Konzept und Implementierung einer Sprache zur Model-to-Model Transformation

Master’s Thesis
in Computer Science

by

Alma Baboci

First Referee: Prof. Dr. Kurt Schneider
Second Referee: Prof. Dr. Wolfgang Nejdl
Supervisor: M.Sc. Sebastian Meyer

March 22, 2011
Abstract

Model Transformations are increasingly recognized for their importance in Model Driven Development approaches. Different works and researches have attempted to understand the features of model transformation and describe the desired characteristics of transformation languages.

In recent years, several proposals have been introduced and different implementations exist for transformation languages. This thesis presents a new concept and implementation of a model to model transformation Language.

The ECore Model Transformation (ECMT) Language is a new unidirectional stateless model transformation language, implemented as an external DSL in Java. ECMT transforms instances of ECore metamodels into new instances of ECore metamodels and is mostly based on declarative patterns to match against model elements.

Kurzfassung

Modelltransformationen haben eine immer zunehmende Bedeutung in der Modellgetriebenen Softwareentwicklung. In verschiedenen Arbeiten und Untersuchungen wurde versucht Eigenschaften der Modelltransformationen zu verstehen und gewünschte Charakteristiken einer Transformationssprache zu beschreiben.

In den letzten Jahren wurden mehrere Vorschläge vorgestellt sowie verschiedene Implementierungen einer Transformationssprache. Diese Arbeit stellt ein neues Konzept und Implementierung einer Model2Model-Transformationssprache vor.

Die ECore Model Transformationssprache (ECMT) ist eine neue unidirektionale zustandslose Modelltransformationssprache, die als eine externe DSL in Java implementiert vorliegt. ECMT transformiert Instanzen des ECore-Metamodells in neue Instanzen eines neuen ECore-Metamodells. Die ECMT is hauptsächlich basiert auf dem deklarativen Einsatz für Matching und Relationen von Modellelementen.
Statement of originality

I declare that the work presented in the thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text, and that the material has not been submitted in any form any other university or institution.

Alma Baboci
Acknowledgments

On this page I would like to briefly thank the people who have helped me throughout this thesis.

First of all, I would like to thank my supervisor, M.Sc. Sebastian Meyer, for his outstanding assistance but most of all for his availability and helpful suggestions. I would also like to thank Prof. Dr. Kurt Schneider for his support and encouragement. Special thanks, to the helpful staff of Software Engineering Group, for making the laboratory room available during this work.

Additional appreciation is extended to my friends who have assisted me for all these months, but most especially to my family for their unconditional love and support during my studies.
Contents

Abstract ........................................................................................................................................... 2
Kurzfassung ................................................................................................................................... 2
Statement of originality .................................................................................................................... 3
Acknowledgments ............................................................................................................................. 4

1 Introduction ................................................................................................................................... 7
1.1 Motivation .................................................................................................................................. 8
1.2 Structure of the Thesis ............................................................................................................... 8

2 Background .................................................................................................................................. 9
2.1 Metamodelling Architecture ..................................................................................................... 9
2.2 Model Driven Engineering (MDE) ............................................................................................ 11
   2.2.1 Model Driven Architecture (MDA) .................................................................................... 12
   2.2.2 Eclipse Modeling Framework (EMF) ................................................................................ 13
2.3 Model-to-Model Transformation (M2M) .................................................................................. 14
   2.3.1 Model Transformation Terminology ................................................................................ 14
   2.3.2 Classification of Model Transformations ......................................................................... 16

3 Related Work .............................................................................................................................. 22
3.1 Languages and Technologies .................................................................................................... 22
   3.1.1 Query/View/Transformation Specification (QVT) ......................................................... 23
   3.1.2 Atlas Transformation Language (ATL) ........................................................................ 26
3.2 Discussion ................................................................................................................................ 29

4 The ECMT Language .................................................................................................................. 31
4.1 Language Overview .................................................................................................................. 31
4.2 The Relational Approach ....................................................................................................... 32
4.3 Basic Features .......................................................................................................................... 32
4.4 Pattern Language .................................................................................................................... 34
4.5 Matching Source Model Elements ........................................................................................ 37
4.6 Creating Target Model Elements .......................................................................................... 38
4.7 Relating Source and Target Model Elements ....................................................................... 39
4.8 Tracing Model Elements ........................................................................................................ 40

5 Language Implementation .......................................................................................................... 42
5.1 Architecture .............................................................................................................................. 42
1 Introduction

The requirements in software development have been changed significantly in recent decades. The importance of designing, adapting and developing software in a time efficient way, without losing the quality of products, has been increased. To meet these requirements, Model Driven Engineering (MDE) proposes the use of models at different levels of abstractions for developing systems [2]. The main objectives of MDE are to raise the level of abstraction in system specifications; therefore, to control the increasing complexity, to hide the accidental complexity of the underlying technologies [3] and to increase the automation in system development by performing model transformations. Different implementations of this approach include Software Factories from Microsoft or Model Driven Architecture (MDA) from Object Management Group OMG [13].

Model transformations are considered the key part of Model Driven Engineering (MDE). In general, model transformations can be divided in two different categories: Model to Model transformation, a transformation where a model is converted to another model of a system and Model to Text transformation, a transformation where a model is converted directly to an arbitrary fragment of text. In case the text produced by transformation is source code or a fragment of source code, this transformation is called Model to Code Transformation. Model to Code transformation is usually used to implement modeling languages whereas Model to Model transformation is usually used for refactoring, migrating models to another modeling language or changing the abstraction levels. In some cases, when the syntax of source code would be considered as a model, then Model to Code Transformation can also be categorized as a Model to Model transformation (M2M). Another category is Code to Model Transformation, usually used to extract models from code.

In the recent years, several proposals have been introduced and different implementations exist for model to model transformation (M2M) languages. They offer different features and follow different approaches; consequently, each of them may provide certain advantages and disadvantages for different situations of model to model transformations.

Different works consider domain-specific transformation language as being better suited for implementation in some specific transformation problems. The idea of using domain-specific language (DSL) [31] is already supported by the model transformation communities with several DSLs having emerged in the past.

A well-known model to model transformation language is represented by Query/View/Transformation Specification (QVT) [14], which is a proposal from Object Management Group (OMG) and is described as a general purpose transformation language.

This thesis is focused on Model to Model Transformation (M2M) and offers a new concept for a model to model transformation language, and is implemented for models defined in ECore metamodelling language [16].
The ECore Model Transformation (ECMT) Language is a new unidirectional stateless model transformation language, implemented as an external DSL in Java. ECMT transforms instances of ECore metamodels into new instances of ECore metamodels and is mostly based on declarative patterns to match against model elements.

1.1 Motivation

In complex software systems, models play an essential role. A concrete example of this is the representation of text documents though models for enabling the extraction of semantic information from them. The materials are available as Word documents and are constructed according to a predefined document structure. Some specific information of documents is to be represented in a new structure. In order to extract the semantic information from documents, a transformation description should be defined. This transformation should be able to convert specific instances of source models and build new instances of other models based on a transformation description that is defined by the user.

1.2 Structure of the Thesis

This thesis is structured as following.

In Chapter 2, the foundation and important background information, relevant for this master thesis, is briefly described. It gives an overview of Metamodelling Architecture, Model Driven Engineering (MDE) and Model to Model (M2M) Transformations in Sections 2.1, 2.2 and 2.3 respectively. Chapter 3 gives an overview of existing model to model transformation languages. Two of the most popular transformation languages, Query/View/Transformation Specification (QVT) and Atlas Transformation Language (ATL), are described in Section 3.1.1 and 3.1.2 respectively. A general discussion about model to model transformation languages is made in Section 3.2. The ECMT Language and its features are described in Chapter 4. Implementation of the ECTM language is described in Chapter 5. Implementation architecture and transformation engine and its environment are described in Sections 5.1 and 5.2. Abstract and concrete syntax is shown in sections 5.3 and 5.4. A simple example of an ECMT transformation description between two example models will be presented in section 5.6. Chapter 6 makes an evaluation of ECMT language based on language characteristics classification proposed on [4]. In Chapter 7, a summary of the conclusions and outlook of the work is presented.
2 Background

This chapter provides the foundations and the theoretical basics of this thesis. Section 2.1 introduces the Metamodelling Architecture and its standards. Section 2.2 describes an overview of Model Driven Engineering (MDE) and its development cycle. Eclipse Model Framework and ECore are briefly described in section 2.3. Section 2.4 introduces Model to Model Transformation (M2M)’s terminology and general classification of features and approaches.

2.1 Metamodelling Architecture

The interpretation of the terms model and metamodel depends strongly on domain where the models are used. This thesis is focused on definitions made in the domain of software engineering. In [1] a model is a simplified representation of a system that helps to gain a better understanding of the system. According to [26] a model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system. A model can be considered as an abstract, compact and formal description of the system or system’s environment for some certain purpose. In accordance to [21] a model is a description of a (part of) system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer. To enable the automation in development processes, a model must be described in a well-defined language, proper for the computer interpretation, with some sort of syntax for describing models and semantic for defining the meaning of corresponding models. Since models are not always defined in a text based language, a new mechanism is needed that supports the concept of visual languages, like for example the Unified Modeling Languages (UML) [11].

This mechanism, called metamodelling, implies that a modeling language is a model itself, which also must be described in a well-defined language. The term metamodel is defined in [12] as a model that defines the language for expressing a model. According to [1] it makes statements about what can be expressed in the valid models of a certain modeling language. In this context a model is said to conform to its metamodel like a program conforms to the grammar of the programming language in which it is written [17]. A metamodel should also be part of a metamodel architecture in which a metamodel can be viewed as a model and it is itself described by another metamodel called meta-metamodel.

The question of how the meta-metamodes are to be defined, results in an infinite loop of meta-levels. To avoid this, the meta-metamodels and their language (meta-language) are often described as self-defining; therefore, the metamodel of the meta-metamodel is in return the meta-metamodel itself.
Many approaches define the metamodeling architecture with four abstraction layers: system, model, metamodel and meta-metamodel. The figure 2.1 illustrates the modeling levels of Object Management Group (OMG) approach, which will first be described individually:

The *meta-metamodeling* layer or *M3-Layer*: This layer represents the meta-metamodels which contains the basic elements for specifying metamodels. It defines the language for specifying the metamodels.

The *metamodeling* layer or *M2-Layer*: In this layer reside the metamodels. Metamodels are valid instances of meta-metamodel in M3-layer and define languages for specifying models.

The *model* layer or *M1-Layer*: Here can be found the models which are meant to be representations of real world or real entities of systems.

The *instance* layer or *M0-Layer*: This layer contains the concrete data that are defined through models in M1-layer.

![Figure 2.1: The four-layer metamodeling architecture](image)

In general, it is said that metamodels are used to define a set of models and meta-metamodel specifies all the metamodels including itself.

One benefit of metamodeling is its ability to describe domain specific languages in a unified way. This means that languages can be more easily manipulated and managed. Metamodelling is also used for constructing semantically rich languages that abstract from specific technological implementation and focus more on the problem domain. By using metamodels, many abstraction levels can be defined and combined to create new languages that are more suited for a specific domain. As a result automation and productivity are greatly improved.

