be distinguished into four major categories and encompass the generation and processing of the models. These categories are:

- **Model Generation**. Is probably the biggest part of MDE. It allows to create models which can be used for further development. Model generation can be done in several ways. The most common is to "draw" the models with a graphical editor. But it is also possible to generate them out of existing sources. Such as Java classes or databases.
- **Model Testing**. Not all constraints of reality are possible to model. Errors in models are hard to find. Model Testing Frameworks allow to check models against given constraints.
- **Code Generation**. It allows to generate code out of models for example SQL [DD87] statements or Java classes. One representative is XPand[2]. It is based on the Eclipse Modeling Framework [Bud04] and allows to create any sort of textual fragments.
- **Model Transformation**. Sometimes it is necessary to change a model and transform its data to the new model. Model transformations [SK03] providing a mapping mechanism from a source model to a target mode. There are many different languages, like ATL [Atl, JK06b], QVT [JK06a, OMGc] or RubyTL [CMT06].

2.2 Modeling languages

They are the staring point for any model development. Any model will be described with a modeling language and is a computer language like any other. Therefore it has typical language characteristics [Kle09]. They have an abstract and concrete syntax and a semantic. The abstract syntax defines the language elements that the concrete syntax can use. It also covers the relationship between each element. The concrete syntax describes how the element will be represented. For example a rectangle for a class or a keyword for a Java command. But a language can have more than one concrete syntax. For example, UML has a graphical and a XML [BPSM$^+$] notation. The last characteristic of each language is the semantics. It describes the meaning of each model element.

2.2.1 Basic model development

There are many definitions to describe the model development structure. The most known is the 4-layer definition [Bé05a], shown in Figure 2.1. It has four different layers and are named *Meta-Metamodel*, *Metamodel*, *Model* and *System*. Each layer gets defined

[2] http://www.eclipse.org/modeling/m2t/?project=xpand#xpand

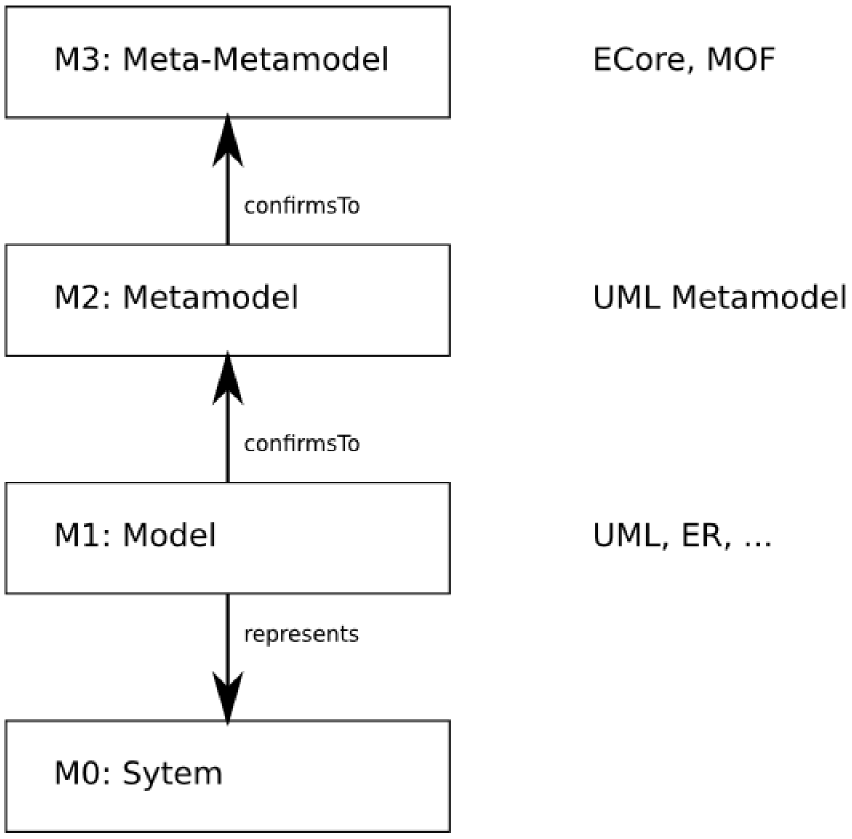


Figure 2.1: 4-layer metamodel definition

by the upper one and is an instance of it, except the meta-metamodel it is defined by itself. Many further development aspects uses the metamodel and model definition [Sei03].

M3: Meta-Metamodel

The meta-metamodel specifies a meta language which will be used to develop a metamodel. Figure 2.1 shows that the meta-metamodel is the top-layer, also called M3-layer. Due the top position a meta-metamodel can only be defined by itself or another meta-metamodel. It defines the different elements that are usable in a metamodel, like classes, attributes or generalization. The most prominent two representatives are:

- **MOF**. MOF stands for Meta Object Facility [OMGa] and is specified by the Object Management Group (OMG)[3]. MOF itself is described in an object oriented way, simplified UML classes to be more precise. The basic idea behind MOF is to use it as a base definition for all OMG modeling projects. The most prominent metamodel which is described in MOF is the Unified Modeling Language (UML)

[3]`http://www.omg.org`

[OMGd][OMGe]. But MOF is very complex definition. The majority of the metamodels do not use all components. Therefore was MOF derived into two different version. One complete version, now called *CMOF* (complete MOF), and a reduced one, *EMOF* (essential MOF). EMOF is designed to cover the standard elements that are needed for common metamodels. CMOF is used for more complex metamodels or inside developing tools.

- **ECore**. Is the default meta-metamodel of the Eclipse Modeling Framework [Fra] and is based on EMOF. The main idea behind ECore is to have a Java oriented and MOF compatible standard. Also UML has an ECore implementation.

M2: Metamodel

The next layer is the metamodel on the M2-level. A metamodel gets described by a meta-metamodel like MOF or ECore and describes the structure of a model. It specifies what elements a model can use, also the relationship between each other. For example it can specify a class *Person* with the attributes *Name* and *Age*. Popular metamodels representations are UML or ER diagrams.

M1: Model

A model is an abstraction of the reality, but only containing the relevant parts which are necessary. The structure and allowed elements are defined by a metamodel. For example the class *Person* from the metamodel example (Section 2.2.1). The attributes are real values, like *John Smith* for the *Name* and *28* for the *Age*. Another popular representative for models are UML diagrams, if an UML metamodel is used on the M2-level. The model is located at the M1-level of the 4-layer definition. There is only one layer beneath, namely the M0-layer.

M0: System

Is the lowest layer, the reality. It would be the real person *John Smith*, the runtime objects or a process in a software program. They get described in an abstract way by a model on the M1-layer.

2.2.2 Unified modeling language

The Unified Modeling Language (UML) [OMGd, OMGe] was developed by OMG. It is one of the most used modeling languages. The main idea behind UML is to provide an universal language or a general purpose language for modeling complex systems. It does not matter if the system is a software or another technical component. This is achieved with many different diagrams which are able to represent all aspects of a system. Normally are UML diagrams created in a graphical and object oriented way. This allows an easy and productive development. Every single UML element has its own graphical representation. But on the other hand are they also serialized to XML, or other data exchange formats. This enables interoperability and exchange.

UML itself is designed with MOF, so it is actually a metamodel on the M2-layer. And any UML diagram is a model on the M1-layer. Through this circumstance it is very easy to use UML diagrams for advanced topics like model transformation or model manipulation. But there are also other implementation beside MOF. The Eclipse Modeling Framework has an implementation based on ECore and is fully compatible within the Eclipse ecosystem. There are also some proprietary versions of UML. Some of them do not support the complete set of diagrams, but the most of the supported one follow the UML guidelines from the OMG.

Version is 2.3 from 2010 defines 13 different diagrams in three categories [QRZ07]. A short overview will be given in the next few sections.

Structure diagrams

Structure diagrams describing the structure of a system. This ranges from simple classes to complex components. They are not only describing the architecture, also how the components look like during runtime. UML has six different diagrams to describe the architecture of a system.

- **Class diagrams** are the central diagrams in UML and describing the basic structure of a system. They will be represented with classes and the different relationship between them. To be more precisely associations, generalizations and aggregations. Every class can be specified with attributes. Every object of a system gets modeled with a class and the object properties are the class attributes. Probably the most known and used UML diagram.
- **Package diagrams** allowing to separate a complex system into packages to get a better overview. Every package describes an aspect of the system and can contain other packages as well to create a hierarchical composition of the system. It is also possible to model relationships between different packages on the same hierarchical level. A relationship can be a simple access or a generalization.
- **Object diagrams** describing instances with real values of the system during runtime. Like a snapshot. In the most cases it is a class oriented representation, like the class diagram. Only with the additional information of the real values beside the attribute definition.
- **Composite structure diagrams** are specifying the internal structure of an object (e.g. a class) and its interaction with other parts of the system.
- **Component diagrams** describing the structure of a system and its interfaces to other components during runtime. The diagram reuses modeling elements from the class and object diagram but with some different meaning. Additionally there

are several elements to model a component and its defined interfaces and the relationship between them.

- **Deployment diagrams** are showing which software part runs on what hardware device. Mostly used to design client-server applications and clusters.

Behaviour diagrams

It is often not enough to model the structure of a system. It is also necessary to design the behaviour of it and Behavior diagrams will close this gap. They describing how a system will act and what it should do. Not only the internal view also the user view.

- **Use Case diagrams** showing the user view on the system. It describes all the functionality a user can execute, like log-in into a bank account. But every action can trigger another action. Therefore it is possible to model a chain of events a user can execute. It is also possible to divide a complex system into subsystem for a better overview. A user can not only trigger an event, he can also be the target of actions.
- **Activity diagrams** describing how to process a defined task. For example to transfer money to another account. It allows to model the different steps to achieve the defined task. It is also possible to model if-conditions for more complex tasks. An activity can only execute one task at once. A token placed on the executed task ensures that.
- **State Machines** specifying all the internal states of a component and its transitions into other states. The modeling elements are similar to the *activity diagram*. But they showing the different states of the system and not the tasks.

Interaction diagrams

Is a subcategory of Behavior diagrams. Interaction diagrams are used to model the internal communication and function calls. All interaction diagrams have a time component. Which means that all processes follow a chronological order.

- **Sequence diagrams** describing function calls and messages between different actors. It is also possible to to send the messages between the actors over the system boundaries. The sequence diagram uses a lifeline for each actor for the time constraints. The lifeline contains the the duration of a function and and a message call to other actors lifelines. This diagram also allows to model parallel calls.
- **Communication diagrams** showing the chronological behaviour of all actors of a task of the system. When happens a state change to what state. All transitions between the actors have a counter to mark the order of execution and a direction arrow.

- **Timing diagrams** are similar to the *sequence diagram.* But with a timing component. It shows when a state transition happens. All transitions are lined up on a time line. It is also possible to model the duration between transitions.
- **Interaction overview diagrams** are a mix of *activity diagrams* and other *interaction diagrams.* It describes, like the *activity diagram*, the execution order of the tasks and under what condition. But additionally it is possible to use one of the other three *interaction diagrams* as elements. For example a *sequence diagram* to add a time behaviour.

2.3 Model extensions

Sometimes the available notation of a metamodel is not suitable for the given model. Some models require additional domain specific knowledge. This mainly happens if an existing metamodel should get extended. The only way to adopt a model to a new model is to extend it. Basically there are 3 different approaches. But all of them have different advantages and disadvantages.

1. **New metamodel**. A possibility is to create a hole new metamodel for the desired domain. But this is the most time consuming method and it is also nearly impossible to apply changes made in the original metamodel after the separation. This option is only a choice if the new model do not need any changes from the original metamodel.
2. **Heavyweight extension**. It extends the existing metamodel. For example with adding new elements or changing existing one. But it changes the meaning of the original metamodel. And it is also problematic to adapt it to a changed original metamodel. The effort and time exposure depends on the changes. It would be easy for a few elements. But it grows with the number and type of changes. It is easier to add an element than changing the meaning of it.
3. **Lightweight extension**. Also known as *profile.* It extends the existing metamodel in a separated file and leaves the metamodel unchanged. Changes in the metamodel may not destroy the profile. But if, it is easy to adapt the extension to the new situation. Lightweight extensions are the most promising and used extension type. A closer look to it is done in the next chapter.

2.3.1 Lightweight extension

Usually heavyweight extension are too complicated to use and every created metamodel needs to have tool support. Therefore it is better to use existing lightweight extensions. Lightweight extensions can be compared with *Domain Specific Languages.* They are providing additional specific knowledge for an existing model, like from a UML model. They are extending elements of a metamodel without destroying the meaning of it.

Lightweight extensions are only allowed to extend existing elements. It is not possible to change any meaning of the original metamodel, like removing elements or changing relationships.

The possible modeling elements depends heavily on the used extension mechanism. But most profiles use some standard elements like classes with attributes and relationships like generalization and aggregations. Also they have an extension point which marks the element of the metamodel which should be extended.

Another characteristic of lightweight extensions is that they are separated from the metamodel. This means that it is possible to change the metamodel at every time and due to the separation and the restrictions it is easy to adapt the extension.

But there are many different lightweight extensions and they are not compatible with each other. Every meta-metamodel can have its own extension. But also normal metamodel can have a profile mechanism. For example UML profiles [FFVM04] for UML diagrams or EMF profiles [LWWC11] for ECore based metamodels.

The benefits of lightweights extension are:

- Unchanged metamodel
- Easy to replace
- Easy to share
- Easy to create
- Easy to extend

2.3.2 UML profile

UML profiles [FTV02, Sel07] are probably the most used extension and are designed for UML diagrams. Figure 2.2 shows an example. Every UML element can be extended by several stereotypes. A stereotype is a class with attributes which represents the additional knowledge and will be described in the next section in detail. UML profiles uses many modeling elements from the *class diagram*.

- *Stereotype*: Contains the additional knowledge.
- *Extension relationship*: Connects the stereotype with the extended element from the metamodel.
- *Profile*: Is similar to the *package* element. It allows to group profile elements to logical groups. The keyword for this element is *profile*.
- *Relationships*: Nearly all relationships from the *class diagram* can be used. For example generalization and aggregations.

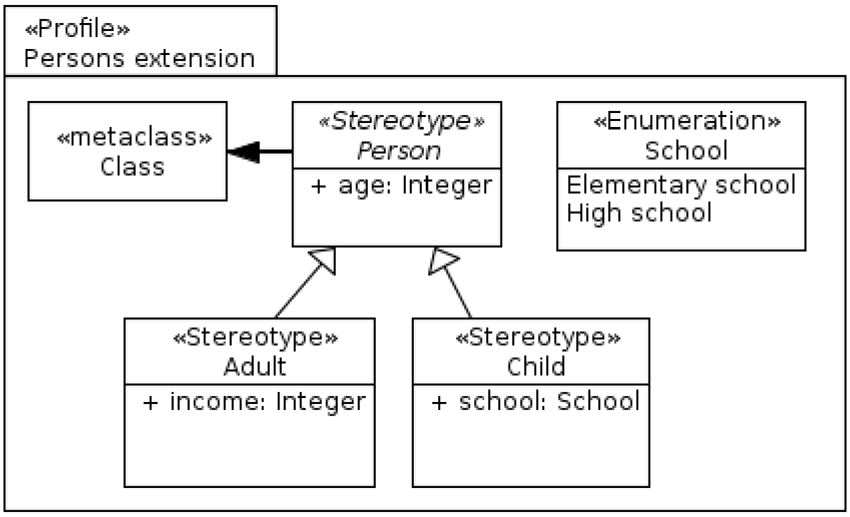


Figure 2.2: UML profile example

- *Enumeration and classes*: They can also be used to define additional data types for the attributes of the stereotypes.

Every UML model can import a profile. Afterward it is possible to apply stereotypes for the corresponding modeling element. But applying a stereotype is not a must. The user can decide if an element should get extended by a defined stereotype or not. It is possible to extend class A with stereotype A and class B with stereotype B. But a UML element can only be extended once by the same stereotype. It is possible to extend almost all UML elements like Classes, Activities, Use cases or Associations. But it is not possible to change the UML language itself. The next few sections will describe all important components of an UML profile in detail.

The profile definition is completely separated from the metamodel. They are also in separate files. But this is not the case for the model and profile application. Thy are in the same file but logically separated.

Stereotype

Stereotypes describing with what additional knowledge an UML element can be extended. They are similar to classes of an UML class diagram. Every stereotype consists of three different areas. The first is the name of the stereotype. Followed by a list of *tagged values* (see Section 2.3.2) as attributes. The last area is a list of operations.

The standard graphical notation is a rectangle with the keyword *«Stereotype»* followed by the name of the stereotype. Beneath are the lists of the tagged values and the operations.

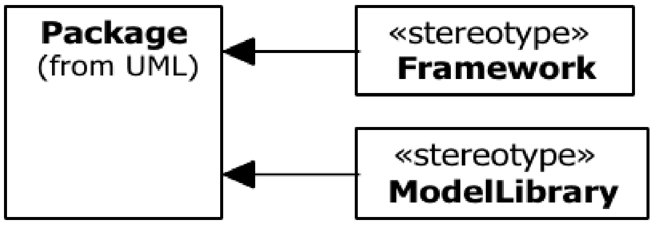


Figure 2.3: Standard UML profile example from UML package

Name	Extends	Description
«Focus»	Class	Marks a class as important for the logic. Typically used together with Auxiliary classes.
«Auxiliary»	Class	Supports another more important class. Is used together with Focus.
«File»	Artifact	Marks an artifact in a deployment diagram as a physical file.
«Framework»	Package	The elements inside the package are reusable.
«Create»	Usage	The source of the Usage relationship creates an instance of the target.

Table 2.1: List of a some stereotypes of the L2 UML profiles

Figure 2.2 shows three stereotypes, *Person*, *Adult* and *Child*. *Person* is abstract, represented with the italic *«Stereotype»* keyword. *Adult* and *Child* extends with a generalization from *Person*.

Besides the self defined stereotypes has UML two predefined profile [OMGe] sets. Named L2 with 31 stereotype and L3 with 3 stereotypes. Table 2.1 shows some L2 stereotypes. They do not have any tagged values to describe the extended element in detail because they are only used to mark elements. Figure 2.3 shows the stereotypes for the Package element as an example.

Model extension

The model extension connects the stereotype with a UML element. These UML elements are called *metaclasses*. Only predefined UML elements are able to get extended by a stereotype and are optional by default. This means a stereotype can extend the metaclass, but it is not a must. They have the multiplicity of *0..1*. But with the keyword *«required»* or with the multiplicity *1* the extension is marked as mandatory. Figure 2.2 shows an optional model extension between a UML Class and the stereotype *Person*. Through the generalization of Person can the UML Class also be extended by *Adult* or *Child*. But not *Person* because of the abstract property.

A model extension is represented as an arrow with a filled arrowhead. The source is a

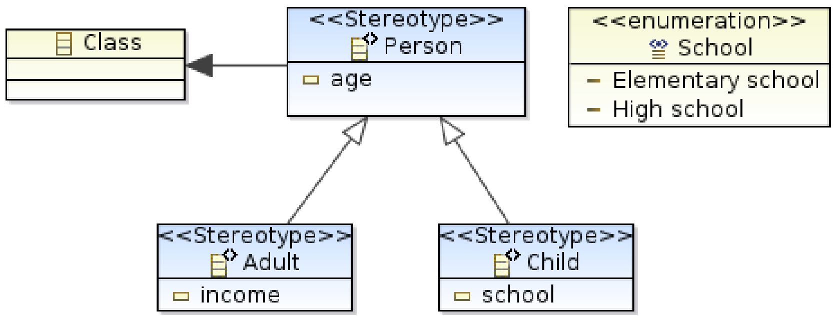


Figure 2.4: EMF profile example

stereotype and the target a metaclass. The metaclass is a rectangle and has the keyword *«metaclass»* and the name of the UML element.

Tagged value

Tagged values are similar to attributes for classes. They describing with what properties the metaclass can be extended. A tagged value is a simple key-value pair. It has a name and a data type. The data type range from primitive types over Enumerations to complex types like classes. See tagged value *school* of the example in Figure 2.2. It has the enumeration *School* as data type.

2.3.3 EMF profile

EMF profiles [LWWC11] are a new type of lightweight extension. They were developed during the AMOR (Adaptable Model Versioning) project by Business Information Group of the Vienna University of Technology, the Johannes Kepler University of Linz and the Ecole de Nantes. They are designed for the Eclipse Modeling Framework [Fra] and is similar to UML profiles. But they extend a ECore based metamodel. The profile is separated in two different files. The first is the *profile definition*. It describes which ECore elements can be extended which stereotypes. The second file is the *profile application* which confirms with the profile definition. It also contains the references to the regular model. The profile application gets serialized in a XMI file. One of the benefits of EMF profiles are that they are compatible with ECore. In theory every transformation language which can use ECore based models should also be able to use EMF profiles.

Figure 2.4 shows an example how an EMF profile may look like. The example is the same as the UML example. The *Class* gets extended with the stereotype *Person*, but through the generalization it is also possible to use *Adult* -or *Child*. EMF profile supports many different modeling elements, like classes, attributes, generalization, references and enumeration. But the most used elements are similar to UML profiles.