10
Currently different self-defining meta-metamodels are being proposed: Model Object Facility (MOF) [12], EMF ECore [16] as an implementation of Essential MOF (EMOF), Kernel metamodel (KM3) [27], Kermeta [38], Corel Simple Metamodel Description [28] and many others. All of them serve a MDE implementation as meta-metamodel but differ from each other in expressiveness.

In context of this thesis, only meta-metamodel MOF and ECore are considered, as the most popular standards. Figure 2.2 shows an example of OMG’s four layer architecture. Object Management Group (OMG) uses MOF as meta-metamodel. Complete MOF (CMOF), Essential MOF (EMOF) and Semantic MOF which together build MOF are also self-defined. A well-known and widely used language for defining metamodels is the UML specification, which consist of UML Infrastructure and UML Superstructure.

![OMG’s four layer architecture example](image)

### 2.2 Model Driven Engineering (MDE)

The use of models in Software Engineering has been introduced for some time. Because the appliance of object oriented software development has had a strong establishment in development processes, the importance of modeling has also been increased.

Model Driven Engineering (MDE) promotes the idea of using models at different levels of abstraction for developing systems [2]. Using models, MDE mostly aims to raise the level of abstraction in system specifications and therefore control the raising complexity. It hides the accidental complexity of the underlying technologies [3] and increases the automation in system development by performing model transformation.

Other terms such as Model Driven-Software Development (MDSD) [8] or Model Driven Development (MDD) share common concepts but do not always imply the same methods. The MDE approach is not related to any specific standard therefore several implementations...
have been emerged that follow the principles of MDE: Software Factories from Microsoft, Model Driven Architecture (MDA) from OMG [13], Eclipse Modeling Framework (EMF) [16], Model Integrated Computing (MIC) from DARPA etc. Most of them have different techniques and definitions but share the same concepts.

In the following, two different technologies of MDE are explained more in detail: Model Driven Architecture (MDA) as the most popular standard of Model Driven Engineering (MDE) and Eclipse Modeling Framework (EMF), a technology that supports MDA.

2.2.1 Model Driven Architecture (MDA)

The vision of Object Management Group (OMG) for MDE is the Model Driven Architecture (MDA) [13]. MDA tends to be restrictive in using the UML-based modeling languages for models which are its primary artifacts. The main objective of MDA is the separation between the specification of the operations of a system and the implementation details of it. It also aims for portability, system integration, system interoperability, reusability, being domain oriented and an efficient software development [9].

In the system architecture of MDA, models provide different levels of abstractions. They are arranged into four levels: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) and the Program-Code.

**CIM** abstracts the software details of a system. It is also sometimes called the *Business Domain Model*. It describes the business context and requirements. It is a computational independent viewpoint focused on system requirements and system’s environment. **PIM** describes the structure and functions of a system, abstracting the specific details of software technology platform. It is a platform independent viewpoint of the system focused on operation and functions of a system. **PSM** represent a view of a system from the platform specific point of view. **Program code** is the implementation of PSM in a free chosen programming language.

![Figure 2.3: Code Generation in MDA](image)
Model transformation forms a key part of MDA. A common use of transformations is the transformations of Platform Independent Model (PIM) to Platform Specific Model (PSM), and PSM to code.

Figure 2.3 describes how code is generated across the different refinement levels in MDA. Model like Platform Description Model (PDM) and Transform Description Model (TDM) contain the necessary information for a transformation from PIM to PSM.

### 2.2.2 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and other applications based on a structured data model. It provides a pluggable framework to store the model information, using default XMI (XML Metadata Interchange) to persist the model definition. From a model specification described in XML Metadata Interchange (XMI), EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor [16].

On top of EMF a number of other frameworks are built, like GEF (Graphical Editor Framework) allowing it to create graphical editor and GMF (Graphical Modeling Framework) for creating a domain specific modeling language editor.

EMF uses ECore as meta-metamodel for describing models and offers runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective Application Programming Interface (API) for manipulating EMF objects generically.

A simplified ECore class hierarchy of the ECore metamodel is shown in figure 2.4.

---

![ECore Class Hierarchy Diagram](image-url)

**Figure 2.4: ECore Class Hierarchy**
2.3 Model-to-Model Transformation (M2M)

In general, the following categories of model transformations can be distinguished: Model to Model transformation, a transformation where a model is converted to another model of a system and Model to Text transformation, a transformation where a model is converted directly to an arbitrary fragment of text. In case the text produced by transformation is source code or a fragment of source code, this transformation is called Model to Code Transformation. Model to Code transformation is usually used to implement modeling languages whereas Model to Model transformation is usually used for refactoring, migrating models to another modeling language or changing the abstraction levels. In some cases, when the syntax of source code would be considered as a model, then Model to Code Transformation can also be categorized as a Model to Model transformation (M2M). Another category is Code to Model Transformation, usually used to extract model form code. This thesis is focused on Model to Model transformation.

2.3.1 Model Transformation Terminology

Model Transformation is a key technology and a relatively young field of MDE, therefore with several, partly overlapping definitions. In context of MDA, it is defined as the process of converting a model to another model of the same system [22] or as the automatic generation of a target model from a source model, according to a transformation description [21]. According to [20] it is simply a program that mutates one model into another. Considering the number of models taking place in a transformation, [6] defines model transformation as the automatic generation of one or multiple target models from one or multiple source models, according to transformation description.

A model transformation implies the need of different languages for specifications of the models taking place in it; at least one language that defines the source and target models and another which defines the transformation itself is needed.

In this context, it is important that Unified Modeling Language (UML), ECore and any other modeling tools, to be considered as languages. In figure 2.5 the meta-levels of a model transformation are shown as well as how the metamodels of source and target models are prerequisite for model transformations.

In the following, some terminology of the model transformation is being defined.

Source model is a model used as input for model transformation. It conforms to the source metamodel. One or more source models can be used as input for a transformation.

Target model is a model used as output for model transformation. It conforms to the target metamodel. One or more target models can be used as output for a transformation.
**Model transformation language** is a grammar with well-defined semantics for performing models transformations. It can be textual or graphical. The different designing principles of model transformation languages and their classification are described in section 2.3.2.

**Model transformation description** is written in a model transformation language. It expresses how one or more source models are transformed in one or more target models. In case the model transformation language is a rule based language, a transformation description is defined as a set of rules [21].

![Diagram](image)

**Figure 2.5: The meta-levels of a model transformation** [15]

**Model transformation rule** is the smallest entity of transformation description. It describes how a fragment of the source model can be transformed into a fragment of the target model [21]. Rules can be composed of other rules or extends other rules. A rule contains a matching and/or an application part.

**Model transformation engine** interprets or execute the model transformation description. It applies the transformation description on source model using them as input and produces target models as output. For interpreting the model transformation description an engine typically perform the following steps [20]:

15
First, it identifies the elements in source models to be transformed and then for each found element it produces, as in transformation description defined, the associated target elements. Secondly, it may produce some tracing information to map the effected source and target elements.

**Transformation model** is the model representing the model transformation description. It also enables it to be as a source or a target model of another transformation.

**Higher-order Transformation (HOT)** is a transformation where source or target model are also transformation models [23].

**Query** is an expression that is executed on a model. The result is one or more objects of source model or objects of types defined by the query language. Queries have no side effects.

**View** is a model that contains certain information of the source model from a certain point of view. It may represent a subset of the information of the source model or organize it in a different way. That is, a view model can have a different structure and hence a different metamodel from that of source model. A view needs therefore its own metamodel and a transformation that generate it from a source model. A view model can often not be changed. If however changes are allowed, it will take place directly on source model and the transformation should be bidirectional.

**Matching part (left-hand side, LHS)** intends to match objects in one or more source models, classifying parts of source models which elements fulfill some certain properties. A match is a part of the source model, represented as a set of object. In case of multiple source models, a match consists of several parts of them.

**Mapping part (right-hand side, RHS)** defines how a match in a source model should be transformed into objects in a target model. Objects of target models are generated either implicitly or explicitly. Implementations can be interactive and unidirectional or declarative and bidirectional.

**Pattern** term is not always used uniformly. In different works it represents both matching and mapping, whereas in [20] it represents expression written in pattern language. The matching part is in this case, a composition of several such expressions.

### 2.3.2 Classification of Model Transformations

Several works [4] [5] [6] have attempted to understand the features of model transformations and describe the desired characteristics of transformation languages. In the following, as first, the different features of usage scenarios concerning the source and target models for model transformation are defined, secondly, a classification of different approaches for model transformations languages is explained and as last an overview of features of transformation languages is described.
1. Classification of Model Transformations, concerning Source and Target Models

Model transformation in which the metamodels of source and target models are the same are called *endogenous* transformation. In this case, when producing the target model, the transformation usually changes only a specific part of the source model, so that to the largest extent the models are the same. The number of models involved in transformation decides whether it is an *in-place* or *out-place* transformation. An *in-place* transformation has a source model as a target model. In this case the target model is created by modifying specific part in the existing source model. The *out-place* transformations uses at least two distinct models, assuming that the target model is empty otherwise the information will be rewritten; it creates model elements in a model based on properties of another model, even though models conform to the same metamodel [6]. Examples for *endogenous* transformations, also called *rephrasing* transformations include refactoring, optimization, simplification or normalization of models. Contrary to *endogenous* transformations, *exogenous* transformations have source and target models conforming to different metamodels. They also called *translation* transformation because of the mapping between the different metamodels. Examples of *exogenous* transformations are synthesis, reverse engineering and migration of models.

Model transformation can also change the level of abstraction between source and target model. A *vertical* transformation uses different levels of abstraction for source and target models [4]. It can decrease the level of abstraction adding additional information or details. This kind of transformation is also called a *refinement* transformation. Examples of this are the transformation of Platform Independent Model (PIM) to Platform Specific Model (PSM) and the code generation in MDA (showed in fig. 2.3). On the other hand, if details from source model are removed then the abstraction level of target model will be increased. Examples of vertical transformations are, for instance, refinement (specialization) or abstraction (generalization).

The opposite of *vertical* transformation is the *horizontal* transformation. In this case, the representations of models are changed but the abstract levels of source and target models remain the same. Examples of horizontal transformation are merging, refactoring or translation of a model to a similar metamodel.

Another feature of model transformation is the *technical space* or *technological space* (TS) in which models are represented. A technical space is a model management framework with a set of tools that operates on the models definable within the framework [24].

Transformation engines that can be used for a model are limited through the *technical space* of the model, since not all transformation supports crossing the boundaries of different technical spaces. Examples of technical spaces are MDA, XML or Grammar Syntax TSs.

In figure 2.6 is depicted a graphical presentation of technological spaces in the meta-levels architecture.
Numerous different transformation languages have emerged nowadays. Since each of them is developed with some specific goal in mind, they do not always share the same characteristic and properties. The following classification of transformation languages is based on [4] [6]. Taking in consideration only the Model to Model transformation, the languages paradigms can be categorized as follows:

**Direct Manipulation** is a technique where the model transformations are implemented using general purpose languages. An internal model representation is needed. Accessing models and metamodels is provided through an Application Programming Interface (API). The main advantage of this approach is that the user does not need to learn a new languages syntax or development tool; on the other hand the maintenance of the implementation is more difficult. Example of this approach is the Java Metadata Interface (JMI) [25].

**Imperative (Operational)** is a technique very similar to imperative paradigm in general programming languages like JAVA or C++. It consists of a sequential control flow with intention to describe how the transformation description is to be executed. Although there is some similarity with direct manipulation technique, here is no external API needed. The well-defined control flow means that all statements in the transformation code are to be executed in a pre-defined order. Easy learning for an experienced programmer, high level of control and flexibility in implementation are some of the advantages of this technique. On the other hand, it suffers under overhead code and explicit effort from programmer for accomplishing a specific task.
Declarative (Relational) is an approach where the focus is based on what is to be transformed into what. This is mostly specified through relations between elements in source and target models. Constrains using for example OCL [19] are also used for querying or putting conditions on elements of models. Many declarative languages support bidirectionality which for imperative languages is hardly possible. Since procedural information is hidden, the language is more compact and easier to comprehend compared to the imperative languages, but contrary to the latter, there is no explicit control flow or rule order and complex transformation are harder to be specified.

Hybrid approach tries to use the advantages of both declarative and imperative languages by offering both methods in a language. Addressing the most appropriate method to specific situation, hybrid languages seem to be more flexible.

Structure-driven is a technique that uses two phases for a transformation. The first phase is concerned with creating the hierarchical structure of the target model; whereas, in the second phase the attributes and references in the target model are being set. The user has to write the transformation rules, while the framework is responsible for scheduling.

Graph Transformation is a technique where models are interpreted as graphs and graph theory foundation is used for transformations. Triple Graph Grammars (TGG) is the common way for describing the transformation. Transformation rule are specified by left-hand side graph, right-hand side graph and correspondence graph which respectively define a sub-graph of source graph, a sub-graph of target graph and the mapping between them. The mapping implies a similarity with declarative paradigm that is why the graph transformation languages are also considered as subcategory of declarative languages.

3. Classification of Model Transformation concerning Language Features

Different characteristics and properties of model transformation languages are collected and divided into categories. This makes possible the exploring and evaluating for the most of model transformation approaches. A classification of features is made in [4] [6]:

Specification Some approaches provide a dedicated specification mechanism, using pre-conditions and post-conditions expressed, for example, in Object Constraint Language (OCL)[19] A particular transformation specification may represent an expression or a function that must be validated between source and target models. In general it describes relations and is not executable.

Transformation Rules are subdivided into domains that are the part of the rule responsible for accessing one of the models that are used in the transformation rule. A domain is again subdivided in domain language, static mode, dynamic mode restriction, body, and typing.

The domains subdivisions define how many domain languages the rule operates, how the domains are declared, if they are implicitly or explicitly declared, such as if a specific domain is defined as in, out, or inout, or if the rule body supports variables for example. Other classifications from the transformation rules are also the syntactic separation, application
Rule Application Control is divided in two aspects, the location determination and the rule scheduling. The location determination is a strategy for determining the location where the matches in models is to be located for a certain rule. This strategy can be deterministic, nondeterministic, or interactive. One example of a deterministic strategy is a depth-first search in the source model, an example of a nondeterministic strategy is when a rule is applied to a random element in the source model. The nondeterministic strategies are again subdivided in concurrent, and one-point. An interactive strategy approach is when the user is allowed to interact with the transformation for determining the location where the rule is going to be applied.

The rule scheduling is related to the order of rule application. It is subdivided into four other features: form, rule selection, rule interaction, and phasing. The form refers to the way scheduling is being expressed. It can be implicit or explicit. The rule selection defines how the rule is selected and controlling when a rule is to be applied. It can be deterministic (interactive, explicit, conflict resolution) or nondeterministic. The rule interaction refers to rules mechanisms for interactions, and it can be recursion, looping, or fixpoint operation. Phasing is a strategy for dividing a transformation process several phases and determining a specific set of rules that can be executed in a certain phase.

Rule Organization is a strategy for organizing the rules. It is subdivided in modularity mechanism, reuse mechanism and organizational structure. The modularity mechanism spreads rules into modules. The reuse mechanism enables the transformation rules to be reused from other transformation rules; examples are rule inheritance, inheritance between modules and rules extensions. The organizational structure refers to the mechanism of organizing rules, for example grouping rules depending on source or target models.

Source Target Relationship was broadly explained in the classification of model transformations, concerning source and target models.

Traceability refers to the mechanism that provides a footprint for the transformation. It maps the elements of source and target model that were affected from transformation rules. Tracing information is useful for analyzing the connection between the source and target elements produced from transformation. It may be implemented as a tool or as part of transformation description. It can be stored in the source model, target model or in a separate model that defines it.

Incrementality is related to the ability for propagating the changes of a transformation without performing the whole transformation again. The changes made in source model are updated in the target model in a way that the target model does not need to be regenerated completely.

Transformations are in [20] categorized as statefull or persistent. Non-incremental or statefull transitions regenerate the whole target models, whereas the incremental or persistent transitions allow update in target models propagating only the changes in the source model.

Incrementality is also subdivided into three groups: source incrementality, target incrementality, and preservation of user edits in the target. The source incrementality minimizes the number of source elements that must be reexamined in a transformation when a
source model is changed. The target incrementality refers to the ability of changing a target model, based on changes performed in the source model without completely rebuilding the target. The preservation of user edits in the target refers to the ability of user to make manual modification in a target model and transformation preserving it when the target is regenerated.

**Directionality** refers to the transformation direction, a transformation can be *unidirectional* in cases that the target model is created or modified based on a source model or *multidirectional*, in cases where the transformations are performed in several directions, if a single source and target model are used and the transformation is called *bidirectional*. Multidirectional transformations can be achieved by defining multidirectional rules or by defining several unidirectional rules, one in each direction.
3 Related Work

This chapter provides an overview of model to model transformation languages. Two of most popular transformation languages, Query/View/Transformation Specification (QVT) and Atlas Transformation Language (ATL), are described in Section 3.1.1 and 3.1.2 respectively. A general discussion about model to model transformation languages is made in Section 3.2.

3.1 Languages and Technologies

Several specifications, implementations and executions of Model to Model Transformations (M2M) were proposed. Each of them offers different features and follows different approaches, therefore providing certain advantages and disadvantages for different situations of model to model transformations.

In the following, some well-known model to model transformations languages are mentioned, whereas two of them will be discussed in more detail; Query/View/Transformation Specification (QVT) and Atlas Transformation Language (ATL).

Query/View/Transformation Specification (QVT) [14] is a standardized language for model transformation and is established from Object Management Group (OMG). QVT is a hybrid language that integrates declarative and operational parts. It defines three languages: a high level declarative language called QVT-Relations. QVT-Core, a low level declarative language and QVT Operational, an imperative language that extends QVT-Relations with imperative constrains.

Two implementations of QVT-Relations are ModelMorf [32] and mediniQVT [33] languages. ModelMorf does not support all features of QVT-Relations, like incrementality or transformation extensibility. Both of them have also no support of QVT-Relations graphical syntax. The QVT-Relations graphical syntax is implemented from Fuja [34] and MOFLON [35] supporting the Triple Graph Grammars (TGG).

SmartQVT [36] is an implementation of QVT-Operational language. It is written in Java and is offered as an Eclipse plug-in. Another imperative transformation language is Kermeta [37]. It can be described as a metamodelling language that allows a description for both the structure and the behavior of models.

Atlas Transformation Language (ATL) [38] is the ATLAS INRIA & LINA research group's answer to the OMG QVT. It is a hybrid model transformation language incorporating both imperative and declarative paradigms. Other model to model transformation languages include: Tefcat [39] a declarative, open source and unidirectional language. Model Transformation Framework (MTF) from IBM, oriented to QVT with support of declarative mapping and describing a transformation through relations [40]. Bidirectional Object-oriented Transformation Language (BOTL) [41]. EMF Henshin [42] as a continuation of the EMF Tiger, both as Triple Graph Grammars (TGG) based languages. Yet another Transformation Language (YATL) [43]. Epsilon Transformation language (ETL) [44] etc.
3.1.1 Query/View/Transformation Specification (QVT)

Query/View/Transformation Specification (QVT) [14] is a new standard for model to model transformations, established from Object Management Group (OMG) with support for the Meta Object Facility (MOF) and Object Constraint Language (OCL) standards. QVT defines three languages for model transformation: QVT-Relations, a high level declarative transformation language. QVT-Core, a low level declarative transformation language and QVT-Operational, an imperative transformation language that extends QVT-Relations with imperative constrains. These languages are organized in a layer architecture shown in Figure 3.1.

![Figure 3.1: QVT languages layered architecture](image)

The languages Relations and Core are at two different levels of abstractions.

The Relations language is a user-friendly transformation language that provides capabilities from specifying transformation as a set of relations among model elements. Relations are built upon the concept of object-patterns. It supports complex pattern matching and handles the manipulation of traceability links automatically. The user must define the relationships between the source and target models but is not responsible for the creation and destruction of objects.

The Core language is simpler than the Relations language, therefore transformation definitions written in it are longer than the equivalent definitions written in Relations language. Traceability is not made automatically so the developer is responsible for creating and using the links. It provides semantics of Relations language which is given as a transformation RelationsToCore.

Listing 3.1 shows the syntax of a relation in QVT-Relations language.

```plaintext
transformation transformation_name (model_name_1 : model_type_1, ..., model_name_n : model_type_n )

[checkonly | enforce] domain : model_type_1
<domain_1_variable_set> {
  <domain_1_pattern_set>
  [domain_1_condition]*
}
...

[checkonly | enforce] domain : model_type_n
<domain_1_variable_set> {
  <domain_1_pattern_set>
  [domain_1_condition]*
}
```


The transformation keyword is followed by the name of the transformation. All models that take place in transformation are declared and passed as parameters. There are two kinds of relations, the top-level relation and the non-top-level relations. A top-level relation is defined by using the top keyword and is required from execution of transformation to be held. Non-top-level relations are required to be hold only in cases where they are invoked directly or transitively from the where clause from another relation. After the keyword relation, the relation name is defined. After the keyword var, the set of variables occurring in the relation is defined. The checkonly and enforce keywords constrain the way a transformation will occur in a given direction. Checkonly indicates that the domain elements cannot be changed (i.e. they are read only) by the transformation execution. Enforce indicates that the transformation engine should change the elements of the domain to ensure the relation. After the keyword domain, the name of the domain is defined. Next, a set of variables that is a subset of relation variables occurring in domain is declared followed by the set of constraints implied by the pattern of current domain. A pattern can be viewed as a set of variables and constraints that model elements bound to those variables must satisfy in order to qualify as a valid binding of the pattern. The when and where clauses specify the predicates for a QVT relation. The when clause specifies the conditions under which the relation needs to hold. The where clause specifies the condition that must be satisfied by all model elements participating in the relation, and it also offers the possibility for invoking other relations.

In some cases, it is difficult to find a solution for a given transformation problem using the declarative languages. For this purpose QVT proposes two mechanisms for extending the declarative languages Relations and Core: Operational Mappings Language (OML) and a mechanism for invoking imperative transformation functionality in an arbitrary language (Black Box implementation).