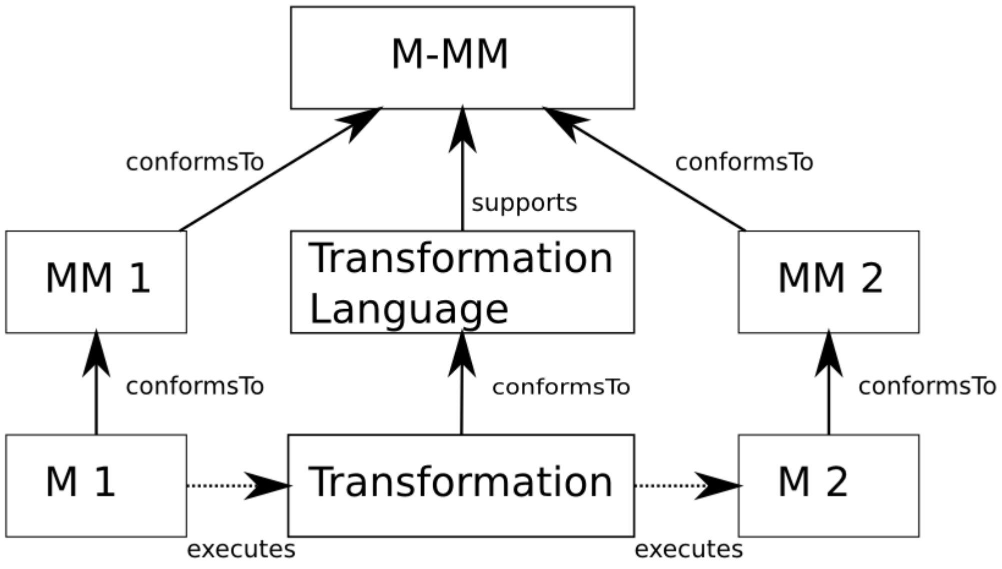


Figure 2.5: Overview of a transformation process

2.4 Transformation languages

Transformation languages are a major part of Model Driven Engineering. They are similar to other programming languages, like Java or Ruby, but they are used to transform a model to a specific output. The output depends on what kind of transformation language is used because there are many different languages [CH06] which support different types of transformations. The most popular is the *model-to-model* transformation [CH03]. It takes a source model and transforms it, according to transformation rules, to an output model like an UML model to another UML model. Figure 2.5 shows an overview how a *model-to-model* transformation will work. It starts with the source model *M 1* and its corresponding metamodel *MM 1*. As shown in Figure 2.1 needs the metamodel a corresponding meta-metamodel like MOF or ECore. The target has only a metamodel *MM 2* because the transformation will transform model *M 1* to the output model *M 2*. The transformation logic varies between transformation languages but their basic structure are the same. The logic gets defined in a transformation file which corresponds with the syntax of the transformation language. The transformation language will execute the *transformation* and generate the output model *M 2*. The most prominent languages are ATL or QVT.

Another kind of transformation is the *Model-to-Text* transformation. It transforms the model to any kind of textual artefacts. Possible outputs may be SQL statements or any source code of a programming language. The most known language is XPand. The basic structure is similar to a model-to-model transformation. The source side is the same but the target varies. The transformation takes the source model and generate any kind of textual artefacts, according to some transformation logic.

2.4.1 RubyTL

RubyTL [CMT06], short for Ruby Transformation Language, is a transformation language written entirely in Ruby and developed at the University of Murcia, Spain. It can be completely integrated into any Ruby code because it is only a library for Ruby and not an independent language. This was realized with the domain specific language extension of Ruby. Due the nearness to Ruby it is possible to use the complete Ruby syntax. RubyTL supports the classic *model-to-model* transformation but also validating models against metamodels and code generation like SQL or Java Code.

RubyTL is able to use different model formats for its transformations. One format is the classic textual form like in XMI [OMGf] serialized models. Another source are Ruby classes with the attributes matching the corresponding elements of the metamodel. Because RubyTL transforms every XMI serialized model to Ruby classes before staring the transformation, it uses the Ruby classes to access the model attributes. This enables the use of syntax highlighting and syntax correction for any standard Ruby development environment. But the situation is a little bit different with metamodels. RubyTL only supports ECore [Fra] and EMOF [OMGg] as metamodels and the metamodel need to be in textual form.

A transformation consists of many different *rules*. Every rule is responsible for transforming one element of the source model to an element of the target model. They are similar to functions in other programming languages. All rules have a name and four, transformation specific, parts. Listing 2.1 shows a rule as example. It transforms an attribute from a class model to a Java class method.

Listing 2.1: RubyTL rule

```
rule 'Attribute2Field' do
  from Class::Attribute
  to   Java::Method

  filter do |attribute|
      attribute.visibility.name == 'Private'
  end

  mapping do |attribute, method|
      method.name = attribute.name
      method.type = attribute.type
      method.visibility = 'private'
  end
end
```

Every rule starts with the keyword *rule* followed by the name. The first of four transform specific parts is *from*. It specifies the source element from the source model. The rule will only be executed if RubyTL finds the element in the source model. The corresponding target element gets defined in the second part and is named *to*. Both values begin with the name of the metamodel. Followed by a *::* and the element name. The name of the metamodel is necessary because the models are transformed to Ruby classes and every class is inside a module to avoid naming conflicts.

filter is the third part and allows to skip rule execution. The source element is accessible over the specified parameter. The rule gets only executed if the filter returns true. It is optional and the rule will always be executed if no filter is set.

The actual transformation is done in the last part. It is called *mapping* and has two or more parameters. The first is the source and the rest are target elements. This allows to generate more than one target with the same source. This is necessary because every source element gets only mapped once. Every target element gets separated from another with a colon. Since Ruby is a imperative language is also the transformation in an imperative style and not in declarative. But a declarative way can be simulated by only using bindings inside the mapping. The example above shows the declarative way. But it is possible use any Ruby syntax like *if-conditions*, *for-loops* or functions for an imperative way.

Rule execution

The rule execution is simple. The transformation starts with the first rule which has an existing *from* part. The next step after the rule matching is to handle the filter. The mapping only gets executed if there is no defined filter or the filter returns *true*. If the filter returns *false* RubyTL will abort the evaluation of this rule and continue with the next rule.

But when RubyTL continues with the rule evaluation it starts with the target element generation. All elements which are defined in the *to* part gets generated. The next and last part is the mapping. It takes the source and all target elements as parameter. The elements are represented as a Ruby class and the element properties are class attributes. Therefore is the access to get and set values the same as in Ruby. RubyTL also works with reflection which allows to change the behaviour of bindings. RubyTL checks both sides of the binding to find a corresponding match of the types in the *from* and *to* part in other rules. If it finds a rule, it will get executed instead of making a direct mapping. After executing the mapping block, will RubyTL insert the generated element into the output model.

Execution of a transformation

The easiest and most common way to execute a transformation is to use rake files[4]. Rake files are similar to Makefiles under Linux or Ant for Java and allows to execute pre configured Ruby scripts. All the configuration will be done in this file, like source models, target models and the transformation file. Listing 2.2 shows such a configuration.

[4]`https://ruby.github.io/rake/`

Listing 2.2: Rake configuration to start a RubyTL transformation

```
model_to_model :class2java do |t|
  t.sources :package    => 'ClassM',
    :metamodel => 'metamodels/Class.ecore',
    :model     => 'models/class.rb'

  t.targets :package    => 'JavaM',
    :metamodel => 'metamodels/Java.ecore',
    :model     => 'models/java.xmi'

  t.transformation  'transformations/class2java.rb'
end
```

Instead of using the normal rake classifiers, like *Task* or *File*, has RubyTL introduced for every task an own command. This is necessary to configure and execute the RubyTL task properly. The most used command is *model_to_model* to execute a model-to-model transformation. Each transformation task gets configured in a code block and has three mandatory parts. The first two parts are the source and the target models and has three parameters. *package* defines the name that will be used in the transformation rules to address the correct metamodel. *metamodel* and *model* are the relative path to the corresponding files. All parameters are separated with a comma. The last part of the block is the transformation file. It also takes the relative path to the file. Other commands are *model_to_code* for code generation and *validate_model* for verifying model constraints.

Plugins

RubyTL is designed to be an extensible language [CM06]. There are some predefined extension points to extend or change the behaviour of RubyTL. Possible features are new types of rules with other behaviour, new keywords or changes in the transformation process. Plugins have different advantages. They allows to extend the behaviour of RubyTL easily and makes it easier to maintain and integrate it into an existing RubyTL installation. Like RubyTL is every plugin written in Ruby.

As mentioned in Chapter 2.4.1 will the first rule with an existing source element be used as entry point for the transformation. This default behaviour is done with a plugin and an extension point. But this behavior can be changed. RubyTL already provides a plugin for an alternative behaviour. This plugin is called *top_rule* and will always be executed if the *form* part matches. Even if there are more than one top rule so that an element can have more than one entry point. The usage inside rules is also simple. Just use *top_rule* instead of *rule* in Listing 2.1.

Another restriction is that a source element can only be executed once by a rule with a new target element. But sometimes it is necessary that a source element gets executed more often. To achieve the behavior has RubyTL another rule, called *creator*. Every time the rule is executed will RubyTL create a new target element. Other examples are one-to-many bindings instead of the one-to-one bindings.

2.4.2 Atlas transformation language (ATL)

The Atlas Transformation Language (ATL) [BDJ+03] is the most used transformation language and is part of the Eclipse Modeling Framework. It is a model-to-model transformation language. An ATL transformation consists of three major parts. An configuration, some transformation rules and helpers. The transformation will be done with transformation rules similar to RubyTL. The rules are supported by helpers, which are like methods in General Purpose Languages. The transformation is stored in a single *.atl* file.

Configuration

An ATL transformation gets configured at the beginning of each file and is mandatory. It starts with the keyword *module* followed with the name of the module. The next part describes the variables for the source and the target metamodel. The target get specified by the *create* keyword and the source with *from*. Each of the target and source has a pair of variables as parameter which are separated with a colon and containing the metamodel and the name of the metamodel. These names will be used to access the elements of the models inside the rules. Listing 2.3 shows a UML to WSDL transformation configuration as example. The module has the name *UML2WSDL*. The target *OUT* has the name *WSDL* and the source *IN* the name *UML*. *OUT* and *IN* are used as variables for the metamodels. The metamodel path will be assigned in the execution dialog of Eclipse.

Listing 2.3: ATL configuration

```
module UML2WSDL;
create OUT : WSDL from IN : UML;
```

Rules

Rules are the main construct of an ATL transformation. Every transformation consists of several rules. Their main purpose is to map the source element attributes to the target element attributes. Each rule has the keyword *rule* followed by the name. ATL also supports rule inheritance which means that every rule can inherit from a superrule. A superrule can be an abstract or a normal rule. ATL distinguishes three different rule types.

Matched rule Is the most important rule. It has a source and a target part and has the keyword *rule* followed by the name of the rule. This rule is split up into two different parts. The first is the specification of the source element and starts with the keyword *from*. The source part begins with the name of the element and will be used to address the attributes of the model. Followed by a colon and the element itself. The element is split up into two classifiers which are separated with a *!*. The first classifier is the name of the source metamodel, which was defined in the configuration part. The second classifier is the name of the element. The source has an additional and optional *guard* condition. A guard condition is a OCL [OMGb] boolean expression inside round braces after the source definition. The rule will only get executed if the source matches with an element and guard returns *true*.

The other part of the rule is the target definition. It begins with the keyword *to*. The target can specify several target elements that should be generated. The target contains the declarative programming style inside its braces. Source properties can be mapped to target properties. Listing 2.4 shows an example of a matched rule with the name *Class2Message*. It transforms a UML Class to a WSDL Message. The source has also a guard condition and should only executed if the class has more than 0 attributes. The target part maps the name of the UML Class to the name of the WSDL Message.

Listing 2.4: ATL matched rule example

```
rule Class2Message {
  from
    c : UML!Class ( c.attribute.size() > 0 )
  to
    m : WSDL!Message (
      qName <- c.name
    )
}
```

Called rule Are similar to matched rules. But they need to get called explicitly because they do not have a *from* part. It is similar to a function in General Purpose Languages but only in rule style. Due the lack of a from part has the called rule the possibility to have some call parameters. They get defined inside braces after the name of the rule. They have the typical ATL variable definition. At first the variable name then the model element separated with a colon. It is possible to use more than one parameter. Every parameter is separated with a comma. Listing 2.5 is the same example as in Listing 2.4 but it takes the source from a call parameter and not from the *from* part. The target element and the logic are the same. The rule can be called from any other rule and can return the generated element back to the caller. But only if the rule uses an imperative block. See Section 2.4.2 for an example. A called rule can also only be called inside the imperative code block.

Listing 2.5: ATL called rule example

```
rule Class2Message ( class : UML!Class ) {
  to
    m : WSDL!Message (
      qName <- class.name
    )
}
```

Lazy rule Lazy rules are also similar to matched rules. They have a *from* and *to* part. But they do not get automatically called during the matching phase of a transformation. They getting executed in the last phase of a transformation, the target model elements initialization phase. To distinguish a lazy rule from a matched rule has the lazy rule the keyword *lazy* in front of the definition. Listing 2.6 is also like the example of the matched rule (Listing 2.4) but only as a lazy rule.

There is also a special type of lazy rules and is called *Unique lazy rules*. Working in the same way as normal lazy rules, but they can only be called one time. If they get called more than once with the same match the already created target element will be used instead of creating a new element. It uses the keyword *unique* in front of *lazy rule*.

Listing 2.6: ATL lazy rule example

```
lazy rule Class2Message {
  from
    c : UML!Class ( c.attribute.size() > 0 )
  to
    m : WSDL!Message (
      qName <- c.name
    )
}
```

Imperative block Normally are ATL transformation rules in an declarative programming style. But ATL also allows to program rules also in an imperative way. This allows to use flow control commands like *if-conditions*, *for-loops* or other OCL commands. If the imperative block is used in a called rule it is possible to return the generated element back to the caller. The block starts with the keyword *do* and after the *to* block and gets executed after it. Listing 2.7 shows a called rule with an imperative part. Every imperative code is inside braces. Line *5-7* shows a for-loop which iterates over all activities of the source model. These activities are used in line *6* as a parameter for another called rule. *thisModule* refers to the transformation file itself and *Operation* to a called rule. The last statement will be used as the return value. Line *8* shows that the generated element *pt* is the return value.

Listing 2.7: Complex example of a called rule with imperative part

```
1 rule PortType () {
2   to pt : MM2!PortType()
3
4   do {
5     for(a in MM!Activity.allInstances()) {
6       pt.eOperations <- thisModule.Operation(a);
```

```
      }
      pt;
    }
}
```

Helpers

Helpers are similar to functions in General Purpose Languages like Ruby or Java. They should help to compute, store and return values to rules. Listing 2.8 shows the syntax of a helper.

Listing 2.8: ATL helper definition

```
helper [context type] def : name (parameters) : return_type = code
```

They are usually right after the configuration and consists of three parts. Every helper starts with the keyword *helper* and is followed by an optional type definition. These type, called*context type*, allows to bind the Helper to an element and only the defined element can call this helper. But every element can call a helper if there is no defined context type. *def* completes the first part of the definition. Every part is separated with a colon. The name of the helper is defined in the next part with optional call parameters. The parameters are between round brackets and have a key-value form. First the name and followed by the type with a colon as separator. A comma separates several different parameters. A helper must have a return value and is defined in the last part. It can return every OCL element like *Set(UML!Class)*. The code is separated from the return type with a =. The last statement inside the code block is the actual code.

Listing 2.9 shows three different helper examples. The first example returns a simple String with the value *Integer*. The second example is similar. But it returns an empty Set. The last example can only be called from a *UML!Class* element and returns the number of attributes of the class.

Listing 2.9: ATL helper examples

```
helper def : setName : String = "Integer";

helper def : clear : Set(String) = Set{};

helper context UML!Class def : getAttributeSize : Integer =
    self.attributes.size();
```

CHAPTER 3

Profiles and their use

Chapter 2 gave an overview of Model Driven Development and its transformation languages and lightweight extensions. But only modeling profiles are often not enough. Sometimes it is necessary to transform a profile enriched model into an other model. Therefore it is crucial that transformation languages are able to process lightweight extensions. This chapter will describe the state of the art possibilities to process a profile enriched model with the Atlas Transformation Language (ATL). Another big topic in this chapter are the possibilities to extend a transformation language to process profiles with its advantages and disadvantages. The chosen variant will be described in detail in the next chapter.

3.1 Support in transformation languages

Lightweight extensions support for transformation languages is not widely spread. Only the minority of tools supports it and then only rudimentarily. RubyTL supports no profile extension at all. ATL, on the other side, supports UML profiles only trough the underlying Java API. Listing 3.1 demonstrates the complex and problematic situation when using UML profiles in ATL.

Listing 3.1: Current possibility to read UML profiles in ATL

```
module UML2WSDL;
create OUT : MM2 from IN : MM, PRO : pro;

rule Model {
  from
     model : MM! Model

  to
```

```
 9        wsdl : MM2!Definition ()
10
11     do {
12        if(not model.getValue(pro!Stereotype.allInstances()->select
              ( e | e.name = 'WSDL').first(), 'targetNamespace').
              oclIsUndefined()){
13           wsdl.targetNamespace <- model.getValue(pro!Stereotype.
                 allInstances()->select( e | e.name = 'WSDL').first(),
                 'targetNamespace');
14        }
15     }
16  }
```

The example should transform an UML model to a WSDL definition. The WSDL itself is a UML diagram with a profile for the WSDL specific parts. See Appendix A.2 for the profile definition and a test instance. The listing shows only the namespace mapping and is not complete and should only help to illustrate the problem.

The first profile related code is in line 2. The profile is defined as a second input source with the name *pro*. The rest of the related code is in the imperative part of the rule *Model*. This is necessary because accessing the tagged values requires the use of the Java API and can only accessed in the imperative part and not in the declarative part of the rule. The mapping of a tagged value will be done in two steps. The first, line *9*, should check if the desired tagged value is present. Line *10* shows the second step, the actual assigning part of the property. At the first view look both lines similar because both lines need to query the property. The actual problem lies in the query and can be subdivided into two smaller problems. But at first a closer look on how the query works in detail. The code fragment *pro!Stereotype.allInstances()->select(e | e.name = 'WSDL').first()* searches for the first stereotype with the name *WSDL*. The found element will be used along with the tagged value name *'targetNamespace'* as parameter for the method *model.getValue*. The method returns the tagged value with the name *'targetNamespace'* from the found stereotype. But this function call has a major disadvantage with the parameter for the tagged value because it is just an ordinary String. ATL supports code completion and syntax checking during development for normal elements but Strings can not be checked. Any misspelling or other mistake can not be detected by the system because it does not know which elements are present. The only way to check if the tagged element exists is to run the transformation and see if it works. Any access of a non existing tagged value results in an error. To minimize the problem has line *9* a check against *OclUndefined*. *OclUndefined* is same as *null* in Java or *nil* in Ruby, a non existing element. But the additional check rises another possible problem. Both queries need to be the same and more String variables increasing the possibility of a spelling mistake.

As showed is the support for UML very rudimentarily. Although the circumstance that UML is the most used modeling language. The second lightweight extension in this thesis are EMF profiles. It is not possible to use it with RubyTL but, with a little trick, can ATL work with EMF extensions out of the box. ATL has the advantage that it is able to use more than one metamodel during a transformation and the EMF profile metamodel is written in ECore. Those two circumstances allows ATL to use this extension. In order to get a transformation to run it is necessary to make two configurations to the Virtual Machine (VM) which executes the transformation. The regular Virtual Machine, which is used by ATL, is not capable to match all EMF Profile elements. Only the EMF-specific VM will work with all elements. The other configuration is to allow **inter-model references**. This is necessary to link the stereotype to the correct model element. Listing 3.2 shows a short example how such a transformation look like.

Listing 3.2: Current possibility to use EMF profiles in ATL

```
1  module UML2WSDL;
2  create OUT : MM1 from IN : MM, IN1 : PRO;
3
4  rule Class2Class {
5    from
6      class : MM!Class, pro : PRO!Class(pro.appliedTo = class)
7    to
8      k : MM1!Class (
9        name <- class.name,
10       abstract <- pro.abstract
11     )
12 }
```

This example uses the test models from Appendix A.4. It transforms a profile enriched class diagram to a Java diagram. The first important part is in line 2. It defines the input and output models. The Java diagram has only one single output metamodel called *MM1*. The Class diagram instead has two input models. One for the normal metamodel, named *MM* and one for the EMF profile, called *PRO*. The transformation is similar to the approach used for UML Profiles. But EMF Profiles do not need any special underlying API to access Profile information. The standard mechanisms of ATL are enough. The next important code is in line *6* and is the source element definition. It is necessary to define the normal model element from the metamodel and the corresponding stereotype from the profile. In this example the element *Class* from the metamodel and the stereotype *Class*. But it is also necessary to link both to each other. Because the rule should only take the stereotype which belongs to the model element. This is done with a guard condition. Every stereotype holds also a reference to the actual model element. This, in combination with the guard condition, allows to select the stereotype which belongs to the specific element. In this example holds the attribute *appliedTo* a direct reference to the class element. But this method has a little drawback. It may be very performance expensive because the guard condition calculates the cross product between the model element and the stereotype. The more stereotypes and elements the longer it takes to find the correct stereotype. But an advantage are the easy mappings due to the use of the declarative style, see line *10*. It allows direct mappings from the stereotype to the target element. No underlying API or other special code is necessary. Also checks against non existing fields are not necessary because ATL handles all of that.