The Operational Mapping language extends the Relations language with imperative constructs and Object Constraint Language (OCL) constructs with side effects. Object-patterns are instantiated by using imperative constructs providing constructs commonly found in imperative language like loops, conditions etc.

An OML module is basically composed of a header and an implementation part. Listing 3.2 shows the header syntax.

```plaintext
modeltype model_referring_name_1 uses
    model_name_path_1;
    where { where_conditions }
    ....
modeltype model_referring_name_n uses
    model_name_path_n;
    where { where_conditions }
```

Listing 3.2 OML header syntax

After the keyword modeltype the model should be defined. The name is used to be as reference to the model in the transformation module. The keyword uses is followed by the actual model location. After the keyword where, some additional contrasts of the model type are defined.

An example of OML implementation section is shown in listing 3.3.
  main()
  {
    Docu.objectOfType(Hyperlink)->map hyper2part
  }
}

Listing 3.3 OML example

The implementation section is defined by the transformation keyword. Two models are passed as parameters. The in and out keywords define the transformation direction. The inout keyword can also be used, in case a model should not be rewritten.

A transformation has always an entry point from which the execution starts. It is defined with the keyword main. From main other mappings may be invoked. In the example, the main entry operation firstly retrieves the list of objects of type Hyperlink and then applies a mapping operation called hyper2part on each object of the list. The mapping operations provide a crucial behavior in the transformation, the Listing 3.4 below shows a general syntax of a mapping operation.

mapping [direction_keyword]?
  <context_type> :: <mapping_name>
  ( [direction_keyword_1 parameter_name_1 : parameter_type_1, 
    ... 
    direction_keyword_n parameter_name_n : parameter_type_n]?)
  : return_type_1, ...
  return_type_n
  [when {statements}]?
  [where {statements}]?
  {
    [init {statements}]?
    [population {statements}]?
    [end {statements}]?
  }

Listing 3.4 OML mapping syntax

A mapping operation is syntactically described by: a mapping signature, a when clause as pre-condition, a mapping body and a where clause as post-condition. Its signature is defined after the keyword mapping. The keywords in, out or inout are optional and can be used to show the direction of mapping operation. A context and context type must be defined, followed by the mapping name.

Each mapping operation may take one or more parameters. A mapping can return one or more elements which are referred by the keyword result. The when and where clauses are optional and executed before the mapping body. The mapping body contains an init, population and end section. The init section is optional and is executed before the other two blocks. It contains some code to be executed before the instantiation of the declared outputs. The population section is optional and contains code to populate result parameters while the end section contains some additional code and is to be executed at the end of the transformation.
3.1.2 Atlas Transformation Language (ATL)

Atlas Transformation Language (ATL) [38] is implemented from ATLAS INRIA & LINA research group as an answer to OMG MOF/ Query/View/Transformation Specification [14]. In terms of semantic, ATL is though similar to QVT, but its syntax differs. It is a hybrid model to model transformation language incorporating both imperative and declarative paradigms, designed to express model transformation as required by MDE. It provides a set of tools, including a transformation repository, sample transformations and an ATL transformation engine. It is a component of Atlas Model Management Architecture (AMMA) [29] which is implemented on the top of EMF.

The abstract syntax of ATL conforms to a metamodel which implies that an ATL transformation definition is also a model. The output models can be generated in the XML Metadata Interchange (XMI) [18] or Kernel Meta Meta Model (KM3) format. ATL integrate Object Constraint Language (OCL) [19] and offers a standard library composed of OCL data types, some own data type extensions and several operations.

The source code is composed of a mandatory header section, optional helpers and several transformation rules. In the header section, the module name is defined and the source and target models are declared. It starts with the keyword module followed by the name of the module. The keyword create indicates the target models while the keyword from indicates the source models. It is also possible to import other libraries to be used in the transformation. The uses keyword refers to the name of the library. An example of header section is given in listing 3.5.

```java
1 module Word_Document2Protocol;
2 create OUT : Word_Document from : Protocol;
3 uses library_name;
```

Listing 3.5 ATL header section example

Helpers are comparable to Java methods. They are used to avoid code redundancy by defining global variables and function which can be specified with or without context. Helpers can be called from any part of an ATL module.

```java
1 helper [context context_oclType]? def :
2 helper_name ((parameter_name : parameter_oclType)*)? : return_oclType = oclExp ;
```

Listing 3.6 ATL helper syntax

Transformation rules are the central part of ATL; they specify how target elements are to be generated. Two different kinds of rules are defined: the matched rules and the called rules.

The matched rules specify which source elements must be matched, the number and the type of the generated target model elements, and the way the target model elements are to be initialized from the matched source model elements. In listing 3.7 is shown the syntax of ATL matched rule.

```java
1 matched_rule [context context_oclType]?
2 (parameter_name : parameter_oclType)*)? : return_oclType = oclExp ;
```
Listing 3.7 ATL matched rule syntax

```
rule rule_name {
from
  in_var : in_type [in model_name]? [(
    condition
  )]? [using { var1 : var_type1 = init_exp1;
    ...
    varn : var_typen = init_expn;
  }]? [to
    out_var1 : out_type1 [in model_name]? ( bindings_1
    ),
    out_var2 :
    distinct out_type2
    foreach (e in collection) ( bindings_2
    ),
    ...
    out_varn : out_typen [in model_name]? ( bindings_n
    )
  )
  [do {
    statements
  }]
}
```

The `rule` keyword is followed by the name of the rule. The matched rule name must be unique inside an ATL module. The next two blocks are mandatory, they refers to the declarative part of the rule. The `from` keyword refers to source model pattern to be matched and also an optional `condition` that is to be satisfied from the source model elements.

The `using` keyword refers to an optional block where local variables can be initialized. Each variable is identified by its name and its data type, and it must be initialized using an OCL expression.

The `to` keyword corresponds to the target pattern of target model elements.

The `do` block represents the imperative part of the rule and it enables the use of a sequence of ATL imperative statements that will be executed once the target model elements generated by the declarative part of the rule has been completed.

The matched rules are also subdivided into three other categories: `standard` rules, `lazy` rules and `unique lazy` rules. They differ from each other in the way they are triggered. Standard rules are applied once for every match that can be found in source model element. Lazy rules are triggered by other rules and they are applied on a single match as many times as it is referred to by other rules. Unique lazy rules are also triggered by other rules but the target model elements will be created only one time for a match.
The *called* rules define an additional kind of rules to explicitly generate target model elements. Except for the entry point rule, this kind of rule must be called explicitly from an ATL imperative block. The called rule syntax is shown in listing 3.8.

```plaintext
[entrypoint]?
rule  rule_name (parameters) {
  ...
  varn : var_type_n = init_expn;
}

[using]{
  var1 : var_type1 = init_exp1;
  ...
  varn : var_type_n = init_expn;
}

to
out_var1 : out_type1 (bindings_1),
out_var2 : distinct out_type2
foreach (e in collection) (bindings_2)
  ...
out_varn : out_type_n (bindings_n)
}

[do{
  statements
}
```

Listing 3.8 ATL called rule syntax

After the *rule* keyword the rule name must be defined. A called rule name must be unique within an ATL module and can optionally be declared as the transformation *entrypoint*.

A called rule accept parameters It is composed of three optional blocks referred by *using*, *to* and *do* keywords. Compared to a matched rule, a called rule has no *from* section and its *to* section is optional. The semantic of the available sections is similar to those defined for the matched rules.

The listing 3.9 shows one rule example:

```plaintext
rule Participant2Hyperlink {
  from
  p : Prot!ProtocolPerson (Prot!Protocol.allInstances()->select(e | e.participants->includes(p))->notEmpty())
  to
  h : Docu!Hyperlink (text <- p.name, target <- p.organization)
}
```

Listing 3.9 ATL rule example
In the example above (Listing 3.6), the from block will only be executed if source model element ProtocolPerson satisfies the condition defined in the from block. The condition is an OCL expression and it checks whether instance $p$ of class ProtocolPerson that is defined in source model Prot, is included to collection participants of class Protocol. The features of instance $p$ are then copied to the features of class Hyperlink that is defined in target model Docu. The to section creates for each of this matched element a target element which features are initializes with the values of variable $p$.

### 3.2 Discussion

Different observations of transformation problems have shown that for diversity of scenarios different techniques are required. A logical question would be if it is possible to handle these scenarios by a single transformation language. The fact that QVT provides three different languages, implicitly imply that this is probably not possible. ATL like QVT also covers imperative and declarative paradigm to address several transformation scenarios to different solutions.

OMG proposes QVT as a general purpose transformation language, similar to the role that XSLT plays in the XML domain. Different works consider domain-specific transformation language as being better suited for implementation in some specific transformation problems. The idea of using domain-specific language (DSL) [31] is already supported by the model transformation communities with several DSLs having emerged in the past.

Domain-specific transformation languages offer several advantages for users and developers. DSLs are more suitable for end-user programming because of their better readability, write-ability, expressiveness and high abstractions. Their concise nature and domain fitting notations make DSLs up to a certain degree self-documenting. On the other hand, they have some drawbacks, like for example, high efforts for development and maintenance, which are needed for a new language or tooling and language performance.

In general, no transformation language considered in this thesis offers all the features that were mentioned in previous sections, making the developer struggle with the decisions of choosing the most appropriate transformation language for their transformation problem requirements.

Experience with ATL has shown that this is a very powerful language. Even though it covers broad aspects in model transformation and offers most of the features, complex transformations are harder to be implemented than in QVT. Both languages encourage users to use the declarative constructs.

In cases where there is structural similarity between source and target metamodels, the declarative approach is possibly the most appropriate way for defining the transformation description, since model objects can be easier related to the corresponding transformation object. Metamodels which differ structurally and where model elements have no semantic
equivalence are more complex to transform. For this situation, imperative approaches are more suitable. Hybrid approaches are therefore more suitable in cases where the metamodels have structural similarity but direct mapping of model objects is not possible.

A more detailed evaluation of model to model transformation language that is based on the classification of [4], and an analysis of the interoperability between ATL and a set of model transformation languages can be found in [47].
4 The ECMT Language

The ECore Model Transformation (ECMT) Language is a new unidirectional stateless model transformation language, implemented as an external DSL in Java. ECMT transforms instances of ECore metamodels into new instances of ECore metamodels and is mostly based on declarative patterns to match against model elements.

This chapter gives an overview of ECMT language. In section 4.1, an overview of ECMT language is given. The relational approach is described in section 4.2. The basic features of ECMT language are described in section 4.3. The pattern language is defined in section 4.4. A description how ECMT language matches the source model elements and create target model elements are described in sections 4.5 and 4.6 respectively. A description about of how to relate source and target model elements is given in section 4.7. In section 4.8, an overview of approaches of tracing models elements is given.

4.1 Language Overview

The ECMT Language was initiated to enable the transformation between instances of ECore models, which may differ structurally but consist of elements with semantic equivalence.

Transformation description is defined through the concrete language syntax. The abstract syntax is represented as an ECore metamodel.

The ECMT language is able to define relations between source and target elements. It is able to match a set of elements in source models according to a pattern language and create a set of elements in target models as in relations description defined.

The language aims to be easily used by users without expert programming experience, which implies avoiding that the user makes too much effort on how the transformation description is programmed.

The transformation allows multiple source models and multiple target models.

A target element can be defined from even more than one transformation rule; therefore, more than one rule can provide a value for the same element property. Due to the fact that the rule order is explicit, the creation of target model elements is also explicit.

Patterns used for matching the model elements can be reused in multiple transformation rules. Transformation rules can match a single, a set of model elements or collections; the iteration over collection is implicit.

The language aims to make data typing dynamically, allowing the transformation description to match and create elements at different metalevels.
4.2 The Relational Approach

The term *relation* is defined in the encyclopedia of Wikipedia as follows: “*In mathematics, especially set theory and logic, a relationship is a property that assigns truth values to combinations (k-tuples) of k individuals. Typically, the property describes a possible connection between the components of a k-tuple. For a given set of k-tuples, a true value is assigned to each k-tuple according to whether the property does or does not hold*”

The relational approach is based in the mathematical concept of *relations*. A transformation between the source and target models is specified as a set of relations that must be held between the source and target models elements. The relational approach enables declarative specification of transformations where declarative statements specify relations between elements in source and target models.

An ECore model can be interpreted as a set of elements which include: Packages, Classes, Attributes and References.

According to the dentition of *relation*, a relation $R$ over the sets $M_1, \ldots, M_n$ is a subset of their Cartesian product, written $R \subseteq M_1 \times \ldots \times M_n$, where in our case, $M_1, \ldots, M_n.$ are the models referred as set of elements.

The ECMT Language is based on the relational approach which makes it a declarative language that is mainly focused on *what* is to be transformed into *what*. Due to the fact that procedural information is hidden, the language is more compact and easier to comprehend from the end-user. In ECMT Language, a transformation description is defined through a set of declarative statements with the intention of keeping a transformation description as simple as possible for the end-user, avoiding the effort of using complex algorithms in transformation descriptions.

Based in the relational approach, the ECTM language offers good solutions for certain scenarios in model transformation situations, including the described scenario of extracting semantic information from documents as presented in previous section.

Even though the relational approach is considered as the most appropriate language paradigm for ECTM language requirements, it also has certain drawbacks, like for example, making complex transformations between models with no structure similarity and semantic equivalence, more difficult to be implemented.

4.3 Basic Features

An EMT transformation consists of mainly two sections: an *import* section and a *transformation rules* section. The *import* section is used to define the files or libraries needed in the transformation description and is followed by the *transformation rules* section. A *transformation rules* section is identified by a name within a module and consists of one or more statements called rules, the ordering of which is explicitly given.
A statement can specify a relation between source and target elements and is called creator rule or it may consist of other statements which effectively can be functions with a number of parameters that are used for matching model elements. The latter are user-defined patterns also called getter patterns which either succeed or fail depending on whether model elements match against the given constraints. A relation specifies what target elements are to be created based on certain constraints and arguments, and the values with which element features are to be initialized. A getter pattern can be called from a creator rule; thus, offering reusable code and reducing unwanted redundancy.

As already mentioned, the transformation rules are mainly divided in two categories: the getter patterns and the creator rules. The general form of an ECMT Transformation is shown in Listing 4.1. Terms that are written inside ‘< >’ specify rule sections which are to be defined as in the following sections described.

```
1 import <import references>+  
2 transformation transformation_name  
3 input <input declarations>+  
4 output <output declarations>+  

5 get <key> getter_name <parameter_definition>?  
6   <getter_pattern_definition>  
7 condition  
8   <getter_condition_definition>?  
9 return  
10   <getter_return_definition>  
11 ...  
12 create creator_name  
13 in_model  
14   <creator_in_defintion>  
15 out_model  
16   <creator_out_defintion>  
17 condition  
18   <creator_condition_defintion>?  
19 ...  
20 ....
```

Listing 4.1 General form of an ECMT transformation

The keyword import is followed by a list of file references or libraries that a transformation description needs. These may include different ECore metamodels. After the keyword transformation, the name of the transformation description must be defined. The direction of transformation is specified by the keywords input and output, where the input files are source model and the output files will serve as the target models.

The get keyword defines a getter pattern rule; it is followed by the name of the getter which must be unique within a transformation description and a set of parameters used inside the pattern. The getter pattern definition is done through different variable declarations and assignments for model elements and is responsible for matching model elements which satisfy the condition defined after the condition keyword. The keyword return is used to define the value that the getter returns; it may be a Boolean value that implies whether the getter will
succeed or not in matching the model elements or user defined value. A more detailed explanation of getter patterns is made in the following section.

The creator rules are defined though the keyword create. Based on the relations concept it provides the relationship between source and target model elements, which are specified respectively by in_model and out_model sections. Similar to a getter condition, the condition keyword in a creator rule is used for specifying certain constraints the model elements must satisfy in order the creator rule to be successfully accomplished. A creator rule is identified by its name, which must be unique in a transformation description and is also used for traceability in the ECMT language.

The information for tracing model elements is held during the execution of creator rules from the transformation engine, in form of tuples between a source model element and a target model element.

### 4.4 Pattern Language

The pattern language of ECTM is essentially a set of variables, assignments and predicates. A getter pattern is based on the pattern language and can be used to match both elements in source models and target models. Getter Patterns can be subdivided into three types: Getter If Pattern, Getter All Pattern, and Getter Value Pattern. This division of getters enables an explicit declaration of three types of values a getter can return if the constraints defined in pattern condition are held. This can be useful to the end user because it provides more modularity and expressiveness in the pattern language.

Getter patterns use some user-defined parameters, and offer effectively function-like methods for defining pattern for model elements. They own variables that are defined in the variable declaration clause and enable variable bindings for binding and assigning values.

Getter patterns are used to match elements in source models without having any side effect to the elements, and they can be used to create templates for elements in target models without changing elements in it. They can be used to make queries in both source and target models or to return user defined values or model elements values as in return clause specified.

Getter patterns can be part of another getter pattern and the order of their execution is defined explicitly by calling them from creator rules. In the Listing 4.2 - 4.4, for each different type of getter patterns, a more general form of syntax, is shown.

```plaintext
get if getter_name (parameter_1 : EDataType, ..., parameter_n : EDataType)
{
variable_1 : EDataType;
.. variable_n : EDataType = Expr;
condition
Pattern_Condition
```
The *getter if pattern* is defined with the keywords *get if* that are followed by the name of the getter. It returns a Boolean value depending on whether the matching of model elements will succeed or not. This kind of pattern can be reused in *pattern condition* or as a variable assignment in variable declaration part of a getter pattern.

```java
get value getter_name (parameter_1 : EDataType, ..., parameter_n : EDataType)
{
variable_1 : EDataType;
.. variable_n : EDataType = Expr;
condition <Pattern_Condition>
return Constant
}
```

Listing 4.3 Getter Value Pattern syntax

The *getter value pattern* is defined with the keywords *get value* and returns a constant value of any EDataType declared in the models or returns null in case the matching of model elements will not succeed. This kind of pattern can be reused as a variable assignment in variable declaration part of a getter pattern, or as a condition term in *pattern condition* clause.

```java
get all getter_name (parameter_1 : EDataType, ..., parameter_n : EDataType)
{
EDataType variable_1;
.. EDataType variable_n;
condition <Pattern_Condition>
return Collection
}
```

Listing 4.4 Getter All Pattern syntax

The *getter all pattern* is defined with the keywords *get all* and returns a list of values that is a collection of any EDataType declared in the models or returns null in case the matching of model elements will not succeed. This kind of pattern can be reused as a variable assignment in variable declaration part of a getter pattern, or as a condition term in *pattern condition* clause.

A *getter pattern condition* enables the user to put constraints on model elements that must be held for a getter to succeed. They are defined in a pattern language, the metamodel of which is
shown in Figure 4.1. It has similarity with the metamodel that is specified by MOF of OMG, but is adapted for the needs of ECMT language.

A pattern condition is composed of a set of different terms. Three abstract metaclasses, SimpleTerm, CompoundTerm and Expression deal with Boolean algebra and represent the main classes for defining a pattern condition. The CompoundTerm defines the Boolean operators and SimpleTerm the operands to the Boolean operators. SimpleTerm can be a condition term or an ECMT term, for example a getter if pattern, and may contain Expressions. Expression can be defined through a SimpleExpression which can be a constant of Integer, String, Boolean or Enumerate type, or a variable declared in the getter pattern variable declaration clause. CompoundTerm defines the operator of the pattern language; It can be defined through an AndTerm, OrTerm or NotTerm which respectively represent the AND, OR and NOT, the main operators of the ECMT pattern language. Condition is a boolean valued expression and offers several comparison operators: equal operator defined by ‘==’, greater than by ‘>’, less than by ‘<’, greater equal by ‘>=’, less equal by ‘<=’ and not equal defined by ‘!=’. ECMT Term is a term that can define an instance of ECore class, attribute, reference or an ECMT getter pattern. Variable is the metaclass for the use of variable declaration in expressions. Variables may contain any valid ECore model Type and they can be dynamically typed.

The ECMT pattern language can be extended with new pattern terms definitions, for example the IfTerm or ForTerm that use logic programming for defining more complex operators in a declarative way. The IfTerm use if else condition statements whereas the ForTerm use for loops statements, with the semantic as already known from general programming languages. In the scope of this work, these terms were left out with the intention of keeping the language simple for the end-user. They can be used for the future works based on this language.

Figure 4.1: The pattern language metamodel
4.5 Matching Source Model Elements

The matching of source model elements can be addressed to the two kinds of transformation rules; the getter patterns which make queries on model elements and return instances of any model data types, and the creator rules which enable generation of target model elements based on relationship between source and target model elements. In both cases, the matching of source model elements follows the same principles and concepts.

The matching of source model elements in creator rules is carried out in the rule sections that are defined after the in_models. In_models correspond to the alias of files as from the user defined in the input section. These sections are responsible for matching source elements from different models; where for each of the source models an in_model section must be declared. Model elements to be matched are declared in sections variable declaration clause respectively.

For a certain model element, which is declared in the variable declaration clause of a getter pattern or of in_model section, the language will first prove the validation of this element. That is done by checking the existence of data type and name element in corresponding metamodel. The matching algorithm is intentionally simple; it creates for each matched model element a variable and uses variable binding for making them available for other patterns. The order of patterns in variable declaration clause is significant, since it implies the correctness of scope where variable is used or assigned.

The variable binding is based on principles of QVT specification which is briefly described in the following:

A variable binding is a unique set of values for all variables of the pattern. It binds all variables of the pattern to a value other than undefined, and where all predicates of the pattern evaluate true. For example, a match of an empty pattern that has no variable and no constraints results in one variable binding, with an empty set of values; a match of a pattern with one variable and no constraints results in one variable binding per instance of the type of the variable in the model corresponding to the model type of the metamodel where the pattern is to be matched; a match of a pattern with two variables and no constraints, result in one variable binding per element of the Cartesian product of all instances of the type of one variable and all instances of the type of the other variable in the model corresponding to the model type of the metamodel where the pattern is to be matched [14].