Unfortunately it is not so easy to apply a stereotype to an element. The problem lies in the profile definition. ATL can not create the profile application correctly. Because the modeled profile definition does not represent the necessary model completely. There are differences between the modeled profile application and the generated one. Listing 3.3 shows a correct profile application. Listing 3.4 the generated one. The differences are big. It starts with the root element. Normally it should be of type *ProfileApplication* with the namespace *emfprofileapplication* but ATL uses *XMI*. Also the type of the stereotype is different. Instead of using *stereotypeApplications* as type and the stereotype type as attribute, it uses the stereotype type directly. There are also elements which can not be modeled. The first is *importedProfiles* which contains the namespace of the profile. The other is *extension* and belongs to the stereotype. It holds a reference to the stereotype in the profile definition. All this wrong mappings and missing elements are the result that ATL can only create the model according to the metamodel. But the modeled profile definitions does not contain this elements and therefore is ATL not capable of creating it.

Listing 3.3: Correct modeled EMF profile application

```
<emfprofileapplication:ProfileApplication xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" xmlns:emfprofileapplication="
    http://www.modelversioning.org/emfprofile/application/1.1"
    xmlns:myprofile="http://www.modelversioning.org/myprofile">
  <stereotypeApplications xsi:type="myprofile:Class" abstract="
      true">
    <appliedTo href="platform:/resource/ATLTest/class.xmi#/1"/>
    <extension href="default2.emfprofile_diagram#
        _aYtRgGsrEeKOMKj3EhJTvA"/>
  </stereotypeApplications>
  <importedProfiles nsURI="http://www.modelversioning.org/
      myprofile"/>
</emfprofileapplication:ProfileApplication>
```

Listing 3.4: Generated EMF profile application

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:myprofile="http://www.modelversioning.org/myprofile">
  <myprofile:Class abstract="true">
    <appliedTo href="out_class.xmi#/1"/>
  </myprofile:Class>
</xmi:XMI>
```

3.2 Approaches to extend languages

Section 3.1 showed the state of the art support of lightweight extensions in transformation languages. This section will focus on the possibilities to extend an existing transformation language so that they are able to process profiles. Manuel Wimmer and Martina Seidl from the Business Information Group at the Vienna University of Technology already pointed out three different methods to extend a language [WS09].

3.2.1 Method 1: Merging the profile with the metamodel

It is the simplest solution. The profile is getting part of the metamodel. The changes are only done in the source metamodel and there are no changes in the transformation language. Just use the changed metamodel as source for the transformation. The access to the tagged value are the same as the access to the normal properties. Listing 3.5 shows how such a transform may look like. It is similar to the example in Listing 3.1 but only with the simplification.

Listing 3.5: Merged profile example

```
rule Model {
  from
    model : MM!Model

  to
    wsdl : MM2!Definition (
      targetNamespace <- model.targetNamespace
    )
}
```

Because the circumstance that RubyTL does not supports any profile extension is the example based on ATL. This has the benefit to show the difference between the built-in and the possible solution. The biggest difference is that the imperative part was removed. The binding can be done in the declarative part because all tagged values are now normal attributes of the class. Additional commands are not needed anymore. But this has one major drawback. A class can have more than one stereotype and they can have tagged values with the same name and more than one element with the same name is not allowed and will rise an error. A possible solution for the naming problem will be not to merge the stereotype into the class. Instead convert the stereotype to a class and change the model extension relationship to a normal reference relationship. The developer can choose from which stereotype the tagged value will be used.

The most important drawback is that this solution infer with the original meaning of lightweight extensions. Because a lightweight extension is a separate model which enhance an existing model with domain specific knowledge. But the combination of both is more like a heavyweight extension. Also transformation languages may have problems with a merged metamodel. Because a transformation language which only understands a clean UML standard may have problems to process an extended UML metamodel.

But a normal ECore based transformation language, like ATL or RubyTL, will have no problem with this as long the metamodel and model are also based on ECore. This method has the advantage that no transformation language needs to be change. The transformation just need to use the merged metamodel and model. This method suits best to support many transformation languages.

3.2.2 Method 2: Pre processor for the transformation language

This method is a little bit different than Method 1. The transformation gets extended with new commands to process profiles. Before execution will a pre processor transform the extended transformation to a standard transformation. This has the advantage that the core of the transformation language does not need to be changed. Taking Listing 3.1 and ATL as transformation language as example. The pre processor will be used to transform all new introduced commands into the native and complex method (see Section 3.1). It first needs a profile definition in the header, see the *using* keyword in Listing 3.6 line *2*. It marks the profile for the pre processor so that it will use the correct file. Any further command will depend on the desired way to process a profile. An easy solution would be to add another new keyword for the source element. The keyword *apply* will extend the target model element with a stereotype. The bindings are the same as in Method 1. All new keyword are from [WS09].

Listing 3.6: An extended ATL for pre processor use

```
1  module UML2WSDL;
2  create OUT : WSDL using from IN : UML;
3
4  rule Model {
5    from
6      model : MM!Model
7
8    to
9      wsdl : MM2!Definition apply WSDL (
10       targetNamespace <- model.targetNamespace
11     )
12 }
13 end
```

The benefits of this method are that the transformation process do not need to be changed. Just running the pre processor before execution of the transformation or integrate it into the editor. A developer does not need to know the partially complex mechanisms of the transformation language to process a profile. It will also reduces the lines of code and may increase the readability of the source code.

But on the other side has this method also some major drawbacks. Every transformation language will need its own pre processor. It is not possible to use a specific pre processor for another language. Because every transformation works different and has another syntax. It is also not easy to ensure that the pre processor will transform the transformation file into the correct format. The access to the tagged values can be difficult, like ATL. An error is not be easy to find. Because the transformation language will only point to the transformed file and not to the extended one. Specific knowledge on how the transformed transformation file will look like is necessary. It is also not possible to debug the extended code, only the transformed one. Another major drawback is that this method can only be used with transformation languages which already have some profile support because this method can only simplify the native usage, not add one.

Andrea Randak used this method in her thesis. She made it possible to read and apply UML profiles in ATL. But more on that in the related work in Chapter 7.

3.2.3 Method 3: Extend the transformation language

Is the last and most drastic method. It aims to extend the transformation language directly. But extending an existing language is not an easy task. In Method 2 it was only necessary to extend the language syntax. But this is not enough for this approach. Furthermore it is necessary to extend the transformation process itself. The extended syntax can be the same as in the second method.

A native implementation has significant advantages over the other approaches. At first it has a complete tool support and a debugger can be used without problems, because there is no differences between source code and execution code or between source model and used model.

But it shares some drawbacks with Method 2. Every transformation needs its own extension. One solution can not be used for another language. Also every lightweight extension needs its own extension. It is also more time-consuming than the other methods. Because it needs specific knowledge about the transformation process. This requires also the source code of the transformation language.

CHAPTER 4

Theoretical approach

The last chapter showed the problems of transformation languages. Beginning from the complex use of the underlying Java API in ATL to the non existent support in RubyTL. The second part of the last chapter listed the possible approaches to extend an existing language with the pros and cons.

This chapter will focus on the used method in detail. The solution is based on Method 1 (see Chapter 3.2.1), but with a few little modifications. The result will not only include reading tagged values of a stereotype but also the possibility to apply tagged values and save them in the corresponding model file. The last part of this chapter will cover a generic example to demonstrate the complete approach.

4.1 Model processor

The solution is based on Method 1 of the last chapter (see Chapter 3.2.1). Because it is the easiest method and can work with different transformation languages. It requires no additional changes in any transformation language. Another reason is the possibility to add new lightweight extensions. Method 1 made the changes manually, but this solution will make it automatically. The model transformation will be done with model processors. A model processor is a technique to alter metamodels and models. They are usually used to change elements before or after processing them. Like removing elements which cannot be processed or adding special behaviour that a transformation language can not generate. In this case are model processors used to process the profile extension. To be more precisely are two model processors necessary. Each one will be used to process another task. The first processor will integrate the profile definition into the metamodel and the profile application into the model. This model processor is called **pre processor**. It runs before the transformation to provide the source files and allows to read tagged values from the profile. The other processor is called **post processor**. It runs after the transformation to split up a generated model into a profile application and a clean model.

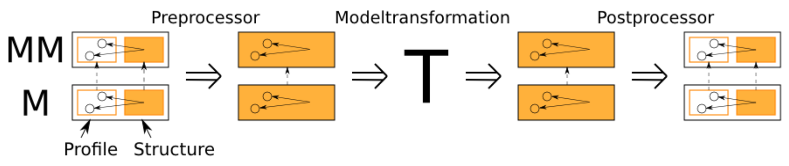


Figure 4.1: Schematic of the model processor

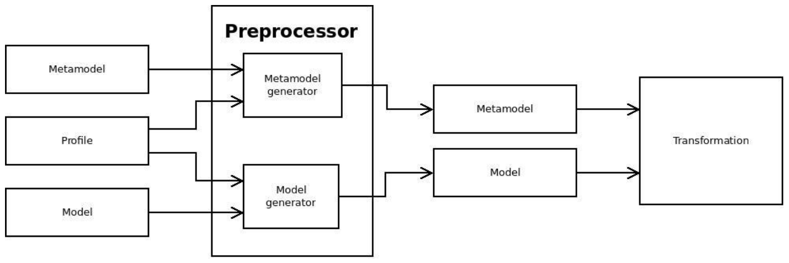


Figure 4.2: Schematic of a pre processor

With that processor it is possible to apply tagged values to a stereotype. The use of the pre or post processor is not a must. They are only used to process profiles. If the source or target does not use a profile can the corresponding model processor be omitted.

Figure 4.1 shows a schematic illustration how a transformation with both model processor works. The first step is that the pre processor takes all required input models and processes them. The result will be used as the source for the transformation. The transformation will use this generated models to create a profile enriched model. Afterwards will the model be used as input for the second processor. The post processor will split up this model into a profile application and a clean model again. Both model processor are described in detail in the next two sections.

4.1.1 Pre processor

A pre processor is responsible for merging the profile with the model. But a model consists of a metamodel and a model application, so it is necessary to process both parts separately. The metamodel gets merged with the profile definition and the model with the profile application. The result are a new metamodel with the included profile definition and a model that conforms with the new metamodel. Now it is possible to access the profile information like any other attribute of an element in the transformation. Figure 4.2 shows an overview of the pre processor. The pre processor contains two different processing units. One is responsible for merging the metamodel with the profile definition. The other merges the model with the profile application. The generated metamodel and model are used as the source for the transformation.

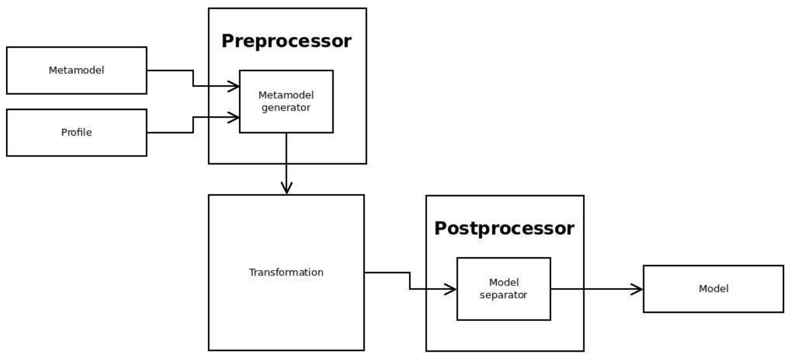


Figure 4.3: Schematic of a post processor

4.1.2 Post processor

The second model processor is the post processor. The main goal of this processor is to provide a system which allows to apply stereotypes to an element. But this system is a little bit more complex than the pre processor. The post processor can not work alone to apply a profile. Because the transformation language can not use a metamodel with a separate profile as target. Therefore it is necessary to merge the target metamodel and profile into a new and single metamodel. This will be done with the metamodel generator of the pre processor. With the new metamodel as target can the transformation language generate the model which includes the desired profile application. After the transformation is done will the post processor separate the profile application from the model. The result of this process is that it is now possible to apply a profile and have a clean model and a profile application at the end. Figure 4.3 shows a schematic illustration of this process.

4.2 Mapping

As mentioned in Chapter 3.2.1 there are two different ways to connect a stereotype to a class. The first will integrate the tagged values as attributes into the element. The other way is to transform the stereotype into a class and connects it with a reference to the extension class. The tagged values are normal class attributes. This solution will use the second approach. Because this way will preserve some of the meaning of lightweight extensions. The stereotype will not be separated and all information about will be at one place. It is also possible to distinguish between the attributes of the class and the tagged values of the stereotype. But the stereotype can easily be accessed over the relationship. Through the multiplicity of the reference a class may have an extension, but it is not a must.

But there are a few things to change because a simple copy may cause some conflicts. All stereotypes need to be renamed to avoid name conflicts because a model element and a stereotype can have the same name. But it is not possible to have more than

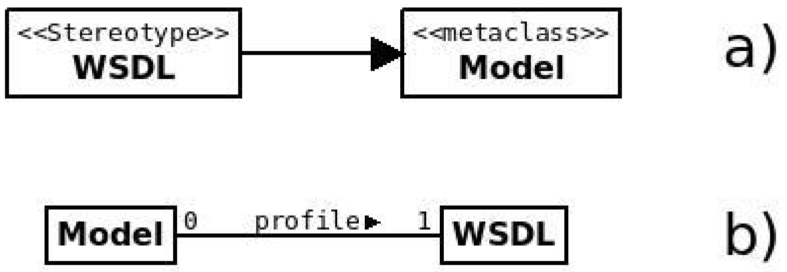


Figure 4.4: Mapping from stereotype extension to class relationship

one element with the same name in a model. The new name is a combination of the prefix *Profile*, the profile name and the stereotype name. All parts are separated with a _. This will look like **Profile_<profile name>_<element name>**. For example **Profile_profile_WSDL**. Figure 4.4 illustrates the process. The starting point is *a* and will be transformed to *b*.

The model extension relationship between the stereotype and the ECore element will be replaced with a normal reference relationship. It has the name *profile* and the multiplicity *0...1*. The multiplicity preserves the characteristic that an extended class can have one stereotype or not.

4.3 Example

This section will show with some generic examples how the results of the model processor look like. There are two different examples. One for the pre processor and one for the post processor. Those are generic examples. No real lightweight extension like UML profiles or EMF profiles are used.

Pre processor

This example shows the result of the pre processor. The pre processor merges the lightweight extension into the model. Figure 4.5 shows the both source files. The diagram marked with *a* is the model. It has only a single class named *Person*. It represents a person and has one attribute of type *String* named *name* which represents the name of the person. Due to simplicity has the class only this single attribute. On the right side of the model in Figure 4.6 is the stereotype. It is marked with a *b*. This diagram exists of two different parts. The first part is the stereotype. It is named *Student* and represents a student. Due to simplicity has the stereotype also only one single attribute. It represents the student number and is named *MatNr*. The other part is the extension point. It marks that the stereotype *Student* extends the class *Person*. But there is one more information necessary which is not modeled. The name of the profile is *University*.

After the pre processor the result will look like Figure 4.6. The profile is now merged into the model. The tagged values can be accessed over the model. The stereotype and

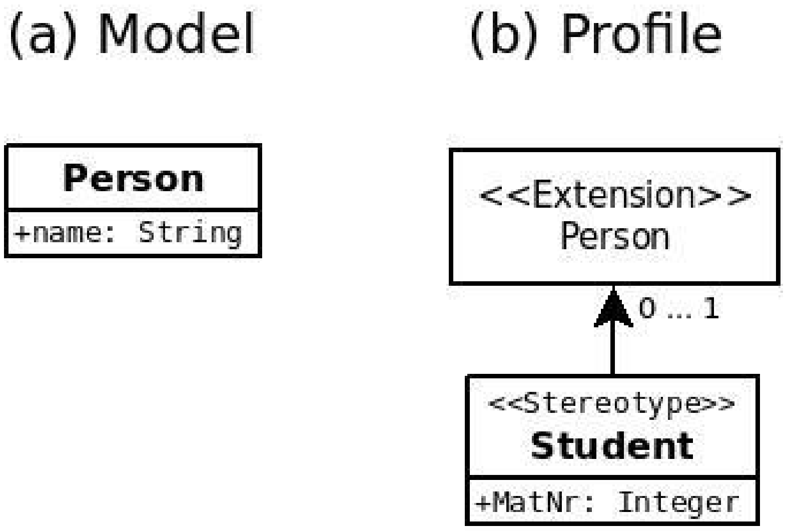


Figure 4.5: Input files for the generic pre processor example

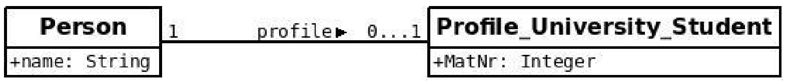


Figure 4.6: Result of the generic pre processor example

the tagged values are transformed to a normal class with attributes. Despite the changes are the graphical notation nearly the same. Only the *«Stereotype»* marker is missing and the class got renamed to *Profile_University_Student*. The extension relationship is replaced with the approach in Section 4.2. It is now a simple association with the name *profile*. Any transformation language can access this class without any special changes.

Post processor

This example works vice versa than the pre processor example. It gets executed after the transformation to separate the profile application from the model. The transformation has as target the model and profile of Figure 4.5. But the most transformation languages are not able to use profiles directly. Therefore it is necessary to use the pre processor to transform the input to an usable form, which is shown in Figure 4.6. Only with this model is the transformation language able to generate an output. This output will be separated by the post processor into a clean model and profile again.

CHAPTER 5

Practical approach

The last chapter covered the theoretical approach to achieve lightweight extension support in different transformation languages. This chapter will describe the practical approach for it.

This chapter starts with a general overview about the solution. It will explain the basic software environment. The solution includes the two introduced lightweight extension, UML profile and EMF profile. Everyone if this lightweight extensions consists of two model processors and overall three processing units (see Section 4.1). The sections gives an overview of most important steps of every processing unit.

Another part in this chapter will show how to use this solution in RubyTL and from the command line. This will be used in the last part of this chapter to show some practical examples. The examples will show how the solution works with the different transformation languages.

5.1 Solution

Both model processors are written in Ruby to allow an easier integration in RubyTL. The transformation itself is done with XML manipulation. Ruby supports in version 1.8 or higher REXML [1]. REXML is fast XML processing library which provides an intuitive API, which allows quick transformations with little code.

The development is test driven [Bec03]. Test Driven Development (TDD) means that all test models are modeled before starting the development of the model processors. This allows continuous testing of the solution and errors are easier to find and to correct. There are two different tools used to test the generated metamodels and models. The first is a ECore validation tool. It ensures that the generated output corresponds with

[1] `http://www.germane-software.com/software/rexml`

the valid ECore syntax. This implies also that the output is a valid XML. The other tool is the transformation language itself. The transformation language is necessary because the generated model may be a valid ECore model but not accessible within the transformation. The test models can be found in the Appendix A.

The solution is based on two classes. One class contains the pre processor. The other class the post processor. Furthermore follows the solution the object oriented programming paradigm. The object oriented way ensures that every different transformation logic is separated from each other. And also separated from the different starting procedures. This ensures the use of the model processors in other applications.

There are two ways to use the model processors for a transformation. The first is through the extension for RubyTL. It allows to configure and start the transformation within any Ruby code. The extension will run the corresponding pre processor if necessary and executes an appropriately configured RubyTL transformation afterward. If the target model uses a profile extension it will automatically execute the post processor after the finished transformation. The other way to use the model processors is a simple command line interface (CLI). It executes the pre processor or post processor directly, without starting a RubyTL transformation. The CLI allows the use of the model processors with any other transformation language.

5.1.1 UML profiles

There are many different ways to represent and save UML diagrams and profiles. This model processor works only with the textual form of ECore based models from the Eclipse Modeling Framework. A profile enriched UML Diagram consists of three different files. A model and a corresponding profile definition. The profile application is integrated in the model. The last file is the ECore metamodel of UML. The metamodel is designed to cover the model and the profile definition in a single file. The profile file actually contains two different description of the extension. One is in the UML definition which is described in the UML metamodel. The other is in standard ECore. Both are the same, only in a different languages. A XML document can only contain one root element, therefore is the ECore notation embedded into the the UML version. Both share the same root element but are complete independent from another.

Pre processor

As mentioned in Section 4.1 are two pre processor for UML necessary. The first merges the UML profile definition into the UML metamodel. The pre processor benefits from the circumstance that both metamodels, the UML metamodel and the profile definition, are represented in ECore. ECore defines no stereotypes in its definition. Therefore uses UML normal classes for representing stereotypes. This made it easy to integrate it into the UML metamodel. The other pre processor is responsible for integrating the profile application into the model.

Metamodel processing This pre processor works in three steps. The first step is to find all *eClassifiers* objects in the profile and copy them into the UML metamodel. They containing all modeled stereotypes. There is no search for unused elements, it is assumed that all modeled elements are used. All elements get renamed to the schema described in Section 4.2 to avoid name conflicts. All references and generalizations are part of the elements and therefore also copied.

The second step is to remove two unnecessary parts from the copied element. At first the extension reference. This is the reference between the stereotype and the UML element. In the ECore representation it is a element of type *eReference*. The other part are annotations which references to the corresponding UML representation in the profile. This is necessary because only the ECore part gets copied and the UML elements does not exist in the UML metamodel.

The last step is to create the relationship the between extended UML element and the copied element. Only those elements which got extended from a stereotype gets a relationship. As described in Section 4.2 is this relationship a simple reference from the UML element to the copied stereotype. This is realized with an element of type *eReference* from the element to the stereotype. The reference is named *profile* and has a multiplicity of *0...1*.

Model processing The other pre processor transforms the UML model. As mentioned before are the model and the profile application in the same file. Different namespaces distinguishes them from another. The normal UML model uses the namespace *uml*. The namespace of the profile application gets defined in the profile. Both are serialized in XMI. This makes it easy to transform the model. Listing 5.1 shows a simplified version of the testmodel in Appendix A.2. It illustrates only the model element and its corresponding stereotype. The first step is to find every element which is extended by a stereotype and append the stereotype as a child. Every extended element has an *reference* element which *href* attribue points to the stereotype. Additionally it is necessary to remove the namespace and change the name of the element to *profile*. Listing 5.1 shows that *uml:Model* is an extended element and *wsdl:WSDL* is the corresponding stereotype. The attribute *base_Model* in the stereotype contains the ID of the extended element. But the attribute name is not always the same. It consists of *base_* and the element name. In this case *Model*.

Listing 5.1: Simplified source model of the testmodel in Appendix A.2

```
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/
    spec/XMI/2.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore
    " xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML"
    xmlns:wsdl="http:///schemas/wsdl/_zwd9ENCAEd-q-MT17yoFPQ/0"
    xsi:schemaLocation="http:///schemas/wsdl/_zwd9ENCAEd-q-
    MT17yoFPQ/0␣wsdl.profile.uml#_zwfyQdCAEd-q-MT17yoFPQ">
  <uml:Model xmi:id="_i2oHsKCUEd-jTfNz9tMbgQ">
```

```
    <profileApplication xmi:id="_CFtkUNCBEd-q-MT17yoFPQ">
      <eAnnotations xmi:id="_CFuycNCBEd-q-MT17yoFPQ" source="http://www.eclipse.org/uml2/2.0.0/UML">
        <references xmi:type="ecore:EPackage" href="wsdl.profile.uml#_zwfyQdCAEd-q-MT17yoFPQ"/>
      </eAnnotations>
      <appliedProfile href="wsdl.profile.uml#_wlh6QCeLEd-04aLxBSBikA"/>
    </profileApplication>
  </uml:Model>
  <wsdl:WSDL xmi:id="_DNu8UNCBEd-q-MT17yoFPQ" Address="www.test.com" targetNamespace="tns" base_Model="_i2oHsKCUEd-jTfNz9tMbgQ" encoding="UTF-8"/>
</xmi:XMI>
```

Every profile defines its namespace location and its physical location in the root XMI element. These elements are *xsi:schemaLocation* and *xmlns:<profile namespace>*. Those definitions need to be removed because they are not needed any more and the defined location is in the most cases a special Eclipse path. It is used to address the profile and this is in the most cases inside an Eclipse project. But RubyTL does not run inside an Eclipse environment and tries to to load all schemas while parsing the metamodel. This will throw an exception because of the invalid path. Both are also unnecessary because the profile gets removed and there is no need for the elements any more. The last step is to remove the *profileApplication* element. It points to the ECore and UML version of the profile definition. Listing 5.2 shows the result from the example in Listing 5.1 after the pre processor. The stereotype is now an integrated part of the model.

Listing 5.2: Result of the UML pre processor for example in Listing 5.1

```
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML">
  <uml:Model xmi:id="_i2oHsKCUEd-jTfNz9tMbgQ">
    <profile xmi:id="_DNu8UNCBEd-q-MT17yoFPQ" Address="www.test.com" targetNamespace="tns" encoding="UTF-8"/>
  </uml:Model>
</xmi:XMI>
```

Post processor

As illustrated in Figure 4.3 is only one model processor for the output model of the transformation necessary. But compared with the pre processor it is more complicated. The target metamodel for the transformation needs to be provided by the UML pre processor. Because the transformation language does not accept a profile as additional target. Therefore is the generated model, from the transformation, based on this new metamodel. The profile application is part of the model. It is necessary to separate the profile from the model to get the result.

The first is step is to restore the root element and its attributes, like the namespace. An UML model can have *xmi:XMI* or *uml:Model* as root element. But the profile application is described outside the *uml:Model* element. Therefore it is necessary to use *xmi:XMI* as root element and has the *uml:Model* element and the profile application as children. If a UML model has no profile it is possible to use *uml:Model* as root element. Changing the root element also requires to move the attributes from the old root element to the new one.

The next step is to identify the profile stereotypes and append it to the root profile element. A stereotype element is easy to find, because it has the name *profile*. But to make sure that the found element is really a stereotype is a check against the profile definition necessary. The profile definition is also required to get the name and the namespace for the element.

The last step is to create the *profileApplication* element. It defines both, the UML and the ECore version, extension elements. See lines *3-8* in Listing 5.1. This element is split into two major parts. One references to the root element of the profile and is called *appliedProfile*. The other refers to the ECore notation. It is called *eAnnotations*.

5.1.2 EMF profiles

EMF profiles working a little bit different than UML profiles. They are using four files instead of three. The profile application is separated from the model. The metamodel and the profile are in the other two files. The EMF model processors also using the textual representation for the models. The metamodel and the profile definition need to be in the ECore format and the model and profile application need to be XMI serialized.

Pre processor

Like the UML pre processor need the EMF pre processor two different model processors. One for the metamodel and one for the model. The stereotypes of the profile are derived from normal ECore *EClasses*. The attribute definition is the same as in a normal *EClass*. This makes it easy to reuse the stereotype without bigger adaptions.

Metamodel processing The merging of the profile into the metamodel only needs a few steps. The first step is to find all model elements in the profile definition. There is no search for unused elements. It is assumed that all modeled elements are used. After renaming the elements to the schema described in Section 4.2, will the elements be appended to the metamodel.

The next step is to change every stereotype to a normal *EClass*. The EMF profile uses an element which is derived from *EClass* for the stereotype declaration and has the name

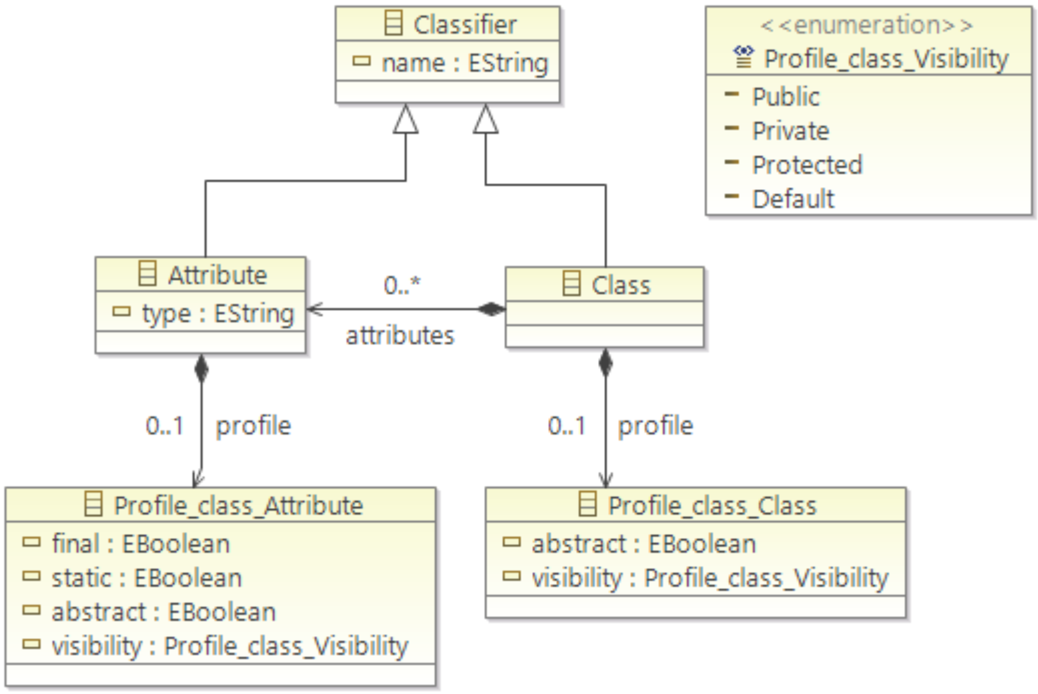


Figure 5.1: Result of the EMF pre processor

emfprofile:Stereotype. The model extension relationship is an attribute. Every stereotype contains its own model extension relationship to its extended element in the metamodel. The structure of ECore makes it easy to change the stereotype. The first step is to remove the *eSuperTypes* child element which refers to the stereotype defined in the EMF profile definition. All other super classes with the type *emfprofile:Stereotype* need to be changed to the type *ecore:EClass*. Because those elements are modeled super classes and need to be preserved. The next step is to change the type of the metamodel element from *emfprofile:Stereotype* to *ecore:EClass*. Other elements, like an enumeration, do not need to be changed.

Before removing the model extension relationship it is necessary to add the relationship between metamodel element and stereotype. Because only the model extension element contains the name of the extended element. The new relationship is described in Section 4.2.

RubyTL needs two other adaptions to work with the new metamodel. The attribute which describes the type of the element need to be changed from *xmi:type* to *xsi:type*. EMF profile uses *xmi:type* and the metamodel *xsi:type* but RubyTL only works with *xsi:type*. The other RubyTL related change is to ensure that all references to other elements are addresses with the full name and not with the ID of the element. Figure 5.1 shows the final metamodel.

Model processing The other pre processor for the EMF profiles is responsible for merging the model and the profile application. This model processor is also more complicated than its UML counterpart. Because the model and the profile application have a different format. Every tagged value is embedded in the *stereotypeApplications* XML element. It also contains the reference to the extended model element. The first step is to change the element from *stereotypeApplications* to *profile*. The element also contains the type of the stereotype and need to be removed in order to get a clean element.

The most complex part is to find the correct element in the model to add the stereotype as a child. The EMF profile describes the reference with the absolute position of element inside the model. Listing 5.3 shows an example how this reference may look like. All parts are separated with a /. The first number points to the position of the first element. To the third element in case of the example of listing 5.3, because the counting starts at 0. The following parts referring to the name of the child element and its position within the element. *@attributes.0* points to the first element with the name *attributes*. All the following parts have the same schema as the *attributes.0* example. After appending the profile as a child element can the *appliedTo* element be removed.

Listing 5.3: Example of the EMF profile application reference to a model element

```
#/2/@attributes.0
```

Post processor

This post processor will create a new profile application file. The first step is to create a basic structure. See Listing 5.4. It contains the root element and all necessary namespaces and configurations. Every profile application has its own namespace. This namespace needs to be added after creating the basic structure. The namespace is defined in the EMF profile definition.

Listing 5.4: Basic structure of an EMF profile Application file

```
<emfprofileapplication:ProfileApplication xmi:version='2.0'
    xmlns:xmi='http://www.omg.org/XMI' xmlns:xsi='http://www.w3.
    org/2001/XMLSchema-instance' xmlns:emfprofileapplication='
    http://www.modelversioning.org/emfprofile/application/1.0'>
</emfprofileapplication:ProfileApplication>
```

The next step is to find all stereotypes and change them to the proper format. The first part is to restore the stereotype type. It can be found in the profile definition. Also the name needs to be changed from *profile* to *stereotypeApplications*. The most difficult part is to calculate the position of the element. See Listing 5.3 for an example. It is necessary to travel the elements back to the root element and record the path in order to get the absolute position. The position will be used with the path to the model as attribute to reference to the correct position inside the model.

The last part is to restore the reference to the profile definition. This is done with the full namespace of the profile.

5.2 How to use the solution

There are two ways to start the model processors. The first way can be used inside every Ruby application. It allows to configure the complete transformation and execute all steps automatically, including the RubyTL transformation. The other possible way to use the model processors is from a command line interface (CLI). It allows the start of every model processor and enables the use within other transformation languages. But all model processors and the transformation need to be executed manually.

5.2.1 From Ruby

Allows the usage of a profile transformation inside any Ruby code. It executes all necessary model processors and the RubyTL transformation automatically. The configuration for the transformation is similar to RubyTL. It has three main configuration parts. The source models, the target models and the transformation. The source model and the target model have additional configuration options, but are in both parts the same. They specifying the different modeling files, like the metamodel or the model. The parameters *package*, *metamodel* and *model* are the same as in RubyTL. The *package* specifies the name which will be used to address the models in the transformation. The paths to the metamodel and the model gets defined by the corresponding parameters *metamodel* and *model*. To address the profile and profile application were two new parameters introduced. They have the name *profile* and *profileApplication*, but are not mandatory. Profile enriched UML models only needs the profile parameter. EMF profiles instead need both parameters. The last transformation configuration is the transformation file.

But the transformation configuration is not enough to run the transformation. It also needs the name of the transformation and the base path to the modeling project. Because all paths in the transformation configuration are relative paths but the transformation needs absolute paths. The method *transform* will start the transformation. Listing 5.5 shows the schema of a transformation where the source has an EMF profile and the target no profile.

Listing 5.5: Example how a transformation from Ruby may look like

```
task = ProfileTransformationTask.new('<name>', '<
    path_to_project>') do |t|
  t.sources :package => '<source_name>',
    :metamodel => '<relative_path_to_source_metamodel>',
    :model => '<relative_path_to_source_model>',
    :profile => '<relative_path_to_source_profile>',
    :profileApplication => '<
       relative_path_to_source_profile_application>'

  t.targets :package    => '<target_name>',
     :metamodel => '<relative_path_to_target_metamodel>',
     :model => '<relative_path_to_target_model>'

  t.transformation '<relative_path_to_transformation>'
end

task.transform
```

The automatic execution works in three steps. At first it checks what model processor is needed. This will be done over the specified profile and profile application. It will use the EMF model processors if both, the profile and the profile application, parameters are present. The UML profile instead only needs the *profile* parameter. Or no model processor if none of the new parameters are set. It will also generate the target metamodel if the target uses a profile. Those generated models will be used within a standard RubyTL transformation. This RubyTL transformation is the second step. The method generates a standard RubyTL starting script with the merged metamodels and models as source and target and executes a transformation. In step three will the corresponding post processor separate the profile application from the model, but only if a post processor is required.

5.2.2 From the command line

It is also possible to start all model processors with the specific parameters from the Command Line Interface (CLI). The CLI allows the use of the model processors within other transformation languages, like ATL. The only difference is that all steps need to be done manually. First use the pre processor to create the corresponding metamodel and model. Then use the generated models for the transformation. The resulting output model will be used, along with the metamodel and profile, as parameter for the post processor. There are six different configuration for the parameters. Every profile type has its own pre and post processor. But the number of arguments are different. It depends on the model processor and the used profile type. The first two arguments are always present. They defining what model processor should be executed. But all arguments need to be in the specified order and the paths for the metamodels and models need to be the absolute path to the corresponding file.

To start any model processor from the CLI it is necessary to have an installed Ruby which can be accessed from the command line. All model processors can be started from the same file. Only the arguments distinguishes them. Listing 5.6 shows how to start a model processor from the command line.

Listing 5.6: Start command for the CLI

```
ruby modelprocessor.rb <agruments for the different model
    processors>
```

The different arguments looks like followed.

UML profile pre processor Is used to generate the merged metamodel and model for UML diagrams. It has seven different arguments which are shown in Listing 5.7. The first two are *-uml -pre* for defining to use the UML profile pre processor. The next three arguments defining the metamodel, model and the profile path. The generated metamodel and model will be written to *output_metamodel* and *output_model*.

Listing 5.7: CLI arguments for the UML profile pre processor

```
-uml -pre <model> <metamodel> <profile> <output model> <output
    metamodel>
```

UML profile post processor This command (see Listing 5.8) separates the model and the profile application from each other. The configuration parameters for this model processor are *-uml -post*. It also needs the model, the metamodel and the profile as arguments. The separated model will be written to *output_model*. The target metamodel for the transformation can be generated with the command shown in Listing 5.9. It takes only the metamodel and the profile as input and writes the generated metamodel to *ouput_metamodel*.

Listing 5.8: CLI arguments for the UML profile post processor

```
-uml -post <model> <metamodel> <profile> <output model>
```

Listing 5.9: CLI arguments for the UML profile post processor to generate the target metamodel for the transformation

```
-uml -postmetamodel <metamodel> <profile> <output metamodel>
```

EMF profile pre processor Generates the metamodel and model for EMF profile enriched models. It gets configured with *-emf -pre*. The next four arguments are the input models. The last two arguments are the output models. See Listing 5.10 for the complete command.

Listing 5.10: CLI arguments for the EMF profile pre processor

```
-emf -pre <model> <metamodel> <profile> <profile application> <
    output model> <output metamodel>
```

EMF profile post processor Separates the profile application from the transformation output model. It has seven different arguments and are specified in Listing 5.11. The first two arguments are *-emf -post*. Followed by the model, the metamodel and the profile. The output will be written to *output_model* and *output_profile_application*. The target metamodel for the transformation can be generated with the command in Listing 5.12. It takes the paths to the metamodel, the profile and the output as arguments.

Listing 5.11: CLI arguments for the EMF profile post processor

```
−emf −post <model> <metamodel> <profile> <output model> <output
    profile application>
```

Listing 5.12: CLI arguments for the EMF profile post processor to generate the target metamodel for the transformation

```
−emf −postmetamodel <metamodel> <profile> <output metamodel>
```

5.3 Practical examples

This section will show how to use this solution with a few practical examples. All example models can be found in the Appendix. There are four different example, which cover all scenarios. Two for the UML profiles and two for the EMF profiles. Every extension type has an example for reading and applying stereotypes. Part of the examples are the way to describe the transformation rules. This includes all important steps to read and apply stereotypes. Another part will describe the starting mechanics in detail. It will shown how to configure the model processors for RubyTL as well as the Atlas Transformation Language (ATL). ATL will be used to demonstrate that the solution also works with other transformation languages than RubyTL.

5.3.1 RubyTL

Since RubyTL and the model processors share the same basis, Ruby, it is very easy to use them together. Therefore it was possible to extend the original starting procedure to support the model processors. But it is also possible to use the model processors alone inside any Ruby script. The programming language itself was not extended. All standard commands can be used to describe the mapping. The basic programming flow is the same as with no model processors.

Start a transformation

There are two possibilities to use the model processors. The first can be used inside any Ruby code and will also start a RubyTL transformation. It has a basic and a transformation configuration part. It is very similar to the configuration of a standard RubyTL transformations. Only the corresponding profile and profile application are added. Listing 5.13 shows an example how a transformation configuration may look like.

Listing 5.13: Example of a transformation configuration for EMF profiles

```
task = ProfileTransformationTask.new('Class2Java', 'C:\\
    Class2Java') do |t|
  t.sources :package => 'Class',
    :metamodel => 'metamodels/Class.ecore',
    :model => 'models/class.xmi',
 5      :profile => 'metamodels/class.emfprofile_diagram',
 6      :profileApplication => 'models/class.emfprofile.xmi'
 7
 8    t.targets :package => 'Java',
 9      :metamodel => 'metamodels/Java.ecore',
10      :model => 'models/java.xmi'
11
12    t.transformation 'transformations/transformation.rb'
13 end
14
15 task.transform
```

The example configures a transformation which uses a model with an EMF profile as the source and a normal model as target. The example models are the same as in Appendix A.4. The transformation is called **Class2Java** and the project is located at **C:\\Class2Java**. This was the first main configuration. The path to the project is required because all other paths for the models are relative paths inside the project folder. The next configuration part are the source files. The source model is named **Class**. The name is needed to identify the model elements inside the transformation and to distinguish the elements from the target. The metamodel and model definition are the same as in RubyTL. The EMF profile definition **class.emfprofile_diagram** gets defined with the **profile** keyword. The corresponding profile application **class.emfprofile.xmi** gets defined with **profileApplication**. All paths are relative to the base path defined in line 1. This means **C:\\Class2Java** and **metamodels\class.emfprofile_diagram** gets to **C:\\Class2Java\metamodels\class.emfprofile_diagram**. The target is called **Java** and defines only a metamodel and model. The transformation file is the last configuration part and is located at **transformations/transformation.rb**. The last line of the example will start the transformation. This code snippet can be used inside any Ruby application. The automated configuration and execution is done object oriented. The configuration of the transformation is done in the Object creation. The execution can be started at any point with the call of the **transform** method.

The second possibility to use the model processors is from the command line. The main purpose of the command line interface is to allow the use of the model processors in other transformation languages besides RubyTL. But it does not start a transformation, all steps need to be done manually. It supports all types of model processors. To distinguish them from another are the first two command line parameter a switch. Listing 5.14 shows an UML pre processor. The UML test models can be found in the Appendix A.2.

Listing 5.14: Command line example of an UML pre processor

```
ruby modelprocessor.rb −uml −pre
  C:\\Class2Java\models\diagram.uml
  C:\\Class2Java\metamodels\UML.ecore
  C:\\Class2Java\metamodels\wsdl.profile.uml
  C:\\Class2Java\output\generated_diagram.uml
  C:\\Class2Java\output\generated_metamodel.ecore
```

The first line shows the program start with the necessary parameters for the UML pre processor. The next three lines are the necessary model, UML metamodel and UML profile. These three models are used as inputs for the pre processor. The output is written into the last two defined files.

The command line interface also supports the post processor. The command line switch for the post processor is **-post**. Listing 5.15 shows an example of an UML post processor. This model processor is used to merge an UML based WSDL model and a corresponding WSDL profile. See Appendix A.2 for a detailed description of the metamodel and the corresponding profile.

Listing 5.15: Command line example of an UML post processor