Since pattern may depend on other patterns, predicates declared for a pattern as constraints may refer to variables declared in another pattern. Matching a pattern that depends on another pattern requires that the variable binding used in it to be valid, for ensuring that all variables have a value when evaluating the predicates.

ECTM Language uses the variable binding in patterns declared in creator rules and uses no variable binding for getter pattern. The name of a getter pattern is bound to the value of the variable declared in the return clause. It is an open issue, if variable bindings should be used for getter pattern and it may be taken in consideration for future work based on this language.
Failing of matching a pattern in a *getter pattern* will return the values *false* or *null* based on the type of the *getter pattern*, whereas failing of matching a pattern in a *creator rule* will cause the variable binding put an *undefined* value to the corresponding instance of variable in the pattern that could not be matched.

### 4.6 Creating Target Model Elements

The generation of target model elements is done from *creator rules*. The sections responsible for building templates for target model elements are identified by *out_models*. Templates are specified through the pattern language and may fail or succeed depending on whether the template is valid or not. A template of target model elements is considered valid if its constraints hold and if the instance, the type and the features values of model element declared in the template are asserted. In ECMT language, the matching of target model elements is based on the target metamodel that is explicitly given, and the matching of model element instances is based on the tracing information.

When a creator rule executes, it produces one or more target elements. The number of elements produced is determined by the number of variables declared in the corresponding section. For each target element, a new variable is declared in template. The pattern language of templates differs from that used for matching model elements.

The templates not only relate element features to certain values, but they also enable assignment of values, given as parameters in creator rules, or as values returned from a *getter pattern*; therefore, the pattern language for templates can also support imperative constructs.

Template constraints are declared in the *condition* section of a *creator rule*. They are defined through the pattern language and deal with Boolean algebra for checking source and target elements conditions that must hold for a *creator rule* to be executed. Listing 4.5 shows a more detailed form of a *creator rule*.

```plaintext
1  create creator_name (parameter_1, … , parameter_n)
2  in_model
3  variable_1 : EDataType;
4   ..
5  variable_n : EDataType;
6  out_model
7  variable_1 : EDataType;
8   ..
9  variable_m : EDataType = value_i;
10  condition
11  <Rule_condition_definition>
```

Listing 4.5 Creator Rule Syntax

The ECMT language offers no inheritance mechanism of *creator rules*, with the intention of keeping transformation description simple, but to allow transformation rules to be reused and
specialized, this mechanism could be taken in consideration for future works that are based on this language.

4.7 Relating Source and Target Model Elements

The relations between source and target elements are defined by the creator rules. The file alias in_model and out_model specify the direction of transformation. The file alias declared as input models imply that the model elements defined in this section are source model elements that are used only for matching, whereas those alias declared as output imply that the model elements are target model elements that are to be created from the source elements defined in the in_models sections.

The new created target elements and the source model elements which led to their creation build a relation in the ECMT language; consequently, relations are specified as mapping between objects of the same type. Mappings are done explicitly through assignments that are made in templates of target model elements. ECMT language enables the specification of mappings based on objects of metamodel instances, but also on objects in metamodel levels. In both cases, transformation engine checks for object validation in the corresponding model. Target model elements will be created if their declaration in template conforms to the data types defined in the metamodel and if they have not already been created from another template before. If a target model element was already created, then only its features will be updated. Each time a target model elements is created it produces some tracing information based on the relations a target model element was part of.

Each variable declaration in out_model sections corresponds to a new target model element. For each variable a new target model element will be created. The assignments in template specify how the new created elements relate to the source elements that the creator rule had to match.

The relations declared on the transformation imply that the end-user knows the structure and the semantic equivalence between objects of the models taking place in it. Defining relations also imply that the end-user knows what is to be transformed into what.

Since transformations can take source and target models that have different structures and a different number of elements, the relation between them cannot be assumed to be bijective. A definition of bijective transformation is made in [46] as follows: A transformation between metamodels M and N given by a relation R is bijective if for every model m conforming to M there exists exactly one model n conforming to N such that m and n are related by R, and vice versa.

The relation R defines a transformation description and is built upon the relations that each creator rule defines. In [46] is explained why bidirectional transformations should not necessarily be based on a bijective relation. Even though ECTM language is based on relations for mapping the source and target elements, which makes it a mostly declarative language; the use of getters introduces some imperative code makes it difficult for ECMT to
produce a bidirectional transformation. In some cases, when the end-user uses no *getter patterns* or if the use of such *getter patterns* keeps the language fully declarative, an ECTM transformation can be bidirectional.

Bidirectional transformation in ECMT would also require some effort from the end-user to redefine the directions in *creator rules*.

### 4.8 Tracing Model Elements

*Creator rules* are responsible not only for generating the target elements, but also for providing tracing information about how source and target elements are related to each other. Tracing information is based on the relationship between source and target elements and defines from which source elements a target element was created. This information is especially important for matching target elements in the target models that are to be generated.

In ECMT language, templates of target elements are used for generating target elements, whereas the matching of target elements is made only for those elements which have been already created.

In general, there are three main mechanisms for relating elements. In [30] are given the following descriptions:

**Relating target elements via tracing information**, this mechanism uses tracing information to relate which source element led to the creation of particular target elements. There exist various tracing information creation mechanisms. In [30], tracing information in a rule is created between all source elements matched by non-nested model element patterns, with the intention to minimize the source elements used in tracing information, and all target elements produced by model element expressions. The tracing information can be used in different scenarios of model transformation: it can be used for enabling users to visualize and understand transformations in their models. It also enables change propagation in the language. It enables querying dynamically created target elements during transformation execution, such as in ECMT language.

**Distinguishing target elements by an identifier** is a technique that can be seen as a slight variation on distinguishing target elements by tracing information. It defines a *target element identifier* which contains, at a minimum, the concatenated identifiers of all source elements which led to the creation of the target element. This method gives the possibility to store extra information in the target element identifier and allow, for example, transformation to encode information which may not be present in tracing information.

**Relating elements by key** involves relating elements by a collection of attributes, which collectively and uniquely identify any given element. Thus, the transformation only needs to know about the relations of the source and target elements’ keys; other information of model elements is essentially irrelevant. This is an approach that is supported by QVT Specification
and is similar to notion of key in the standard database. This method has two main problems: firstly, an extra effort is expected by the user for defining the keys, which may not always be trivial. Secondly, after the initial transformation, the user cannot safely change source or target elements’ keys since changing a key’s value means the transformation may no longer correctly relate the elements.

ECMT language uses the tracing information as an internal utility of the language for managing target model elements. The tracing information is produced after each executed creator rule and cannot be accessed or changed during the transformation by the end-user.

Tracing information could be helpful for: debugging, detecting conflicts between model elements, providing an overview of the mapping between the model elements for a better understanding of the transformation description to the user or enabling incrementality in transformations. These features are not part of the ECMT language, but can be considered as a possible extension for future works based on this language.
5 Language Implementation

In the previous chapter, the theoretical concept of a model to model transformation language was presented. This chapter provides details about the implementation of the ECMT language. The architecture of the language implementation is presented in section 5.1. The transformation engine and its environment are described in section 5.2. In section 5.3, the ECMT abstract syntax is shown, whereas the concrete syntax is shown in section 5.4. An example of an ECMT transformation description between two example models will be presented in section 5.6, as an illustration of the syntax and the semantics of the ECMT language.

5.1 Architecture

The ECMT language is implemented as a Java external Domain Specific Language (DSL). This key design choice was based on several important features of an external DSL and considerations about the ECMT language requirements, as is explained in the following.

In figure 5.1 is shown the architecture of ECMT language implementation.

![Figure 5.1: Implementation architecture of ECMT language](image)

ECMT language is designed with the intention of being expressive, easy to use by the end-user and related to the domain language of the document models that are to be transformed. To provide this, ECMT makes use of several advantages of DSLs.
Some advantages of DSLs are expressiveness, productivity and maintainability. They are also more suitable for the end-user programming. A relevant definition of DSLs is given in [48] as follows, “DSLs are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with General Programming Languages (GPLs) in their domain of application”.

It is implemented as an external DSL, because in contrary to an internal DSL, its syntax is not limited to any other language. An internal DSL is built on top of a host language, which implies restrictions for the language’s syntax and evaluations rules.

A transformation description is provided as a DSL script to the parser which based on the semantic model of the ECMT grammar, generates model element classes. A transformation description is written based on the ECMT grammar that is defined through its concrete syntax. The concrete syntax is a natural way of describing the syntactic structure of a DSL. For parsing the grammar file a parser is needed. For this purpose, ECMT language uses a Parse Generator. A Parse Generator uses the grammar file to generate a parser which can be updated and regenerated each time the grammar is being changed.

The transformation between the source and target model is performed by the Transformation Engine. The Transformation Engine is responsible for interpreting and executing the transformation rules and makes use of the generated libraries by the parser. It takes one or more models or metamodels as input, and produces one or more models or metamodel as output, as in the transformation description defined. The following section gives a more detailed description of the transformation engine and its environment.

5.2 Transformation Engine and Environment

The ECMT transformation engine represents an important part of ECMT language implementation. It is written in Java and relies on several libraries generated in Java by a Parser Generator.

Figure 5.2 shows a process diagram, which describes the components and the data, involved in the whole transformation process. In the following, the main components and the steps of a transformation process are described.

1. A transformation description is defined based on the ECMT concrete syntax. It is parsed and evaluated by the transformation language parser and based on the semantic model of the language, a set of Java classes are provided as representation of the transformation rules. These rules are used by the transformation engine to perform the transformation.

2. Source metamodels and target metamodels are used as input files. They are defined in the transformation description and are read by a metamodel language reader. A set of Java classes are generated and made available to the ECMT interpreter. These classes represent the classes defined in the source and target models.
3. Source models are input files and parsed by a model instance reader which leads to the creation of a set of instance objects.

4. The transformation execution is based on the relations between model elements in the **creator rules**. Interpretation of **getter patterns** and **creator rules** is performed.

5. The output of the transformation process is one or more instance files containing the elements of the target models which conform to the corresponding target metamodels.

The environment of the transformation engine is the Eclipse Modeling Framework (EMF). It uses XText as a Parser Generator, ECore API for accessing metadata generically, and EMF persistence framework for working with XMI resources.

The source and target metamodel are ECore models, whereas the source and target models are represented as XMI files. The transformation description is written with Xtext editor and Xtext parser libraries are being used. In the following, the components that were mentioned above are described in more detail.
5.2.1 Xtext Parser

Xtext [47] is part of Open ArchitectureWare (oAW) which is an Eclipse based platform suited to develop DSL corresponding to the MDE approach. Xtext enables text to model transformations where the metamodel is derived from an Xtext grammar file.

The grammar describes the concrete syntax of the DSL and is transformed to an ANTLR grammar by Xtext. It also generates the abstract syntax of the DSL as an ECore metamodel. The parser generated of the ANTLR grammar creates model elements conforming to the metamodel. The generated metamodel corresponds to an AST specification for the DSL.