```
ruby modelprocessor.rb -uml -post
  C:\\Class2Java\models\diagram.uml
  C:\\Class2Java\metamodels\UML.ecore
  C:\\Class2Java\metamodels\wsdl.profile.uml
  C:\\Class2Java\output\final_diagram.uml
```

The first line is similar to the pre processor of Listing 5.14. It shows the program start and the parameter switch for the UML post processor. The first three files defining the involved models. The last file is for the separated model.

The post processor can not work alone. The transformation also requires a merged metamodel as a target to generate the output model. Therefore it is necessary to run an additional command before using the transformation. Listing 5.16 shows the associated command for the metamodel used in the command of Listing 5.15.

Listing 5.16: Command line example to generate a target UML metamodel for the transformation

```
ruby modelprocessor.rb -uml -postmetamodel
  C:\\Class2Java\metamodels\UML.ecore
  C:\\Class2Java\metamodels\wsdl.profile.uml
  C:\\Class2Java\output\generated_diagram.uml
```

The command is very similar to the command of the pre processor. But with the difference that the parameter switch is **-postmetamodel** and only needs the metamodel and the profile definition as input. They will be defined in the first two arguments. The generated metamodel will be written into the last file.

UML profiles

Unfortunately it is not possible to use UML profiles with RubyTL. The problem are not the model processors. It lies in RubyTL or in the used ECore parser of RubyTL.

One of them is not capable to parse complex ECore metamodels correctly. ECore has the same features as EMOF, but with some Java extensions. Every model element can be enhanced with Java information, like a Java type. Those elements are the problem. Every metamodel, not only UML, which uses such a special element will end up with errors. Even ECore as metamodel has this problem.

Even after the elimination of the problematic elements was is not possible to use UML profiles. RubyTL was able to read the metamodel without errors and runs through all rules. But it did not executed any mapping procedure. It seems that RubyTL was not able to find appropriate element from the model in the metamodel.

EMF profiles

Accessing profile information in a RubyTL transformation is pretty simple. Listing 5.17 shows a rule which transforms a generic class with an EMF profile stereotype to a Java class. No special commands are necessary to access the stereotypes. See Appendix A.4 and Appendix A.5 for the complete metamodels and EMF profile.

Listing 5.17: Example for reading EMF profile information in RubyTL

```
1  rule 'class2javaclass' do
2    from   Class::Class
3    to     Java::Class
4
5    mapping do |klass, java|
6      java.name = klass.name
7
8      if klass.profile != nil
9        java.visibility = klass.profile.visibility
10       java.abstract = klass.profile.abstract
11     end
12
13     java.attributes = klass.attributes.select { |f| f.kind_of?
           Class::Attribute }
14   end
15 end
```

The profile reading part is covered in the if-condition at line *8-11*. According to Section 4.2 is the profile information stored in a separate class. The reference to that class is called *profile*. But an element does not need to have a stereotype and due the mapping of the stereotype to a simple class will the access to it result in an error. Therefore it is necessary to check if there is a existing stereotype. RubyTL maps the model internally to concrete Ruby classes to processes them. This makes a check very easy, because if there is no stereotype then there is also no Ruby class. Line *8* shows the check against *nil*. *nil* in Ruby is the same as *null* in Java. It stands for a non existing object.

Lines *9* and *10* are two simple bindings. Because of the internal class representation of the model it is possible to access other classes attributes. Both, **klass.profile.visibility** and **klass.profile.abstract**, will access the tagged values **visibility** and **abstract** of the stereotype. Those values are used to set the appropriate attributes for the target model.

Applying EMF profiles

Applying information to stereotypes is applying information to another classes. According to Section 4.2 gets every stereotype transformed into a class. Therefore it is possible to use normal rules with normal bindings to apply any tagged value. Listing 5.18 shows an example how such a transformation may look like. This example is also based on the test models of Appendix A.4 and Appendix A.5. The Java metamodel is used as source and the class diagram with the EMF profile is the target.

Listing 5.18: Example for applying EMF profile information in RubyTL

```
rule 'klass2javaclass' do
  from   Java::Class
  to     Class::Class

  mapping do |source, klass|
    klass.name = source.name

    klass.profile = source

    klass.attributes = source.attributes.select { |f| f.kind_of? Java::Attribute }
  end
end

rule 'klass2profile' do
  from   Java::Class
  to     Class::Profile_Class_profile_Class

  mapping do |klass, profile|
    profile.abstract = klass.abstract
    profile.visibility = klass.visibility.name
  end
end
```

This example code transforms a Java class to a generic class. All additional information which can not be modeled have corresponding tagged values in a separate stereotype. A complete transformation of a stereotype enriched element needs two different rules. One for the normal element and one for the stereotype. Both rules have the same *from* element, because the same source element provides the information for both elements. The rule *klass2javaclass* contains the bindings for the normal class element. Rule *klass2profile* instead contains the bindings for the stereotype information. The *to* part of the stereotype rule generates the stereotype class **Class::Profile_Class_profile_Class**. The bindings inside the mapping part are just normal bindings like in the other rule. The reflection in line *8* calls the second rule and takes care that the result, the transformed stereotype, will be set as stereotype for the element.

5.3.2 ATL

It is possible to use the model processors with ATL. But they can only be used manually and not automatically as with RubyTL. Without changes in the ATL starting procedure it is not possible to integrate the model processors in ATL for automated use. In order to use it, it is necessary to use the command line interface of the model processors. The generated metamodel and model are used as source or target models for the ATL transformation. With the model processors is ATL capable to use both profile types. No other tools or extensions are required.

UML profiles

To use ATL with UML profiles it is necessary to run the UML specific pre processor before using the transformation. The generated metamodel and model will be used as source model for the ATL transformation. Listing 5.14 shows the command to generate the metamodel and the model. This example uses the models from Appendix A.2 as source and Appendix A.3 as target. It transforms a simplified UML based WSDL model with an UML profile extension for some WSDL specific parts to a normal WSDL metamodel. Due to the native support of UML Profiles in ATL will this part of thesis show the difference between both ways. Listing 5.19 shows the transformation of the base WSDL element with the native method of ATL to access UML profiles.

Listing 5.19: Example for reading UML profile information with the native ATL method

```
rule Model2WSDL {
  from model : MM!Model
  to wsdl : MM2!WSDL (
    name <- model.name
  )

  do {
    if(not model.getValue(pro!Stereotype.allInstances()->select
        ( e | e.name = 'WSDL').first(), 'targetNamespace').
        oclIsUndefined()) {
      wsdl.namespace <- model.getValue(pro!Stereotype.
          allInstances()->select( e | e.name = 'WSDL').first(),
          'targetNamespace');
    }
    if(not model.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'WSDL').first(), 'Address').oclIsUndefined()) {
      wsdl.address <- model.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'WSDL').first(), 'Address');
    }

    if(not model.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'WSDL').first(), 'encoding').oclIsUndefined()) {
      wsdl.encoding <- model.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'WSDL').first(), 'encoding');
    }

    for(a in MM!Activity.allInstances()) {
      wsdl.operations <- thisModule.Activity2Operation(a);
    }
  }
}

rule Activity2Operation (acc : MM!Activity) {
  to op : MM2!Operation (
    name <- acc.name
  )

  do {
    for(p in acc.ownedAttribute) {
      if(p.name.startsWith('error')) {
        op.errors <- thisModule.Class2Data(p);
      } else {
        if(not p.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'InOut').first(), 'InOutString').oclIsUndefined()) {
          op.InOut <- p.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'InOut').first(), 'InOutString');

          if(p.getValue(pro!Stereotype.allInstances()->select( e | e.name = 'InOut').first(), 'InOutString').equals('In')) {
                op.input <- thisModule.Class2Data(p);
            }
            if(p.getValue(pro!Stereotype.allInstances()->select(
                e | e.name = 'InOut').first(), 'InOutString').
                equals('Out')) {
                op.output <- thisModule.Class2Data(p);
            }
        }
      }
    }
    op;
  }
}
```

This rule reads the three base attributes from the stereotype and maps it to the target model. As described in Section 3.1 requires ATL two major steps to read a tagged value. The first checks if a tagged value exists. The second reads the tagged value. It is complex and error prone. Listing 5.20 instead uses the UML pre processor for the transformation with the same source. The profile mapping is done at the lines *8-12*. At the first view has the model processor method a code reduction and a simplification. It starts with the existence check of the stereotype, as with RubyTL. This is necessary because an element can have a stereotype, but it is not a must. But the existing check with the model processor is easier than with the native method. Because with the native method it is necessary to check for every single tagged value. The model processor method instead only needs to check if the stereotype exists. But with ATL is that easy because the check supports syntax completion and OCL commands. And therefore very faultless. Otherwise will the access to an attribute of a non exiting stereotype result in an error. The three lines inside the if-condition are the mapping of attributes.

Listing 5.20: Example for reading UML profile information with the model processors in ATL

```
1  rule Model2WSDL {
2    from model : MM!Model
3    to wsdl : MM1!WSDL (
4      name <- model.name
5    )
6
7    do {
8      if(not model.profile.oclIsUndefined()) {
9        wsdl.address <- model.profile.Address;
10       wsdl.encoding <- model.profile.encoding;
11       wsdl.namespace <- model.profile.targetNamespace;
12     }
13
```

```
    for(a in MM!Activity.allInstances()) {
      wsdl.operations <- thisModule.Activity2Operation(a);
    }
  }
}

rule Activity2Operation (acc : MM!Activity) {
  to
    op : MM1!Operation (
      name <- acc.name
    )

  do {
    for(p in acc.ownedAttribute) {
      if(p.name.startsWith('error')) {
        op.errors <- thisModule.Class2Data(p);
      } else {
        if(not p.profile_InOut.oclIsUndefined()) {
          op.InOut <- p.profile_InOut.InOutString;

          if(p.profile.InOutString.equals('In')) {
            op.input <- thisModule.Class2Data(p);
          }

          if(p.profile.InOutString.equals('Out')) {
            op.output <- thisModule.Class2Data(p);
          }
        }
      }
    }
    op;
  }
}
```

The major difference between both methods is that the model processor method needs only one existing check and that check is easier than the native method. No stereotype search is necessary. Through the fact that the stereotype is now an integral part of the metamodel, it is possible for ATL to use its build-in syntax highlighting and syntax correction. A mistake in the attribute name can be detected and fixed. Also the tagged values have this advantage, because they are normal attributes of a class and therefore it is also possible to use the syntax highlighting and syntax correction. This will reduce the error-proneness of the transformation. It also reduces the total line of code and increases the readability.

Unfortunately needs the UML model processor a special workaround. The workaround is necessary because of the possibility to use an **UML Port** as an **owned attribute**. A port specifies a specific interaction point to exchange an UML element. It has a name and a datatype. But the UML element **ownedAttribute** does not point to an UML port element in the ECore notation. It points to the datatype of the port. The port element is never created. The ownedAttribute element uses the **xmi:type** attribute instead to mark the owned attribute as port. But technically it has not the type **UML Port**. But a port can be extended with a stereotype. The model processor detects the extension relationship and makes a normal reference from the port to the stereotype. If a port is used as an owned attribute, then the stereotype will point the to owned attribute. But as described before is the owned attribute not from the type **UML Port**. So it is not possible to access those stereotype. Therefore needs the **ownedAttribute** element to be extended with a reference to the stereotype to support the Port stereotype. Afterwards it is possible to access the stereotype in the same way as every other stereotype. The lines **26-42** of Listing 5.20 shows such an access.

Applying UML profiles

It is also possible to apply stereotypes. The main process of applying stereotypes is the same as with RubyTL. Every extended UML element has a reference to the stereotype. And every stereotype can be generated. Listing 5.21 shows the same example as in Listing 5.20 but with swapped source and target. It shows the transformation of the main WSDL element in rule *WSDL2Model* and the stereotype generation in rule *WSDLprofile*. Line *8* calls the rule which maps all required values from the WSDL element to the stereotype. After a successful transformation can the UML post processor be used to separate the profile application from the model to generate a pure UML model with profile extension.

Listing 5.21: Example for applying UML profile information with the model processors in ATL

```
1   rule WSDL2Model {
2     from wsdl : MM!WSDL
3     to model : MM1!Model(
4       name <- wsdl.name
5     )
6
7     do {
8       model.profile <- thisModule.WSDLprofile(wsdl);
9
10      for(x in wsdl.operations) {
11        model.packagedElement <- thisModule.Operation2Activity(x)
              ;
12      }
13    }
14 }
15
16 rule WSDLprofile(wsdl : MM!WSDL) {
17   to profile : MM1!Profile_wsdl_WSDL()
18
19   do {
20     profile.Address <- wsdl.address;
21     profile.encoding <- wsdl.encoding;
22     profile.targetNamespace <- wsdl.namespace;
23     profile;
24   }
25 }
```

As mentioned before is the process very similar to RubyTL. It requires two rules to transform an element with an extended stereotype. The first rule will transform the base element. The second one the stereotype. This is necessary because of the transformation of the stereotype to a class. Line *8* will call stereotype rule, which is actually a called rule and need to be called separately. Rule *WSDLprofile* from line *16-25* shows three simple bindings to set the tagged values. The last line in the do-block will will return the completed stereotype and will be set in line *8* as stereotype for the element.

Using EMF profiles

With the model processors it is also possible to use EMF profiles with ATL. The usage is very similar to the usage with UML profiles. Listing 5.22 shows the transformation of a generic class model with a Java profile extension to Java model. The used metamodels and profile are illustrated in Appendix A.4 and Appendix A.5. The listing only transforms the classes and the corresponding attributes. As with UML it is necessary to check if a profile is present. Shown in line *8* and *26*. The mapping of the profile attributes are done in the lines *8-9* and *27-28*. The mapping need to be done in imperative part, because the declarative part does not allow imperative code, like if-condition.

Listing 5.22: Example for reading EMF profiles with ATL

```
1  rule class2javaclass {
2    from class : MM!Class
3    to java : MM1!Class (
4      name <- class.name
5    )
6
7    do {
8      if(not class.profile.oclIsUndefined()) {
9        java.visibility <- class.profile.visibility.toString();
10       java.abstract <- class.profile.abstract;
11     }
```

```

    for(a in class.attributes) {
      java.attributes <- thisModule.attribute2attribute(a);
    }
  }
}

rule attribute2attribute (source : MM!Attribute) {
  to att : MM1!Attribute()

  do {
    att.type <- source.type;
    att.name <- source.name;

    if(not source.profile.oclIsUndefined()) {
      att.visibility <- source.profile.visibility;
      att.abstract <- source.profile.abstract;
    }

    att;
  }
}
```

Applying EMF profiles

It is also possible to apply stereotypes to the elements as well. The basic way is very similar to RubyTL and the UML method of ATL. Through the circumstance that the stereotype is an ordinary element it is necessary to create the stereotype like any other element. This is done with a called rule, named *ClassProfile*. The rule takes the extended element as input parameter and returns the generated element to assign it as a stereotype. Line *8* shows the call of the called rule and the assignment. The actual mapping will be done in the lines *20-21*. The appropriate post processor will separate the profile application from the model at the end. The commands for the EMF profile post processor are described in Section 5.2.2.

Listing 5.23: Example for applying EMF profiles with ATL

```
1  rule klass2javaclass {
2    from java : MM!Class
3    to class : MM1!Class (
4      name <- java.name
5    )
6
7    do {
8      class.profile <- thisModule.ClassProfile(java);
```

```

    for(a in java.attributes) {
      class.attributes <- thisModule.attribute2attribute(a);
    }
  }
}

rule ClassProfile(source : MM!Class) {
to profile : MM1!Profile_Class_profiles_Class ()

  do {
    profile.abstract <- source.abstract;
    profile.visibility <- source.visibility;
    profile;
  }
}
```

CHAPTER 6

Evaluation

The previous chapter discussed the solution in detail with detailed steps to achieve the model processors and the different ways to start a transformation.

This chapter is dedicated to the evaluation of the solution. The evaluation is concerned with the quality of the model processors and the approach, because only a correctly performing solution can be used in a productive way. There are some common metrics which may give an impression how the solution performs. This evaluation is structured in a set of research questions and evaluates another aspect of the solution. The first part of this chapter gives an overview of the questions. It will describe the questions in detail and the approach to answer it. The last part of this chapter answers the questions according to the described approach.

6.1 Research questions

As mentioned before is the evaluation is based on questions. It will answer three major questions concerning the solution besides the usage. Every question describes the subject in detail and contains the approach on how to solve it. The first question is the most crucial one. Are the model processors able to transform all described lightweight extensions? Because it is essential that all elements can be transformed without information loss. The next question will handle the problems in general and describe the existing problems as well as the solved ones. The last and third question will answer how the model processors perform in numbers with the advantages over other methods.

6.1.1 RQ1: Is it possible to transform all metamodels?

This question tries to answer the most crucial one. Is it possible to transform all parts of a model or metamodel? It is important that the model processors are able to recognize and process all necessary elements. Otherwise it will not be possible to use the model

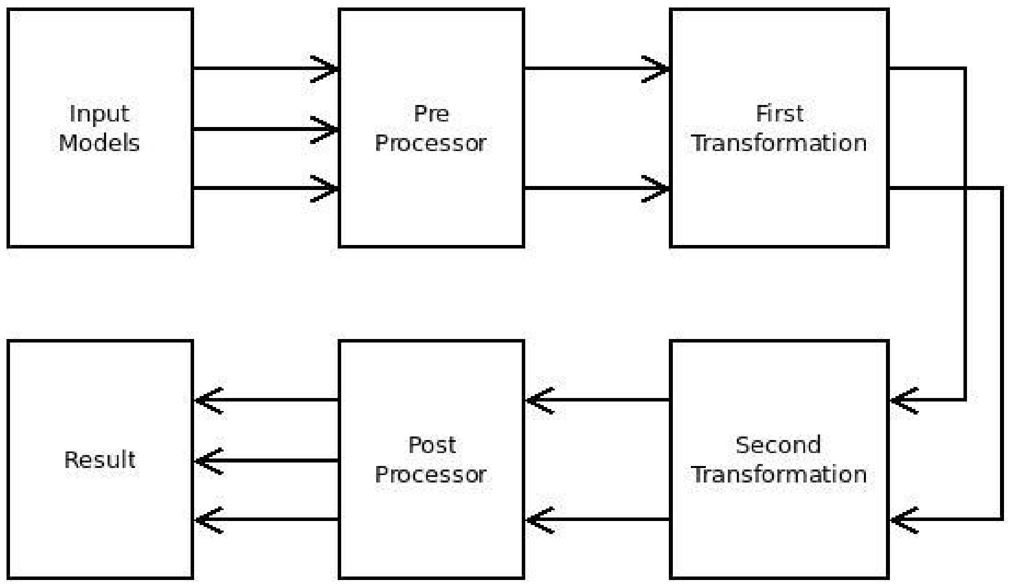


Figure 6.1: Round trip of the model processors

processors in a productive way. Also part of the question is if the processors are able to transform the profiles without loosing information.

A roundtrip over all model processors will try to answer this important question. A round trip over all model processors means that a source model with a profile is used with all model processors in sequence and vice versa. At first will the input model be used with the pre processor. Afterwards the result will be used as input for a transformation. Those transformed models are used as input for a second transformation. But with swapped source and target. The intention of this swap is to have the same output model as result. After the second transformation will a post processor separate the profile from the model. Figure 6.1 shows the round trip in detail. The main purpose of this round trip is to show if the result is the same as the input after both model processors. Only if both contain the same information is there no information loss. But the answer will only compare the information and not the structure of the XML file. Because the transformation language and the model processors may change the XML. But it is only important that all elements are present. Due the support of two different lightweight extension, namely UML profiles and EMF profiles, the answer is split up into two independent parts with one for every extension.

6.1.2 RQ2: Are there any problems or limitations in the solution?

Although the last question covers the possibility that the model processors can transform all elements it is possible that the model processors may have other problems. But this question should not only contain unsolved problems. Also problems which came up during programming and which are already solved. Those problems can range from problematic metamodel elements to limitation in the transformation language with the use of the model processors.

6.1.3 RQ3: Advantages of the model processors?

This question covers some quantitative aspects of the solution as well as a direct comparison between the build-in UML profile capabilities of ATL and the model processors. The quantitative aspects will be answered with a numeric comparison and will be shown in several tables. Since every profile and every model processor has its own behaviour and therefore another result will the comparison use three different metrics to compare all parts. Those qualifiers are

- the execution time of the transformations,
- the number of rules,
- and the Lines of Code (LOC).

The first metric is the execution time of the transformations. But to have a more representative result is the execution time the average of ten independent execution rounds because the execution may depend on some outside criteria. The most time changing factor is the used computer. A faster computer can process the models faster. Also the load of the computer during testing plays a role. Therefore the time metrics uses the average of ten different test rounds. The time will be measured in milliseconds, because in the most cases are the transformation to fast to measure it in seconds. The reason of this metric is to show how the model processor will affect the execution time. How much will a transformation will take longer. Because the measure will also count all model processor processing as well as the transformation time. Only ATL offers the option to compare the build-in UML profile support with the model processor and allows to answer if the model processor take longer or not compared to the build-in solution.

The next metric are the number of rules of a transformation. All tests are using the same models as input. Therefore it is possible to see which transformation needs how many rules to transform the example models. But the most interesting question is how many rules the model processors need compared to the build-in method. Unfortunately is this comparison only possible with ATL and not with RubyTL.

The last metric is the Lines of Code (LOC) metric. But it has to be noted that LOC is not a standardized measurement [Jon94]. It is not specified which lines are actually counted and which are not. This evaluation will only count necessary lines as LOC. This means only lines with actual code and no empty lines or lines with only braces. Also a single instruction over several lines are counted as a single LOC. This metric allows to measure how many lines the different transformation languages need with the use of the model processors. Additionally with ATL it is possible to make a direct comparison with the build-in method for UML profiles.

The UML transformation uses the models from Appendix A.2 and the EMF transformations the models from Appendix A.4.

6.2 Answering the research questions

This section will answer the questions from Section 6.1. Every question has its own approach which were also described in the previous section.

6.2.1 RQ1: Is it possible to transform all metamodels?

One of the most crucial quality characteristics is that the solution is able to make a correct transformation. A correct transformation consists of different aspects. At first it is necessary that all required profile extension elements can be transformed. This includes integrating the stereotypes into the metamodel. But the migrated metamodel should also be valid and has no information loss.

EMF profiles

This roundtrip uses the examples from Appendix A.4 and Appendix A.5 as source models. Listing 6.1 shows the profile application in XML representation. The EMF profile is split up into two different files but it is only necessary to check the profile application file. Because the profile definition is always the same and will not get changed. Listing 6.1 shows the original source file which can be found in the Appendix A.4. The profile application is listed in its XML representation because it is easier to find the differences. Listing 6.2 is the result after the roundtrip.

Listing 6.1: EMF profile information of the input model after the evaluation roundtrip

```
<emfprofileapplication:ProfileApplication xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" xmlns:classpro="http://class/
    profile/1.0" xmlns:emfprofileapplication="http://www.
    modelversioning.org/emfprofile/application/1.0">
  <stereotypeApplications xsi:type="Class" visibility="Private"
     >
    <appliedTo href="platform:/resource/test/class_model.ecore.
       xmi#/1"/>
  </stereotypeApplications>
  <stereotypeApplications xsi:type="Attribute" visibility="
     Private">
    <appliedTo href="platform:/resource/test/class_model.ecore.
       xmi#/2/@attributes.0"/>
  </stereotypeApplications>
  <importedProfiles nsURI="http://class/profile/1.0"/>
</emfprofileapplication:ProfileApplication>
```

Listing 6.2: EMF profile information of the input model after the evaluation roundtrip

```
<emfprofileapplication:ProfileApplication xmlns:xmi='http://www
    .omg.org/XMI' xmlns:xsi='http://www.w3.org/2001/XMLSchema-
    instance' xmi:version='2.0' xmlns:classpro='http://class/
    profile/1.0' xmlns:emfprofileapplication='http://www.
    modelversioning.org/emfprofile/application/1.0'>
  <stereotypeApplications xsi:type='Class' visibility='Private'
     >
    <appliedTo href='file:///test/temp_model.xmi#/1'/>
  </stereotypeApplications>
  <stereotypeApplications xsi:type='Attribute' visibility='
      Private'>
    <appliedTo href='file:///test/temp_model.xmi#/2/@attributes
        .0'/>
  </stereotypeApplications>
  <importedProfiles nsURI='http://class/profile/1.0'/>
</emfprofileapplication:ProfileApplication>
```

There are only minor differences between the result and the input after the roundtrip. Every *stereotypeApplication* stands for a single applied stereotype and its attributes represents the tagged values. As showed are the stereotypes between input and result nearly the same and the attributes are completely the same. Except the child *appliedTo*. This element is a reference to the extended element. Due the circumstance that the model and the profile application are in two separate files, it is necessary to have an absolute path to the element. Otherwise it would not be possible to address to the correct model. But the source model uses a special eclipse based reference and addresses the model inside an Eclipse workspace. But the model processor are running outside an Eclipse environment. This makes it hard to address resources inside an Eclipse workspace. But Eclipse also supports absolute paths of the file system as well. Therefore are both the same just with another path.

UML profiles

The UML roundtrip uses the examples from Appendix A.2 and Appendix A.3. The first Listing 6.3 shows the source profile application in XML representation but only the profile definition and not the entire document. Because the rest of the document is the model and is not relevant for this evaluation. The second Listing 6.4 shows the result after the complete roundtrip.

Listing 6.3: UML profile information of the input model before the evaluation roundtrip

```
<wsdl:WSDL xmi:id="_DNu8UNCBEd-q-MT17yoFPQ" Address="www.test.
    com" targetNamespace="tns" base_Model="_i2oHsKCUEd-
    jTfNz9tMbgQ" encoding="UTF-8"/>
```

```
<wsdl:InOut xmi:id="_kgZ24NCBEd-q-MT17yoFPQ" InOutString="In"
    base_Port="_4-CqwKCUEd-jTfNz9tMbgQ"/>
<wsdl:InOut xmi:id="_naS1INCBEd-q-MT17yoFPQ" InOutString="Out"
    base_Port="_WyqtEMsGEd-ovM9h017xRA"/>
```

Listing 6.4: UML profile information of the input model after the evaluation roundtrip

```
<wsdl:InOut xmi:id="_UYNt9NCAEd-q-MT17yoFPQ" InOutString="In"
    base_Port="_yV4a6Ll3EeGKLoIlOWlTQQ"/>
<wsdl:InOut xmi:id="_XJiaSNCAEd-q-MT17yoFPQ" InOutString="Out"
    base_Port="_yV4a6rl3EeGKLoIlOWlTQQ"/>
<wsdl:WSDL xmi:id="_YO-yfNCAEd-q-MT17yoFPQ" Address="www.test.
    com" targetNamespace="tns" base_Model="
    _yV4a4Ll3EeGKLoIlOWlTQQ" encoding="UTF-8"/>
```

Both profiles are nearly the same. At the first look is the order of the elements different. But this is not a problem because XML is not an ordered document. This also counts for the attribute order. The *InOut* stereotype is present both times and has the same amount of attributes. The two attributes *xmi:id* and *base_Port* are generated from ATL and therefore different. The first attribute *xmi:id* is only an unique identifier to address the element inside profile and to distinguish it from similar elements of the same type. The other attribute *base_Port* is the reference to the extended element. In this case to a *Port* element. The value of the attribute is the ID of the *Port* element. But both attributes are generated from the transformation language and do not contain any domain specific information from the profile. Therefore it is not problematic that they are different after the roundtrip. The only attribute which contain the information is the tagged value *InOutString* and is in both cases the same after the roundtrip. *In* for the one stereotype and *Out* for the other.

The other stereotype *WSDL* has also two attributes which do not contain any profile specific knowledge. They are named *xmi:id* and *base_Model*. The *xmi:id* attribute is the same as in the other stereotype and is the unique identifier for the element. The other attribute *base_Model* is also similar. But the stereotype extends the *Model* element and not the *Port* element. The *WSDL* stereotype has three tagged values and are named *Address*, *targetNamespace* and *encoding*. All three are the same after the roundtrip. This means that there is no information loss and both model processor, the pre processor and the post processor, are working for UML correctly.

6.2.2 RQ2: Are there any problems or limitations in the solution?

This question tries to answer if there are any problems or limitations with the solution. Unfortunately has the model processors a few limitations and issues. Some of them are theoretical but some other have an actual impact on how to describe transformation rules. The probably biggest issue is the violation of the original meaning of profiles. A lightweight extension is defined as an extension mechanism for models which only extends an existing metamodel. It does not remove or change an element or relationship of the original metamodel. But the model processors are changing this meaning. Because the profile gets merged into the metamodel and all stereotypes, tagged values, model extensions and all other model elements are now in a single metamodel. There is no separate metamodel and profile definition any more. The result resembles more a heavyweight extension. Only the post processor can transform it back to a lightweight extension. But during the transformation it is integrated into the metamodel. Before and after the transformation it is still a separate profile and metamodel. This problem is only theoretical and has no effect on the practical use.

Another difficulty is the mapping itself. The usage of the model processors requires some knowledge about the mapping mechanism. To be more precisely about the naming of the elements and the model extension. Every element from the profile gets renamed to the schema **Profile_<profile name>_<element name>**. This is necessary to prevent name conflicts. But as showed in the examples in Chapter 5.2 it is necessary to know the name of the elements to access the stereotypes. This naming is mostly used when applying stereotypes. Another naming issue is the reference to the stereotype, which is named **profile** and is necessary to access all stereotypes. Without this knowledge it is not possible to access the stereotypes.

There are other more problematic limitations. It is currently not possible to use more than one profile at once with the model processors. But it would be theoretically possible to use several of them at once. The only difficulty is to assure the unique names of the elements. Because different profiles can have different stereotypes with the same name. Also the name of the profiles can be the same. A possible approach to use several profiles is to execute one after another. The first profile gets merged with the metamodel. Then the second profile with the generated metamodel. This can be repeated several times. But the naming problem still exists. And all extension references to stereotypes have the same name. Therefore it is currently not possible to use more than one stereotype per element. There is no check if such a duplicate already exists. So it is possible that the final metamodel may have some errors and inconsistencies. Only a name check can prevent this.

An additional limitation is the number of stereotypes per element. The model processors supports only a single stereotype per element. The problem lies in the model extension reference. All references have the name **profile**. It is not possible to have more than one reference with the same name.

6.2.3 RQ3: Advantages of the model processors?

This section answers the question how the model processors perform. This will also cover their advantages and the disadvantages. The different evaluations contains metrics like the average execution time of ten test runs, the Lines of Code (LOC) and the number of rules of the transformations. The LOC and number of rules metrics are absolute values and do not depend on outside effects. This is different to the execution time. That depends

Round #	Pre processor transformation [ms]	Post processor transformation [ms]
1	485	662
2	460	669
3	542	672
4	458	722
5	490	691
6	468	676
7	490	690
8	489	674
9	461	671
10	494	710
Avg.	483.7	683.7

Table 6.1: Execution time evaluation of the EMF profile example for RubyTL with the pre and post processor method

on outside criteria as descried in the research question in Section 6.1.3. Unfortunately is there a problem with the transformation languages and the Java testmodel. It was not possible to apply the profile information from the **Method** element. Therefore all transformations will omit the information to ensure comparable results.

RubyTL

Unfortunately it was not possible with RubyTL to transform any UML profiles. Therefore is there no evaluation of UML profiles. The RubyTL evaluation will only contain EMF profiles and due the non existing build-in support can only the model processor be evaluated.

Because of the integration of the model processors in RubyTL are all times the sum of the specific model processor and the transformation. The ATL evaluation will contain the execution times of the model processors and the transformation separately.

The first metric of Table 6.1 is the time RubyTL needs for the complete transformation with the pre processor. After ten test rounds it has an average execution time of **483.7** milliseconds. This looks not very fast. But it contains the time of the pre processor and the transformation. The second metric is for the complete post processor transformation. It has a higher average execution time with **683.7**. But it is important to keep in mind that the complete post processor transformation needs two different model processors. The post processor to separate the profile from the model and the metamodel generation of the pre processor to make the target metamodel. Therefore it is normal to expect that the post processor takes longer than the pre processor.

The next evaluation are the metrics for the number of rules and the lines of code. Those are absolute values and have no average.

	Pre processor	Post processor
Number of Rules	4	4
LOC	30	22
LOC (profile)	8	12

Table 6.2: Rules and lines of code evaluation of the EMF profile example for RubyTL with the pre and post processor method

The first column of Table 6.2 represents the pre processor method and needs only **4** rules for the sample model. One for each element of the metamodel. The profile does not need any additional rules because the tagged values can be accessed inside the rule of the element. Due the internal class representation of the model in RubyTL it is possible to access all attached classes. The next metric are the total lines of code. They are representing all lines which are necessary for the complete transformation. It needs only **30** lines for the example. The last metric shows only the lines of code for the profiles and needs only **8** lines. This eight lines including six mappings and two stereotype existing checks. The remaining **22** lines are the normal mappings for the metamodel and the rule headers.

The last column represents the post processor method. It also needs only **4** different rules. But in contrary to the pre processor transformation needs the post processor transformation only two rules for the normal metamodel elements and additional two rules for the stereotypes. The two rules for the stereotypes are creating the stereotypes and containing the mappings for the tagged values. But more interesting are the lines of code. It only needs **22** lines of code in total and both stereotypes **12** lines. That is more than the pre processor needs. But this is only logical because the post processor need two additional rules. This includes the mappings and the rule header. That means only ten lines are needed for the rest of the transformation.

ATL

In contrary to RubyTL is it with ATL possible to transform both lightweight extension. Therefore contains the evaluation both lightweight extensions. It is also possible to make a direct comparison between the model processors method and the build-in method. But due the non existing model processor integration it is necessary that the model processor process the models over the command line interface before transformation. Due the manual usage of the model processors are the measure times split up into an execution time of the model processor and the execution time of the transformation.

EMF profiles The EMF profile evaluation will contain the evaluation of the pre and post processor and their transformations. Due the trick descried in Chapter 3.1 it is possible to compare the reading part of both solutions. The build-in and the model processor method. But unfortunately it was not possible to apply the trick to all elements. It was not possible to access the profile information of the **Attribute** element. The

Round #	Pre processor [ms]	Pre processor transformation [ms]	Build-in transformation [ms]
1	72	2	4
2	69	1	4
3	70	2	4
4	71	1	5
5	69	1	4
6	70	2	3
7	72	2	4
8	75	1	6
9	80	2	5
10	77	2	4
Avg.	72.5	1.6	4.3

Table 6.3: Execution time evaluation of the EMF profile example for ATL with the pre processor method

reason is that the trick requires a matched rule and the transformation uses a called rule. Due this circumstance all of the ATL transformations will omit the profile information for the element to ensure comparable result. The trick only works for reading profile informations, so it is not possible to compare the applying part. Table 6.3 and Table 6.4 are showing the metrics for the EMF profile.

The first metric of Table 6.3 represents the execution time of the pre processor in milliseconds. It merges the metamodel with the profile definition and the model with the profile application. The average execution time is only **72.5** milliseconds. The corresponding transformation has a fast average execution time of **1.6** milliseconds. The last metric is the build-in method of ATL. The procedure is described in Chapter 3.1. It has an average execution time of **4.3** milliseconds. That is **2.7** milliseconds longer. The longer time can be explained with the additional work with the profile. The transformation needs to calculate the cross product between the model element and stereotype to find the stereotype which belongs to the element and can be very time expensive. The time difference is not much but during development of the transformation a developer usually executes the transform several times. The example has only a few models but more elements will increase the time the calculation needs for the cross product. It is also necessary to take into account that the model processor method requires the pre processor. That makes the pre processor method **69.8** milliseconds slower. But the pre processor does not need not be executed every time the transformation starts. Only after a change in metamodel or the profile definition.

The post processor needs nearly the same time as the pre processor and is shown in Table 6.4. It has an average execution time of **62.8**. Also the corresponding transformation needs the same average time as the pre processor transformation. Only **1.6** milliseconds. But the post processor method can not work alone. It needs the metamodel from the

Round #	Post processor [ms]	Pre processor metamodel [ms]	Post processor transformation [ms]
1	81	69	2
2	60	68	1
3	60	70	2
4	68	67	2
5	60	67	2
6	67	67	1
7	30	69	2
8	67	71	1
9	61	66	1
10	74	68	2
Avg.	62.8	68.2	1.6

Table 6.4: Execution time evaluation of the EMF profile example for ATL with the post processor method

	Build-in reading part	Model pre processor	Post processor
Number of Rules	5	5	4
LOC	32	31	23
LOC (profile)	3	3	12

Table 6.5: Rules and lines of code evaluation of the EMF profile example for ATL of all three methods

pre processor which is only slightly faster as the complete pre processor. It needs **68.2** milliseconds. This also means that the model processing part of the pre processor has only an average processing time of **4.3** milliseconds. Those fast times can be explained with the simple metamodel and model. The metamodel is not very complex and the model has not many instances. It is to expect that the execution time will grow with more complex models.

The next evaluation are the number of rules and the lines of code. Table 6.5 contain the metrics for both model processors.

At the first view are the metrics values very similar to the RubyTL evaluation. See Table 6.2. The first column represents the build-in method of reading profiles. It needs **5** rules with **32** lines of code. The profile specific parts only needs **3** lines. One for each mapping and one in the matched rule. The pre processor method is represented in the second column. It has also only **5** different rules, which is **1** more than RubyTL. This is because in ATL the getter and setter generation is split up into 2 rules and in RubyTL it is just a single one. This is also responsible for an higher lines of code count. It take **31** lines. But the lines cannot be compared directly. As stated before omits the ATL transformation the profile information and RubyTL not. The last column represents the

Round #	Pre processor [ms]	Pre processor transformation [ms]	Build-in transformation [ms]
1	171691	4	10
2	247916	5	11
3	260422	4	11
4	266465	5	10
5	274139	4	11
6	277359	4	9
7	284571	5	8
8	284592	5	12
9	290654	4	10
10	286893	4	11
Avg.	264470.2	4.4	10.3

Table 6.6: Execution time evaluation of the UML profile example for ATL of the pre processor method

other model processor, the post processor. It also needs only **4** rules, like RubyTL or the pre processor. Also the lines of code are similar. It needs only **23** total lines and **12** lines for profiles specific part. The cause for the similarities are the similarities between both transformation languages and the approach of the model processor. Both transformations need two rules for the metamodel and two rules for the stereotypes.

UML profiles Due to the build-in UML profile support this evaluation contains a comparison between the build-in method and the model processors. The first evaluation is for the pre processor and contains the execution times for the pre processor, the pre processor transformation and the build-in transformation method.

The first metric represents the execution time of the UML profile pre processor and has an average time of **264470.2** milliseconds or 4 minutes, 24 seconds and 470.2 milliseconds. Compared to the execution time of the EMF profile preprocessor is this very high. The answer of this high time is the UML metamodel. The UML metamodel is 1,1 megabyte in size and has 293 elements. This also includes additional, non useable, elements, like comments or annotations. Compared to the simple example metamodel for the EMF profile is the UML metamodel very complex. Every additional element in the metamodel requires additional time to process for the model processors. Because the model processors read the metamodel internally as DOM tree and this takes time. More elements will take more time to read. Also the model processor uses XPath to search for the required elements. A bigger DOM tree means that XPath will need more time for the search. All this will result in a higher execution time. This matters for all metamodels and not only for the UML metamodel. But this does not mean that the pre processor need to be executed on every transformation. It need only be executed after a change in the metamodel or the profile. The pre processor will always generate the same merged

Round #	Pre processor metamodel [ms]	Post processor [ms]	Post processor transformation [ms]	Build-in transformation [ms]
1	171673	1854	4	309
2	247899	2353	4	217
3	260405	2389	3	220
4	266447	2476	3	293
5	274120	2260	3	215
6	277342	2533	4	212
7	284555	2658	3	406
8	284573	2557	3	219
9	290638	2371	3	215
10	286885	2764	4	213
Avg.	264454	2421.5	3.4	251.9

Table 6.7: Execution time evaluation of the UML profile example for ATL of the post processor method

metamodel with the same input models.

The other two metrics are the execution times of both transformations. The first is the time for pre processor transformation and has an average time of **4.4** milliseconds. The other metric is the time for the build-in method of ATL. It has an average execution time of **10.3** milliseconds. The model processor method is **5.9** milliseconds faster. That may not be much. But in all ten test rounds was the build-in method slower. The reason is that the build-in method need to handle two different metamodels. The normal metamodel and profile definition. Additionally need ATL to do a full search in the profile every time a tagged value will be accessed and this need some time. ATL helper methods may help to speed up the access but it need some time. On the other side need the model processor method handle only one single metamodel. Every tagged value can be accessed directly without a search for the stereotype.

The next evaluation is done for the post processor. It has similar metrics as the pre processor evaluation and contains the execution times of the post processor, the metamodel generation of the pre processor, the post processor transformation and the build-in transformation. The input models are the same as for the pre processor. Only the source and target are swapped. This means a normal WSDL metamodel as input and a UML with an UML profile as target.

The first of the four metrics represents the time the pre processor need to generate the metamodel for the target. It will only generate the metamodel and not the model. It takes also a long time to process the XML. As described before is the reason for the long execution time the complex metamodel. The average execution is **264454** milliseconds. This also means that the model processing for the UML method takes only **16** milliseconds. The next metric is the post processor after the transformation. It has

	Build-in method	Model processor method
Number of Rules	4	4
LOC	36	34
LOC (profile)	10	8

Table 6.8: Rules and lines of code evaluation of the UML profile example for ATL of the pre processor method

an average execution time of **2421.5** milliseconds. Compared to the pre processor is this fast. The reason is that the post processor does not need metamodel to separate the profile from the model. Like the pre processor it is not mandatory to execute the post processor after each transformation. Only if a separate profile is required.

The last two metrics are the execution times of the transformations. The first is the post processor transformation and has only an average execution of **3.4** milliseconds. The other metric is the build-in method of ATL. But compared with the post processor transformation need it a very long time with **251.9** milliseconds. This time difference lies also, like as reading profiles, in the circumstance that ATL requires two different input metamodels. It takes time to generate the profile and insert the stereotypes.

The next evaluations are the metrics for the number of rules and lines of code. Due to the possibility of ATL to transform UML profiles this evaluation also contains the metrics for the build-in method. The first evaluation are for reading the profiles tagged values and are shown in Table 6.8. The evaluation for applying profiles are represented in another table.

The first column contains the metrics for the build-in method of ATL. It needs **4** different rules. This is because the example model uses only four different elements which should be mapped. The profile specific parts does not require any additional rules. The other two metrics are the lines of code. The build-in method requires **36** total lines of code. The profile specific part only **10**. The other column represents the model processor method metrics. It also needs **4** rules for the transformation, like the build-in method. They have the same number because both ways do not require any additional rules for processing profile information. All tagged values can be read inside the corresponding rule of the element of the metamodel. The model processor method requires only **34** total lines of code. But more interesting are the lines for the stereotypes. The model processor only needs **8**. That are two less than the build-in method. Because the model processor method needs only a single existing check for the stereotype. The build-in method requires an existing check for every single tagged value. Three mappings need only four lines in the model processor method, but six lines in the build-in method. Every additional tagged value will increase the difference between both methods.

The evaluation Table 6.9 contain the metrics for applying a profile. The first column contains the metrics for the build-in method of ATL. It requires **5** rules for the complete transformation and needs **53** total lines of code. The profile specific part only needs

	Build-in method	Model processor method
Number of Rules	5	7
LOC	53	48
LOC(profile)	9	12

Table 6.9: Rules and lines of code evaluation of the UML profile example for ATL of the post processor method

10 lines of code. The other column represents the post processor method. It needs **7** rules. That are two more than the build-in method. This is because the model processor require additional rules to create the stereotype. One for each stereotype and the example has two stereotypes. Therefore need the model processor two more rules. The tagged values in the build-in method can be set in the rule of the element and does not require additional rules. The total lines of code are **48**. That are five less than the build-in method. The difference are because of the different approaches to deal with profile. More interesting are the lines of code for the profile. The model processor need more lines than the build-in method. It need **12** lines of code and can explained due to the two additional rules. Also requires the build-in method less lines compared to its reading part.

6.3 Summary

Research Question 1 showed that both model processors are able to make correct transformations. No information was lost during the process. This is the key question. Because without a complete and correct transformation this approach is useless. But unfortunately, Research Question 2 showed that there are some limitations. Some are more serious than others. Chapter 8 shows how to eliminate some of those limitations. A final result for Research Question 3 was a little bit more complicated to answer. It showed the advantages and disadvantages in numbers. All in all is the model processor method easier to use, easier to read and requires less code.

CHAPTER 7

Related Work

This chapter covers related work and includes transformation languages as well as model extensions and in particular tools which enables the use of model extensions in transformation languages.

7.1 Transformation languages

There exist a multitude of transformation languages and most of them have different approaches to perform a model-to-model transformation. Not only the approach may be different but also the supported metamodels. Some may only work with ECore and some other only with MOF or any other type of metamodel. This goes also for the models. Some can use XMI serialized models while some other use proprietary formats. For example is RubyTL also capable to use special Ruby classes. Two other transformation languages are described in the next subsections. The first is Epsilon [KPRGD12] and the other is QVT (Query/View/Transformation) [OMGc].

7.1.1 Epsilon

Is a complete tool suite [KPRGD12] and supports code generation, model-to-model transformations, model validation, comparison, migration and refactoring for EMF and other metamodels. It is part of the Eclipse Modeling Framework. Epsilon is designed to provide a wide range of different languages to support all of these modeling parts. The syntax of all languages is a little bit different but similar to JavaScript. This allows to have languages which are specialized in their specific field but easy to learn and similar to each other.

Epsilon Object Language (EOL)

EOL [KPP06c] is designed to create, query and modify any EMF model. It is as mixture between JavaScript and OCL. JavaScript adds the functional and imperative parts, like variables, loops and if-conditions to EOL. OCL adds the modeling commands to the language. The combination allows to have an easy to use language with explicit model support. It is also possible to access any Java objects, which makes the language even more powerful. Listing 7.1 shows an example how EOL looks like. The command iterates over all **Class** objects and prints their names.

Listing 7.1: Example of the Epsilon Object Language (EOL)