Xtext offers a number of helper artifacts to embed the parser in an oAW workflow and an Eclipse based editor that provides an outline view, syntax highlighting and other DSL coding facilities. Listing 5.1 shows the getter pattern syntax written in Xtext.

```
getter:
  "get" key=keydef id=ID parameter_def=parameter_definition?
  
  local_def+=local_variable_declaration*
  condition_def=condition_clause_definition?
  return_def=return_definition?
  
  
  "\}"
keydef:
  value="if" |value="all"|value="value";
condition_clause_definition:
  "condition" cond=sCondition;
return_definition:
  "return" (parametername=ID | booleanConstant);
parameter_definition:
  open="(" (parameters+=parameter ("," parameters+=parameter)*)? ")";
parameter:
  name=ID ";" dataType=ID;
local_variable_declaration:
  name=ID ";" dataType=ID ";" expression=Expr ";";
```

Listing 5.1 Getter Pattern Syntax in XText

The Xtext approach of defining DSL has a disadvantage. It suffers from the inability to create a custom metamodel and the XText language itself is limited, so only simple DSLs are possible. Nevertheless, Xtext is a good choice for ECMT language since ECMT concrete syntax aims also to be simple.

The abstract syntax metamodel presented in the next section, is the abstract syntax that the ECMT concrete syntax conforms to, but is not used by Xtext for generating the parser. Xtext abstract syntax representation is generated automatically from the concrete syntax, and is not presented in this thesis.

The Xtext grammar is similar to the Backus-Naur Form (BNF) grammar and consists of a set of rules. Each rule can be composed out of a combination of three token types: keyword,
identifier or string, and references to other rules. It offers modifiers such as multiplicity, optionality or alternation. String token or identifiers token are not customizable; nevertheless, it is possible to define a simple custom token type for strings.

5.2.2 ECore Reader

ECore is the metamodel language of EMF. In ECMT implementation, source metamodels and target metamodels are defined in ECore and are read by an ECore reader. The EMF runtime framework enables the use of ECore metadata at runtime.

Since ECore is considered as a model, EMF provides a generated API for it. The Java package `org.eclipse.emf.ecore` contains interfaces corresponding to each of the classes that are modeled in ECore with the corresponding getters and setters for the attributes and references.

Since metadata information is available at runtime, the ECore API is used for querying the structure of the model, and the reflective `EObject` API for accessing or manipulating generically object instances.

5.2.3 XMI Reader

To allow the exchange of metadata information of models or metamodels, the OMG has defined the XML Metadata Interchange Standard (XMI). This standard describes how MOF models can be mapped to XML. EMF offers a powerful framework for model persistence with XMI support for ECore as its main exchange format. This enables interchangeability between ECore models.

In ECMT implementation, source models and target models are represented in XMI format, and are read by a XMI reader using the EMF persistence API.

5.2.4 Rules Interpreter

Rules Interpreter is implemented in Java and is responsible for the translation and execution of transformation rules. It uses EMF libraries for model and metamodel representations, and Xtext parser for generating the transformation rules representation. An ECMT transformation is translated to certain Java classes with a number of function and methods for matching, creating and holding tracing information.

The interpreter recognizes the first rule from the keyword `start` of the `creator rules` and uses the standard fetch-translate-execute cycle for the `creator rules` that follow. It recognizes the `getter patterns` through the `getter ID`, which is called in a `creator rule`, and executes them automatically. If a `getter ID` is called in another `getter pattern`, recursive execution is needed.

For interpreting rules, the interpreter must consider some predefined conventions:
Accessing the attributes is done through specification of the class name followed by a ‘.’ symbol and the attribute name.

Accessing the collections is done through specification of the class name followed by a ‘->’ symbol and the collections name.

Classes, attributes and collections can be defined through predefined variables in variable declaration clauses of the rules.

For iterating a collection, an explicit statement is needed; the `for all var in collection_name {statements}` is responsible for iterating through the collection with name `collection_name` and returns each element in the `var` variable. The statements between the `{}` are executed for each element. The variable `var` can be only accessed inside the scope of the `{}`. The data typing of variable `var` is done dynamically, in this case, from the interpreter.

To enable work on both levels, the interpreter must be able to work both with instance data of the models and metadata of the metamodels. For this reason, the interpreter makes the difference between the name and the value of a classifier or a feature by using a ‘.name’ after their definition. The ‘.name’ convention implies that only the metadata of this classifier or feature is to be accessed. This convention also implies that the specification of the classifier or the feature that is to be accessed is done through a typed variable, the name of which is to be filtered.

The comparison operators include: ‘==’, ‘!=’, ‘<’, ‘>’, ‘<=>’. Which operation can be effectively used, depends on the comparative values. If an invalid comparison operation is used, an exception will be thrown.

The variable assignment is done by using the ‘=’ symbol. Data typing is done dynamically in runtime.
5.3 Abstract Syntax

The abstract syntax represents the metamodel of language grammar. It contains semantic information about the grammar and it may have several concrete syntaxes as instances. In figure 5.3, the abstract syntax of ECMT language is shown. The concrete syntax defined in the following section is the BNF representation of ECMT grammar that conforms to the abstract syntax metamodel.

The ECMT abstract syntax metamodel is used for defining an ECMT grammar instance, but it is not directly involved in the ECMT implementation. Xtext dynamically generates an abstract syntax metamodel based on the grammar file.

![Diagram of abstract syntax of ECMT]

**Figure 5.3: Abstract syntax of ECMT**
5.4 Concrete Syntax

The complete concrete grammar of ECMT language in Backus–Naur Form is given in Listing 5.1. It conforms to the abstract syntax given in the previous section. For the implementation of ECMT language, the concrete grammar was adapted to the Xtext grammar editor without changing the semantic.

```
1 model ::= imports | transformation_definition;
2 imports ::= ‘import’ import_reference*;
3 import_reference ::= ‘file’ ID ‘;’;
4 transformation_definition ::= transformation | transformation_element*;
5 transformation ::= ‘transformation’ ID
6 ‘input’ filemappings ‘;’
7 ‘output’ filemappings ‘;’;
8 filemappings ::= ID ‘as’ alias_ID(‘ID ‘as’ alias_ID)* ;
9 transformation_element ::= getter | creator;
10 creator ::= ‘start’? ‘creator_ID parameter_definition?’
11 ‘{’
12 local_variable_definition*
13 in_clause_definition +
14 out_clause_definition +
15 condition_clause_definition?
16 ‘}’;
17 in_clause_definition ::= ‘in’ alias_ID (clause_var_def(‘,’clause_var_def)*);?
18 out_clause_definition ::= ‘out’ alias_ID (clause_var_def(‘,’clause_var_def)*)? | for_all_def;
19 clause_var_def ::= ID (‘:’ dataType_ID )? ‘=’ Expr | getter_call;
20 for_all_def ::= ‘for’ ‘all’ Interator_ID ‘=’ ID | getter_call
21 {;
22 (clause_var_def(‘,’clause_var_def)*)?
23 };
24 getter_call ::= getter_ID ‘(’ (constant (, constant)* ‘)’ )? ;
25 getter ::= ‘get’ key ID parameter_definition?
26 ‘{’
27 local_variable_definition*
28 ‘}’
29 condition_clause_definition?
30 return_clause_definition? ;
31 key ::= ‘if’ | ‘all’ | ‘value’;
32 condition_clause_definition ::= ‘condition’ sCondition;
33 return_clause_definition ::= ‘return’ ID | booleanConstant;
34 parameter_definition ::= (‘(parameter (‘,’ parameter) ’)’) ? ;
35 parameter ::= ID (‘:’ dataType_ID;
36 local_variable_definition ::= ID ‘:’ dataType_ID ‘:=’ expr ‘;’;
37 expr ::= constant;
38 constant ::= intConstant | stringConstant | booleanConstant;
39 intConstant ::= INT;
40 stringConstant ::= STRING;
41 booleanConstant ::= yesConstant | noConstant;
42 yesConstant ::= ‘True’;
43 noConstant ::= ‘False’;
44 sCondition ::= andCondition | orCondition | simpleCondition;
```

49
5.5 Example Models

In the following, two models that are used for the transformation example in the following section are shown. They are made to this work available from M.Sc. Sebastian Meyer and represent parts of Protocol Document and Word Document metamodels respectively.
Figure 5.4: Document ECore Metamodel
5.6 Transformation Example

In the following, a simple ECMT language transformation example is shown. It is based on the metamodels that were given in the previous section.

```ecore
import tran_mod : file "models/ProtocolModel.ecore"; file "models/DocumentModel.ecore";

transformation proc2docu
input "instantmodels/sample-Protocol.xmi" as in_model
output "instantmodels/out.xmi" as out_model

get value getTopic
{
  e : EClass = "Protocol";
  condition
    e.topic != ""
  return e.topic
}

get all findPersons (class : EString, collect : EString)
{
  e : EClass = class;
  return e->collect
}

creator start createDocument
{
  in_model
    e: EClass = "Protocol"
  out_model
    d: EClass = "Document"
    h: EClass = "DocumentHeading"
    h.text = getTopic()
    h.level = 1
    d->content = h
}

creator writeParticipants
{
  in_model
    persons : EClass = "ProtocolPerson"
  out_model
    t: EClass = "DocumentTable"
    for all I = findPersons("ProtocolPerson", "participants")
    {
      r: EClass = "DocumentTableRow"
      c: EClass = "DocumentTableCell"
      paragraph : EClass = "DocumentParagraph"
    }
}
```
The previous example represents a transformation description in the ECMT language. The running module start with the `import` section defined after the keyword `import`. After it, the import metamodels files `ProtocolModel.ecore` and `DocumentModel.ecore` are specified.

The transformation name is defined as `proc2docu` and is followed by the name of the instance models `ProtocolModel.ecore` and `out.xmi` which are used as input and output files respectively. The getter pattern defined by the name `getTopic` queries for a class called `Protocol` and returns the value of its attribute with the name `topic`. This getter is then called explicitly in the creator defined by name `createDocument`. The `createDocument` creator is used for producing two classes in the target model. A `Document` class and a `DocumentHeading` class. It also initiates features of these classes, as in the following explained:

In `h.level = 1`, the attribute `level` of the class referred by the `h` variable is accessed using the ‘.’ convention and is initiated with a constant. In `h.text = getTopic()`, the attribute `text` is initiated by value the getter `getTopic` returns. In `d->content = h`, the collection content is initiated with a class `h`.

The getter pattern `findPersons` takes two parameters which are used to define the name of the class and the name of its collection. This getter is used to return a list of elements that belongs to the reference the name of which is passed as parameter.

The creator `writeParticipants` generates as first, a class `DocumentTable` in the target model. The `findPersons` getter is then called through a `for all` convention for iterating the collection explicitly. For each element of this collection, several elements of target model are being produced.
6 ECMT Language Evaluation

In this Chapter, an evaluation of the features supported by ECTM language based in the criteria catalogue [4], which has been discussed in section 2.3.2, is presented.

6.1 Transformation Rules

In Figure 6.1 and 6.2, the diagrams of transformation rule features and rule domain features are depicted, which follow an overview and explanations of the corresponding features that ECMT language supports.