```
for (c in Class.all) {
  c.name.println();
}
```

Epsilon Transformation Language (ETL)

ETL [KPP08] is the model-to-model transformation language of Epsilon. It is a rule based language which is based on EOL. The syntax is similar to ATL or RubyTL. But the EOL extension allows to use also OCL as well as functional and imperative code from JavaScript. The rule consists of a declarative header and an imperative body, similar to RubyTL. It has different rule types, rule inheritance, guard conditions and automated rule execution. Listing 7.2 shows an example of a simple rule. Like ATL, a rule starts with the keyword **rule** followed by the name. The first statement in the rule, **transform**, is like the **from** part. It defines the source element. The target is defined in the **to** statement. It has as parameter the target element. It is also possible to have multiple target elements. Line *4* shows a guard condition. The result has to be a boolean expression. It executes the rule only if the example class is not abstract. Line *6* is a simple binding from source to target.

Listing 7.2: Example of the Epsilon Transformation Language (ETL)

```
1 rule Class2Java
2   transform c : IN!Class
3   to j : OUT!Class {
4     guard : not c.abstract
5     j.name = c.name;
6   }
```

Epsilon Validation Language (EVL)

Is able to validate a model against given constraints and uses OCL constraints because EVL is also built on EOL. A feature in EVL [KPP06a] is that it can distinguish between errors and warnings with customizable errors messages. Another features are dependencies between constraints and a repair mechanism to repair inconsistencies. Unlike to OCL is it possible to model inter-model constraints. EVL has also an integration into the EMF validation framework. All these features allowing a full fledged validation of many different models. Listing 7.3 shows a simple validation.

The first line defines that the validation is for the element **Class**. The second line, **critique**, defines the start of a validation followed by the name. Every element can have several validations. The **guard** statement in line *4* will run before the actual validation procedure and makes sure that the following check can be done. The actual validation happens in line *6*. This case checks if the class is not abstract. If the class is abstract then the message in the command **message** will be shown. See line *7* for an example. Line *9* defines the start of the repair mechanism. **title** defines a short message. The actual repair will be done in the **do-block**. In this case it sets abstract to false.

Listing 7.3: Example of the Epsilon Validation Language (EVL)

```
1  context Class {
2    critique noAbstract {
3      guard : self.abstract.isDefined()
4
5      check : self.abstract = false
6
7      message : "Class " + self.name + " should not be abstract"
8
9      fix {
10       title : "Makes class " + self.name + "abstract"
11
12       do {
13         self.abstract = false
14       }
15     }
16   }
```

Epsilon Generation Language (EGL)

EGL [RKPP08] is a model-to-text and template based language similar to XText [EV06]. It allows to generate any textual fragment, like source code, documentation or other text. Templates can be called with parameters and are able to call other templates as well. Listing 7.4 shows an example which generates a Java class with its name. The EGL syntax is very different to rest of the languages of the Epsilon suite and is more similar to JSP (Java Server Pages) [KFB01]. Every EGL command is inside [% and %] tags. All text outside these tags are part of the output text. EGL is also based on EOL and therefore able to use OCL commands.

Listing 7.4: Example of the Epsilon Generation Language (EGL)

```
public class [%= self.name %] {
}
```

Epsilon Wizard Language (EWL)

Is similar to ETL, but unlike to ETL is EWL [KPPR07] designed to do automated and recurring modeling tasks. It can be executed within any EMF and GMF editor. In case of a mistake can every action be reverted to the old version. Like any other Epsilon language is EWL also based on EOL. Listing 7.5 is similar to the EVL example in Listing 7.3. The only difference is that this example has no check condition. The example sets every abstract attribute of the Class model to false.

Listing 7.5: Example of the Epsilon Wizard Language (EWL)

```
wizard noAbstract {
  guard : self.abstract.isDefined()

  title : "Makes class " + self.name + "abstract"

  to {
    self.abstract = false;
  }
}
```

Epsilon Comparison Language (ECL)

It allows to compare homogeneous or heterogeneous models [Kol09]. The syntax is similar to ETL and is also rule based. The result can be exported to the Epsilon Merging Language (EML). Listing 7.6 shows an example which compares the name of two **Class** objects. The **match** statement in line *2* defines the to matching element. Line *3* with the **with** statement the other element. The comparison will be done in the **compare-block** and has to return a boolean expression if both models are the same.

Listing 7.6: Example of the Epsilon Comparison Language (ECL)

```
1 rule MatchName
2   match c : IN!Class
3   with j : OUT!Class {
4     compare {
5       return c.name = j.name;
6     }
7   }
```

Epsilon Merging Language (EML)

EML [KPP06b] allows to merge two homogeneous or heterogeneous models with each other. The syntax is similar to ETL because EML needs features from a transformation language. But the ETL syntax is not enough. EML is extended with merging specific syntax. Before merging it is necessary to compare both models with each other. This can be done with ECL or plain JavaScript. Listing 7.7 shows a short example how to merge two **Class** objects. The **transform** statement defines the source element while the **to** keyword defines the target element. The actual merging is done in the **to-block**.

Listing 7.7: Example of the Epsilon Merging Language (EML)

```
rule MergeClass
  transform c : IN!Class
  to k : OUT!Class {
    k.name = c.name;
  }
```

Epsilon Flock

Flock [RKPP10] allows to update models in response to metamodel changes. A copy algorithm automatically copies any non changed elements. The rest will be done in a rule based transformation style.

7.1.2 Query/View/Transformation (QVT)

In contrary to Epsilon is QVT only a transformation language and is developed by the Object Management Group (OMG). The language syntax and the way how QVT transforms a model differs from the other transformation languages, like ATL, RubyTL or ETL. QVT is divided into three different transformation languages. Every language is able to transform MOF 2.0 compatible metamodels, but have a different syntax and scope of use. Those languages are the **Relation** language, the **Core** language and the **Operational** language. All three will be described in detail in the next section. QVT is split into a two level architecture. A declarative and imperative level. Every, of those three, languages supports one level. But they are also in relationship to each other. Figure 7.1 shows the relationship. As showed are the three languages not the only involved parts. There is also a black box implementation and transformation and allows to plug in any other external code. It can be in any programming language but it has to have a MOF binding. For example has Java JNI (Java Native Interface) [Lia99] bindings. This may be necessary if the desired operation can not be expressed with one of the languages or OCL. Due the external plugin can the black box be considered as an imperative part, according to the two level architecture. The binding also allows direct access to the model properties. But the use of the black box can be a risk too. The external program may break any visibilities or any other model constraints.

Relation language Is the most used language of the three QVT transformation language. It has a declarative syntax style and supports complex objects. The language also generates the trace models automatically. Trace models are used to record the transformation and can be used with other matched patterns.

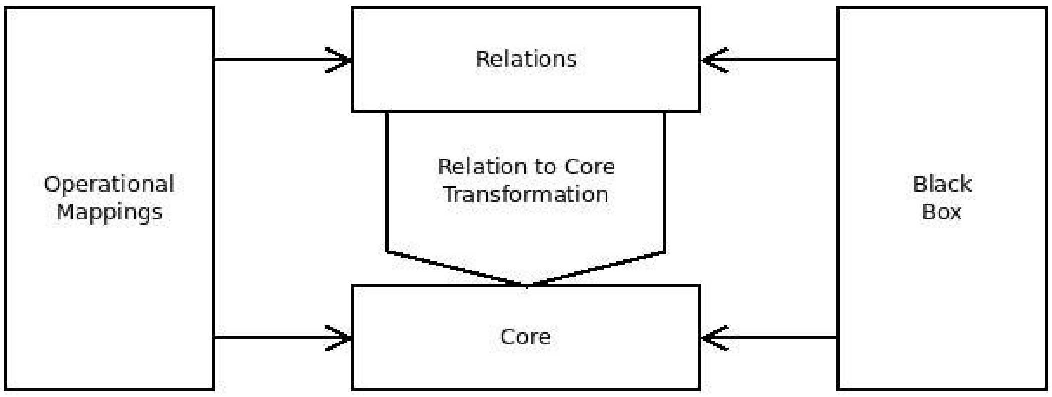


Figure 7.1: Relationship between the three QVT languages and the implementation

Core language Is the second declarative styled language. It has a simple and small syntax, but as powerful as Relation language. But the syntax is so simple that the language supports only a pattern matching over a flat set of variables. Those variables are evaluated against given models. In addition and in contrary to the Relation language, is it necessary to define the trace model directly and is not derived from the transformation description. The Core can be used directly or as a reference, mapped from a Relation.

Operational language Is the only imperative language and is nearly the same as the Relation language. It also provides an OCL extension which allows a more procedural programming style. Every operational mapping can have one or more Relation operations or one or more other Operational operations. But if it uses other Operational operation is it necessary to include a Relation for the trace model. This allows to write a complete transformation in an imperative code style. Listing 7.8 shows a small example of a single mapping and uses the test model from Appendix A.4. The mapping maps the class from the class diagram to a Java class from the Java diagram. The only attribute mappings is the name. It is not necessary to specify the target type directly, like in ATL. The name of the attribute is enough. The source object can be accessed with the keyword **self**.

Listing 7.8: Example of the QVT Operational language

```
mapping Class::Class::class2class() : Java::Class {
  name := self.name;
}
```

7.2 Model extension approaches

EMF and UML profiles are not the only model extensions. There are also other extension approaches. One of them is **Model Decorations** from Kolovos et al. [KPR+10]. But their approach is a little bit different than to model the additional information. They annotate the additional information in *diagram notes*. Which means that all information

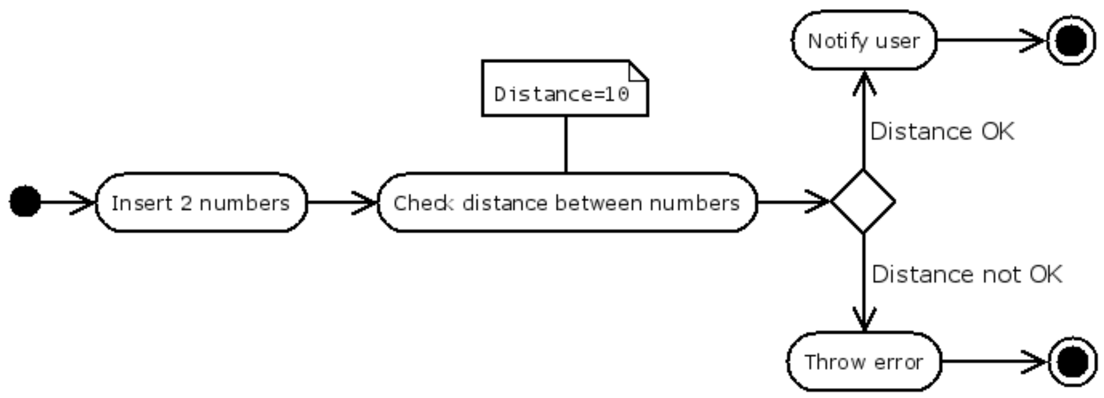


Figure 7.2: Example activity diagram with a model decoration

are in textual form. Those textual fragments getting translated to an actual model and also vice versa. A model can be transformed into a diagram notes. This will be done with self written *extractor* and *injector* transformations. The extractor is responsible for getting the information out of the notes. The injector for creating the notes. Other model extensions, like EMF profiles, do not require such transformations and in contrary to other model extensions are the Model Decorations only useable with GMF based diagrams. Because the notes are GMF notes and only transformation languages which are capable of using GMF diagrams are able to use this extension method. But this is also an advantage. It is not necessary to write special extension for different transformation languages. They only have to support GMF notes. Another major disadvantage is that the notes are not structured. They save the information in plain text. Any misspelling or typo is hardly to track and to solve. This may result in an unexpected behaviour. It is also hard for programming environments to work with that. A syntax highlighting or syntax completion do not work. Also the programmer has to be familiar with the syntax. Without knowing the syntax is it not possible to use the extension.

Figure 7.2 shows a short example how such a diagram may look like. The example shows an activity diagram which contains the steps to check the distance between two numbers. It starts on the left side with the activity *Insert 2 numbers* where the user adds the 2 numbers. The next step is to check if the distance between the numbers matches. If they do the user will be notified. Otherwise will a error be thrown. The activity *Check distance between numbers* has a model decoration. It has only one single attribute in it. It is called *Distance* and says that there has to be a distance of 10 between the two numbers. In this diagram is the attribute represented as a keys-value pair. This is not a must it can be in any other format. Only the model extractor and injection have to parse it.

Epsilon (see Chapter 7.1.1) is one the languages which are able to use the Model Decorations. It is possible with the Epsilon Object Language (see Chapter 7.1.1) to access the diagram notes. Kolovos et al. showed in the article [KPR$^+$10] how to access the notes. It is very trivial because EOL provides all necessary commands. The only pitfall may be the parsing of the attributes. Because, as already stated, are the attributes in textual form and there is no concrete syntax. Every user can define its own syntax with the drawback is that there is no common extractor and injector. The article shows how to handle attributes in the key-value pair format with one attribute per line.

Another model extension approach are EMF Facets[1] and allows to extend an ECore based metamodel. But this extension goes further than other model extensions. Normally extensions only allow to add attributes. But EMF Facets can add attributes, operations, types and references. It is also possible to extend existing features. This is done with a *query abstraction framework*. This is independent from existing implementation, like ATL, OCL or Java. Which means that all languages should be able to use it. In contrary to other model extension approaches work EMF Facet during runtime and allows to dynamically add features depending on some criteria.

7.3 Model extension support in transformations

The most similar work is from Andrea Randak [Ran]. Her solution was already mentioned in Chapter 3.2.2. The solution is called *ATL4Pros*. She used Higher Order Transformations to transform the abstract syntax to the concrete ATL syntax and is completely integrated in ATL. But this has pros and cons. It is necessary to enhance ATL with new approaches. Everyone who wants to use the extension needs a special version of ATL which contains the solution. But it is possible to access the profile without any other steps, like executing another script. The complete procedure will be done automatically by ATL4Pros. Due the lack of existing commands for accessing profiles is it necessary to learn the new introduced syntax. The new syntax is based on Manuel Wimmer and Martina Seidls work on the approaches to extend an existing transformation language with profile support (see Chapter 3.2).

[1]`http://www.eclipse.org/modeling/emft/facet`

CHAPTER 8

Conclusion and Future Work

8.1 Conclusion

This thesis introduced a way to use UML and EMF profiles with RubyTL and other transformation languages. This is necessary because of the increased use of model driven development and the associated increased use of lightweight extensions. But only the minority of transformation languages are capable to process profiles and without an extension it is complicated or impossible to use them. RubyTL is one of such languages. Also the most used transformation language ATL has only a very rudimentary support for both extensions. The introduced model processors have solved this problem in a non-iterative way.

A model processor is a system which changes a model to get a desired characteristic. In this case model processors are used to merge the profile with the metamodel. This allows access to the stereotypes like to any other element. The model processor transforms the stereotype to a normal element and adds a reference from the element to the stereotype with the name **profile**. This allows the access to the stereotype without any additional syntax and the tagged values can be used like any other attribute from an element.

The theory behind this is very simple. There are two different types of model processors, the pre and the post processor. The pre processor integrates the profile into the metamodel. The stereotypes are converted into normal classes in the metamodel. Therefore they are now accessible like any other element of the metamodel. The tagged values are getting converted into attributes of the new class. The last missing part is the model extension. It will be represented as a simple reference with the multiplicity of *0...1* and the name *profile*. Now the profile is an integral part of the metamodel and usable like any other metamodel. The profile application is merged with the model in the same way. The practical part looks a little bit different because of the many different formats for the metamodel. Therefore the focus of this thesis lies only on the most used notation, the

XML representation of ECore. This makes it easy to process UML and EMF profiles because both are representing their stereotype as EClasses. The same element can be used in the metamodel with their tagged values as attributes. This makes it very simple to merge the stereotypes into the metamodel. The other model processor is responsible for separating the profile from the metamodel. This processor is used if stereotypes getting applied to an element. It will generate a plain profile as result.

The last implementation is the RubyTL integration. It takes all required models as input and checks what processors are needed and executes them. The pre processor before the transformation and the post processor after the transformation. It will also executes a configured RubyTL transformation with no additional steps. This was achieved with the RubyTL extension mechanism. ATL and all other transformation languages requires manual control over the command line interface. It allows to execute every model processor with the appropriate metamodel, model and profile as argument. Those generated models can be used with any other transformation language. But the three steps, execute the preprocessor, execute the transformation and execute the post processor, need to be done manually.

At the moment only the minority of transformation languages are capable to process profile enriched models. But they are getting used more and more. Therefore it is necessary that the transformation languages have support for lightweight extensions. But unfortunately the shown UML and EMF profiles are not the only profiles. It will be difficult for a language to support them all. The model processor method of this thesis is a good starting point to enable the use of profiles for many different transformation languages at once. It has the advantage that the profile support need only be added for one tool and not for all languages. This may be a solution to solve the problem with the rising use of profiles, despite its limitations described in Chapter 6.2.2.

8.2 Future work

There are several possibilities for a future work. The limitations from Section 6.2.2 are a good starting point. Not all limitations, but some, are solvable by changing the model processors. The UML profile problem lies not at the model processors. It is located in RubyTL or the used ECore parser. This may be solved with another metamodel or with another model processor. The model processors can change the metamodel so that all problematic parts are removed and replaced with a working solution.

But other limitations are easier to solve with rewriting the model processors. Currently it is only possible to use only one profile at the same time. The main problem are name conflicts. But with name checks and element renaming it is possible to solve that problem too.

With a rewrite it is also possible to use more than one stereotype per element. The problem is the static reference from the stereotype to the element. They all have the same name **profile** and an element can only have one reference to an element with the same name. The only thing to do is to rename the name from **profile** to some other name. But this will make the use of the model processors more difficult. Because it is necessary to know the additional naming convention. The reference name need to be extended with an unique identifier like the target stereotype name. It may be an option to rename the reference name only if an element has more than one stereotype. But this has the drawback that the model extension has now two different naming conventions, the normal one with the pure name and the other with the unique name extension.

The model processors can only use UML and EMF profiles. But there exists more lightweight extensions, for example EMF Facets. The architecture of the model processors allows an easy extension with other profiles. The only requirement is that they have a textual notation like XML. Without it would be very difficult to extend the metamodel with the profile.

Another advancement could be to support more meta-metalanguages besides ECore, for example MOF. Many different metamodels are modeled in MOF and has a textual notation in XML. This makes it easy to extend the model processors to merge MOF based profiles.

APPENDIX A

Appendix

A.1 Testmodels

Because of the test-driven development approach it is necessary to create testmodels before starting to develop both model processors. Therefore it is possible test the solution and find problems at every stage of the development. To test every circumstance it is necessary to create several testmodels.

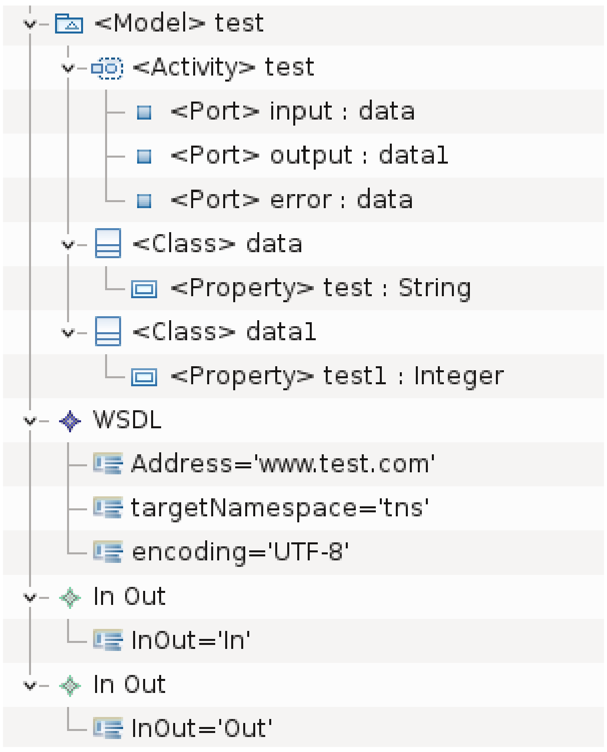
A.2 UML Model with an UML profile for describing WSDL documents

This testmodel tries to describe a WSDL document with normal UML elements. But not all aspects are able to be modeled. They will be described in a separate UML profile. The following two sections will describe the profile structure and a testmodel. The metamodel will be the ECore UML metamodel provided from the Eclipse Modeling Framework.

UML Model

It needs four UML elements to define a rudimentary WSDL. Figure A.1 shows a treeview of the model with the profile values.

- **UML Model**: Represents the WSDL in its entire. All other used elements are children or children of children. The name of the element is the name of WSDL document.
- **UML Activity**: Are the operations and direct children of an UML Model. The name of the Activity is the name of the operation. Since a WSDL can have multiple operation can an UML Model also have multiple activities.