**Rule features**

Considering the declarative approach of creator rules, ECMT basically enables a clear syntactic separation. Both the source and target models have separate corresponding parts, represented by in and out sections in the creator rules. It enables application condition through the definition of constraints, by using Boolean conditions in condition section of a creator rule. Application condition specifies constraints that must be true in order for the rule to be executed. ECMT enables bidirectionality only if a transformation description is defined by creator rules without using imperative code injected by getter patterns; otherwise, only unidirectional transformations are possible. ECMT support some intermediate structures for holding tracing information. The tracing information is not based in a metamodel and it is only temporarily available. Parameterization is the use of parameters for allowing passing values. ECMT supports the use of control parameters, which are used as control flags for implementing policies. ECMT supports no generic parameters, which are used for passing data types, and no higher-order rules for passing other rules as parameters. ECMT supports no reflection for allowing reflective access to transformation rules during the execution of transformations.

![Figure 6.1: Features of transformation rules [4]](image-url)
**Domain features**

ECMT Language uses ECore as the *domain language* for the metamodels involved in the transformation. It requires an explicit declaration of input and output models. As a consequence of the ECMT rule execution approach, domains can only be specified as either input or output, but not combined. ECMT also does not support *dynamic mode restriction* for restricting the static modes at execution time. It uses *patterns* with term based structure that may have variables, expression or statements embedded. The pattern syntax is represented in a concrete, textual way. It supports the use of *variables* for holding intermediate elements or elements from source and target models.

*Logic* expresses computations and constraints on model elements. ECMT uses a declarative approach language paradigm based on relations defined by *creator rules* and functions like methods for querying model elements from *getter patterns*. The *creator rules* are not executable, whereas the *getter patterns* represent executable logic.

*Value Specification* is done through binding values to variables or values are specified through constraints. ECMT uses syntactical *typing*, based on the types defined in the metamodels that involved in the transformation. However, the typing is only performed at run-time, which means that for example the assignment of a String value to an attribute of type Integer is not detected only during the execution of the transformation.

![Diagram of domain features](image)

**Figure 6.2: Features of rule domain [4]**

### 6.2 Rule Application Control

In the following, the diagrams of two rule application control aspects, the location determination and the rule scheduling, are depicted respectively in Figures 6.3 and 6.4.

**Location determination**

Due to the fact that ECMT language follows a declarative approach for defining *creator rules*, its *application strategy* is not interactive. It uses a deterministic rule application based on the order of rule elements explicitly specified by the user.
**Rule Scheduling**

*Rule Scheduling* is related to the order of rule applications. The rule scheduling form of ECMT can be both implicit and explicit. The *creator rules* are called implicitly each time they are defined in a transformation description, whereas the *getter patterns* are an exception for this behavior, because they are only executed when explicitly called in another *getter pattern* or in a *creator rule*. *Rule selection* is done explicitly in ECMT language and supports no conflict resolutions. ECMT support *rule recursion* for getter patterns that are called from the same getter pattern.

The *phasing* refers to the transformation process that may be organized into several phases, with each phase having a specific purpose, and only certain rules can be invoked in a given phase. ECMT language provides the users the possibility to specify *getter patterns* for defining explicitly when certain transformation should take place, but it does not offer a specialized build in phasing mechanism.

Figure 6.3: Features of Rule Application Strategy [4]

Figure 6.4: Features of Rule Scheduling [4]
6.3 Rule Organization

ECMT language supports basic *modularity mechanisms*, because it allows importing rules from different files. Using the import statement, another ECMT transformation can be imported, and its rules can be used in the current transformation. *Reuse mechanisms*, such as rule inheritance and logical composition are not supported. The *organizational structure* of a rule file is independent from both source and target model.

In Figure 6.5, the diagram of rule organization features is depicted.

![Figure 6.5: Features of Rule Organization [4]](image)

6.4 Source Target Relationship

ECMT language enables horizontal and exogenous transformations with the Eclipse Modeling Framework (EMF) as its technical space. For each transformation a new target model must be defined. Working with an existing target model would cause the regeneration of that target model after the transformation; consequently, neither in-place, nor out-place transformation are supported by this language.

In Figure 6.6, the diagram of source target relationship features is depicted.

![Figure 6.6: Features of Source Target Relationship [4]](image)
6.5 Directionality

ECMT language enables mostly unidirectional transformations. Due to the declarative approach of ECMT language for defining the creator rules, bidirectional transformation are also possible; the latter would be possible, when no imperative assignments are used in the transformation description, an example of imperative assignments, is the initialization of a new target element by using a getter setter.

In Figure 6.7, the diagram of directionality features is depicted.

![Directionality Diagram]

Figure 6.7: Features of Directionality [4]

6.6 Traceability

Tracing is considered as the runtime footprint of transformation execution. ECMT language creates the traceability links automatically after the execution of the creator rules. The tracing information is stored separately and uses no predefined metamodel for this information.

In Figure 6.8, the diagram of directionality features is depicted.

![Traceability Diagram]

Figure 6.8: Features of Traceability [4]
6.7 Summary

The criteria catalogue of transformation language features that was presented in this chapter is based on the classification described in [4]. It provides an overview of all characteristics of existing transformation languages and it is used for evaluating the ECMT language.

Most of the model transformation languages are designed with a certain goal in mind; consequently, making languages differ considerably in their features and overall usage. Because of this fact, it is sometimes difficult for the user to choose a model transformation language as the most appropriate for a certain situation.

The evaluation provides the user with an overall view of features that ECMT language supports. By providing the necessary information for comparing ECMT language with other existing languages, the evaluation makes a decision easier, based upon the appropriateness of the ECMT language for a certain usage scenario.
7 Conclusion and Outlook

Model transformation languages are a key prerequisite in Model Driven Engineering. Model to model transformation language is a special case of model transformations that enables the transformation between models in a system.

The goals of this thesis are to provide a concept and an implementation for a model to model transformation language and to provide a solution for enabling the extraction of semantic information from structured text document. Structured documents are defined in ECore metamodelling language and represent the models that are to be transformed.

ECMT language was designed taking in consideration different language requirements and environment. ECMT aims to be easily used by users without expert programming knowledge, to provide the necessary expressiveness for defining transformation description, and to conform to the domain language the models are defined. It follows the relational approach, which is mainly focused on what is to be transformed into what and hide procedural information from the end-user. This keeps an ECMT transformation description simple and prevents the effort of using complex algorithms.

ECMT was implemented as a Domain Specific Language (DSL) in Java, for using DSL advantages like expressiveness, productivity and maintainability. It is implemented as an external DSL to avoid restrictions in language syntax and evaluations rules.

Another important aspect that was taken in consideration for the design of ECMT language is the representation of models that are to be transformed. Source and target models are defined in the same modeling language ECore. Models differ in structure but model elements have semantic equivalence. Since models have no structure similarity it is difficult to make a direct mapping of model elements; consequently, the transformation between models is difficult to be made with a language that is fully declarative. For this purpose ECMT introduces the use of function like methods to define patterns for model elements; nevertheless, making complex transformations between models with no structure similarity and semantic equivalence with this approach would be more difficult to implement.

For the creation of a target model, ECMT requires that the metamodel it conforms to, should be explicitly given. For a particular situation, where the target model is not available, an implicit generation of the target model would have been more appropriate.

In general, transformation languages offer those features that best fulfill the requirements of the given transformation and MDE environment where the transformation takes place. Like most of the existing languages, ECMT does not provide all the features; nevertheless, it enables transformation based on those features that were chosen as the most appropriate considering language requirements and a certain transformation scenario.
ECMT language was restricted only for the problem of extracting semantic information from models written in the same language that differ in structure, but have semantic similarity. This makes ECMT have some limitations.

Considering future work, additional features like inheritance, modularity, update and in-place transformations, or a more powerful pattern language could be taken into consideration.
Bibliography


[33] IKV Technologies. medini QVT Available at: http://projects.ikv.de/qvt. 2007

[34] Fujaba development team. Fujaba Available at http://www.fujaba.de, 2007
[35] MOFLON development team. MOFLON Available at http://www.moflon.org


A ECMT Language Grammar in Xtext

grammar org.xtext.example.eCMT.Ecmt with
org.eclipse.xtext.common.Terminals

generate semt "http://www.xtext.org/example/eCMt/Ecmt"

Model:
   imports=importstatement transformation=transformation_def;

importstatement:
   "import" name=ID ":" references+=import_reference*;

import_reference:
   "file" filename=STRING ";";

transformation_def:
   transformation=transformation elements+=transformation_element*;

transformation:
   "transformation" name=ID "input" input=filemappings "output"
output=filemappings;

filemappings:
   fileidmapping+=fileIdMapping (',', fileidmapping+=fileIdMapping)*;

fileIdMapping:
   file+=STRING "as" id=ID;

transformation_element:
   getElement=getter | setElement=setter;

setter:
   "creator" start="start"? name=ID
parameterdef=parameter_definition? "{" 
local_def+=local_variable_declaration*
   in_clause=in_clause_definition out_clause=out_clause_definition
where_clause=where_clause_definition? "}";

in_clause_definition:
   in="in" id in_var+=clause_var_definition*;

out_clause_definition:
   out="out" id in_var+=clause_var_definition*
assignments+=assignment_clause* for_all+=forAll_clause*;

assignment_clause:
   to=varAccess "=" from=Expr ";";

forAll_clause:
   "for" "all" iterator=ID "=" (var=varAccess | getter=getter_call)
"{" 
   vars+=clause_var_definition*
   assignments+=assignment_clause* 
}";
clause_var_definition:
  (name=ID "=" dataType=ID "=" expr=Expr) ";" | getter_call ";";

getter_call:
  getter_id=ID "(" parameter_value+=ParameterValue? ("," parameter_value+=ParameterValue)* ")";

ParameterValue:
  stringValue=STRING | intValue=INT | idValue=ID;

getter:
  "get" key=keydef id=ID parameter_def=parameter_definition? "{"
    local_def+=local_variable_declaration*
    where_def=where_clause_definition? return_def=return_definition?
  "}";

keydef:
  value="if" | value="all" | value="value";

where_clause_definition:
  "condition" cond=sCondition;

return_definition:
  "return" (const=booleanConstant | var=varAccess);

parameter_definition:
  open="(" (parameters+=parameter ("," parameters+=parameter)*)? ")";

parameter:
  name=ID "=" dataType=ID;

local_variable_declaration:
  name=ID "=" dataType=ID ("=" expression=Expr)? ";";

Expr:
  constant | var=varAccess | getter_call;

constant:
  intConstant | stringConstant | booleanConstant;

intConstant:
  value=INT;

stringConstant:
  value=STRING;

booleanConstant:
  yesConstant | noConstant;

yesConstant:
  value="true";

noConstant:
  value="false";

sCondition:
  andCondition | orCondition | simpleCondition;
simpleCondition:
    name=varAccess op=sCompare expr=Expr;

varAccess:
    varNames+=ID+ (accesstype+=("." | "->") varNames+=ID)*;

andCondition:
    left=simpleCondition "and" right=simpleCondition;

orCondition:
    left=simpleCondition "or" right=simpleCondition;

sCompare:
    equal="==" | not="!=" | lt="<" | gt=">" | ge=">=" | le="<=";