```
<Model> test
    <Activity> test
        <Port> input : data
        <Port> output : data1
        <Port> error : data
    <Class> data
        <Property> test : String
    <Class> data1
        <Property> test1 : Integer
WSDL
    Address='www.test.com'
    targetNamespace='tns'
    encoding='UTF-8'
In Out
    InOut='In'
In Out
    InOut='Out'
```

Figure A.1: Treeview of the WSDL model

- **UML Port**: Are children of an Activity and representing two different WSDL elements. The first type are the errors an operation may raise. The second are the different input or output methods. The data type of the Port are the parameter and needs an UML Class as type.
- **UML Class**: Are used to describe the parameter of a method. Will be the data type of an UML Port. Every property of the class is a single parameter an operation may need or return.

UML profile

Is a very simple profile. The model was not able to cover all necessary information of a WSDL document and the profile will be used to describe them. It contains the following stereotypes and tagged values.

- Stereotype **WSDL**: Extends the UML Model element. It specifies the general properties of a WSDL document. Those properties are:
  - Tagged Value **Address**: Defines the internet address on which the WSDL will be discoverable.

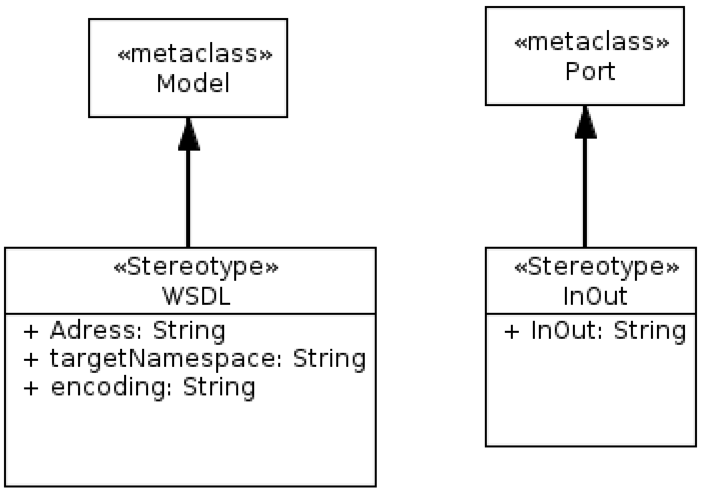


Figure A.2: Profile for the WSDL special parts

  - Tagged Value **targetNamespace**: The namespace that the WSDL will use.
  - Tagged Value **encoding**: Every WSDL needs a separate specification of the used character encoding.

- Stereotype **InOut**: Represents the input and output methods that a WSDL supports. The stereotype extends the UML Port element. It has only one tagged value with the name *InOut*. This tagged value should only have *In* or *Out* as value. Because they describing the type of the method if it should be an input or output.

A.3 WSDL Metamodel

Is the companion metamodel for UML diagram with WSDL profile extension (see Section A.2). This metamodel describes a very simplified WSDL diagram. The metamodel contains the following elements and is also illustrated in Figure A.3:

- Class *WSDL*: Representing the WSDL Document and is the starting point and contain all operations. It has three different attributes for further configuration. The *name* attribute defines the name of the WSDL. The second attribute is *encoding* and defines the text encoding of the WSDL document. The last attribute is *namespace*. It defines the used namespace of the WSDL document. The class has also a reference to *Operation* to define all supported operations. The number of operations is not limited, but it does not need to have one.

- Class *Operation*: Describes a single operation of the WSDL. It has an attribute, called *name* to define the name of the operation. It also has three references with

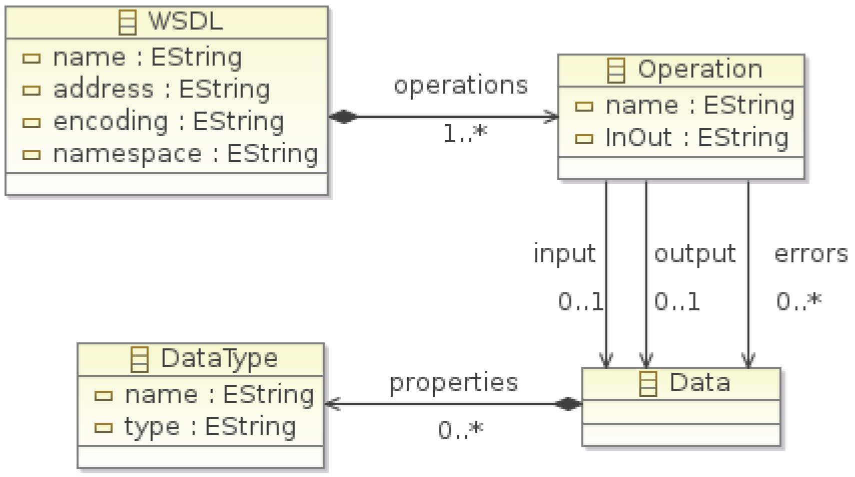


Figure A.3: Metamodel of a very simplified WSDL document

to the element *Data*. The first reference defines the input parameter and is called *input*. Another reference is called *output* and describes the return type of the operation. Both references share the same behaviour. They are not mandatory but can only occurs once. The last reference defines the errors an operation may raise. Every error has a data type to describe the error in detail. Another attribute is called *InOut* and defines the type of operation. It should contain **Input** if the operation should act like a normal function. Or *Output* if the operation can call the caller from alone.

- Class *Data*: The element represents the data type for the operation. It holds all parameters an operation uses. The parameters are described with a reference to *DataType* with a multiplicity of *0..**.

- Class *DataType*: Represents the a single parameter. It has two attributes. The *name* and the *type* of the parameter.

A.4 Model with EMF profile

A model with an EMF profile has four files. A metamodel, a model, a profile definition and a profile application. This testmodel describes a very simplified class diagram with an extension for Java classes.

Class metamodel

The metamodel describes a basic class diagram. Figure A.4 represents the complete diagram. The diagram consists of the following elements:

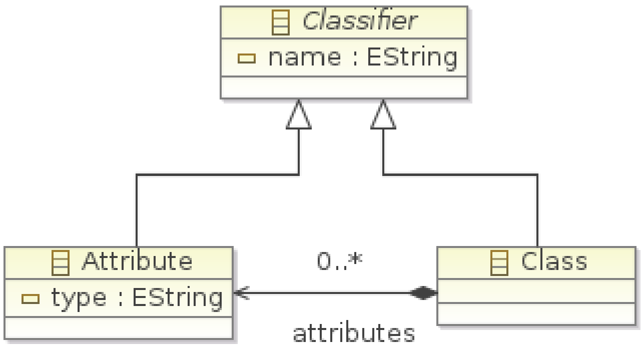


Figure A.4: Testmetamodel to represent a generic class diagram

- Class *Class*: Represents the class itself. It extends from *Classifier* and owns 0 or many *Attributes*. The name of the class gets derived from the property *name* of the super class. Every class can own several classes of the type *Attribute* to represent the attributes of the class.
- Class *Attribute*: Represents the attributes of the class. It also gets extended from *Classifier* and the name of the attribute is derived form the property *name*. The type of the attribute gets defined with the property *type*. The property has the type String and takes the name of the type as value.
- Class *Classifier*: Is abstract and is the super type of *Class* and *Attribute*. It has the property *name* with type *EString* to represent the names of the classes and attributes.

Java profile

It describes the Java extension for the generic class diagram. Figure A.5 shows the profile. The diagram consists of the following elements:

- Abstract stereotype *Classifier*: Is the basis for all other stereotypes and therefore abstract. It has two attributes. The first attribute, *abstract*, marks the class or the attribute as abstract and has the type Boolean. The other attribute has the name *visibility* and defines the visibility. The different visibilities are defined in the enumeration *Visibility*.
- Stereotype *Class*: Extends the Class element. It has no direct attributes, only derived attributes from its super class *Classifier*.
- Stereotype *Attribute*: It extends the Attribute element and has two direct attributes and two derived attributes from the super class *Classifier*. Both attributes have the type Boolean. The attribute *Final* marks the attribute as final and the attribute *static* marks it as static.

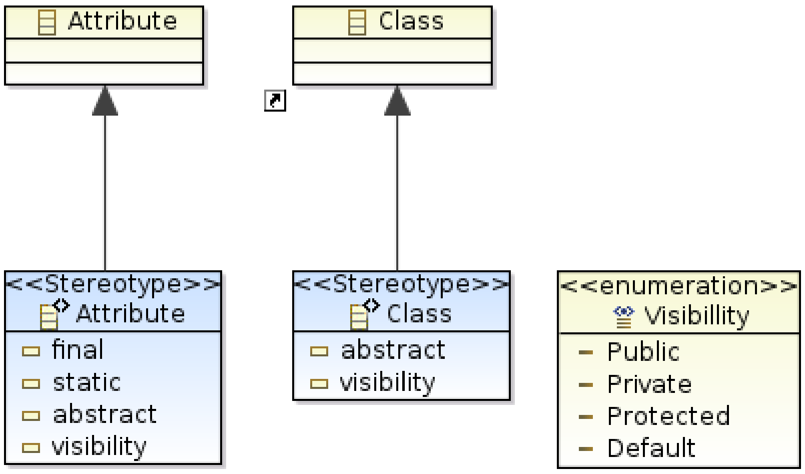


Figure A.5: Testprofile to extend the class diagram with Java attributes

- Enumeration *Visibility*: Contains the four Java visibility types, *default*, *private*, *public*, *protected*.

A.5 Java metamodel

This model is also a Java class diagram but has no profile like the model of Section A.4. All Java specific properties are covered in the metamodel. The metamodel is structured into the following classes, also illustrated in Figure A.6:

- Class *Classifier*: Is the basis for all other classes and therefore abstract and not instantiable. It has three different properties. The first property *name* represents the name of the object. The Boolean property *abstract* marks the object as abstract. The last property is an enumeration with the name *visibility* and defines the visibility of the object.
- Class *Class*: Represents a Java class and has no direct attributes but inherits from the class *Classifier* and has therefore three derived properties. Every Java class can have several children of type *Attribute* which are defined by the reference *attributes*. Also every class can have several *Methods* which are represented through the reference *methods*.
- Class *Attribute*: Are the class variables of a Java class. It extends from *Classifier* and has therefore three derived properties. In addition it also has the property *type* to define the type of the variable.
- Class *Method*: Representing the different methods of a Java class and extends from *Classifier*. In addition to the three derived properties from the super class has the

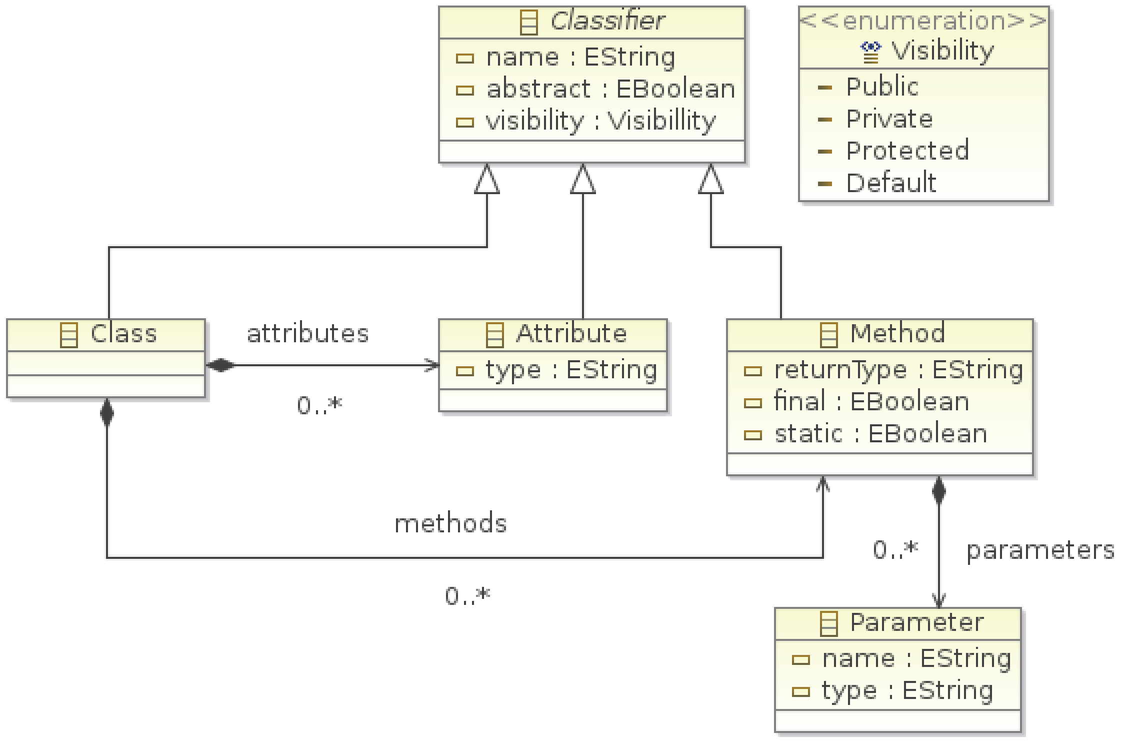


Figure A.6: Testmetamodel for the Java metamodel

Method three direct properties. The first is similar to the property of *Attribute*. It defines the return type of the method. The other two properties have the type Boolean and marking the method as static or final. The parameters for the method is a reference to with a *0..** multiplicity to *Parameter*.

- Class *Parameter*: Defining the parameters for methods and has only two properties. One for the name and the other for the type.
- Enumeration *Visibility*: Contains the four Java visibility types, *default*, *private*, *public*, *protected*.

A.6 Source

The source code of the RubyTL extension, the testmodels and the test source code can be found under the git repository https://github.com/mdopplinger/modelprocessors.

List of Figures

List of Tables

Bibliography

[Atl] Atlas Transformation Language. http://www.eclipse.org/atl. Accessed: 2011-12-27.

[BDJ$^+$03] J. Bézivin, G. Dupé, F Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[Bec03] K. Beck. *Test-driven development: by example.* 2003.

[BPSM$^+$] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML).

[Bud04] F. Budinsky. *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.

[Bé05a] J. Bézivin. In search of a basic principle for model driven engineering. *UML and Model Engineering*, 2005.

[Bé05b] J. Bézivin. On the Unification Power of Models. *Software and System Modeling*, 2005.

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. 2006.

[CM06] J. S. Cuadrado and J. G. Molina. A Plugin-Based Language to Experiment with Model Transformation. In *MoDELS*, 2006.

[CMT06] J. S. Cuadrado, J. G. Molina, and M. Tortosa. RubyTL: A practical, extensible transformation language. In *in 2nd European Conference on Preliminary Version Preliminary Version Model Driven Architecture*, 2006.

[DD87] C. J. Date and H. Darwen. *A Guide to the SQL Standard*, volume 3. 1987.

[EV06] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[FFVM04] L. Fuentes-Fernández and A. Vallecillo-Moreno. An Introduction to UML Profiles. *UPGRADE, European Journal for the Informatics Professional*, 2004.

[Fra] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf. Accessed: 2011-12-27.

[FTV02] L. Fuentes, J. M. Troya, and A. Vallecillo. Using UML profiles for documenting web-based application frameworks. *Annals of Software Engineering*, 2002.

[JK06a] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing*, 2006.

[JK06b] F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, Lecture Notes in Computer Science, 2006.

[Jon94] C. Jones. Software metrics: good, bad and missing. *Computer*, 1994.

[KFB01] M. A. Kolb, D. K. Fields, and S. Bayern. *Web Development with Java Server Pages*. 2001.

[Kle09] A. Kleppe. The Field of Software Language Engineering. In *Software Language Engineering*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009.

[Kol09] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *Model Driven Architecture-Foundations and Applications*, 2009.

[KPP06a] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[KPP06b] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Merging models with the epsilon merging language (EML). *Model Driven Engineering Languages and Systems*, 2006.

[KPP06c] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The epsilon object language (eol). In *Model Driven Architecture–Foundations and Applications*, 2006.

[KPP08] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The epsilon transformation language. *Theory and Practice of Model Transformations*, 2008.

[KPPR07] D. S. Kolovos, R. F. Paige, F. A. C. Polack, and L. M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 2007.

[KPR+10] D. S. Kolovos, R. F. Paige, L. M. Rose, N. Drivalos Matragkas, F. A. C. Polack, and K. Fernandes. Constructing and navigating non-invasive model decorations. *Theory and Practice of Model Transformations*, 2010.

[KPRGD12] D. S. Kolovos, R. F. Paige, L. M. Rose, and A. García-Domínguez. The Epsilon Book. *Structure*, 2012.

[Kuh06] T. Kuhne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 2006.

[Lia99] S. Liang. *The Java Native Interface: Programmer's Guide and Specification.* 1999.

[LS06] M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *Satellite Events at the MoDELS 2005 Conference*, 2006.

[LWWC11] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. From UML Profiles to EMF Profiles and Beyond. In *TOOLS*, 2011.

[MSUW02] S. Mellor, K. Scott, A. Uhl, and D. Weise. Model-Driven Architecture. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems*, Lecture Notes in Computer Science. 2002.

[OMGa] OMG. *Meta Object Facility (XMI) Specfication.* Version 2.4.1.

[OMGb] OMG. *Object Constraint Language (OCL) Specfication.* Version 2.2.

[OMGc] OMG. *Query/View/Transformation (QVT) Specfication.* Version 1.1.

[OMGd] OMG. *Unified Modeling Language Infrastructure Specfication.* Version 2.4.1.

[OMGe] OMG. *Unified Modeling Language Superstructure Specfication.* Version 2.4.1.

[OMGf] OMG. *XML Metadata Interchange (XMI) Specfication.* Version 2.1.1.

[OMGg] OMG's MetaObject Facility. http://www.omg.org/mof. Accessed: 2011-12-27.

[QRZ07] S. Queins, C. Rupp, and B. Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung.* 3. edition, 2007.

[Ran] A. Randak. ATL4pros: Introducing Native UML Profile Support into the ATLAS Transformation Language.

[RKPP08] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon generation language. In *Model Driven Architecture–Foundations and Applications*, 2008.

[RKPP10] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model migration with epsilon flock. *Theory and Practice of Model Transformations*, 2010.

[Sei03] E. Seidewitz. What models mean. *IEEE Software*, 2003.

[Sel07] B. Selic. A systematic approach to domain-specific language design using UML. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, 2007.

[SK03] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 2003.

[WKB03] J. Warmer, A. G. Kleppe, and W. Bast. *MDA explained: the model driven architecture: practice and promise.* 2003.

[WS09] M. Wimmer and M. Seidl. On Using UML Profiles in ATL Transformations. In *MtATL*, 2009.